CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Specification and Verification of Dynamic Topology Systems

On the Applicability of Query- and Data-Type-Reduction-based Abstractions

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inform. Bernd Westphal

Gutachter:
Prof. Dr. Werner Damm
Prof. Dr. Andreas Podelski

Tag der Disputation: 17. April 2008

Dem König.

Abstract

Formal methods for embedded systems currently mainly focus on single components or fixed configurations of finitely many components. Examples for the former case are open finite-state systems as models of single discrete controllers in a discrete environment. Examples for the latter comprise models of clusters of controllers with fixed inter-connection, that is, no controllers are added or removed at runtime and the inter-connection is not re-configured, e.g., the numerous controllers in a car for airbags, braking assistance, etc.

These concepts are not sufficient when many autonomous systems interact. A characteristic example is the *Car Platooning* application as studied by the California PATH project. The intention of car platooning is to reduce fuel consumption by dynamically merging cars into platoons where they drive with significantly reduced safety distance and hence benefit from slipstream. To faithfully model car platooning, there have to be means to describe (i) unbounded appearance and disappearance of cars within the system "highway", (ii) topologies, that is, selective connections between cars like between leader and follower, and (iii) (asynchronous) communication.

We propose to extend the particular finitary abstraction *Data Type Reduction* (McMillan, 2001) known for parameterised systems to the class of *dynamic topology systems* as characterised by (i)–(iii). As computational model, we introduce labelled transition systems where states are labelled with graphs. This allows us to model nasty but possibly critical effects like dangling links, i.e. connections to already disappeared processes.

Furthermore, we introduce a first-order extension of classical temporal logic. It is process-oriented in the sense that quantified variables range over processes and follow their evolution over time. We can express properties requiring that, for instance, *the particular* car, which initiated a merge, will finally complete the merge. The semantics of this logic for the first time completely and explicitly treats issues such as pre-mature disappearance of processes.

By re-stating the finitary DTR abstraction in terms of the graph-labelled transition system, we gain insight into the potentials and limitations of this technique; beforehand, it has only been described in terms of a construction procedure. Individual-oriented properties, which are easily lost in many other abstractions, are essentially preserved by following what we call the *spotlight principle* (Wachter & Westphal, 2007).

Finally, we demonstrate the applicability of this approach by sketching a translation from a relevant fragment of UML and of the DCS language (Bauer, Schaefer, Toben & Westphal, 2006) into graph-labelled transition system, the latter allowed us to establish safety and liveness properties for the car platooning case-study.

Zusammenfassung

Formale Methoden für eingebettete Systeme konzentrieren sich heutzutage überwiegend auf einzelne Komponenten oder endlich viele Komponenten in einer während der Laufzeit statischen Konfiguration. Beispiele für den ersten Fall umfassen offene, endliche Transitionssysteme als Modelle eines einzelnen diskreten Controllers in einer als diskret angenommenen Umgebung, Beispiele für den zweiten Fall sind Modelle von mehreren Controllern die etwa durch Bussysteme in einer festen Topologie verbunden sind, etwa verteilte Controller für Airbags, Bremsassistent etc. in modernen Fahrzeugen. Insbesondere werden zur Laufzeit keine Controller hinzugefügt oder entfernt und die Verbindungstopologie wird zur Laufzeit des Systems nicht rekonfiguriert.

Diese Konzepte sind nicht ausreichend, wenn mehrere solcher, für sich autonomen, Systeme miteinander interagieren sollen. Ein charakteristisches Beispiel hierfür ist die ,,Car Platooning" Anwendung, die seit den 1990er Jahren im California PATH Projekt betrachtet wird. Die Idee des Car Platooning ist, das̈ Fahrzeuge auf der Autobahn autonom Konvoys bilden, innerhalb denen mit verringertem Sicherheitsabstand gefahren wird, um Energie zu sparen und die Fahrzeugdichte zu erhöhen. Um Bremsmnnövern weiterhin sicher durchführen zu können, nimmt das erste Fahrzeug im Konvoy die Rolle des ,,Leaders" ein, der den anderen Fahrzeugen, den sog. ,,Followers", Bremsmnnöver rechtzeitig ankündigt, um deren Reaktionszeit zu minimieren.

Um Systeme wie das Car Platooning adequat zu modellieren, Bedarf es Mitteln zur Beschreibung von (i) unbeschränktem Erscheinen und Verschwinden (,,appearance and disappearance") von Fahrzeugen im System ,,Autobahnabschnitt", (ii) (Kommunikations-)Topologien, d.h., logische Verbindungen zwischen Fahrzeugen, wie etwa zwischen dem Leader und seinen Followers und (iii) asynchroner Kommunikation. Und zwar sowohl zur operationellen Beschreibung von Implementierungen als auch zur deklarativen Beschreibung von Anforderungen an Implementierungen. Zur Analyse, ob eine Implementierung gegebene Anforderungen erfüllt (,,model-checking"), bedarf es Abstraktionstechniken, da das Problem, aufgrund der unbeschränkten Anzahl von Sub-Systemen, im allgemeinen unentscheidbar ist.

Wir untersuchen, inwiefern und unter welchen Randbedingungen die für Parametrisierte Systeme bekannte finitäre Abstraktion ,,Data Type Reduction" (McMillan, 2000) auf die allgemeinere, durch (i)–(iii) charakterisierte Situation anwendbar ist.

Zu diesem Zweck führen wir, in Ermangelung eines geeigneten operationellen Modells in der Literatur, das Konzept der *Evolving Topology Transition Systems* (ETTS) ein, im wesentlichen zustandsbeschriftete, unendliche Transitionssystem, deren Zustände mit Graphen beschriftet sind. Unser Modell erlaubt insbesondere, ,,unangenehme" aber kritische Effekte wie logische Verbindungen zu bereits verschwundenen Fahrzeugen zu modellieren. Als deklarative Beschreibungssprache verwenden wir eine Variante von

Temporallogik erster Stufe, die insbesondere den Effekt des vorzeitigen Verschwindens von referenzierten Sub-Systemen adäquat behandelt, was in existierenden Vorschlägen in der Literatur erstaunlicherweise oft nicht erfolgt.

Durch die Formulierung von ,,Data Type Reduction" im ETTS Modell verstehen wir nun erheblich besser, wie diese Abstraktion funktioniert und welche Eigenschaften sie für ETTS erhält; zuvor ist im wesentlichen nur eine Implementierung bekannt gewesen (McMillan, 2000). Das zugrundeliegende Prinzip haben wir in (Wachter & Westphal, 2007) zum ,,Spotlight Principle" generalisiert.

Wir demonstrieren die Anwendbarkeit des Verfahrens für ein relevantes Fragment der bekannten UML und für die spezialisiertere Sprache DCS (Bauer, Schaefer, Toben & Westphal, 2006), indem wir eine ETTS Semantik angeben, die uns erlaubte Sicherheits- und Lebendigkeitseigenschaften für die ,,Car Platooning" Fallstudie nachzuweisen.

Acknowledgements

"Long time span, many people met, many people to thank..."

First of all, I want to thank my supervisor Prof. Dr. Werner Damm for the persistent support and the amounts of freedom and trust, allowing me to host (some say strange) courses, write (some say strange) publications, and supervise (...) a number of interesting student's projects. This may have significantly delayed completion of this thesis, but in retrospect I wouldn't want to trade most of it.

I'm in particular grateful for the opportunity to live in one of the head-quarters of SFB TR/14 AVACS, from the planning in 2003 to the report and application phase in 2007 it has been a unique experience. First to name, the incredible amount of people from Saarbrücken and Freiburg, spanning all kinds of topics, opinions, and experience levels, I had the opportunity to listen to, discuss with, and get to know over time.

From the department, I want to thank the following persons. First to name, the former room-mate, regular co-author, and travelling companion Tobe Toben. I particularly appreciate taking over a number of organisational duties during the "hot phase" of writing this thesis. Well-informed circles wonder whether he spends more time with me than with his own family. Which is certainly wrong, yet he's always been open to discuss paper titles, and raise an eyebrow or frown on any weird idea or proposal, just to participate in the most cooperative form if it promised to be fun in any sense.

Even earlier, I met Jochen Klose, then coordinator of the SPP 1064 project USE on Live Sequence Charts, which has been the offspring of quite some publications and provided first support of my work. Thanks for always critically teaching me the basics of how research projects are run.

Further to name is Ingo Schinz with whom I spent long afternoons on the sunny side on the executive suite of the Uhlhornsweg site debating crystallisation, memory models, analyses, red fog, and SSL. Most of my (some may say) thorough understanding and intuition of the concepts of the, in a broad sense, object-oriented domain stem from this time as loose associate of the IST WOODDES and OMEGA project.

From the many other people, the following stand out as I benefited from their enormous and rock-solid expertise in their respective domains. Tom Bienmüller and Jürgen Bohn for the technical and in particular efficiency- and scalability-oriented views on symbolic model-checking, and Hartmut Wittke for discussions on automata theory, temporal logic, and their special relation.

For distracting and non-technical sessions I thank the former room-mate Eckard Böde and the other members of the "Mittagskarawane", comprising (over time) Ralf, Jan, Thomas, and Matthias, and floor-neighbours like Christian.

From the AVACS-subproject-S2 family, the two other principal investigators Prof. Dr. Andreas Podelski and Prof. Dr. Reinhard Wilhelm are to name, who granted us a notable amount of freedom, which we (some may say) scrupulously exploited for our own benefits, but complemented with advice whenever needed.

Jörg Bauer and Ina Schaefer have been our closest collaborators. Thanks are in order for a thorough introduction into Saarland-ian culture and cuisine, and always inspiring work on rejections`^W` publications. Finally, this work certainly benefited from the intensive collaboration with Björn Wachter on the publication based on his Master's Thesis.

Last but not least I'd like to thank (although this doesn't buy him a cake) the "Grand Old Dame of the Department", Jürgen Niehaus, for his always friendly management of organisational issues like project application and reporting, travelling, student worker funding, and my own funding.

Contents

List of Figures

List of Tables

Part I.

Introduction and Preliminaries

1. Introduction

This work belongs to the domain of formal methods-based development of hard- and software, that is, the idea that errors in hard- and software systems can be reduced by formally and declaratively stating what the expectations on of the system (the *requirements*) and to (at best) automatically check whether a given operational implementation satisfies its requirements.

What we just called the implementation can be manifold: it can range from an abstract model of the system in languages like UML to the actually deployed implementation in form of a programming language. But we always envision this duality between declarative requirements and operational implementation.

Such approaches have three ingredients: there is a need for two rigorous formal languages for precise specifications of the requirements on the one and the implementation on the other hand, and there is a need for techniques to formally verify whether the implementation satisfies the requirements. Although this work is formally self-contained, we have to assume some background on the related ideas and concepts, otherwise we refer to textbooks such as [33].

Formal methods in this sense are on their way into the industrial practice, yet primarily in the domain of finite, statically structured systems. That is, systems with a fixed finite number of processes and static communication links between processes. Examples are isolated embedded systems, single controllers, or strictly bounded networks of such controllers. Assuming finiteness and fixed interconnections is perfectly reasonable where it applies, as it matches the final deployment in trains or cars.

New questions arise if multiple of such super-systems, like trains or cars, are supposed to interact in an intrinsically dynamic setting. Cars, for instance, can freely enter and leave a highway and different relative positions may require adjustment of interconnection relations. Even if the single objects are finite state, that is, if all local variables of a sub-system have finite domain, the whole system may grow arbitrarily large. We'll see examples like car platooning and a handler pattern in railway management applications in more detail in Sections 1.1.2 and 1.1.1).

In this work, our goal is to investigate a certain technique for automated formal verification for the class of systems with an unbounded number of objects or processes which seemed promising in the pre-study [177], in particular for requirements in form of scenarios.

Note that, from the at least five "sources of infiniteness" in a system [66], namely the number of processes, infinite-domain data, asynchronous, message-queues based communication, recursion and call stacks, and real-time, we address exactly the first. Only if the local representation of links, e.g. the address of the platoon leader car, counts a data, the second aspect is addressed. But in the sense of arithmetical data like a real-valued car-speed.

One our main findings is that, in some cases, we can treat this first aspect orthogonally to the other ones. That is, if we have a procedure for finitely many processes with infinite-domain data, like hybrid systems, then for a system with an unbounded number of hybrid processes we're confident that one can derive from our work strategies to construct an abstraction that lies in the class of bounded-number hybrid systems and is thus treatable with the methods for hybrid systems that assume finitely many processes.

More precisely, subject of this work is an investigation of the applicability of the approach presented in [124, 127] for parameterised systems. The difference from parameterised systems to truly dynamic topologies is, even if each process in the latter executes the same program locally, that a parameterised system describes all infinitely many, but as such *finite* and fixed topology instances.

Taking the step to Dynamic Topology Systems as we shall call them raises a couple of issues, both on the side of the computational model of implementations and on the side of the requirements specification language. Furthermore, the abstractions employed in [124, 127] are proved sound and successfully applied [28, 31], but far from understood, in the sense that before this work there has been no assessment of which properties can be expected to hold in the abstraction.

To get a more detailed idea of this problem domain and the proposed solutions, this chapter briefly answers the most pushing questions and gives some background for the whole work.

First of all, Section 1.1 introduces two examples of DTS, a railway system with autonomous cars and car platooning on highways, to give a better intuition of the characteristics of this class of systems. In Chapter 3 on the computational model, we'll address the Car Platooning application and its relevant properties in more detail, and in Chapter 10 we'll revisit both applications as case-studies and apply our methodology.

Assuming that DTS can precisely and unambiguously be described, Section 1.2 points out what kinds of properties are desired to be formally verified for a DTS, in particular what we understand as scenarios. A complete treatment follows in Chapter 4 on the specification logic.

After Section 1.3, it shall be clear what we understand under the term *formal verification* and why formal verification in this sense is both difficult and relevant for DTS. Section 1.4 briefly outlines the approach we investigate in order to overcome the apparent difficulties in DTS verification. The whole Part III is dedicated to this topic.

Finally, Section 1.6 gives a more informed overview over the structure of this work in terms of the concepts introduced in the previous sections.

1.1. What is a Dynamic Topology System?

A Dynamic Topology System in the broader sense is basically a system of communicating processes with a dynamically changing communication topology and a dynamically

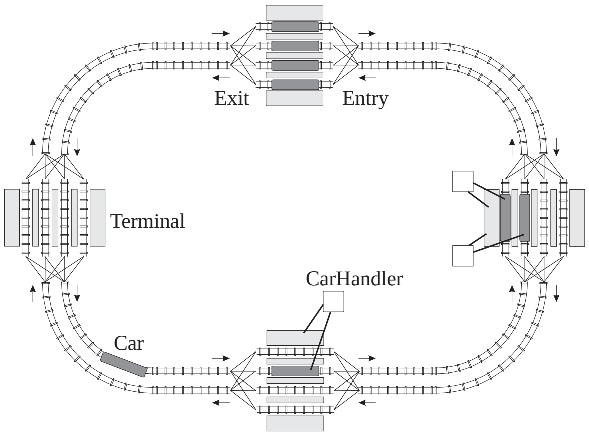


Figure 1.1.: **Automated Rail Cars System.** Closed variant with 8 cars and 4 terminals.

changing number of processes without an upper bound on the number of processes existing simultaneously. It is possibly best understood on concrete examples, so we shall give two prominent ones in the following.

1.1.1. Example 1: The Automated Rail Cars System

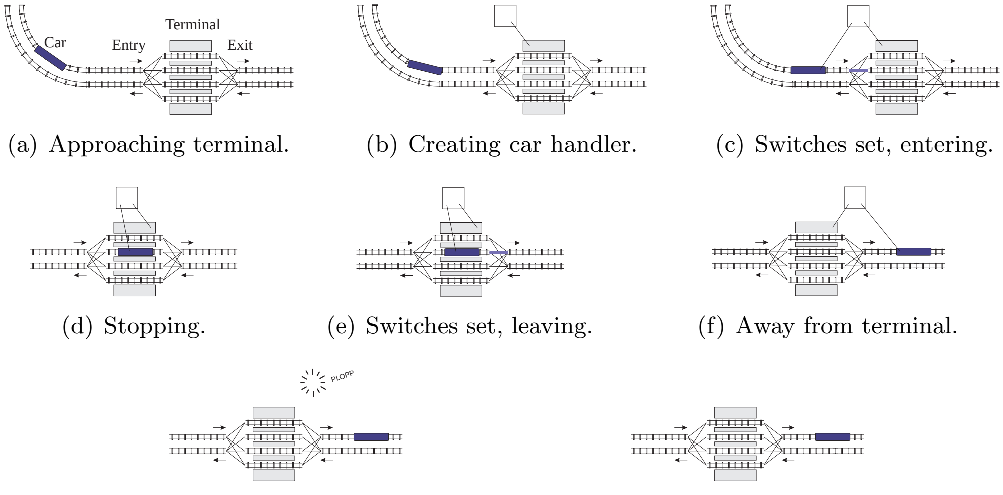
The Automated Rail Cars System [79] (ARCS) serves as a case-study for a particular executable object-oriented modeling employing a variant of the Unified Modeling Language [138, 141, 140].

In the original design, it is a closed system of N autonomous rail cars shuttling on one-way tracks between M terminals (cf. Figure 1.1). Each terminal has a number of platforms, and entry and exit switches. The arrival and departure procedure for rail cars in particular comprises reserving and setting the switches. It employs a *handler pattern*, that is, each time a car approaches a terminal, a new handler is created which has access to terminal internals and negotiates between car and terminal.

A complete arrival and departure procedure is shown in Figure 1.2. It starts by a car approaching a terminal and reading the terminal's address from a track-side barrier (Figure 1.2(a)). On the arrival request by the car, the terminal creates a handler object (Figure 1.2(b)) which reserves a platform at the terminal and negotiates the switches to be set accordingly.

If everything is prepared, the car receives admission to enter from the handler (Figure 1.2(c)). If the admission doesn't arrive in time, the car stops at a safe distance in front of the terminal and awaits the admission.

The car handler is also responsible to support the car when leaving the terminal, that is, the car leaves the terminal only after it has been notified by the handler that the switches are set accordingly (Figure 1.2(e)). If the car is a safe distance away from the terminal, it initiates destruction of the car handler, which then gives up its lock on the outgoing switches and disappears (Figure 1.2(g)). The rail car is again freely cruising afterwards.



(a) Approaching terminal.
(b) Creating car handler.
(c) Switches set, entering.
(d) Stopping.
(e) Switches set, leaving.
(f) Away from terminal.
(g) Destroying handler.
(h) Cruising freely.

Figure 1.2.: **Arrival and Departure Procedure.**

Note that handler is newly created and later destroyed and that the communication topology changes dynamically: the rail car communicates with processes which didn't even exist before. In order to keep the interface between cars and terminals lean, and thus in particular the amount of radio-based communication low, the handlers will typically not be allocated within the cars but on the side of the terminal.

Even in the closed system, it is then not trivial to determine an upper bound for the number of car handlers needed at any point in time without intimate knowledge of the physical circumstances. Allocating as many handlers per terminal as there are cars may not be sufficient if one handler is still alive and engaged in the departure procedure while the car approaches the terminal again and another handler is needed to start the arrival procedure.

Furthermore, we can easily imagine the system to be open, that is, to admit that rail cars enter and leave the scope of the model freely. As the procedure shown in the figure doesn't depend on a fixed number of cars, there is then not even a fixed number of cars.

In practice, that is, for the final implementation it is of course essential to know how much local memory is needed for the controller to work properly where a finite upper bound will be determined. But for verification of the correctness of the arrival and departure protocol one doesn't want to depend on an upper bound but the aim is to establish properties of the protocol in general, with arbitrarily many processes in the system.

1.1.2. Example 2: Car Platooning

A second popular and even more epitomic example of Dynamic Topology Systems Car Platooning as studied by the California PATH project [84, 53].

The idea is that cars shall be equipped with radio-based communication and controllers

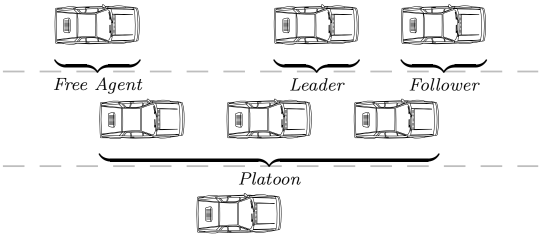
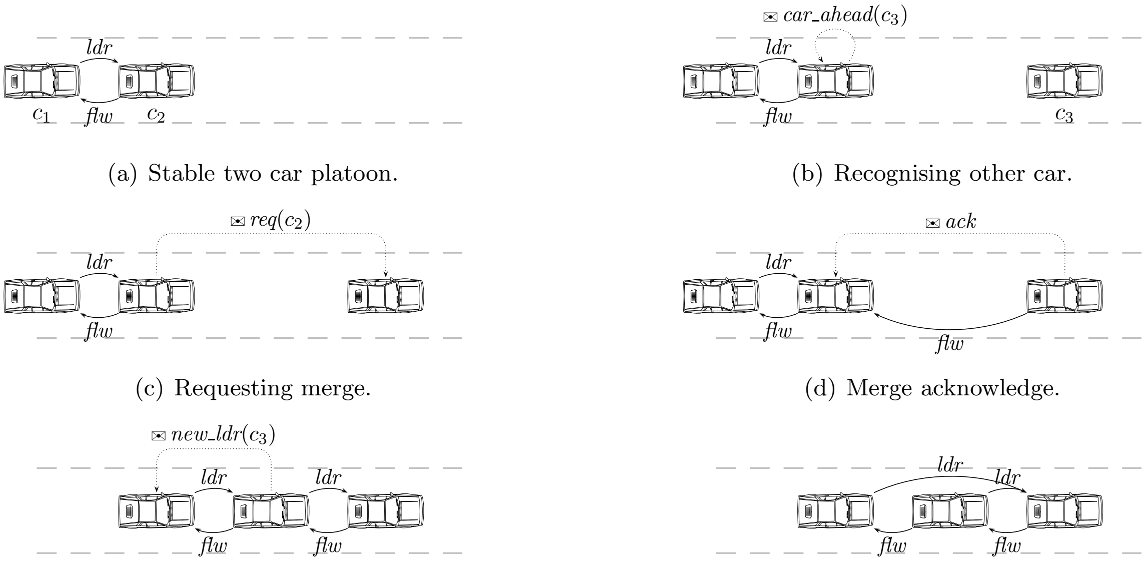


Figure 1.3.: **Car Platooning**



(a) Stable two car platoon.

(b) Recognising other car.

(c) Requesting merge.

(d) Merge acknowledge.

(e) Announcing new leader to followers.

(f) Stable three car platoon.

Figure 1.4.: **Merge Procedure.**

to negotiate the formation of convoys (or platoons). A platoon comprises at least two cars and we distinguish the front car, called *leader*, and the others, the *followers*. Cars not participating in a platoon are called *free agents* (cf. Figure 1.3). As the platoon leader is responsible to announce braking manoeuvres in time, cars may drive with reduced safety distance and safe energy and highway space.

The basic procedures of car platooning are merge and split, the more complex change lane procedure employs both. A merge is initiated by a leader or a free agent who recognises another leader or free agent driving in front.

For example, consider the two-car platoon in Figure 1.4(a). If a third car c_3 enters the highway and gets into reach of the corresponding sensor of the platoon leader c_2, the latter sends a merge request message comprising its own identity to the car in front (cf. Figure 1.4(b) and 1.4(c)). If the front platoon (or free agent) is willing to join, it adds the requester to its followers and replies with a positive acknowledge (cf. Figure 1.4(d)). Then the back leader, in the example this is c_2, notifies all its followers about the join by a message announcing the new leader (cf. Figure 1.4(e)). If all these notifications have been processed, in the example this is c_3, a new stable three-car platoon has been established (cf. Figure 1.4(f)).

A platoon is split into two platoons or free agents in a similar procedure. A split is initiated by the car that wants to leave the platoon, which may be the leader or a follower. The initiating car becomes leader of a new platoon or a free agent.

A third procedure, change lane, comprises becoming a free agent, that is, initiating two split procedures in the worst case, and negotiating with the cars on the neighbour lane and the lane behind to avoid collisions. For example in Figure 1.3, if the free agent on the bottom-most lane wants to change to the middle lane, it would recognise the neighboured three-car platoon and negotiate that it needs space on the middle lane.

This space could be provided by the three-car platoon accelerating, decelerating, or splitting in half. Provided sufficient space, the bottom-most free agent would negotiate with the cars on the top-most lane *who* changes lane in order to avoid collisions.

Note that cars freely enter and leave the highway, and dynamically change communication topology. Cars consider other cars as leader which didn't even exist in their world, which is the highway, until recently.

In practice, it is assumed that there is a maximal platoon length per highway segment communicated by the roadside controller. But the aim is to verify the correctness of the merge and split protocols independent from such bounds.

1.1.3. Dynamic Topology Systems vs. Parameterised Systems

The first example, the ARCS, is in the original definition similar to classical parameterised systems where there are finitely many finite programs $P_1, \dots, P_n$ executed by fixed numbers of $K_1, \dots, K_n$ processes. In the ARCS we had three programs, P_1 for the cars, P_2 for the terminals, and P_3 for the car handlers, and $K_1 = N$ car-, $K_2 = M$ terminal-, and $K_3 = L$ handler processes executing them, assuming we had an upper bound L on the number of car handlers. The challenge is then to prove properties (cf. next section) for all infinitely many instantiations at once.

The ARCS and Car Platooning applications are different to typical parameterised systems in the following aspects:

1. We're not interested in infinitely many finite instances, but a single instance whose extension may grow unboundedly as processes appear and disappear freely in the system.

2. Processes have individual named relations (or links), that is, there are some rail cars approaching a particular terminal now while some aren't. This is in contrast to typical parameterised systems where every process has access to every processes' local memory.

3. Processes communicate according to the individual named relations, that is, a car handler only asks the car it manages to enter the terminal (unless it accidentally considers the wrong car as its managed one).

Note that the particular kind of communication, asynchronous with event queues or synchronous, rendezvous-like, is not important.

These are the constituent aspects of our notion of DTS: that there is a dynamically changing number of processes, each with a unique identity, and that there may be dynamically changing links between processes.

1.2. What Requirements Shall DTS satisfy?

In previous section, we already pointed out a possible error in the ARCS example: a car handler shall not consider the wrong car to be its car as otherwise it may grant terminal access to a car which actually should still wait.

A bit more precise, we're interested in properties of the form

For each car handler h and each car c, whenever h sends a grant message to c, then h is known as the responsible handler to c. (1.1)

In general, there is a universal quantification over individuals (or processes) and a generic temporal property, for instance, LTL. That is, the focus is on single processes, or certain processes standing in a particular relation to each other, which behave in a certain manner when traced over time. It is *not*, as with typical parameterised systems, on the entirety of processes in a state, for instance, that at most one process in a state is in a critical section when considering mutual exclusion examples.

The atomic terms in the temporal properties will comprise the following.

- Generic predicates, for example comparing local variables of multiple processes.
- Predicates indicating whether a process has just been created or will immediately be destroyed to express, for instance, that a car handler manages one and the same car throughout its lifetime.
- Link navigation to refer to links from one process to another, for example, a car handler would have a link to the car it is responsible for.

A particular example of temporal logics for DTS is Mett, which has been introduced in the joint work [9] to complement our definition of the DTS description language DCS (cf. Chapter 10).

It is basically LTL with quantification over DTS processes and atomic propositions referring to process creation and destruction, local state of processes, presence of named links between two processes, and in addition sending and reception of events (cf. Chapter 4 for details and a detailed comparison to other approaches).

In Mett, the natural language property (1.1) would read as follows.

$$\mathsf{G}\, \forall\, h : \mathit{CarHandler}, c : \mathit{Car}\,.\, \mathit{send}[\mathit{grant}](h, c) \rightarrow \mathit{conn}[\mathit{resp}](c, h) \tag{1.2}$$

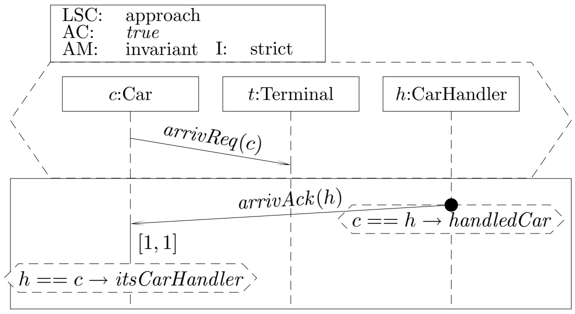


Figure 1.5.: **Live Sequence Chart.** If c contacts terminal t, then it is granted access by the h responsible for it and establishes the link *itsCarHandler*.

Another particular example of temporal properties for DTS are Live Sequence Charts [42, 43, 97, 81] (LSCs), one of the most prominent representative of languages scenario.

LSCs are basically a conservative extension of the well-known Message Sequence Charts by modalities for chart elements and whole charts, which for instance allow to require progress along instance lines. Each vertical so-called instance line binds, like a quantified logical variable, to a correspondingly typed process in the system and gives requirements on how and when this group of processes interacts. For example, Figure 1.5 is a requirement on the ARCS example (cf. Chapter 3 for details).

We will discuss both, Mett and LSCs, in more detail for DTS in Chapter 10.

Note that there is a principal and crucial difference to the indexed logics employed with parameterised systems. The quantifiers in indexed-logic range over finitely many processes, which are present in each system state. That is, these properties can be unfolded into a finite non-quantified formula enumerating all processes of the N-instance of the parameterised system.

With DTS there is true dynamics. So the range of logical variables covers an unbounded set of identities. And in addition, we have to be prepared for what we call pre-mature disappearance. That is, individuals denoted by a logical variable may disappear in the model while we still want to evaluate a logical term on them to decide whether a temporal property holds or not.

Similar issues arise when considering reconfigurable systems in the sense of e.g. [103], that is, systems of finitely many finite sub-systems with principally fixed topology but with the option to replace sub-systems at run-time.

This can be seen as the finite fragment of DTS: there is no unbounded creation and destruction because each sub-component is replaced by exactly one other one, but there are the same critical transitional situations where the old sub-component just disappeared and the new one is not yet fully operational.

1.3. What is Formal Verification of DTS, and Why is it Difficult and Relevant?

Given a formal model M of a DTS, like the ARCS or Car Platooning, the formal verification (or model-checking) problem is to prove that M satisfies the requirement Φ, at best with an automated method to establish or refute this relation.

As there is dynamic and unbounded creation and destruction of processes, DTS are inherently infinite-state systems and the model-checking problem is in general undecidable. For this reason, finitary abstractions are employed which yield a finite-state transition system that simulates the original one.

That is, the abstraction shall be *sound*, i.e. if a property can be established for the abstract system, for instance by finite-state model-checking techniques, then it also holds in the concrete, infinite-state system.

But it is typically not *complete*, that is, not all properties holding in the concrete system can be established in the abstract one; verification of the abstract system may yield so-called spurious counter-examples that lead to a violation in the abstract system but are not possible in the concrete one. The challenge is to chose an abstraction, that is, the procedure mapping concrete to abstract systems, such that it is finite, effectively computable, and preserves interesting properties.

Formal verification of DTS is relevant because in particular UML provides dynamic creation and destruction of interlinked objects, thus UML models immediately classify as DTS.

Assuming one of the many proposed formal semantics for it, the UML is an adequate language to describe systems like the ARCS and Car Platooning in a model-based development process, where a model of the system under design is built first, formally verified, and then provides the starting point for automatic generation of the actual programming language implementation or to serve as a “golden device” from which test-cases for the final, manually conducted implementation can be derived.

1.4. What is the Approach?

We investigate the application of an abstraction technique that has been introduced under the name Data-Type Reduction in [124, 129, 127] and which is typically used together with a technique later called Query Reduction by [185].

In order to state our observation and to outline our investigations, we have do delve a little bit deeper into these two techniques on the example they’ve originally been presented for.

1.4.1. Hardware Verification By Compositional Model Checking

Query and Data-Type Reduction are actually part of a larger strategy [124, 129, 127] that demonstrated how the techniques and procedures of the time are able to formally

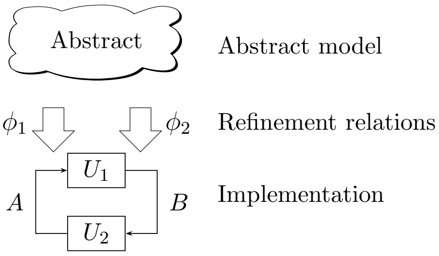


Figure 1.6.: **Compositional Refinement Verification.** [127]

verify properties of parameterised systems (cf. Section 1.1.3) using finite-state model checking.

The strategy actually serves to establish a notion of refinement (cf. Figure 1.6). In the concrete example, there is an abstract description of how a sequential micro-processor executes instructions ("Abstract model" in Figure 1.6). The task is to check whether an out-of-order-execution implemented by components U_1 and U_2 as observed on buses A and B adheres to the refinement relations ϕ_1 and ϕ_2.

The whole strategy employs the following techniques.

1. Temporal Case Splitting [123, 127] breaks the apparent cyclicity when checking ϕ_1 and ϕ_2 for U_1 assuming U_2 behaves as the abstract layer and vice versa (cf. Figure 1.6).

2. Query [124, 127] and Data-Type Reduction [129, 127] treat replicated components.

3. Uninterpreted Functions [27, 14, 15] abstract from large combinatorics that compute certain data values when the focus is on the control part, e.g. the transportation of data over buses.

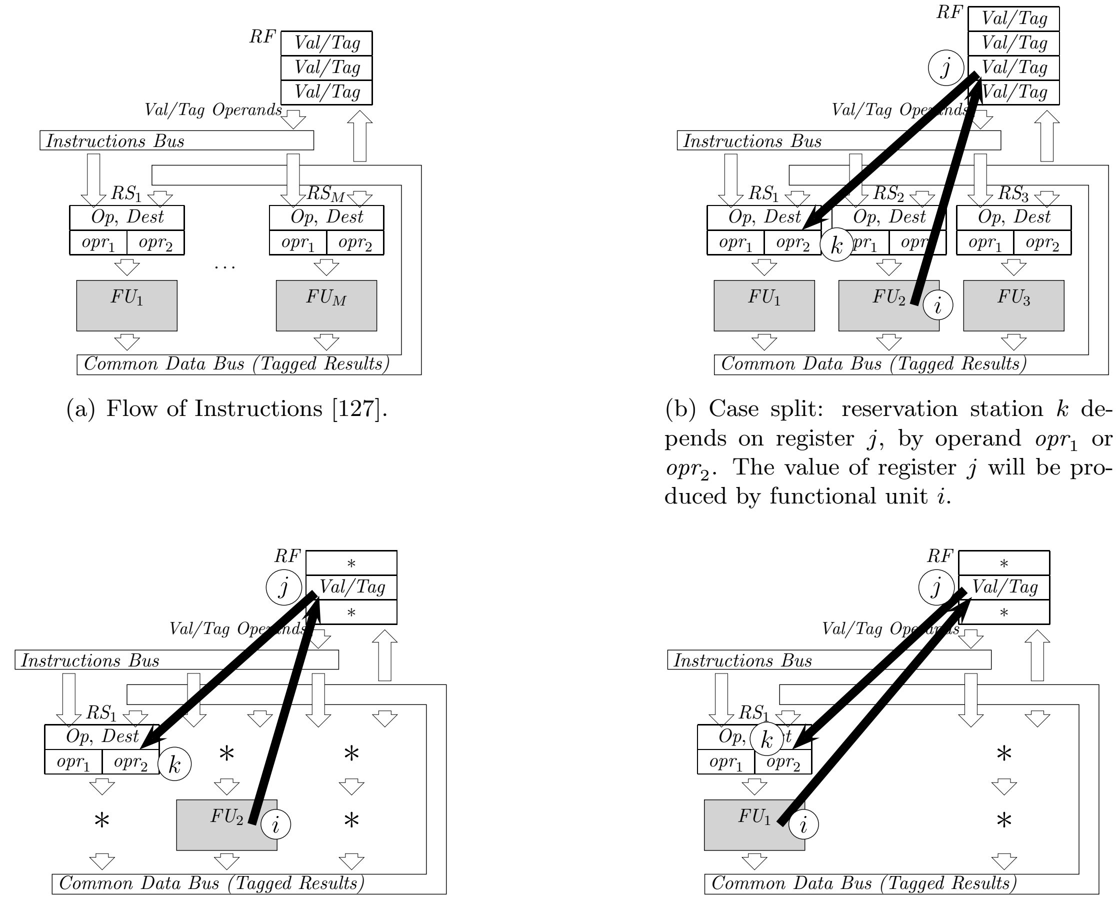
4. In addition, for replicated components organised in certain topologies like a ring, inductive arguments can be applied.

In [144], the approach is described it as "neither compositional nor verification" but the best combination of deduction and model-checking to that time. At the end of Chapter 9, we'll provide an own opinion which justifies to view in particular the Data-Type Reduction abstraction a special variant of compositional verification in the classical sense.

In order to discuss the strategy in more detail, the following Section 1.4.1 briefly introduces the case-study on which the whole strategy is demonstrated in [127]. Namely a microprocessor that employs the Tomasulo algorithm [169] for out-of-order execution.

The subsequent Section 1.4.1 discusses Temporal Case Splitting and Uninterpreted Functions and points out how they apply to the Tomasulo case-study and why we don't pursue them further.

In Section 1.4.1 we get back to Query and Data-Type Reduction and point out the underlying principles in Section 1.4.1 to prepare for Section 1.4.2 where we lay out how the combination of both applies to the domain of DTS.



(a) Flow of Instructions [127].

(b) Case split: reservation station k depends on register j, by operand opr_1 or opr_2. The value of register j will be produced by functional unit i.

(c) Data-Type Reduction: abstract from rest.

(d) Second representative case.

Figure 1.7.: **Data-Type Reduction** for Tomasulo's Algorithm.

Tomasulo Algorithm

The running example in [129] is a microprocessor that employs the Tomasulo algorithm for out-of-order execution (cf. Figure 1.7(a)). That is, operations are not executed strictly sequentially but an operation may be executed as early as its data-dependencies are met.

Basically, when a three-address operation *Op* is loaded via the Instructions Bus, it is assigned a suitable functional unit *FU*, like an adder, a multiplier, or a load/store unit. Each functional unit *FU* is guarded by a reservation station *RS*.

If both operands opr_1 and opr_2 of *Op* are in the register file when *Op* is loaded, then they are read from the register file and stored in the reservation station. *Op* can execute immediately, unless the functional unit is busy.

If an operand is not yet in the register file, but still computed in one of the functional units, the register file entry contains a *tag* that indicates which functional unit will provide the value. Then this tag is entered in the reservation station.

As every reservation listens on the Results Bus, on which tagged values are passed to the register file, they recognise when an awaited value is available and directly read it from the bus.

One of the main properties to be established for the Tomasulo implementation is the following.

$$\begin{aligned}&\forall\, k : \mathit{TAG}.\\ &\quad \mathsf{G}\ \big((st[k].\mathit{valid} \wedge st[k].\mathit{opr}_1.\mathit{valid}) \rightarrow st[k].\mathit{opr}_1.\mathit{value} = \mathit{aux}[k].\mathit{opr}_1\big)\end{aligned} \tag{1.3}$$

We don't want to elaborate on the syntax and semantics of the property specification language used in [127] here, but only quote how it reads out loud.

> "[...] for all reservation stations, [whenever] the station is valid (contains an instruction) and its *opr*$_1$ operand is a value (not a tag), then the value must be the correct operand value that we stored in the auxiliary array aux." [127]

It refers to a set of auxiliary variables that are used to provide the correct values; they are not necessary to understand the abstraction procedure.

Note that the property is in line with the class of properties outlined in Section 1.2, namely outermost quantified temporal formulae.

Temporal Case Splitting and Uninterpreted Functions for the Tomasulo Algorithm

Temporal Case Splitting applies to the Tomasulo case-study by distinguishing the two phases of obtaining correct operands at the reservation stations, and producing correctly tagged values and entering them into the register file. Then the approach of [123, 127] is to prove that each phase functions correctly up to step $n + 1$ assuming the other phase worked correctly until step n.

This approach in some cases yield great savings as for each phase there are parts of the system which don't influence the phase, like the register file in the first phase. These parts can then automatically be removed from the system using the so-called "cone-of-influence" reduction [34] but it doesn't provide an approach to abstract unbounded DTS to finite-state systems.

Uninterpreted Functions are applied to the Tomasulo algorithm to abstract from the functional units. The focus of the verification is on the control part, first of all, the usage of tags on the buses and in the register file.

The actual data on the buses shall not interfere with the control, thus the functional units should be replaceable by simple combinatorics freely choosing from the possible outcomes and possibly choosing the number of clock-cycles it may take the original functional unit to compute the result. The number of clock-cycles varies, for example, for multipliers depending on the data.

If the desired properties hold even with the abstract functional units, then we conclude that it holds for the concrete functional units either. If doesn't hold, then it may in general be a spurious error, but in case of the Tomasulo algorithm it means that the data-related behaviour of the functional units influences the control part, which most probably points out a serious design flaw.

Using Uninterpreted Functions allows to treat data of large or even unbounded domain, for instance, if a module is replaced which internally operates on real-valued data, thus it addresses the source of infiniteness “infinite-domain data” (cf. introduction of Chapter 1), but in general not the infiniteness caused by the number of processes. The use of Uninterpreted Functions is actually orthogonal to the other steps of the verification strategy we discuss here, and also to our approach.

Query and Data-Type Reduction for the Tomasulo Algorithm

Given property (1.3), the idea of query and data-type reduction is as follows. Firstly, the property is obviously equivalent to (1.4) below where we distinguish cases by explicitly considering the functional unit i which shall provide the value for register j.

$$\begin{aligned}\forall\, k, i : \mathit{TAG}, j : \mathit{REG}\,.\, \mathsf{G}\ \big(&st[k].opr_1.tag = i \land aux[k].src_1 = j) \rightarrow \\ \big((&st[k].valid \land st[k].opr_1.valid) \rightarrow st[k].opr_1.value = aux[k].opr_1\big)\big)\end{aligned} \tag{1.4}$$

McMillan calls this *path splitting* because by the transformation of the formula we focus on a particular path that, for instance, data may take through the system. In the example, it involves the source functional unit i, the (whole) destination reservation station k, and the register of interest j, here assuming the dependency is by the first operand opr_1 (cf. Figure 1.7(b)).

Instead of verifying 1.4 at once, it can be approached case by case, for example first considering reservation station RS_1 for k, functional unit FU_2 for i, and a register j.

The idea of Data-Type Reduction is that for the verification of this particular case, it shouldn’t be necessary to consider the complete behaviour of RS_3 and FU_3, it should be sufficient to consider a reasonable abstraction of a functional unit, which emits tagged values to the bus from time to time, as long as it doesn’t pretend to be FU_2.

Following the same thoughts, it shouldn’t be necessary to consider functional unit FU_1, reservation station FU_2, and any register except for j. In Figure 1.7(c), abstract replacements are indicated by stars (“$*$”).

Substituting the named components by abstract replacements that are capable of mimicking the behaviour of any number of concrete ones, possibly adding some behaviour, first of all the model-checking task will typically use less time and space to complete than the one conducted on the whole system. Furthermore, once established, we can conclude that (1.4), in the particular binding, holds for all instances of the Tomasulo algorithm with *any number* of reservation station/functional unit pairs next to RS_1 and FU_2 and any number of registers next to register j. The property even holds for an infinite register file or infinitely many reservation station/functional unit pairs [129].

It is not safe to simply *remove* all components outside the focused data path because in the concrete system, the focused components can interact with the others. This possible interaction has to be preserved.

Note that the approach can only establish a single, particular case of property 1.4. In other words, if the property can be established, then it is independent from the number of accompanying functional units and registers around, but it remains the single, particular case.

What helps us out is that the Tomasulo algorithm as described above is *symmetric* in tag and register numbers, that is, for each run where reservation station k waits for the value bound for j, there is a run, which is identical up to permutation of identities.

By symmetry, we can conclude from a verification of property 1.4 for the particular binding above that the property holds for *all* symmetric bindings, that is, all registers j' and all pairs k', i' of reservation stations and functional units with $k' \neq i'$. When tracing the behaviour of j', k', i' in any computation path of the Tomasulo implementation, we find that we've then already considered a path, which is similar up to exchanging identities, that is, j by j' etc.. Note that we can *not* conclude to the cases with $k' = i'$ because in the case we have considered, k is different from i. It is only representative for the cases with $k' \neq i'$.

The proof of property 1.4 can easily be completed by considering one additional case, for example reservation station RS_1 for k and functional unit FU_1 for i. Then also a different abstraction can be used (cf. Figure 1.7(d)) since we don't need an abstract reservation station for the concrete functional unit and an abstract functional unit behind the concrete reservation station as in the case before.

Note that the symmetry in tags and register numbers is a property of this particular implementation, and not necessarily present in any implementation. For example, if register number 0 had a special meaning, like constantly yielding the value 0, then there is not necessarily a run where a reservation station waits for it to complete, thus we then can't conclude from using register 0 for j to any other register. To recognise these symmetries, [124] uses the procedure of [88, 89]. A data-type can be declared to be a *scalarset*, that is, to be symmetric, and if variables of this type adhere to certain, effectively checkable well-formedness rules [88, 89], symmetry is guaranteed.

Alternatively, we will encounter system description languages, like the DCS description language of [9], where every described system is symmetric in process identities as the language doesn't permit to break symmetry.

The Principles Underlying Query and Data-Type Reduction

In the strategy of [129, 127], Query Reduction can be applied to any outermost quantified temporal logic property. It yields, depending on the number and type of quantifiers, that is, how many quantified variables of each type are used, a finite set of representative cases.

In the Tomasulo example, tags and registers are two different types. The representative cases basically represent all possibilities to bind quantified variables of the same type to different or equal value. Therefore there are two representative cases in the example in Section 1.4.1, one where the two quantified variables of type tag are equal and one where they are different.

Given a particular binding of the free variables in a temporal property, the underlying idea of Data-Type Reduction is to keep the components used within the binding precise and to abstract (as coarse as possible) from the rest. In other words, the labels i, j, and k in Figure 1.7(b) can be seen as putting *spotlights* on the relevant components and in Figure 1.7(c) we've abstracted from the rest. This abstraction, that is, the function mapping concrete to abstract systems is sound in the sense that if we can establish the desired property on the abstract system, then it also holds in the concrete one, but it may yield spurious counter-examples.

Such an abstraction is only usable for formal verification if the abstract transition system can effectively (and efficiently) be computed without enumerating the full, original transition system.

It is a distinguishing property of Data-Type Reduction that given a description of the behaviour of the considered system as a program P and given a few pre-requisites on the language, which we will discuss in detail in Chapter 9, there is an efficient syntactical transformation of the program P to P' which is in the finite fragment of the language and describes a finite-state system that bisimulates the abstract transition system as obtained by the mathematical description of the abstraction.[1]

Note that the abstraction is independent from symmetry and Query Reduction. It applies to any DTS and any non-quantified formula with a particular binding of the free variables, but it is then only conclusive for the particular valuation of free variables. The conclusion to other (or all) combinations of processes is provided by symmetry.

1.4.2. Query and Data-Type Reduction for Dynamic Topology Systems

In [129], it is already recognised that, having established the property once for a particular binding implies that it holds for any larger instantiations. That is, it holds as intended for all finite instantiations of the micro-processor with N registers and M reservation station/functional unit combinations but it also holds for a processor with infinitely many registers. It was left at that side-note as it may have been considered not to be of practical use to verify a property for a micro-processor with an infinitely large register file.

This reasoning immediately seems highly useful if we consider a property like

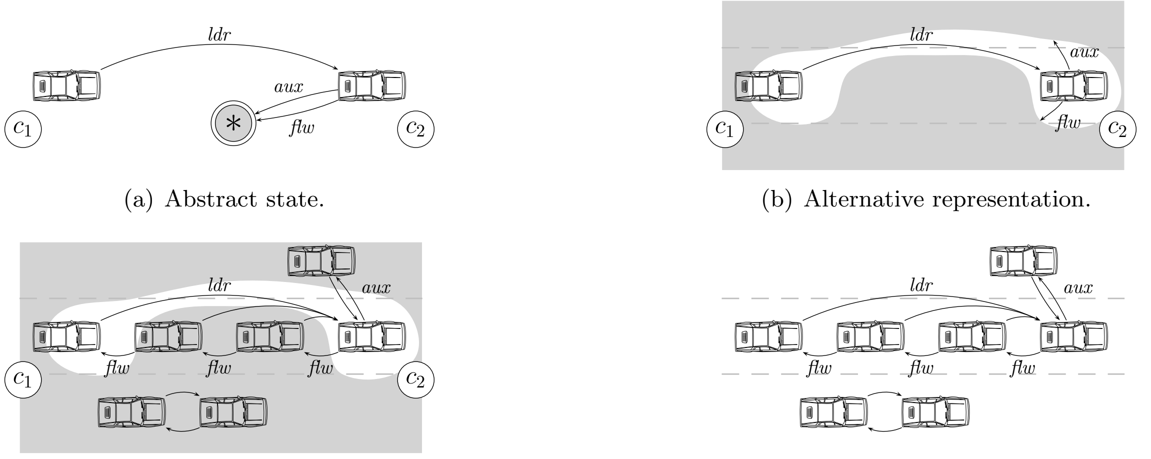
> *For all two different cars, c_1 and c_2, it is never the case that both consider each other to be the leader.* (1.5)

and want to establish it for an implementation of the Car Platooning application.

Recall the principles of Query and Data-Type Reduction from Section 1.4.1. Like with the Tomasulo example, we have a quantified temporal property, in this case with quantification variables c_1 and c_2. Given a particular binding of the two variables (to two different cars), the Data-Type Reduction idea applies immediately.

The abstract system then has states similar to the one shown in Figure 1.8(a). The cars chosen for the particular binding are kept concrete, all other cars are represented by a special identity represented by the star-labelled node. That is, the links from c_2 point to some node different from c_1 and c_2.

[1] Less strictly speaking, the outcome of the syntactical transformation *is* the abstract transition system.



(a) Abstract state.

(b) Alternative representation.

(c) Possible concretisation.

(d) Two platoons, one free agent negotiating lane change.

Figure 1.8.: **Data-Type Reduction** for car platooning.

An alternative representation is shown in Figure 1.8(b), where the special node is unfolded into the complement of c_1 and c_2. One of many possible concretisations is shown in Figure 1.8(c) as a situation with two platoons and one free agent. If we remove the spotlight, we obtain the ordinary topology of Figure 1.8(d) as one concretisation of Figure 1.8(a). Viewed in the opposite direction, Figure 1.8(a) is the abstract representation of Figure 1.8(d) under Data-Type Reduction.

It is a characteristic of Data-Type Reduction that information about the darkness is completely lost thus there are in particular no *explicit* links from the darkness into the highlighted part remaining in Figure 1.8(b). In contrast, Figure 1.8(b) *implicitly* represents all configurations with links originating at the star-labelled node, in particular the configuration shown in Figure 1.8(c) where there is, for example, a *ldr*-link from the shadows to c_2 but not to c_1.

In the pre-study, we played kind of a trick to employ this abstraction for the ARCS case study. We thought of encoding the object system with terminal, car, and car handler in an array program with one array per class, the indices playing the role of identities just as in the Tomasulo example. In addition, and in difference to Tomasulo, each array entry obtained a field to encode whether it is currently alive or not to model dynamic creation and destruction.

Thereby, the overall approach showed as good match for Dynamic Topology Systems, or object-oriented models, but left more questions open than it answered.

For example, what does the abstraction intuitively do? Why does it work? The nice pictures in Figure 1.8 are one not to underestimate result of this work: we now have an individual-oriented view on the abstraction.

Furthermore, what does the abstraction reflect, that is, what properties can we expect to show? Is it limited to certain classes of properties or models, i.e. is there something particular exploited in the Tomasulo example?

Then, how is it practically implemented, why is the computation of the abstract transition relation fast, and can it really match the theory?

Last but not least, issues of refinement. If the good properties, in particular the easy computability of the abstract transition relation are to be preserved, then what can be done to add precision?

1.5. What is the Contribution?

The main and general contribution is an investigation of the implications of the observation from Section 1.4. That is, an in depth assessment of how the application of Query and Data-Type Reduction allows to treat the infiniteness of systems introduced by an unbounded number of processes in presence of dynamic topologies.

In the following, we shall list the major contributions in some more detail. Complete appreciations of our results, in particular in comparison to the literature, is given at the end of each of the following chapters.

For the lack of fully adequate computational models of Dynamic Topology Systems, we firstly contribute a new formal model called ETTS which is basically a topology-labelled transition system with a new tracing of evolution over time.

Similar to computational models, there is a serious lack of truly adequate requirements specification logics. A thorough survey conducted in the joint work [11] shows that there is significant body of proposals for first-order temporal logics, yet they all share a non-satisfactory treatment of pre-mature disappearance of individuals referenced by quantified variables.

We propose a new temporal logic we'll call EvoCTL* which is basically the union of the legitimate features of existing logics, but not more. With certain new results on monotonicity and definiteness we can show that the design is adequate, and as a side-effect provide a common formal setting for possible further work on comparison of the incorporated logics. This is significantly more involved than the indexed temporal logics considered for parameterised systems and the reachability properties considered by other approaches.

As QR/DTR is strictly restricted to formulae in prenex normal form, our results on such normal forms help to estimate the range of properties the approach applies to, we can in particular confirm that LSCs fall into this class.

Given these pre-requisites, we show soundness of the DTR abstraction employing a new simulation relation on ETTS which takes evolution into account. One result of stating DTR in the graph-based setting is a thorough and intuitive understanding of how and why it works, what can be expected to be reflected, and where strategies for refinement should start. Most notably, this is the first closed and declarative characterisation, the original works provide rather an implementation without a good intuition.

Similarly, we introduce QR in the particular setting of DTS based on a notion of symmetry in identities. As new results we have identified the minimal representative set and we discuss singularities, that is, identities breaking symmetry which appear naturally in object-oriented programs in form of NULL.

Having defined DTR formally, we discuss the issue how to obtain the abstract transition system effectively. This is not provided by the original work, and later applications work rather by example.[2]

To this end, we contribute a high-level description language which resembles both, the DCS language and a core of UML. On this level, we're able to discuss further reflection properties relative to the higher-level description, which cannot be discussed on the level of general ETTS. Furthermore, we discuss the symmetry issue completely, including requirements on the particular language instances, like symmetry of communication and scheduling. Interestingly, a prerequisite for DTR/QR as known seems to be an effective interleaving semantics.

And for the first time, we discuss in how far the syntactical transformation matches the theoretical description. We actually find further losses in precision, but are able to sketch ways to overcome these.

We close the circle by reports on verification case-studies conducted for the two examples, ARCS and Car Platooning, in their respective formalisms. This shows the principle effectiveness of the approach to verify scenario properties including true liveness aspects using our implementation of the procedure from Chapter 9.

1.5.1. Publications

Some of the results have already been published. We shall briefly discuss these publications in order of appearance.

The investigated observation first appeared in [177, 48], back then with a focus on the UML dialect and semantics used by the schematic entry tool *Rhapsody in C++* [79, 87] for the description of systems and Live Sequence Charts as property specification language. An integrated, revised, and extended version of these three publications resulted in [49].

In [178], we first reported on experimental results for a simplified version of the arrival procedure of the ARCS case study integrated into the *Rhapsody UML Verification Environment* [159].

The aim of the cooperation [9] was to provide a leaner system specification language to focus on the essential problems with Dynamic Topology Systems, that is, the dynamic extension of the system and the dynamically changing topology assuming asynchronous communication.

In the joint work [174], we view DTR as a canonical abstraction (cf. Chapters 5 and 6). The intuition gained from our closed definition certainly was useful in that undertaking.

Our contribution to [10], where we present an approach to consider topology invariants obtained via static analysis to refine the abstraction, is already further work from the perspective of this thesis.

[2] Most probably because of the incredible tediousness of the complete definition we have.

To the joint work [11], we've contributed a simplified version of our definition of EvoCTL* and preliminary results on monotonicity and definiteness, which are clearly marked in the publication. The survey part and the relation to philosophical logic is completely joint work and not claimed.

1.6. How is it Obtained? (Or: What is the Structure of the Work?)

The work is split into five parts. The first part comprises this introduction and a collection of preliminaries for self-containedness.

The second part is dedicated to a formal model of Dynamic Topology Systems. Chapter 3 introduces Evolving Topology Transition Systems (ETTS) and Chapter 4 the complementing requirements specification logic EvoCTL*.

The third part provides the formal investigation of the QR/DTR approach. Chapter 5 introduces a notion of simulation relation that takes evolution of individuals into account. It shows that simulation corresponds to over-approximation for the universal fragment of EvoCTL*. Proving soundness of DTR in the DTS setting then amount to providing a simulation relation in Chapter 6. Chapter 7 discusses Query Reduction and Chapter 8 the strategy for a combination of both.

The fourth part is dedicated to the practical application. In Chapter 9, we introduce a high-level DTS description language and discuss issues like symmetry detection and how to obtain the abstract transition system effectively. To be of practical use, the chapter concludes by an encoding of the high-level language, and in particular the finite abstract transition system in form of ordinary array programs that are amenable to off-the-shelf finite-state model-checkers. Chapter 10 briefly discusses the two case-studies.

The fifth part comprises only the conclusion, Chapter 11.

2. Preliminaries

The sections of this chapter provide standard definitions in order to achieve self-containedness and to disambiguate our usage or certain standard concepts and notions.

The extension of the respective presentation is driven by the overall characterisation of this work. Alltogether, it is biased towards the classical transition systems and model-checking community when it comes to choosing names, notations, and the amount of further explanation to admit for each topic. For example, by spending more time on the introduction of Galois connections than on transition systems in the following sections.

The underlying computational model of this work is basically Kripke structures, the specification language is temporal logic, and the verification approach is based on abstraction in the sense of [34]. These basics are readily defined on top of notions of relations, functions, and sequences as introduced in Sections 2.1 and 2.2.

Yet there are twists which require additional notions. First of all, the states of our Kripke structures are not labelled with atomic propositions, but by multi-graphs as introduced in Section 2.3.

The temporal logic is a first order variant over varying domains, hence has to cater for unexpected disappearance of objects. We solve this by employing three-valued Kleene logic with the third “indefinite” value $1/2$ as introduced in Section 2.5.

In the section on abstraction, we need notions from the theory of abstract interpretation in the sense of [40], in particular lattices and Galois connections as introduced in Section 2.4.

2.1. Sets, Relations, and Functions

To avoid confusion, we consistently write $\mathbb{N}_0$ to denote the set of natural numbers including 0 and $\mathbb{N}^+$ to denote the set of positive natural numbers.[1]

We use $\mathbb{Z}$ to denote the set of integers and $\mathbb{Z}^-$ and $\mathbb{Z}^+$ to denote the set of negative and positive integers, and $\mathbb{Z}_0^-$ and $\mathbb{Z}_0^+$ to include zero. Similarly, we use $\mathbb{R}$ and, for example, $\mathbb{R}_0^+$ for the set of real numbers.

Definition 2.1.1 (Disjoint Union)**.** *Let A, A_1, and A_2 be sets. We write $A = A_1 \dot{\cup} A_2$ if and only if A is the union of A_1 and A_2, and if A_1 and A_2 are disjoint, i.e. if $A_1 \cap A_2 = \emptyset$.* ♢

Definition 2.1.2 (Partitioning)**.** *Let A be a set. The subsets $A_1, \ldots, A_n$, $n \in \mathbb{N}^+$, of A are called a* partitioning *of A if and only if*

[1] As the former is the natural and non-debatable meaning of $\mathbb{N}$ for many people in computing science while the latter is the natural and non-debatable meaning of $\mathbb{N}$ for many mathematicians.

1. *the subsets are non-empty, i.e.* $A_i \neq \emptyset$, $1 \leq i \leq n$,

2. *the subsets are pairwise disjoint, i.e.* $A_i \cap A_j = \emptyset$, $1 \leq j \neq i \leq n$, *and*

3. *together the sub-sets cover* A, *i.e.* $A_1 \cup \cdots \cup A_n = A$.

We may write $A = A_1 \dot{\cup} \ldots \dot{\cup} A_n$ *to introduce the partitioning* $A_1, \ldots, A_n$ *of* A. ♢

Definition 2.1.3 (Relation)**.** *Let* A *and* B *be sets. A set* $r \subseteq A \times B$ *of pairs is called* (binary) relation *between* A *and* B. *We say* $b \in B$ *is in* r*-relation to* $a \in A$ *if* $(a, b) \in r$*; we sometimes use the alternative* predicate notation $r(a, b)$, infix notation $a\ r\ b$, *or, primarily in diagrams,* $a \xrightarrow{r} b$, *to indicate* $(a, b) \in r$.

The set $\{a \in A \mid (a, b) \in r\}$ *is called the* domain *and the set* $\{b \in B \mid (a, b) \in r\}$ *is called the* range *of* r. *We use* r^{-1} *to denote the* inversion *of* r, *i.e. the relation* $\{(b, a) \mid (a, b) \in r\}$ *between* B *and* A.

A binary relation is called total *if and only if each* $a \in A$ *is related to at least one* $b \in B$, *i.e. if* $\forall\, a \in A\ \exists\, b \in B : (a, b) \in r$. ♢

Definition 2.1.4 (Partial Function, Image of Set, Restriction)**.** *Let* A *and* B *be sets. A relation* $f \subseteq A \times B$ *is called* partial function, *denoted by* $f : A \nrightarrow B$, *if and only if each element* $a \in A$ *is related to at most one* $b \in B$, *i.e.*

$$\forall\, a \in A\ \forall\, b_1, b_2 \in B : (a, b_1) \in f \wedge (a, b_2) \in f \implies b_1 = b_2 \tag{2.1}$$

A partial function is defined *for* $a \in A$ *if and only if* $a \in \mathrm{dom}(f)$. *We may write* $f(a)$ *to denote the unique* $b \in B$, *called the* image *of* a *under* f, *if* f *is defined for* a, *that is, when writing* $b = f(a)$ *we deliberately mean that* f *is defined for* a *and* b *is the image of* a *under* f. *To define a function, we sometimes use the notation* $\{a \mapsto f(a) \mid a \in A\}$.

Let $f : A \nrightarrow B$ *be a partial function and* $A' \subseteq A$*; we deliberately use* $f(A')$ *to denote the set of images of elements from* A' *under* f, *i.e. the set*

$$\{f(a) \mid a \in A' \cap \mathrm{dom}(f)\}. \tag{2.2}$$

Let $f : A \nrightarrow B$ *be a partial function and* C *a set; by* $f|_C$ *we denote the* restriction *of* f *to* C, *that is, the partial function from* C *to* B *that coincides with* f *on* C *where* f *is defined, i.e.* $f|_C = \{(c, f(c)) \mid c \in C \cap \mathrm{dom}(f)\}$. ♢

Definition 2.1.5 (Total Function)**.** *Let* A *and* B *be sets. A partial function* $f : A \nrightarrow B$ *is called (total) function, denoted by* $f : A \to B$, *if and only if* f *is defined for all* $a \in A$, *i.e. if* $\mathrm{dom}(f) = A$.

If not otherwise stated, we use $f^{-1} : B \nrightarrow A$ *and* $f^{-1} : B \to A$ *if the inverse of the relation* f *is again a partial or total function;* f^{-1} *is then called the* inversion *of* f.

In some cases we use $f^{-1}(b)$ *to denote the set of pre-images of* b *under* f, *that is*

$$f^{-1}(b) = \{a \mid f(a) = b\}; \tag{2.3}$$

if f *has an inversion, this set of pre-images comprises exactly one element.* ♢

Definition 2.1.6 (Identity Function)**.** *Let A be a set. The function $f : A \to A$ with $f = \{(a, a) \mid a \in A\}$ is called the* identity function on A *and denoted by id_A.* ♢

Definition 2.1.7 (Power-set, Multi-set)**.** *Let A be a set. We use $\mathfrak{P}(A)$ to denote the power-set of A, i.e. the set of subsets of A, and $\mathfrak{M}(A)$ to denote the set of multi-sets over A, i.e. the set of total functions from A to $\mathbb{N}_0$.*

We use '$\{\!|$' and '$|\!\}$' to indicate multi-set comprehension. ♢

2.2. Sequences

Definition 2.2.1. *Let A be a set.*

1. *We use A^+ to denote the set of finite sequences over A, that is, the set of sequences*

$$a_0, a_1, \ldots, a_n \tag{2.4}$$

with $n \in \mathbb{N}_0$ and $a_i \in A$ for each $0 \leq i \leq n$.

2. *We use A^* to denote the set of finite sequences including the empty sequence ε, i.e. $A^* := A^+ \cup \{\varepsilon\}$.*

3. *We use A^ω to denote the set of infinite sequences over A, that*

$$a_0, a_1, \ldots \tag{2.5}$$

with $a_i \in A$ for each $i \in \mathbb{N}_0$.

4. *Given a finite sequence $\pi = a_0, a_1, \ldots, a_n$ over A, we use π/k to denote the suffix of π starting at the k-th element, $k \in \mathbb{N}_0$, that is*

$$\pi/k = \begin{cases} a_k, a_{k+1}, \ldots, a_n & \text{, if } k \leq n \\ \varepsilon & \text{, otherwise} \end{cases} \tag{2.6}$$

and we use $\pi(k)$ to denote the k-th element of π for $k \leq n$, i.e. $\pi(k) = a_k$.

5. *Given an infinite sequence $\pi = a_0, a_1, \ldots$ over A, we use π/k to denote the suffix of π starting at the k-th element, $k \in \mathbb{N}_0$, that is*

$$\pi/k = a_k, a_{k+1}, \ldots \tag{2.7}$$

and we use $\pi(k)$ to denote the k-th element of π, i.e. $\pi(k) = a_k$. ♢

2.3. Labelled Multi-Graphs

Definition 2.3.1 (Labelled Multi-Graph)**.** *Let Σ_V and Σ_E be (possibly infinite) sets. A* (Σ_V, Σ_E)-labelled multi-graph *is a quintuple*

$$G = (V, E, \psi, f, g) \tag{2.8}$$

of two disjoint (possibly infinite) sets V and E, a total function $\psi : E \to V \times V$, and two partial functions $f : V \nrightarrow \Sigma_V$ and $g : E \nrightarrow \Sigma_E$.

The elements of V are called vertices *of G, the elements of E are called* edges *of G. The function ψ is called* incidence function*. It denotes the two vertices connected by a given edge. If $e \in E$ is an edge and $\psi(e) = (v_1, v_2)$ is its value under the incidence function, then v_1 and v_2 are called* initial *and* terminal vertex *of e and denoted by $ini(e)$ and $ter(e)$.*

The function f is called vertex labelling function*, the elements of its domain Σ_V are called* vertex labels*. Analogously, g is called* edge labelling function*, the elements of Σ_E* edge labels*.*

The vertices and edges of G are referred to as $V(G)$ and $E(G)$, its incidence function as $\psi(G)$, and its vertex and edge labelling as $f(G)$ and $g(G)$. ♢

Note that we permit the labelling functions to be partial. This is highly unusual for graphs, but it provides a method to distinguish alive nodes from non-alive ones in the following chapters in a more natural way than by employing a designated label to indicate the aliveness of a node.

Definition 2.3.2 (Degree)**.** *Let G be a labelled multi-graph. Let $v \in V$ be a node and*

$$E_v = \{e \in E \mid \psi(e) = (v, v'), v' \in V\} \tag{2.9}$$

*the set of outgoing edges. If E_v is of finite cardinality $n \in \mathbb{N}_0$, then n is called the (out-)*degree *of v.* ♢

The following definition is necessary because we consider multi-graphs for which a finite vertex set doesn't imply that the graph is finite.

Definition 2.3.3 (Finite Labelled Multi-Graph)**.** *A labelled multi-graph G is called* finite *if and only if both, the sets $V(G)$ and $E(G)$ of vertices and edges, are finite.* ♢

As we don't use unlabelled, simple or undirected graphs in the following, from now on we may use the term *graph* to denote labelled multi-graphs in the sense of Def. 2.3.1.

2.4. Complete Lattices, Galois Connections, and Abstract Interpretation

In the following, we provide the formal basics for (data) abstraction, namely complete lattices and Galois connections between them. As said in the introduction, we're biased towards the model-checking community regarding our audience. Thus we spend a little bit more time on motivation in this section than would strictly be necessary for the actual purpose, rendering this work self-contained.

Note that we don't need the fixed point theory of the abstract interpretation domain because our approach is far from classical static analysis, but rather related to temporal logic model-checking. Our presentation follows [77] and partly [50].

2.4.1. Partially Ordered Sets

Definition 2.4.1 (Partial Order). *Let L be a set. A binary relation $\sqsubseteq$ on L, i.e. $\sqsubseteq \subseteq L \times L$, is called* partial order *if and only if it is*

1. *reflexive,* *i.e. $l \sqsubseteq l$ for all $l \in L$,*
2. *anti-symmetric,* *i.e. if $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_1$, then $l_1 = l_2$ for all $l_1, l_2 \in L$,*
3. *transitive,* *i.e. $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$ implies $l_1 \sqsubseteq l_3$ for all $l_1, l_2, l_3 \in L$.*

Given a partial order $\sqsubseteq$, we write $l_1 \sqsubset l_2$ if and only if $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$. We may write $l_2 \sqsupseteq l_1$ and $l_2 \sqsupset l_1$ for $l_1 \sqsubseteq l_2$ and $l_1 \sqsubset l_2$. ♢

Definition 2.4.2 (Partially Ordered Set). *A* partially ordered set $(L, \sqsubseteq_L)$ *is a set L equipped with a partial order $\sqsubseteq_L$, where the index may be omitted if the set is clear from the context.* ♢

Example 2.4.3 (Three valued logic). *Let $\mathbb{B}$ denote the set of truth-values $\{0, 1\}$. We use $\mathbb{B}_3$ to denote the domain of three valued logic, $\{0, 1, 1/2\}$. The third truth value $1/2$ is also called* uncertain *(or* indefinite*).*

The relation

$$\sqsubseteq := \{(b, b), (b, 1/2) \mid b \in \mathbb{B}_3\} \tag{2.10}$$

is a partial order on $\mathbb{B}_3$. Intuitively, $b_1 \sqsubseteq b_2$ read as "b_2 is less definite than b_1". ♢

If not otherwise noted, we consider $\mathbb{B}_3$ to be equipped with the partial order from Def. 2.4.3 in the following.

Note 2.4.4.

1. *Let $b_1 \in \mathbb{B}$, $b_2 \in \mathbb{B}_3$. If $b_2 \sqsubseteq b_1$, then $b_2 = b_1$.*
2. *Let $b \in \mathbb{B}_3$. If $1/2 \sqsubseteq b$, then $b = 1/2$.* ♢

2.4.2. Join and Meet

Definition 2.4.5 (Upper and Lower Bound). *Let A be a subset of the partially ordered set L.*

- *$l \in L$ is an* upper bound *of A if and only if it is larger or equal to all elements in A, that is,*

$$\forall\, a \in A : a \sqsubseteq l. \tag{2.11}$$

- *$l \in L$ is a* lower bound *of A if and only if*

$$\forall\, a \in A : a \sqsupseteq l. \tag{2.12}$$

- *An upper bound $l \in L$ of A is called* least upper bound *of A if and only if all other upper bounds of A are larger or equal to l, that is if l' being an upper bound of A implies $l \sqsubseteq l'$.*
- *Similarly, a lower bound $l \in L$ of A is called* greatest lower bound *of A if and only if all other lower bounds of A are smaller or equal to l, that is if l' being a lower bound of A implies $l' \sqsubseteq l$.*

If the least upper bound of A exists, it is denoted by $\bigsqcup A$ and if the greatest lower bound of A exists, it is denoted by $\bigsqcap A$. To explicate in which partially ordered set L bounds are formed, we may write $\bigsqcup_L A$ and $\bigsqcap_L A$. ♢

Note 2.4.6. *Let A be a subset of the partially ordered set L. The least upper and greatest lower bound of A are unique, if they exist, due to anti-symmetry of the partial order.* ♢

Definition 2.4.7. *Let L be a partially ordered set. By setting*

$$\begin{aligned} \sqcup : L \times L &\to L \\ (l_1, l_2) &\mapsto \bigsqcup\{l_1, l_2\} \end{aligned} \quad \text{and} \quad \begin{aligned} \sqcap : L \times L &\to L \\ (l_1, l_2) &\mapsto \bigsqcap\{l_1, l_2\} \end{aligned} \tag{2.13}$$

we obtain the join *and* meet *operators on L, typically written in infix-notation.* ♢

Note 2.4.8. *Let L be a partially ordered set. The join and meet operators on L are*

- *associative, i.e. for all $l_1, l_2, l_3 \in L$,*

$$(l_1 \sqcup l_2) \sqcup l_3 = l_1 \sqcup (l_2 \sqcup l_3) \qquad (l_1 \sqcap l_2) \sqcap l_3 = l_1 \sqcap (l_2 \sqcap l_3) \tag{2.14}$$

- *commutative, i.e. for all $l_1, l_2 \in L$,*

$$l_1 \sqcup l_2 = l_2 \sqcup l_1 \qquad l_1 \sqcap l_2 = l_2 \sqcap l_1 \tag{2.15}$$

- *idempotent, i.e. for all $l \in L$,*

$$l \sqcup l = l \qquad l \sqcap l = l, \tag{2.16}$$

  which is a consequence of, for all $l_1, l_2 \in L$,

$$l_1 \sqsubseteq l_1 \sqcup l_2 \qquad l_1 \sqcap l_2 \sqsubseteq l_1 \tag{2.17}$$

- *absorbing, i.e. for all $l_1, l_2 \in L$,*

$$l_1 \sqcup (l_1 \sqcap l_2) = l_1 \qquad l_1 \sqcap (l_1 \sqcup l_2) = l_1 \tag{2.18}$$

♢

Definition 2.4.9. *Let $f_1, f_2 : A \to L$ be two (total) functions from a set A into a partially ordered set $(L, \sqsubseteq_L)$. We write $f_1 \sqsubseteq f_2$ if and only if*

$$\forall\, a \in A : f_1(a) \sqsubseteq f_2(a). \tag{2.19}$$

$\diamondsuit$

Note 2.4.10. *The order on functions as introduced in Def. 2.4.9 is a partial order on the set of (total) functions between A and L.* $\diamondsuit$

Definition 2.4.11. *Let $f : L \to M$ be a (total) function between two partially ordered sets $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$.*

- *f is called* monotone *(or* order-preserving*) if and only if*

$$\forall\, l, l' \in L : l \sqsubseteq_L l' \implies f(l) \sqsubseteq_M f(l'). \tag{2.20}$$

- *f is called* additive *(or* join morphism*) if and only if*

$$\forall\, l_1, l_2 \in L : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2) \tag{2.21}$$

- *f is called* multiplicative *(or* meet morphism*) if and only if*

$$\forall\, l_1, l_2 \in L : f(l_1 \sqcap l_2) = f(l_1) \sqcap f(l_2) \tag{2.22}$$

- *f is called* completely additive *(or* complete join morphism*) if and only if*

$$\forall\, A \subseteq L : f(\textstyle\bigsqcup_L A) = \bigsqcup_M \{f(l) \mid l \in A\} \tag{2.23}$$

  whenever $\bigsqcup A$ exists in L.

- *f is called* completely multiplicative *(or* complete meet morphism*) if and only if*

$$\forall\, A \subseteq L : f(\textstyle\bigsqcap_L A) = \bigsqcap_M \{f(l) \mid l \in A\} \tag{2.24}$$

  whenever $\bigsqcap A$ exists in L.

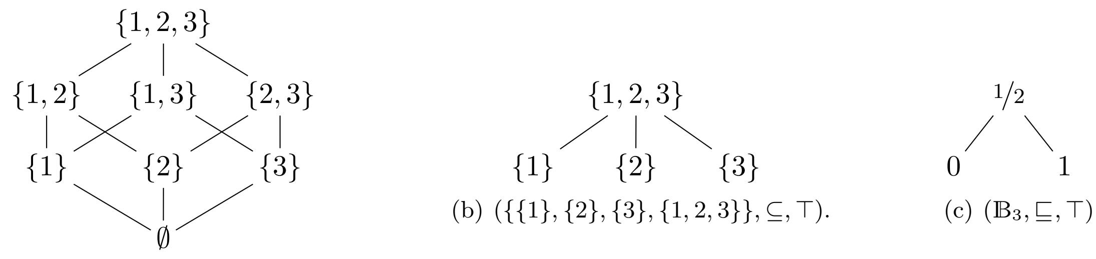
- *f is called* isomorphism *if and only if it is monotone and if its inverse f^{-1} exists and is monotone. The two ordered sets are then called* isomorphic *and said to be* identical up to isomorphism*.* $\diamondsuit$

2.4.3. Lattices

Definition 2.4.12 (Lattice)**.** *A partially ordered set $(L, \sqsubseteq)$ is called* join semi-lattice *if and only if all pairs of elements from L have a least upper bound in L, that is, for all $l_1, l_2 \in L$ the least upper bound $l_1 \sqcup l_2$ exists in L.*

Analogously, it is called meet semi-lattice *if and only if all pairs of elements from L have a greatest lower bound.*

It is called lattice *if and only if it is a join semi-lattice and a meet semi-lattice.* $\diamondsuit$



(a) $(\mathfrak{P}(\{1, 2, 3\}), \subseteq, \top, \bot)$.

(b) $(\{\{1\}, \{2\}, \{3\}, \{1, 2, 3\}\}, \subseteq, \top)$.

(c) $(\mathbb{B}_3, \sqsubseteq, \top)$

Figure 2.1.: **Hasse diagrams.**

Definition 2.4.13 (Complete Lattice). *A triple* $(L, \sqsubseteq, \top)$ *is called* complete join semi-lattice *if and only if*

1. $(L, \sqsubseteq)$ *is a join semi-lattice,*

2. *all subsets of* L *have a least upper bound, that is, for all* $A \subseteq L$*, the least upper bound* $\bigsqcup A$ *exists, and*

3. $\top = \bigsqcup L$.

Dually, a triple $(L, \sqsubseteq, \bot)$ *is called* complete meet semi-lattice *if all subsets of* L *have a greatest lower bound and* $\bot = \bigsqcap L$.

A quadruple $(L, \sqsubseteq, \top, \bot)$ *is called* complete lattice *if and only if*

- $(L, \sqsubseteq, \top)$ *is a complete join semi-lattice and*

- $(L, \sqsubseteq, \bot)$ *is a complete meet semi-lattice.*

$\top$ *is also called* greatest *or* top element*,* $\bot$ *is also called* least *or* bottom element*.* ♢

Example 2.4.14 (Lattices).

1. *Let* A *be a set. The power-set* $\mathfrak{P}(A)$ *with set inclusion* $\subseteq$ *as order is a complete lattice with* $\bot = \emptyset$ *and* $\top = A$.

2. *Let* A *be a set. The set*
$$\{A\} \cup \{\{a\} \mid a \in A\} \tag{2.25}$$
with set inclusion $\subseteq$ *as order and* $\top = A$ *is a complete join semi-lattice.*

   We call it the canonical join semi-lattice completion *of* A.

3. $(\mathbb{B}_3, \sqsubseteq, \top)$ *with* $\top = 1/2$ *is a complete join semi-lattice.*

   Note that $(\mathbb{B}_3, \sqsubseteq, \top)$ *is isomorphic to the canonical join semi-lattice completion of* $\mathbb{B}$. ♢

A lattice like $(\mathbb{B}_3, \sqsubseteq, \top)$ can be depicted as a diagram called *Hasse diagram.* The elements of $\mathbb{B}_3$ provide the nodes and there are edges between immediate successors in the order relation. An edge going upwards from some l_1 to some l_2 indicates $l_1 \sqsubseteq l_2$. Figure 2.1 shows Hasse diagrams of the lattices introduced in Example 2.4.14 above

In the following, we only consider complete join semi-lattices and sometimes only write *lattice* as a shorthand for complete join semi-lattices if it's clear from the context that we don't mean a (possibly non-complete) lattice in the sense of the Def. 2.4.12.

2.4.4. Galois Connection

Definition 2.4.15 (Galois Connection). *Let $(L, \sqsubseteq, \top)$ and $(M, \sqsubseteq, \top)$ be complete join semi-lattices. A quadruple (L, α, γ, M) is called* Galois connection *between L and M if and only if*

1. $\alpha : L \to M$ *and* $\gamma : M \to L$ *are monotone functions,*
2. $\forall\, l \in L : l \sqsubseteq \gamma(\alpha(l))$, *and*
3. $\forall\, m \in M : \alpha(\gamma(m)) \sqsubseteq m$

α *is called* abstraction (function) *and* γ *is called* concretisation (function). *If the two lattices are clear from context, we only use the pair (α, γ) to denote the Galois connection.*

If we read $\sqsubseteq$ as information order, then item 2 ensures that we don't lose *safety* by first abstracting and then concretising again, but we may lose *precision.*

The following note states that iterating abstraction and concretisation neither gains nor loses precision. In other words, if (α, γ) is a Galois connection, then γ doesn't destroy the precision preserved by α and vice versa.

Note 2.4.16. *Let (L, α, γ, M) be a Galois connection.*

1. $\alpha \circ \gamma \circ \alpha = \alpha$ *and* $\gamma \circ \alpha \circ \gamma = \gamma$.
2. *If* $\alpha(l_1) \neq \alpha(l_2)$ *for* $l_1, l_2 \in L$ *then* $\gamma(\alpha(l_1)) \neq \gamma(\alpha(l_2))$ *and if* $\gamma(m_1) \neq \gamma(m_2)$ *for* $m_1, m_2 \in M$ *then* $\alpha(\gamma(m_1)) \neq \alpha(\gamma(m_2))$.

Proof. 1. Let $l \in L$. By Def. 2.4.15, $l \sqsubseteq \gamma(\alpha(l))$. As α is monotone, we have $\alpha(l) \sqsubseteq \alpha(\gamma(\alpha(l)))$, and thus $\alpha \sqsubseteq \alpha \circ \gamma \circ \alpha$.

By Def. 2.4.15, we have $\alpha(\gamma(\alpha(l))) \sqsubseteq \alpha(l)$, thus $\alpha \circ \gamma \circ \alpha \sqsubseteq \alpha$, and hence $\alpha = \gamma \circ \alpha \sqsubseteq \alpha$.

The relation $\gamma \circ \alpha\gamma = \gamma$ follows analogously.

2. Let $l_1, l_2 \in L$ such that $\alpha(l_1) \neq \alpha(l_2)$. Assume $\gamma(\alpha(l_1)) = \gamma(\alpha(l_2))$. Then clearly $\alpha(\gamma(\alpha(l_1))) = \alpha(\gamma(\alpha(l_2)))$ and $\alpha(l_1) = l_2$ follows by (1) from above, in contradiction to the premises.

The second part follows analogously.

□

Alternatively to Def. 2.4.15, Galois connections can be introduced as an adjunction.

Definition 2.4.17 (Adjunction)**.** *Let $(L, \sqsubseteq, \top)$ and $(M, \sqsubseteq, \top)$ be complete join semi-lattices.*

A quadruple (L, α, γ, M) is called adjunction *between L and M if and only if*

$$\forall\, l \in L\; \forall\, m \in M : \alpha(l) \sqsubseteq m \iff l \sqsubseteq \gamma(m). \tag{2.26}$$

Note 2.4.18 ([77])**.** *(L, α, γ, M) is an adjunction if and only if (L, α, γ, M) is a Galois connection.*

Lemma 2.4.19 ([77])**.** *Let (L, α, γ, M) be a Galois connection.*

1. *The abstraction function α uniquely determines the concretisation function γ by $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq \gamma\}$.*
2. *The abstraction function α is completely additive.*

Note that, in general, Galois connections are defined not only for complete semi-lattices, but for complete lattices.

As already noted earlier, we don't do this because the complete join semi-lattices are sufficient for our purposes, we don't need the additional properties of full lattices.

2.5. Kleene Logic

Note that in Example 2.4.3, we've introduced the logical domain of Kleene logic which we'll use to define the semantics of formulae in Chapter 4.

When defining that semantics, we'll form minima and maxima in $\mathbb{B}_3 = \{0, 1, 1/2\}$. Given two boolean values $a, b \in \mathbb{B}_3$, we write $\min(a, b)$ and $\max(a, b)$ to denote the smaller and larger value of the two in arithmetic order of the set of rational numbers. Given a non-empty set of boolean values $\emptyset \neq A \subseteq \mathbb{B}_3$, we write $\min A$ and $\max A$ to denote the smallest and largest element in A and set $\min \emptyset = 1$ and $\max \emptyset = 0$.

Note 2.5.1. *Let $a, b \in \mathbb{B}_3$. Then $1 - \min(a, b) = \max(1 - a, 1 - b)$.* ◇

2.6. Labelled Fair Transition Systems

The definition of labelled (fair) transition system is standard [34]. We always include the fair case because we need fairness when considering liveness properties of individuals in an interleaving semantics in Chapter 9.

Definition 2.6.1 (Labelled Fair Transition System)**.** *A* labelled transition system *(LTS) is a quintuple*

$$M = (S, S_0, R, \mathscr{L}, F) \tag{2.27}$$

of (possibly infinite) sets S and $\emptyset \neq S_0 \subseteq S$, a total binary relation R on S, a function $\mathscr{L} : S \to \mathscr{D}$, and a set of sets $F \subseteq \mathfrak{P}(S)$.

The elements of S *are called* states *of* M*, the elements of* S_0 initial states*. The relation* R *is called* transition relation *of* M*, a state* $s' \in S$ *is called* successor *of state* $s \in S$ *if and only if they are in transition relation, i.e.* $(s, s') \in R$*. State* s *is then also called the* source state *and state* s' *the* destination state *of* r*. The function* $\mathscr{L} : S \to \mathscr{D}$ *is called labelling function of* M*, its (possibly infinite) domain* $\mathscr{D}$ *is called* labelling domain*. The elements of* F *are called* fairness constraints*. The components of* M *are referred to as* $S(M)$*,* $S_0(M)$*,* $R(M)$*,* $\mathscr{L}(M)$*, and* $F(M)$*, respectively.*

If F *comprises only* S*, i.e. if* $F = \{S\}$*, we may omit the fifth component.* $\diamondsuit$

Note that the transition relation is assumed to be total, as usual. The general case reduces to this special one by adding self-transitions to all states with no outgoing transition, and in case of ETTS (see below) additional self-evolution of all non-newly created individuals.

Definition 2.6.2 (Finite- and infinite-state Transition System)**.** *A labelled fair transition system* $(S, S_0, R, \mathscr{L}, F)$ *is called* finite-state *if the set of states* S *is finite, otherwise it is called* infinite-state*.* $\diamondsuit$

For completeness and for reference, the following definition introduces Kripke structures as a special case of LTS.

Definition 2.6.3 (Kripke Structure)**.**
A finite-state labelled fair transition system $(S, S_0, R, \mathscr{L}, F)$ *is called* Kripke structure *if and only if the domain* $\mathscr{D}$ *of the labelling function* $\mathscr{L}$ *is the power-set of a finite set of* atomic propositions AP*, i.e.* $\mathscr{D} = \mathfrak{P}(AP)$*.* $\diamondsuit$

The following definition introduces the standard notions of infinite (fair) computation paths in LTS. In addition, we'll need finite variants.

Definition 2.6.4 (Computation Path)**.** *Let* $M = (S, S_0, R, \mathscr{L}, F)$ *be a labelled fair transition system.*

1. *A sequence*
$$\pi = s_0, s_1, s_2 \ldots \tag{2.28}$$
of successive states *of* M*, i.e.* $s_i \in S$ *and* $(s_i, s_{i+1}) \in R$*,* $i \in \mathbb{N}_0$*, is called* infinite (computation) path *or* path *in* M *from state* s_0*.*

   A finite sequence
$$\pi = s_0, s_1 \ldots, s_n \tag{2.29}$$
of successive states is called finite (computation) path *or (computation) path from* s_0 *to* s_n *in* M*.*

2. *Let* $\pi = s_0, s_1, s_2 \ldots$ *be a path in* M*. We use* π/k *to denote the suffix of* π *starting at* s_k *and* π^k *to denote the* k*-th state of* π*, i.e.* $\pi/k = s_k\, s_{k+1} \ldots$ *and* $\pi^k = s_k$*.*

3. *A path π in M is called* fair *if and only if it each fairness constraint is visited infinitely often by π, i.e. if*

$$\forall C \in F : C \cap \operatorname{inf}(\pi) \neq \emptyset \tag{2.30}$$

where $\operatorname{inf}(\pi)$ *denotes the set of states occurring infinitely often in π, i.e.*

$$\operatorname{inf}(\pi) = \{s \in S \mid \forall i \in \mathbb{N}_0 \; \exists j \in \mathbb{N}_0 : j > i \wedge \pi(j) = s\}. \tag{2.31}$$

4. *We use $\Pi_s(M)$ and $\Pi_s^F(M)$ to denote the sets of paths and fair paths in M from state $s \in S$ and $\Pi(M)$ and $\Pi^F(M)$ to denote the sets of paths and fair paths from an initial state. In addition, $\Pi_{s,s'}(M)$ denotes the set of finite paths in M from state $s \in S$ to state $s' \in S$.* $\diamondsuit$

Part II.

System and Requirements Specification: ETTS and EvoCTL*

3. Computational Model: Evolving Topology Transition Systems

As we'll see in the discussion in Section 3.7, there is unfortunately currently not a single agreed formal model for model-checking the class of systems we're interested in (cf. Chapter 1). In the following, we employ a graph-based approach to model states and employ a classical labelled transition system to model temporal evolution.

To keep the presentation readable, we have separated the motivation from the formal definitions. In Section 3.1, we discuss the class of "real-world" systems and the corresponding intuitive notions of snapshots, agents, and inter-agent connections along the Car Platooning example known from Chapter 1. Then we'll discuss how we formally model these notions by topologies, individuals, and links in Sections 2.6 to 3.5, and *why* we chose these models. For instance, by naming which effects from the "real-world" we want to see preserved in the formal model and from which we abstract right-away.

Beginning with Section 2.6, which introduces classical labelled transition systems as the underlying formal computational model catering for system evolution over (discrete) time, we'll keep motivational notes to a minimum and only discuss how our choices meet the requirements named in Section 3.1. The dynamically changing and possibly infinite number of agents to consider in a particular system snapshot is represented by labelling transition system states with so-called topologies as introduced in Sections 3.2 and 3.3.

The evolution of particular agents over time, that is, telling which agents newly appeared, remained alive, or will disappear, is captured by an annotation of transition system transitions with an evolution relation between the agents in the topologies of the source and destination state. This relation, together with a notion of consistency, which for instance excludes that an agent is new and non-new at the same time, is discussed in Section 3.4.

Putting it all together, we obtain the notion of evolving topology transition system (ETTS) in Section 3.5 and briefly discuss some particular sub-classes, for example the effects of disregarding evolution of agents over time or disregarding identities, and recall the relation to array programs.

Section 3.7 discusses the detailed relation between our approach and the numerous different other approaches to formal models of this class of systems, Section 3.6 contributes definitions of different paradigms to these discussions.

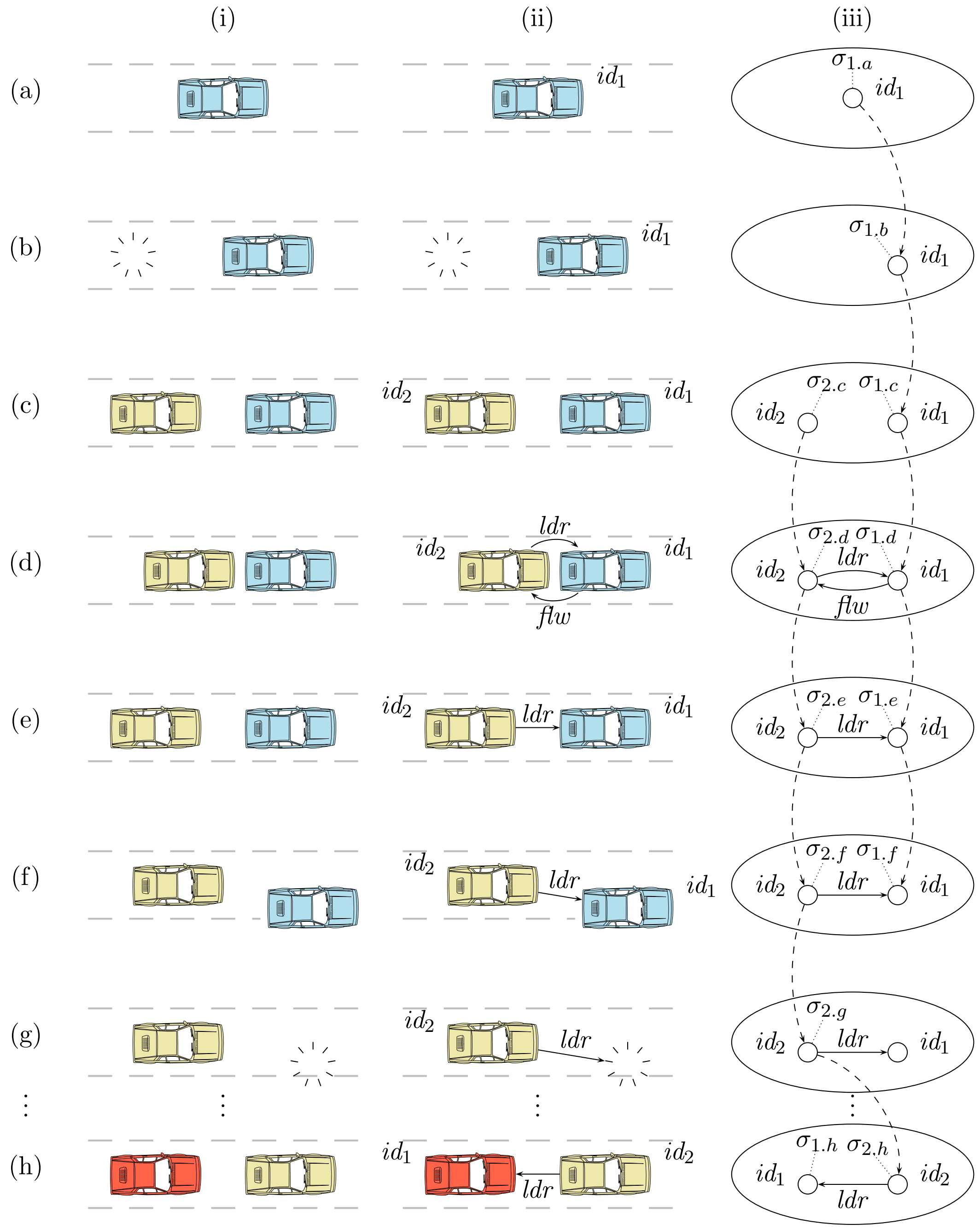

(i)
(ii)
(iii)
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
id_1
id_2
ldr
flw
$\sigma_{1.a}$
$\sigma_{1.b}$
$\sigma_{2.c}$ $\sigma_{1.c}$
$\sigma_{2.d}$ $\sigma_{1.d}$
$\sigma_{2.e}$ $\sigma_{1.e}$
$\sigma_{2.f}$ $\sigma_{1.f}$
$\sigma_{2.g}$
$\sigma_{1.h}$ $\sigma_{2.h}$

Figure 3.1.: **Dynamic Topology Systems** with re-usable identities and dangling links. Each column shows a snapshot of car platooning with level of abstraction increasing from left to right. Time (non-uniformly) increases from top to bottom.

3.1. Modelling Multi-Agent Systems by Evolving Topology Transition Systems: The Intuition

In order to motivate the computational model as introduced in this chapter, we shall first examine a bit closer what the characteristics of the real-world systems we're supposed to cover actually are (cf. Chapter 1). To this end, Figure 3.1 shows side by side different levels of abstraction, from the "real-world" system to an intuition of the formal model.

3.1.1. Multi-Agent Systems

Column (i) of Figure 3.1 shows, with time (non-uniformly) increasing from top to bottom, *snapshots* of real-world car platooning. That is, a system of (traffic) *agents* freely entering and leaving a highway and autonomously negotiating the merge into platoons (or convoys) with reduced safety distance (cf. Chapter 1).

Firstly, in row (a), there may be a single car cruising on the highway. Later another car may enter the highway and recognise the first car driving ahead ((b) to (c)). They may negotiate to form a platoon and reduce the safety distance in between and cruise on as a platoon (cf. snapshot (d)). After a while, the platoon leader may decide to leave the highway and to this end break up the platoon. Consequently, the safety distance is increased again (cf. snapshot (e)). Then the former leader actually leaves the highway, leaving the former follower in situation (f), which looks similar to snapshot (a). Later in time, another car may enter the highway and approach the former follower from behind (cf. snapshot (h)), possibly starting over the procedure to form a platoon.

3.1.2. Multi-Agent Systems with Re-usable Identities and Evolution

Column (ii) of Figure 3.1 takes a more conceptual view. First of all, the cars are distinguished by *unique identifiers* or *identities*. The car present in snapshot (a) has the identifier id_1 and the car appearing in snapshot (b) is assigned identifier id_2. When both have completed the negotiation procedure "*merge*" to form a platoon in snapshot (d), they know each other (by identifier), that is, there are (logical) *inter-agent connections* between the cars. Car id_2 knows that id_1 is its leader, as indicated by the *ldr* connection in snapshot (d), and car id_1 has id_2 as follower. These connections first of all indicate potential communication: the leader may inform the follower of an upcoming braking manoeuvre and the follower may send a request for splitting the platoon to the leader. In general, connections indicate that the source of the connection may interact with the destination. Which needn't be limited to message communication, but may include accessing the local memory of the destination for reading or even for writing. This is just not appropriate for the car platooning model. Cars will in the contrary be interested in encapsulating their local state from the environment and only communicate with the environment via a well-defined protocol.

Via the connection to the follower, the leader may initiate a "*split*" manoeuvre, for instance in order to leave the highway. At the end of the splitting procedure, the leader will have given up the connection to its followers, thus the *flw* connection from id_1

to id_2 is gone in snapshot (e). As cars are autonomous agents, the reverse link from the follower to the leader need not be gone as well at this point in time. That is, because the connection is logically by identification, the follower may consider car id_1 to be its leader even if the car formerly carrying this identification has already left the highway (cf. snapshot (g)). Making matters worse, the identification may principally be re-assigned, for instance by some roadside equipment, that recognises cars entering the highway. Then the formerly dangling connection may again denote a legal destination for communication. For example, in snapshot (h), car id_2 may request a split at car id_1 although the car with identification id_1 may now be in free-agent mode such that it doesn't expect such requests.

3.1.3. Topologies and Evolution

Finally, row (iii) in Figure 3.1 shows how we're going to model such multi-agent systems formally. We'll discuss three aspects in the following sections. Firstly, snapshots are represented by graphs, called *topologies*, whose nodes represent identities. Secondly, over a sequence of topologies, *evolution*, that is, which node evolves into which, is indicated by an evolution relation. And finally, there is a notion of *life-cycle* defined in terms of evolution, that is, distinguishing new, alive, or disappearing identities.

Topologies

A snapshot in column (iii) is a graph, which we'll call *topology.* The nodes of the graph are *identities*, that is, there may be a node even if there isn't a corresponding car as in snapshot (g). The reason for choosing identities as nodes instead of agents, i.e. cars in the particular example, is that agents as nodes would not allow to naturally model dangling connections and re-use of identities.[1] Nodes may principally appear in and disappear from the graph, but we'll see later that there is not much practical difference between a constant graph of possibly unused nodes and a dynamically changing graph comprising only nodes in use.

Nodes may be labelled with a *local state* σ and are then called *individuals.* In the example, the local state may comprise, for example, the current role of a car, that is, whether it is acting as a leader or as a follower. The domain of the local state labels is a priori not assumed to be finite, it may also comprise, for instance, the $\mathbb{R}$-valued current speed of a car. In Figure 3.1, we depict the label of a node by $\sigma_{i.n}$, $i = 1, 2$ and n the snapshot, and connect it to the node with a dotted line in order to resolve possible ambiguities.

Note that nodes intentionally need not have a label, the labelling function is only partial[2]. In Figure 3.1, for example, node id_1 has lost its label in snapshot (g) because the corresponding car *disappeared* beforehand. It models that there no longer is a car

[1] When re-visiting this issue in Section 3.7 where we relate our computational model to other, existing ones, we'll see that there are domains where dangling connections and re-use don't matter or can be assumed not to exist and that our approach is the more general one.

[2] Alternatively, one could choose a designated label to indicate that a node doesn't have a sensible label.

with this identity on the highway. Thus in particular the result of querying the current mode or speed of id_1 becomes undefined.

Further note that disappearance denotes the point in time when the behaviour of the considered individual becomes *undefined*. In the running example, this would be the case when a car leaves the range of the wireless network. In a different setting, like object-oriented programs with dynamic creation and destruction of objects this would clearly be the case after the destructor has completely been executed and the memory has been reclaimed. Accessing the former object via a now dangling link yields undefined results.

If the memory-management in the latter setting *would* guarantee that an access after destruction would yield the last value after destruction or a defined constant value, it would be a matter of choice how to map this memory-management into our computational model. If both, destruction and the guaranteed values after destruction *and* re-use are of interest, a natural approach would be to keep the local state label or set it to a constant value and to shift the aspect of being after destruction into the local state. In other words, to add a boolean component to the local state σ which indicates whether the individual has been destroyed.

If re-use in the sense that former dangling links may become valid again later is not of interest, a natural approach would be to have multiple designated identities like NULL providing the guaranteed constant value and changing the links at disappearance time.

If one is not interested in the time after destruction, one could simply remove the local state label from the corresponding identity at the point in time where destruction completes. In all three cases, re-use would be modelled by a discontinued chain of evolution (see below).

Edges, which we'll call *links*, are directed and are labelled with labels from a finite alphabet. In Figure 3.1, links are depicted by solid arrows, in the example labelled with *ldr* or *flw*. Note that in general, we don't assume that links are uniquely determined by source, destination, and label. For example, one individual is allowed to have multiple links with the same label to a single identity. Thus formally, topologies are labelled multi-graphs (cf. Section 2.3).

Evolution over Time

Until now we've only discussed single snapshots represented by single topologies, but not their evolution over time. As we're considering identities as nodes — instead of completely anonymous nodes like for example in [190, 191] — at first sight a natural notion of evolution of individuals could be to consider an individual to evolve between two topologies if and only if its identity is present and labelled in both topologies.

For example, the combination of identity id_1 being labelled by $\sigma_{1.a}$ in snapshot (a) and by $\sigma_{1.b}$ in snapshot (b) would mean that a particular car with identity id_1 evolved from local state $\sigma_{1.a}$ to $\sigma_{1.b}$ between the two snapshots.

At second sight, this notion falls short of sensibly capturing evolution in the presence of re-use of identities. As an example consider the two steps from snapshot (f) over

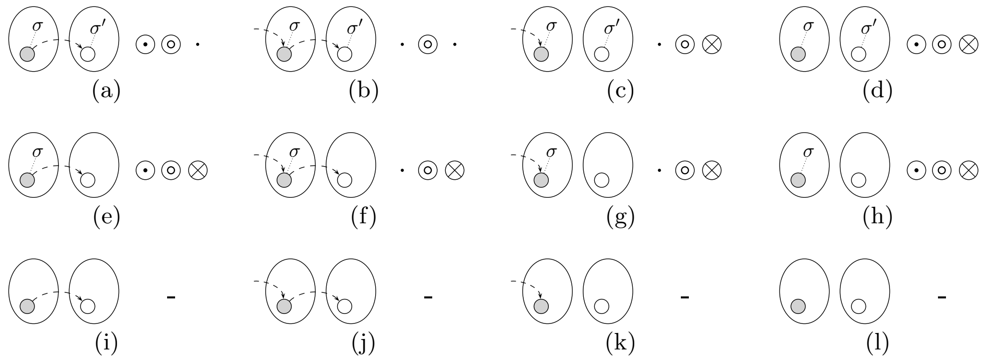


Figure 3.2.: **Life-cycle of individuals** in terms of being labelled with a local state and being source or destination of an evolution.

snapshot (g) to snapshot (h). In the corresponding topologies we have exactly the situation just described: node id_1 is labelled with local state $\sigma_{1.f}$ in topology (f) and with local state $\sigma_{1.h}$ in topology (h) but these are the local states of *different* cars. The evolution of the car denoted by id_1 in snapshot (f) actually ended in snapshot (f) and a different car obtained identity id_1 in snapshot (h). From there on, it may evolve or even choose to immediately leave the highway again, that is, choose not to evolve.

For this reason, we adopt a more elaborate approach, principally following [190, 191] where evolution of anonymous objects is explicitly traced. Similarly, we consider an explicit *evolution relation* between the sets of individuals of different topologies. In Figure 3.1, this relation is indicated by dashed arrows between nodes. These arrows indicate, for example, that the car from snapshot (a), or, more general, the agent denoted by identity id_1 in snapshot (a), evolves until snapshot (f) and then leaves the highway.

Life-Cycle

From the description of the previous section, we can tell that our model distinguishes a four-phase *life-cycle* of individuals: an individual, i.e. a labelled identity, can be

1. *newly created*, that is, alive for the first point in time,

2. just *alive*,

3. *disappearing*, that is, alive for the last point in time, and

4. none of the three phases, where we strictly speaking don't even have an individual but only an identity.

The combination of the evolution relation and the labelling of the node indicates in which phase of its life-cycle an agent is in a certain topology. Partly anticipating Chapter 4, Figure 3.2 shows all possible cases an individual might face. Like in column (iii) of Figure 3.1, a topology is rendered as an ellipsis, individuals as circles, and labels are attached to individuals via dotted lines. The dashed lines indicate the evolution relation, which may originate at an individual in a previous topology which is not shown in Figure 3.2. The individual we're interested in is the one rendered with a solid gray interior in each case.

First of all, an individual is *alive*, denoted by the symbol ⊚, if and only if it is labelled with a local state σ. This is the case for all situations in Figure 3.2 except for the cases 3.2(i) to 3.2(l) in the bottom row.

An individual is *newly created*, as denoted by the symbol ⊙, if and only if it is alive and didn't evolve along the transition to the current topology. In Figure 3.2 this is the case for the four cases 3.2(a), 3.2(d), 3.2(e), and 3.2(h). It is not the case for case 3.2(i) although there is no incoming evolution relation, because in this case the considered individual is not labelled, thus not alive.

An individual is *disappearing*, that is, alive for the last point in time as denoted by the symbol ⊗, if and only if it is alive and isn't in evolution relation with another individual or if it is but the other individual is not alive. The former criterion holds for cases 3.2(c), 3.2(d), 3.2(g), and 3.2(h), the latter for the whole middle row.

Note that an individual cannot be newly created or disappearing without being alive but it may be only alive as demonstrated by case 3.2(b).

From this brief discussion we can already tell that there are some cases where the evolution relation is redundant. Firstly, there are two indications that an individual disappears: it may not evolve, like in cases 3.2(c) to 3.2(k) and 3.2(d) to 3.2(l) in the lower two rows of Figure 3.2, or it evolves into an unlabelled identity, like in cases 3.2(e) to 3.2(h) in the middle column of Figure 3.2. Secondly, the evolution relation doesn't change our view on unlabelled nodes, that is, identities not in use by an alive individual like in cases 3.2(i) to 3.2(l) in the right column of Figure 3.2. The notions of being newly created or disappearing is simply not defined for non-alive individuals.

But without the evolution relation we obtain ambiguous situations. For example, without the evolution relation we couldn't distinguish case 3.2(a) from 3.2(d), that is, couldn't tell whether a re-use of the gray marked identity takes place as in case 3.2(d) or not as in case 3.2(a).

3.1.4. Overview and Graphical Representation

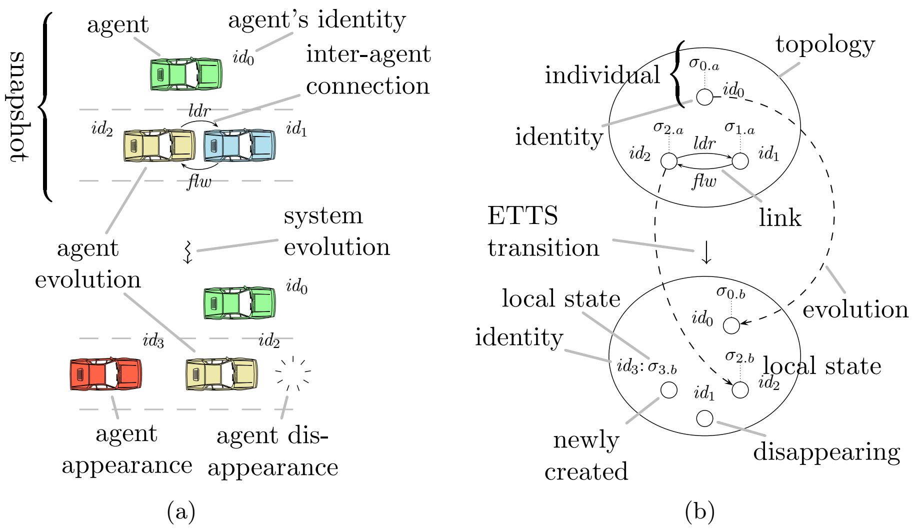
Table 3.1 provides an overview over the concepts mentioned in the previous sections and provides references to the places in the following text where they are formally defined.

The left two columns of Table 3.1 are related to columns (i) and (ii) of Figure 3.1 and Section 3.1.1 and 3.1.2. The right three columns are related to column (iii) of Figure 3.1 and Section 3.1.3.

Figure 3.3 graphically illustrates the concepts from Table 3.1, showing the graphical representation of the modelled system and the formal representation side by side. In the following, we will adhere to the graphical representations introduced in Figure 3.3 in order to illustrate definitions, discussions, or examples, but note that we may omit some of the annotations or relations shown in Figure 3.3 when they are clear from context or not relevant for the illustrated context.

"real-world"	example	model	name	alter. names	cf.
system	car platooning	labelled transition system	ETTS	DCS	Def. 3.5.1, p. 57
snapshot	highway	labelled multigraph	topology	world, configuration, situation, global state, state	Def. 3.2.1, p. 45
agent	car	labelled node	individual	object, process	Def. 3.2.1, p. 45
agent's state	free-agent	node label	(local) state	state, mode	Def. 3.2.1, p. 45
unique identifier	network address	node, identity mapping into nodes	identity	name	Def. 3.2.1, p. 45
inter-agent connection	leader	edge	link	channel, pointer, reference	Def. 3.2.1, p. 45
system evolution	merge procedure	ETTS transition	transition	-	Def. 3.5.1, p. 57
agent evolution	become leader	evolution relation	evolution	-	Def. 3.4.1, p. 51
agent appearance	car enters highway	didn't evolve	newly created	object creation	Def. 3.4.1, p. 51
agent disappearance	car leaves highway	evolves into unlabelled, if	disappearing	object destruction	Def. 3.4.1, p. 51

Table 3.1.: **Concepts and names introduced in this chapter.** The "real-world" column gives the name we'll use to refer to the phenomena we're modelling, like agents. The next column mentions examples or instances of theses names.
The middle column lists the mathematical means we use to model the corresponding concepts and column labelled "name" gives the name we'll use throughout this work to denote parts of the model.
The rightmost column provides alternative names used in related works for the corresponding concept. We shall avoid these alternative names in the following.



(a) (b)

Figure 3.3.: **Graphical representation** of the concepts and names introduced in this chapter (cf. Table 3.1). We elaborate on the right-hand side pictures in Figure 3.4 and Example 3.2.2.

3.2. Topologies

As discussed in Section 3.1, a topology is basically a labelled multi-graph with nodes taken from a fixed set of identities and with the uncommon partial node-labelling.

The set of identities is assumed to be equipped with a three-valued function comparing for identity. This function can be thought of as the natural equality relation until Chapter 5 on abstraction, where we'll have identities which compare neither equal nor unequal to itself.

Similarly, the definition of alive and non-alive individuals is already prepared for the needs of abstraction in Chapter 5 where we'll have individuals which are both alive and non-alive.

Definition 3.2.1 (Topology)**.** *Let* (Id, eq_{Id}) *be a (possibly infinite) set of* identities *equipped with an equality function* $eq_{Id} : Id \times Id \to \mathbb{B}_3$ *and let* Σ *and* Λ *be (possibly infinite) sets of* local states *and* link names*.*

A tuple

$$G = ((U^{\circledcirc}, U^{\not\circledcirc}), L, \psi, \sigma, \lambda) \tag{3.1}$$

is called (Σ, Λ)-topology over *Id, topology for short, if and only if*

$$(\underbrace{U^{\circledcirc} \cup U^{\not\circledcirc}}_{=:U}, L, \psi, \sigma, \lambda) \tag{3.2}$$

is a (Σ, Λ)*-labelled multi-graph with* $U \subseteq Id$*,* $\sigma : U \nrightarrow \Sigma$ *total on* $U^{\circledcirc}$ *(and partial on* $U^{\not\circledcirc}$*), and* $\lambda : L \to \Lambda$ *total.*

The elements of $U^{\circledcirc}$ *are called* individuals *(or* alive*), the elements of* $U^{\not\circledcirc}$ *are called* non-alive*, and the elements of* L *are called* links*. Given an identity* $id \in U$*,* $\sigma(id)$ *is called the* local state *of* id*; given a link* $\ell \in L$*,* $\lambda(\ell)$ *is called the* name *of* ℓ*.*

We assume associated with link name $\lambda \in \Lambda$ *of topology* G *a set* $\mathrm{dom}(\rightarrowtail_\lambda)$ *as the domain of* link navigation *and a function* $\rightarrowtail_\lambda : U \to \mathrm{dom}(\rightarrowtail_\lambda)$ *providing link navigation. We may write* $id \rightarrowtail \lambda$ *in in-fix notation to denote* $\rightarrowtail_\lambda(id)$*.*

The components of G *are referred to as* $U^{\circledcirc}(G)$*,* $U^{\not\circledcirc}(G)$*, their union as* $U(G)$*,* $L(G)$*,* $\psi(G)$*,* $\sigma(G)$*,* $\lambda(G)$*, and* $\rightarrowtail^G_\lambda$*, respectively.* $\diamondsuit$

If not otherwise noted, eq_{Id} is defined as

$$eq_{Id}(id_1, id_2) = \begin{cases} 1 & \text{, if } id_1 = id_2 \\ 0 & \text{, otherwise} \end{cases} \tag{3.3}$$

and the domain of all link navigation functions is $\mathfrak{M}(Id)$, the set of multi-sets over Id, and each function yields for identity id the multi-set of the destinations of all links with source id and name λ, that is

$$\rightarrowtail_\lambda(id) = \{\!|\, id_0 \mapsto |\{\ell \in L \mid \lambda(\ell) = \lambda \wedge \psi(\ell) = (id, id_0)\}| \mid id_0 \in U \,|\!\}. \tag{3.4}$$

Note that $U^{\circledcirc}$ and $U^{\not\circledcirc}$, and thus the derived notions of being alive or non-alive, are a priori not disjoint. This will, as said above, turn out useful in the section on abstraction where identities may indeed be alive and non-alive at the same time.

In this and the subsequent chapter though, where we're referring to concrete topologies, one can in most cases think of both sets as being disjoint.

The notion of finiteness carries over from graphs, i.e. from Def. 2.3.3, to topologies.

Example 3.2.2 (Topology)**.** *Let* $Id \supseteq \{id_0, id_1, \ldots, id_{27}\}$ *be a set of identities with natural equality,* Σ *a set of local states, and* $\Lambda = \{ldr, flw\}$ *a set of link names.*

Then

$$G = ((U^{\circledcirc}, U^{\not\circledcirc}), L, \psi, \sigma, \lambda) \tag{3.5}$$

with

- $U^{\circledcirc} = \{id_0, id_1, id_2, id_{27}\}$, $U^{\not\circledcirc} = \{id_4, id_5\}$,
- $L = \{\ell_0, \ell_1, \ell_2, \ell_3\}$,
- $\psi = \{\ell_0 \mapsto (id_0, id_1), \ell_1 \mapsto (id_1, id_2), \ell_2 \mapsto (id_2, id_1), \ell_3 \mapsto (id_2, id_0)\}$,
- $\sigma = \{id_0 \mapsto \sigma_0, id_1 \mapsto \sigma_1, id_2 \mapsto \sigma_2, id_{27} \mapsto \sigma_3\}$,
- $\lambda = \{\ell_0 \mapsto ldr, \ell_1 \mapsto flw, \ell_2 \mapsto ldr, \ell_3 \mapsto flw\}$.

is a (Σ, Λ)*-topology over* Id*.*

Figure 3.4(a) introduces the graphical representation of topologies on the example of G*, Figure 3.4(b) a possible "real-world" Car Platooning snapshot modelled by topology* G*.* $\diamondsuit$

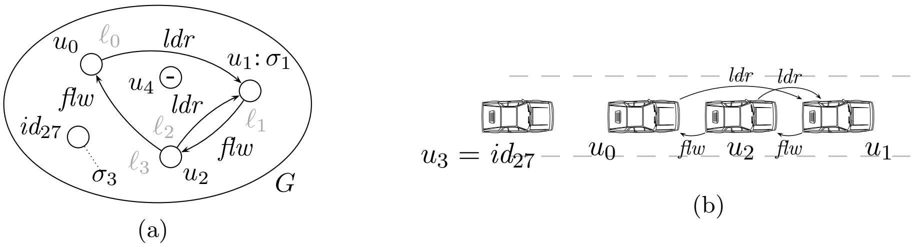
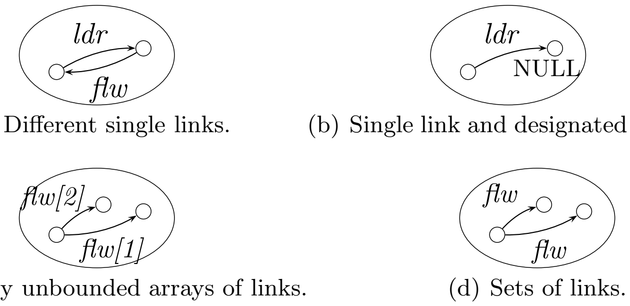


Figure 3.4.: **Graphical representation** of topology G from Example 3.2.2 following Figure 3.3. A topology G is basically represented as a node and edge labelled graph, enclosed in an ellipsis. The ellipsis indicates the extension of G, that is, the set of alive and non-alive identities from $U^{\circledcirc}$ and $U^{\not\circledcirc}$, but identities irrelevant for a certain illustration needn't be shown, in particular unlabelled identities are typically omitted, e.g. u_5. Identities are represented by a circle. The identity from Id may be shown next to a circle denoted by the name $id \in Id$, e.g. id_{27}, alive ones alternatively by u_i.
If the local state of an individual is shown, it either follows the identity separated by a colon, e.g. u_1, or is connected to the circle by a dotted line, e.g. id_{27}. To explicate that an identity is unlabelled, it may be rendered with a dash within the circle as shown for u_4. Unlabelled identities are typically only shown if they're affected by dangling links.
Links from L are depicted by arrows between identities which may be labelled with the link name. The gray labels in Figure 3.4 are only shown for completeness, to explicate the connection between L, ψ, and λ. Links not relevant for a certain illustration may be omitted.



(a) Different single links.

(b) Single link and designated NULL.

(c) Possibly unbounded arrays of links.

(d) Sets of links.

Figure 3.5.: **Different notions of agent connections.**

Note that the weak restrictions on links in Def. 3.2.1, firstly, that there may be multiple links of the same name, and secondly, that the set of labels needn't be finite, provides for the coverage of a broad range of paradigms (or classes) of agent interconnection. Namely, we can have

- at most one link per link name to individuals, for example connecting leader and follower in a platoon; the leader link, for instance, is simply *absent* if a car doesn't have a leader (cf. Figure 3.5(a)), if all topologies have this form, we'll say it has the single-link property (cf. Def. 3.2.3),
- alternatively, a car not having a leader can be modelled having the *ldr*-link point to a designated NULL identity (cf. Figure 3.5(b)),
- the previous models allow to represent followers as kind of a singly linked list (for instance, cf. Figure 3.4), alternatively they can be kept in an array; then each *flw*[*i*] is simply a link name (cf. Figure 3.5(c)), and
- alternatively, an unbounded number of followers can be modelled as a set of links, that is, by having many links with the same name (cf. Figure 3.5(d)).

There may even be multiple links with the same name between two individuals, which corresponds to multi-sets of links (in Figure 3.5(d), there could be an additional link with name *flw* from the left-most individual to one of the two on the right).

Definition 3.2.3 (Single-Link Property)**.** *A topology G has the* single-link *property if and only if for each pair of identities $id_1, id_2 \in U(G)$ and each link name $\lambda \in \Lambda(G)$, there it at most one link $\ell \in L(G)$ with name λ from identity id_1 to id_2, i.e.*

$$\begin{aligned}\forall\, \ell_1, \ell_2 \in L(G) :\ &(\psi(G)(\ell_1) = \psi(G)(\ell_2) \wedge \lambda(G)(\ell_1) = \lambda(G)(\ell_2)) \\ &\implies \ell_1 = \ell_2.\end{aligned} \tag{3.6}$$

$\diamondsuit$

Note that alternatively, we can give an analogous definition of a *c-link* property for any number $c \in \mathbb{N}^+$. Then the class of topologies with the single-link property is identical to the class of topologies with the 1-link property. Further note that the class of topologies with the c-link property is isomorphic to the class of topologies with the single-link property if we use $\Lambda \times \{1, \dots, c\}$ as link names, that is, if we explicitly count the multiplicity of links in link names.

This property will be useful in Chapters 6 and 9 to tell when the abstraction for a topology-labelled transition system is finite. Because if we don't have the single- or c-link property, there may be infinitely many links in a topology even if there are only finitely many individuals.

A distinguishing feature of our model over more natural graphs as employed in [7, 12] or first-order logical structures, as in [190, 191], is that we can represent all kinds of nasty effects emergent in dynamic topology applications.

Dangling links as introduced in the following, that is, links from an alive individual to non-alive individuals, are among the most prominent ones. In a clean graph or logical structure setting in contrast, disappearance of a node causes immediate disappearance of all adjacent links. We will classify ETTS along these concepts in more detail in Section 3.6.

Definition 3.2.4 (Dangling Link). *A link $\ell \in L(G)$ of a topology G is called* dangling *if and only if its destination is non-alive, i.e. if $\psi(G)(\ell) = (u, u')$ and $u' \in U^{\not\varnothing}(G)$.* $\diamondsuit$

A second effect we're able to model is re-use of identities. For example in pointer programs, a chunk of memory denoted by a dangling pointer may later be re-used for another newly created object (also cf. Section 3.1).

The following definition generalises the untyped topologies of Def. 3.2.1 to typed ones. Definition 3.2.1 is then the special case with only a single subset of *Id* providing the partitioning

Typing doesn't affect most of the following definition, and on the level of transition systems we needn't be aware of the further structure. Typing will be considered in Chapter 4 where the specification logic is defined and in the later sections on abstraction, in particular Chapters 6 and 9. One of the properties of the abstraction considered in Chapter 6 is that it is sensitive to typing information in the sense that different types are treated differently.

Definition 3.2.5 (Typed (Or: Many-sorted) Topology). *A (Σ, Λ)-topology G over identities Id is called* typed *if and only if*

- *Id and Σ are partitioned into n partitions*
$$Id_1 \,\dot\cup\, \ldots \,\dot\cup\, Id_n \text{ and } \Sigma_1 \,\dot\cup\, \ldots \,\dot\cup\, \Sigma_n, \tag{3.7}$$
- *individuals from Id_i are assigned local states from Σ_i by σ, i.e.*
$$\sigma|_{Id_i} : Id_i \nrightarrow \Sigma_i \tag{3.8}$$
for $1 \leq i \leq n$,
- *links with the same name connect individuals from the same partitions, i.e.*
$$\forall\, \lambda \in \Lambda \exists\, 1 \leq i, j \leq n \forall\, \ell \in L(G) : \lambda(\ell) = \lambda \implies \psi(\ell) \in Id_i \times Id_j. \tag{3.9}$$
The relation between linknames λ and partitions Id_i and Id_j is called the typing *of link names.*

It is called many-sorted *if $n > 1$ and* single-sorted *otherwise. We call Id_i the* sort *of an individual.* $\diamondsuit$

Finally note that the sets Σ of local states and Λ of link names give hints on how the topologies may look like, but they don't completely determine the possible topologies – they furthermore depend on, for example, the multiplicity (or degree) of links, that is, whether there may be at most one link of a given name originating at each individual (see above), or $N \in \mathbb{N}^+$, or arbitrarily many.

We leave this underspecified here, the higher-level languages in Chapter 10 will briefly discuss how to impose restrictions, like that links are understood to be single links.

3.3. TTS: Topology-Labelled Transition Systems

The simplest model of applications with dynamically changing topologies is an LTS whose states are labelled with topologies over a common vocabulary of local states and link names. These are introduced as TTS in Def. 3.3.1 below.

In TTS, individuals in topologies of different states are a priori *not* explicitly related. Consequently there is not yet a notion of being newly created or disappearing, because these notions are intrinsically connected to considering individuals at different states being related.

For completeness, we also distinguish disjoint-universe TTS, where the topologies of different states use disjoint sets of nodes although this doesn't make a semantical difference for TTS, but it will make a difference for ETTS, the next stage.

Definition 3.3.1 (Fair Topology Transition System)**.** *A labelled fair transition system*

$$M = (S, S_0, R, \mathscr{L}, F) \tag{3.10}$$

is called (fair) (Σ, Λ)-topology transition system *(TTS) over Id if and only if* $\mathscr{L} : S \to \mathscr{D}$ *assigns each state a* (Σ, Λ)*-topology over the set of identities Id.*

Let $s \in S$ *be a state of* M*. The topology* $\mathscr{L}(s)$ *is then called* the topology of s *and we write, e.g.,* $U^{\circledcirc}(s)$ *to access the components of the state's topology, that is, as an abbreviation for* $U^{\circledcirc}(\mathscr{L}(s))$*.*

We say the TTS M *is of the* disjoint-universe kind *if and only if the topologies of all states have pairwise disjoint sets of identities, that is, if* $U(s) \cap U(s') = \emptyset$ *for all* $s \neq s' \in S(M)$*, otherwise it's said to be of the* shared-universe *kind.*

M *is called* typed *if all topologies are typed with the same typing of link names.* ◇

In other words, the notion of *identity* in TTS is a rather local one. This is different to the ETTS defined in the following section. From there on, we have the option to also consider identities to be meaningful across topologies of different LTS states, and may ask questions like whether the car denoted by identity *id* may become a leader within at most three steps from now.

In particular note that the notion of TTS is not yet sufficient to faithfully model the systems discussed in Section 3.1. But they are principally still useful; in Section 3.7 we'll discuss a number of related approaches where one wants to abstract from (global) identities right-away.

For example, if we would only be interested in whether an illegal connection, where two cars mutually consider each other to be the leader, is possible, then to prove the absence of this situation, we needn't necessarily trace which car is which in what situation. For instance, [7, 12] follows this principle.

3.4. Evolution Relation

The final missing concept is evolution, that is, in particular providing the information whether an individual is new, going to disappear, or none of both (cf. Figure 3.2). We will provide it by annotating transitions with an evolution relation[3] as defined in the following.

Note that the definition is slightly more general than necessary for our primary purpose, namely the investigation of the DTR abstraction [127] in Chapter 6. There, we will only consider linear evolution (see below), while the general definition also provides notions of merging and splitting, which don't *directly* correspond to, e.g. in the Car Platooning application, but could serve to represent whole platoons by a single node of type "platoon" next to cars.

Still, the general definition allows us to define the specification logic in Chapter 4 for the general case, and to compare other approaches to ours in our terms in Section 3.7.

Definition 3.4.1 (Evolution Relation). *Let M be a TTS over identities Id.*

1. *Let $r = (s, s') \in R(M)$ be a transition of M. A relation $e\langle r\rangle$ between the identities in the topologies of the source and the destination state of r, i.e. $e\langle r\rangle \subseteq U(s) \times U(s')$, is called* evolution relation *of r.*

   We call it linear *if and only if it is an injective partial function, i.e. $e\langle r\rangle : U(s) \rightarrowtail\!\!\!\!\!\!\;+\; U(s')$. We call it* summarising *if and only if it is non-injective and* splitting *if and only if it is not a partial function.*

   It is called identity-preserving *if and only if $e\langle r\rangle$ doesn't relate different identities, i.e. if $(id, id') \in e\langle r\rangle$ implies $id = id'$.*

2. *Let $e\langle r\rangle$ be an evolution relation of a transition $r = (s, s') \in R(M)$ of M.*

   a) *We say an individual $u \in U^{\circledcirc}(s)$* evolves into *$u' \in U(s')$ along r, denoted by*

   $$u \overset{r}{\rightsquigarrow}_e u' \quad \textit{or (alternatively)} \quad u \xrightarrow{(s,s')}\!\!\!\!\rightsquigarrow_e u', \tag{3.11}$$

   if and only if $(u, u') \in e\langle r\rangle$.

   b) *An individual $u \in U^{\circledcirc}(s)$ is said to* disappear along *r, denoted by*

   $$u \overset{r}{\rightsquigarrow}_e \text{◌} \quad \textit{or (alternatively)} \quad u \xrightarrow{(s,s')}\!\!\!\!\rightsquigarrow_e \text{◌}, \tag{3.12}$$

   if and only if u is not in the domain of $e\langle r\rangle$, i.e. $u \notin \mathrm{dom}(e\langle r\rangle)$, or if u evolves into a non-alive identity, i.e. $u \overset{r}{\rightsquigarrow}_e u'$ and $u' \in U^{\not\circledcirc}(s')$.

   c) *An individual $u \in U^{\circledcirc}(s')$ is said to* appear *or* be newly created along *r, denoted by*

   $$\text{◌} \overset{r}{\rightsquigarrow}_e u \quad \textit{or (alternatively)} \quad \text{◌} \xrightarrow{(s,s')}\!\!\!\!\rightsquigarrow_e u, \tag{3.13}$$

   if and only if u is not in the range of $e\langle r\rangle$, i.e. $u \notin \mathrm{ran}(e\langle r\rangle)$, or if u is not alive in s, i.e. $u \in U^{\not\circledcirc}(s)$.

[3] Called counterpart relation in [116].

3. *Let $\pi = s_0, s_1 \ldots, s_n$ be a finite computation path in M such that each transition $r_i = (s_i, s_{i+1})$, $0 \leq i < n$, has an evolution relation $e\langle r_i \rangle$.*

   We say an individual $u \in U^{\circledcirc}(s_0)$ evolves into *$u' \in U(s_n)$* along *π, denoted by*

$$u \overset{\pi}{\rightsquigarrow}_e u', \tag{3.14}$$

   if and only if there are individuals $u_i \in U^{\circledcirc}(s_i)$, $0 < i < n$, such that

$$u \overset{r_0}{\rightsquigarrow}_e u_1 \overset{r_1}{\rightsquigarrow}_e u_2 \ldots u_{n-1} \overset{r_{n-1}}{\rightsquigarrow}_e u'. \tag{3.15}$$

   Such a sequence of individuals $u, u_1, u_2, \ldots, u'$ is called (finite) evolution chain *of u. An evolution chain may be infinite if the individual doesn't disappear.*

4. *Let $\pi = s_0, s_1, s_2 \ldots$ be a computation path in M such that each transition $r_i = (s_i, s_{i+1})$, $i \in \mathbb{N}_0$, has an evolution relation $e\langle r_i \rangle$.*

   Given a identity $id \in U(s_0)$, we use $\Delta(id, \pi)$ to denote the set of maximal finite and infinite evolution chains of id along π (the destiny *of u), i.e. the set*

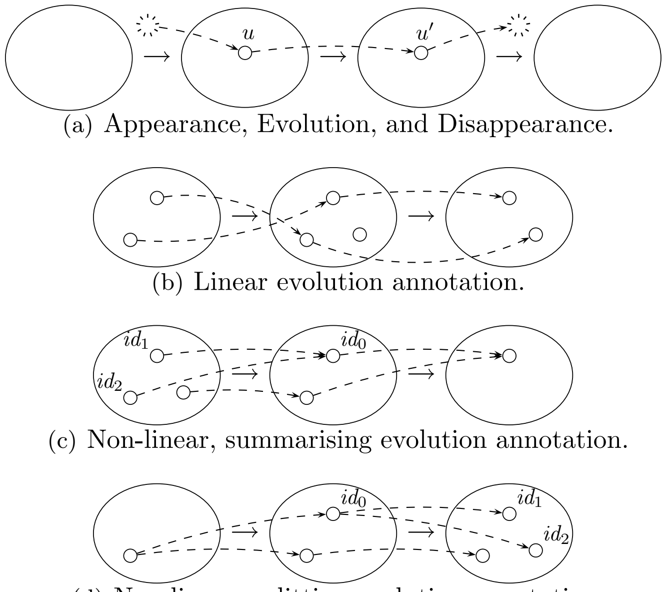
$$\begin{aligned}\Delta(id, \pi) = \{\{u_0, u_1, \ldots, u_n\} \ \mathit{maximal} \mid u_0 = id, u_i \xrightarrow{s_i, s_{i+1}}_e u_{i+1}, 0 \leq i < n\} \\ \cup \{u_0, u_1, \ldots \mid u_0 = id, u_i \xrightarrow{s_i, s_{i+1}}_e u_{i+1}, i \in \mathbb{N}_0\}.\end{aligned} \tag{3.16}$$

5. *An* evolution annotation *of M is a mapping $e : R \to \mathfrak{P}(Id \times Id)$ which assigns each $r \in R$ an evolution relation $e\langle r \rangle$. An evolution annotation is called linear if and only if each evolution relation is linear, and summarising or splitting if and only if at least one evolution relation is summarising or splitting.* ◇

Note 3.4.2. *Let M be a TTS and $u \in U(s)$ an individual. Then by Def. 3.4.1, u evolves into itself along the empty path prefix ϵ, i.e. $u \overset{\epsilon}{\rightsquigarrow}_e u$.* ◇

Having defined evolution as a relation (instead of a function) is rather weak. It obviously covers the natural four-phase life-cycle of individuals as introduced in Section 3.1.3 (cf. Figure 3.6(a) and Figure 3.6(b)). In addition, it principally allows for summarisation, for example Figure 3.6(c) can be viewed as *id* representing both, *id*$_1$ and *id*$_2$, in the second state. Complementary, it allows for splitting, so Figure 3.6(d) can be seen as *id* representing two identities which are unfolded into *id*$_1$ and *id*$_2$ in the last transition.

In the following, we will start with linear evolution and even restrict it further such that identities only evolve into itself and call this consistent evolution in Def. 3.4.4. Then, for instance, the evolution annotation shown in Figure 3.6(a) would not evolve consistently if u was different from u' (cf. Figure 3.7(c) vs. Figure 3.7(e)). This assumption is natural as, for instance, cars in the car platooning example don't merge into single cars and objects in object-oriented programming don't merge either. But as said above, one may wish to model Car Platooning with types for cars and platoons. A different application is in summarising abstractions where abstract nodes represent variable sets of concrete



(a) Appearance, Evolution, and Disappearance.

(b) Linear evolution annotation.

(c) Non-linear, summarising evolution annotation.

(d) Non-linear, splitting evolution annotation.

Figure 3.6.: **Evolution and evolution annotation.** A dashed arrow between nodes indicates an evolution relation between individuals. The arrows from and to locations outside the topologies in 3.6(a) emphasise that some individuals didn't or won't evolve, they are typically omitted.

nodes (cf. Section 3.7 and Chapter 5). We don't consider such abstractions, but with the general definition of evolution, they can be seen as proper ETTS (cf. Section 3.5) with merging and splitting.

We'll see in the Chapter 4 on requirements specification logic, that the formal semantics doesn't depend on this aspect of consistency; we "only" lose a direct intuition of what it means for certain formulae to hold or not to hold when permitting inconsistent evolution.

When redefining consistent evolution to a sense where it is sufficient that an identity *id* evolves into another identity which is equal to or represents *id*, we can have finite abstractions of infinite TTS where we still have a well-defined semantics of property specification formulae. This would not be possible by just annotating "this one does evolve" to an individual, which is obviously sufficient to encode linear evolution.

Note 3.4.3. *Let M be a TTS over identities Id. Let $\pi = s_0, s_1, s_2 \ldots$ be a computation path in M such that each transition $r_i = (s_i, s_{i+1})$, $i \in \mathbb{N}_0$, has an evolution relation $e\langle r_i \rangle$.*

Given an identity $id \in Id$, its evolution chain is the empty sequence if and only if id is not an individual in s_0, i.e.

$$\Delta(id, \pi) = \{\varepsilon\} \iff id \notin U^{\circledcirc}(s_0). \tag{3.17}$$

♢

Proof. Distinguish two cases, namely whether *id* is an individual in s_0 or not.

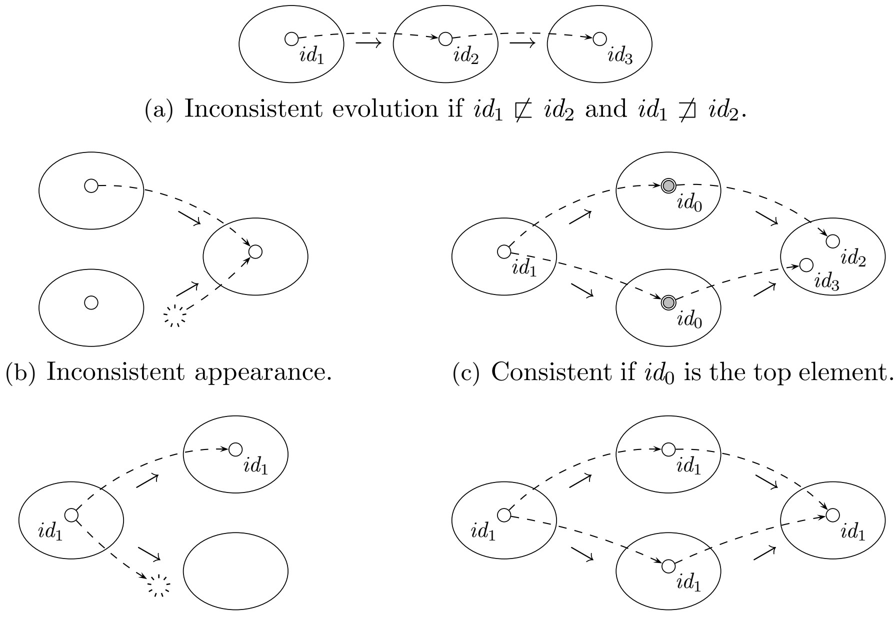
Let identity $id \in Id$ be not alive in s_0, that is, $id \notin U^{\circledcirc}(s_0)$. Any sequence different from the empty sequence is not a legal member of $\Delta(id, \pi)$ because evolution is only defined for individuals (cf. Def. 3.4.1.2a). The empty sequence ε is a legal member of $\Delta(id, \pi)$ because it is consequently maximal and the set-comprehension requirement trivially satisfied as there is no index i to consider.

Let identity $id \in Id$ alive in s_0, that is, $id \in U^{\circledcirc}(s_0)$. Then ε is not in $\Delta(id, \pi)$ because if *id* doesn't evolve, then the sequence *id* is in $\Delta(id, \pi)$, and if *id* does evolve, then there is a sequence with *id* and a successor id' in $\Delta(id, \pi)$. Thus $\emptyset$ is not maximal as both cases are proper extensions of ε. □

The other aspect of the evolution relation, next to tracing who evolves into whom, is that it encodes appearance and disappearance. An individual is considered to be disappearing in a certain state of a path, that is, alive for the last point in time, if it will not evolve along the transition that relates the current to the next state in the path.

In contrast, it is considered to be new in a certain state, that is, alive for the first point in time, if it didn't evolve along the transition which led to the current state. By this description, we'd have to look back in time. In the definition of the property specification logic in Chapter 4, we want to avoid looking back in time, that is, define appearance only in terms of a path suffix.

To this end, we introduce consistent appearance below. Intuitively, it requires that if an individual appears newly in a state, then it does so along all incoming transitions



(a) Inconsistent evolution if $id_1 \not\sqsubseteq id_2$ and $id_1 \not\sqsupseteq id_2$.

(b) Inconsistent appearance.

(c) Consistent if id_0 is the top element.

(d) Consistent evolution.

(e) Consistent evolution.

Figure 3.7.: **Inconsistent vs. consistent evolution.**

of that state (cf. Figure 3.7(b)). Then we can decide newness by considering the state only.

Note that this treatment of appearance and disappearance has the disadvantage of being unsymmetric in the sense that we need to establish additional requirements in order to define appearance, which we don't need for disappearance. This may correspond to the observation that it is perfectly legal that an individual appears along one transition while it remains alive along another one (cf. Figure 3.7(d)), while it seems strange to have appearance along some but not along other transitions.

On the other hand, our treatment has the advantage that it is symmetric in the sense that both notions, being new and being disappearing, lie naturally within the lifetime of an individual, i.e. the individual is also alive at both points in time (cf. Figure 3.3 on 45). This is similar to the approach of [57] while, for instance, [190, 191] chose to have an artificial state of individuals denoting that the individual has last been alive in the previous state.

Definition 3.4.4 (Consistent Evolution Annotation)**.** *Let M be a TTS over Id. An* evolution annotation *e of M is called* consistent *if and only if the following two conditions are met.*

1. *(Consistent Appearance) If an individual appears newly in a state $s' \in S$, then it*

doesn't evolve along any transition with destination s', i.e.

$$\begin{aligned}&\forall\,(s_1, s') \in R\ \forall\,u \in U(s') : \\ &\quad \text{☼} \overset{(s_1,s')}{\rightsquigarrow}_e u \implies \left(\forall\,(s_2, s') \in R : \text{☼} \overset{(s_2,s')}{\rightsquigarrow}_e u\right).\end{aligned} \tag{3.18}$$

2. *(Consistent Evolution) If $(Id, \sqsubseteq)$ is a partially ordered set, identities evolve into $\sqsubseteq$-related identities, i.e.*

$$\forall\,r \in R(M)\ \forall\,(id, id') \in e : id \sqsubseteq id' \vee id' \sqsubseteq id. \tag{3.19}$$

$\diamondsuit$

The rationale behind the definition of consistent evolution is to disallow unconstrained changes of identities. Our intuition is that an evolution relation is either identity-preserving, and thus only an expensive encoding of appearance and disappearance, or adheres to an information order on individuals.

In the first case, the encoding is expensive because it would be sufficient to simply annotate to a transition the set of identities remaining alive, because in this case, the evolution relation consists of pairs of the same identity, which doesn't carry more information.

Alternatively, one can think of annotating being new, alive, or disappearing directly to the individuals. Annotating disappearance is not an option because disappearance is a property depending on the taken transition, not on the state (cf. Figure 3.7(d)); on the other hand, appearance is actually not a property of transitions but rather one of states (cf. discussion related to Def. 3.4.4).

Allowing evolution between individuals that are related by an information order provides means to capture summarisation and splitting, and in particular treat them in the evolution chain-based interpretation of quantification (cf. Chapter 4). In the following, we will in most cases consider equality as partial order, which has the consequence that the evolution relation is linear and identity-preserving, thus not very involved.

Note 3.4.5. *Let M be a TTS over $(Id, =)$ and e a consistent evolution annotation of M with respect to the equality relation as partial order. Then*

1. *e is linear.*
2. *e is identity-preserving.*

Proof. Let M be a TTS over $(Id, =)$ and e a consistent evolution annotation. Let $r \in R(M)$ be a transition and e its evolution annotation. Let $(id, id') \in e$.

As the evolution annotation is consistent, id and id' are partially ordered, thus $id = id'$ because we assumed equality as partial order. Thus e is identity-preserving.

It is also linear because assuming id_1 and id_2 to evolve into id or id to evolve into id'_1 and id'_2 leads to the consequence that $id_1 = id_2$ and $id'_1 = id'_2$. $\square$

As TTS in general have an infinite number of states and transitions, we unfortunately have the following restriction.

Note 3.4.6. *Let M be a TTS and e an evolution annotation of $R(M)$. In general, it is undecidable whether e is consistent or not.* $\diamondsuit$

Still it can be proven in some cases, and, even more relevant, it can be obtained by construction. For example, in Chapter 9 we'll introduce a high-level modeling language which has an ETTS semantics. All ETTS obtained from the high-level modeling language sport consistent evolution by construction.

Further note that the specification logic of Chapter 4 doesn't depend on consistent evolution. It is also defined for ETTS with non-consistent evolution.[4]

Note 3.4.7. *Let M be a TTS and e a consistent evolution annotation. Every individual evolves into itself along a cycle, that is, if there is a path π in M from s to s then $u \overset{\pi}{\rightsquigarrow}_e u$.* $\diamondsuit$

Proof. By Note 3.4.2, $u \overset{\epsilon}{\rightsquigarrow}_e u$ thus by consistency also for any other path as the empty path leads from s to s. $\square$

3.5. ETTS: Topology-Labelled Transition Systems with Evolution

Given the evolution evolution relation by Section 3.4, we can complete TTS to obtain ETTS.

Definition 3.5.1 (Fair Evolving Topology Transition System)**.** *A (fair) (typed) TTS M over Id is called (fair) (typed)* evolving topology transition system *(ETTS) if and only if its transition relation $R(M)$ is equipped with an appearance consistent evolution annotation, i.e. if $R(M) = (R, e)$.* $\diamondsuit$

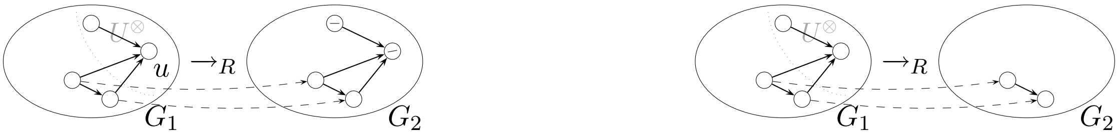
According to the following note we may, in the following, focus the discussion on ETTS as they are a proper superset of TTS.

A given TTS becomes an ETTS by annotating each transition with the empty relation, that is, by explicating that the TTS is not concerned with evolution over time but views all individuals as transient.

Note that the following statement uses the yet undefined notion of *equivalence* between transition systems which won't be formally introduced (as bisimulation) before Chapter 5.

Note 3.5.2 (TTS vs. ETTS)**.** *Each TTS M is "equivalent" to the ETTS obtained by equipping the transition relation of M with the empty transition annotation, that is, with $e = \{r \mapsto \emptyset \mid r \in R(M)\}$.*

[4] As a side-note, we're even able to specify inconsistency:
$\exists\, id \in Id : \mathsf{F}(\neg \circledcirc id \wedge (\mathsf{E}\,\mathsf{X} \odot id) \wedge \neg(\mathsf{A}\,\mathsf{X} \odot id))$ (cf. Chapter 4).



(a) Dangling links.

(b) Vanishing links.

Figure 3.8.: **Vanish vs. Dangling** link semantics.

Finally, we're interested in sufficient criteria for finiteness of ETTS. As the abstractions discussed from Chapter 5 onwards map into ETTS, we are supposed to tell whether the resulting ETTS is finite, which is our overall goal, or not. A simple sufficient criterion is the following.

Note 3.5.3 (Finiteness Criteria for ETTS)**.** *Let $M = (S, S_0, R, \mathscr{L}, F)$ be an (Σ, Λ)-ETTS over Id. Let the states of M be distinguished by their labelling, that is,*

$$\forall\, s_1, s_2 \in S : \mathscr{L}(s_1) = \mathscr{L}(s_2) \implies s_1 = s_2. \tag{3.20}$$

1. *In general, it is not sufficient for M being finite-state that the sets of local states Σ and link names Λ are finite.*

2. *In general, it is not sufficient for M being finite-state that all topologies have the single-link property in addition to 1.*

3. *M is finite state if and only if Id is finite, Σ is finite, Λ is finite, and all topologies have the single-link property.* ♢

3.6. Supporting Vanish and Re-use Semantics

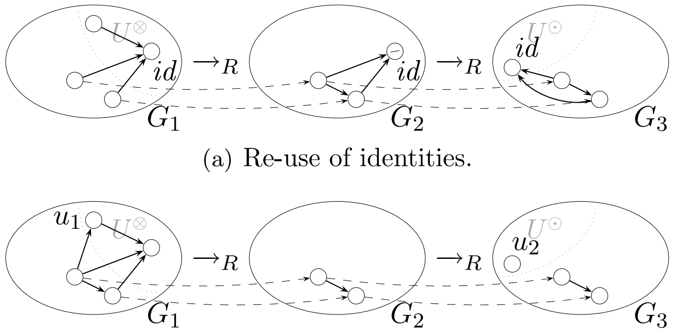
In the following, we elaborate a bit further on the difference between our approach and approaches employing more natural graphs like [7, 12] or first-order logical structures, as [190, 191] as promised in Section 3.2.

The two main issues are the treatment of dangling links and support of re-use of identities when ETTS, or other models, are used as semantical domain of higher-level languages (cf. Chapter 9).

The difference between dangling and vanishing semantics is illustrated by Figure 3.8. In general, disappearance of an individual in ETTS doesn't imply that links directed to it appear along.

For example, individual u is disappearing in topology G_1 in Figure 3.8(a)). Then, in general, the links to u remain, u is just non-alive in the subsequent topology G_2.

This corresponds to many programming languages supporting dynamic memory. Deallocating an object doesn't automatically care to remove all pointers referencing this object.



(a) Re-use of identities.

(b) No re-use of identities.

Figure 3.9.: **Re-use semantics.**

In contrast, with a vanish semantics, the interpretation of a deallocation of an object is that it is (in some not further specified way) cared for removal of all references to this object.

For example, individual u is also disappearing in G_1 in Figure 3.8(b), but it is removed from the topology in G_2 and all links are gone with it.

This corresponds to programming languages with garbage collection, where objects are only reclaimed if they are no longer reachable, that is, remain present at least as long as there are references.

Definition 3.6.1 (Vanish Semantics)**.** *An ETTS M over Id is said to model a* vanish semantics *if and only if there aren't any dangling links in any of its topologies, i.e. in any topology from $\mathscr{L}(S(M))$.* ♢

Assuming that an ETTS models a vanish semantics is obviously simplifying the representation (cf. Figure 3.8(b)). It is then adequate to employ more natural labelled multi-graphs to represent topologies, which in particular includes a total node labelling. This is the strategy of, for example, [7, 12].

Yet this assumption needs to be justified. We cover the more complex case, too, as it doesn't complicate the discussion too much and as our aim to apply the methods discussed here also to UML models with different semantics, in particular with action languages like C++, which clearly don't have a vanish semantics in ETTS.

The topic of re-use is illustrated by Figure 3.9. The individual *id* is disappearing in topology G_1 of Figure 3.9(a), and non-alive in G_2, with two dangling links pointing to it. For G_3, identity *id* is chosen for creation, so it is again alive in G_3 and the formerly dangling links are again pointing to a regular individual.

This effect can also be observed in programming languages with dynamic memory management and pointers, but also with network addresses in the Car Platooning.

In contrast, there may be high-level languages where the semantics of creation is that new individuals obtain unique, unused identities. This corresponds to a setting with strictly disjoint topologies, where new individuals cannot be traced back to individuals

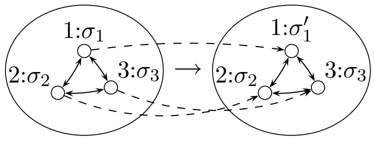


Figure 3.10.: **Parameterised System** of size $N = 3$ encoded as ETTS. In the shown transition, process 1 changes state from σ_1 to σ'_1.

that existed formerly. Figure 3.9(b) gives an example. The ETTS model again supports both cases.

3.7. Discussion

In the following, we compare the definition of ETTS to other well-known formalisms employed to model systems which resemble the dynamic topology systems discussed in Section 3.1.

In the comparison, we're basically interested in whether the considered formalism can encode ETTS and be encoded in ETTS, and in how far it doesn't suit our needs in the following chapters.

Note that some of the comparisons anticipate Chapter 9, where we introduce a higher-level language with ETTS semantics. For example, parameterised systems are often introduced and defined by a description language, which then has a transition system semantics. The latter corresponds to this chapter while the former is more related to Chapter 9 such that these aspects will be primarily considered there.

3.7.1. Parameterised Systems

We characterise the, in details very diverse, class of parameterised systems as follows. There is a single program which is executed by $N \in \mathbb{N}_0$ processes, the program is principally parameterised in identities. Processes are arranged in a fixed topology, in contrast to the dynamics we consider. In many cases processes access central data which is shared between all processes or are fully connected such that each process can read parts of every other processes' memory. There is no creation or destruction, all N processes are present in the initial state and remain alive throughout the system runtime. The challenge is to verify that, for all possible choices of N, the corresponding system instance has a certain property [32, 33].

Typical examples are cache coherency protocols, mutual exclusion, for example, via the bakery algorithm, or leader election algorithms. Variations stem from different fixed topologies, different concepts of shared memory, different limits to local data, etc.

A parameterised system can be encoded as an ETTS in two ways. Firstly it can be seen as a family $\{M_1, M_2, \dots\}$ of ETTS over identities $Id_N = \{1, 2, \dots, N\}$, $N \in \mathbb{N}_0$. Each M_N, $N \in \mathbb{N}^+$, features N newly created individuals in the initial state(s), none of them disappearing along a path. All these individuals are fully connected via one

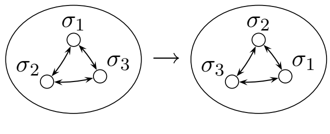


Figure 3.11.: **Abstract Parameterised System.** In contrast to Figure 3.10, identities are not traced.

link, all links are labelled with the same link name. A combination of shared and local memory can be encoded by adding a single designated individual in addition to the N ones modelling the shared memory (cf. Figure 3.10).

Secondly, it can be seen as the union M of the just named families, that is, with $Id = \mathbb{N}^+$ and transitions only between states whose topology is of the same size.

In general, an ETTS doesn't encode directly as a parameterised system because individuals need by no means behave symmetrically. That is, there needn't be a finite program which induces the desired ETTS behaviour when executed on any number of processes.

The higher-level languages considered in Chapter 9 are by far closer to parameterised systems because they assume a finite number of programs executed by processes. The difference to classical parameterised systems remains that they don't consider dynamic creation and destruction of individuals and that they assume fixed connections. That is, ETTS as considered in Chapter 9 are infinite-state systems while classical parameterised systems typically define infinitely many *finite*-state systems; unless there are local variables of unbounded domains like $\mathbb{R}$, or real-time aspects, etc.. In other words, the infinitely many instantiations are not infinite-state by the number of *identities*, but at most for other reasons like variables.

ETTS seem to most closely resemble something like the *limit* of classical parameterised systems for $N \to \infty$, with additional local variables indicating whether a process shall be considered newly appearing, alive, or disappearing.

In case of a given single-link property, links would be encoded as local variables with domain *Id*, that is, this would directly introduce infinite-domain local variables. Encoding ETTS that are lacking the single-link property would require to (ab-)use processes to encode the more general link connections.

Then we were able to capture the dynamics of, e.g., the Car Platooning case study. We'll employ exactly this encoding in Chapter 9 to check abstractions of high-level models in common model-checkers, but for studying the effects of abstractions in Chapters 5 and 6 it is not the right representation.

A variant are abstract parameterised systems (cf. Figure 3.11), for example obtained by counter abstraction. There the only information is how many processes are in which local state, but it is not traced over transitions which process changes local state.

3.7.2. Communicating Finite State Systems

Communicating finite state machines [21] (CFSM) are different from parameterised systems in that there is a fined finite number of processes, yet each one may execute a different (finite) program operating on a finite local state.

The processes are also fully connected, not via shared memory but with unbounded message-queues. A process has one message per possible sender, that is, one queue per other process. Sending is asynchronous, that is, non-blocking, the receiver needn't be ready to consume an event at the time it is sent.

Some model-checking problems are shown undecidable for CFSM in [21], intuitively caused by the unbounded queues.

CFSM are more closely related to the high-level language of Chapter 9, the main difference is that Chapter 9 considers a finite number of programs that can be executed by a variable and unbounded number of processes, while CFSM have a fixed number of processes.

The high-level language in Chapter 9 can, with its included message based communication, be seen as a generalisation of CFSM.

3.7.3. Array Programs

It is said that there is work showing that parameterised systems are equivalent to a kind of array programs.[5]

Following that idea, we can ask whether ETTS are equivalent to array programs with infinite arrays where it is also allowed that the index types may occur as the domain of an array, not only as the index.

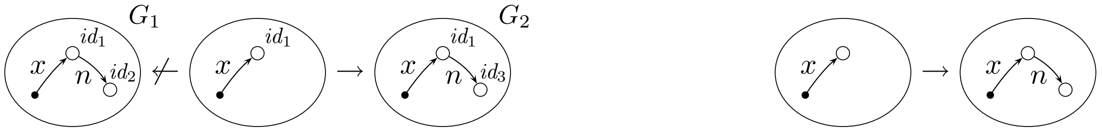
An array program can be encoded in an ETTS even with finite local state for example by having one kind of individual per array with the value as local state. An array index as value can be encoded as a link.

The other direction is similar to the case of parameterised systems. In general, an ETTS can show more behaviour than can be described in a finite program. But if we assume that an ETTS is given by something like the high-level language of Chapter 9, then these ETTS can be encoded in array programs by having one array per kind of individuals indexed by identities. Places of the arrays are chosen such that they represent the local state of an individual and the current life-cycle phase. This way, single-link systems are encoded straight-forwardly, more general cases need more effort. Note we demonstrate exactly this encoding in Chapter 9.

The original proposal of the Data Type Reduction abstraction we investigate in Chapter 6, has actually been discussed in the context of array programs [127] and proposed for parameterised system.

It is one motivation of our presentation that we felt that this very technical presentation hinders a clear view on the effect of the abstraction. A presentation in a setting of topologies seems more natural in order to understand, among others, the strengths and

[5] unfortunately we're missing the exact reference



(a) Distinct cases.

(b) No identities.

Figure 3.12.: **Abstracting from Identities.**

weaknesses of the abstraction, ways to improvements, and side-conditions to be kept up when considering improvements.

3.7.4. Shapes

A wide and diverse area of techniques which is naturally interested in systems with dynamically changing topologies falls under the name *shape analysis*. The typical motivation is the analysis of pointer programs operating on lists, trees, or other dynamic data-structures on the heap. The analysis obtains its name by its abstract domain, finite representations of *shapes* of data-structures, e.g. [60, 155]. Yet viewing the highway in the Car Platooning application as the heap, the class of dynamic topology systems is just that: a data-structure of cars.

In the following, we have to partly anticipate Chapter 5 while referring to some extent to the particular abstractions in order to be able to discuss the motivation behind the computational models employed there.

In the named approaches, abstract graphs closely resemble our topologies – and even more closely our abstract topologies from Chapter 5. Technically, many approaches employ logical structures to represent the *shape* of the heap at certain program points, in particular at the program points corresponding to completion of a heap manipulating function.

One main difference to our approach is that these analyses already assume a rather abstract world, namely where one can disregard particular identities. In terms of our definitions, their topologies aren't over a common set of identities *Id*, but each topology only had a set of own nodes U. Consequently, it is not intended to refer to particular identities, for example, one is not interested in properties like the following.

> *"The car, which asks the car in front for a merge manoeuvre now will finally be the follower of the front car."*

But rather, whether a list reversal program has a memory leak or yields a structure which is not a list.

Other effects are that issues with link navigation are neglected because with binary relations, there is typically no notion of navigation. It is in the best case mapped to quantification, for which it is not clear whether it raises all intricacies we address. And it is not suitable to represent multi-sets. That is, the issue of an unbounded number of links, possibly to the same individual, is lost in a logical structure encoding, while it is present in our model.

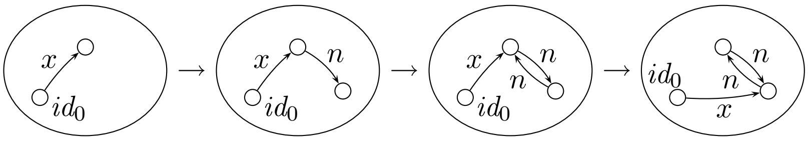


Figure 3.13.: **Abstract Heaps.** Fragment of a run of a TTS model for an abstract heap. The node denoted by global variable x is used to form a doubly linked list of two nodes, which is reversed in the rightmost step by changing the destination of x.

For example, we want to distinguish topologies G_1 and G_2 in Figure 3.12(a) because our approach is able to treat this case. Without identities, topologies G_1 and G_2 are identical so the view is rather similar to Figure 3.12(b).

This assumption is reasonable as, for example, in a pointer program inserting data into a linked list and to this and allocating and deallocating list-nodes, there is no natural interest in *which* particular list-node is allocated or deallocated, i.e. which memory address it has.

The brief discussion of considered properties leads to another main difference. Namely that many of the static analyses belonging to the class discussed above don't trace evolution over time, with the exception of [190, 191]. Then they are interested only in the information which topologies are possible over the whole lifetime of a system or runtime of a program, but they are often not interested in what sequences of topologies are possible.

On the other hand, if possible shapes are annotated to a control flow graph, for instance by a static analysis, the control flow graph can be viewed as a TTS and be queried with temporal properties like the ones from Chapter 4. For instance, whether the node newly added during Figure 3.13 is always finally deallocated This would allow analyses similar to the approach of Huuck et al. [67], which considers temporal properties of control flow graphs.

Figure 3.13 illustrates how a sequence of operations on a doubly-linked list can be modelled by a TTS. The pictures naturally closely resemble the graphical presentation of logical structures in, e.g. [60, 155], yet have a different interpretation here.

Assume a global program variable x pointing to a single heap node, and assume an operation appends one node and establishes the doubly-linked list property, and then changes x to the new node, thus effectively reversing the list. In the leftmost topology in Figure 3.13, the single list node is shown without identity, as the analysis is not interested in distinguishing heap nodes by actual identity. As TTS as such don't support global variables natively, the encoding uses one designated node with identity id_0 as the source of all links corresponding to global variables. In comparison, the representations of [155] are more natural and concise on this aspect, although (finitely many) global variables could easily be added to TTS. In the second topology in Figure 3.13, a new node appeared and has already been connected via the n link to the existing node, the third topology adds a link back to obtain a doubly-linked list, and the fourth topology shows the result of what would be something like an assignment `x := x->n` in pointer programming languages. Note that there are no evolution relations in Figure 3.13, as the named analyses are often also not interested into who evolves into whom; whether the four operations break the list property, or, for instance, render nodes unreachable by updating the wrong link becomes visible even if identities are not traced.

Most closely related to, and actually inspiring, our computational model are the following approaches. [190, 191] aim at checking a first-order extension of temporal logic similar to the one introduced in Chapter 4. The semantics of the logic is given for traces, that is, sequences of disjoint topologies, where appearance, disappearance, and evolution is explicitly traced by an annotation of consecutive pairs of topologies. Behaviour is given in form of a next-step function, which iteratively defines a set of traces given a set of initial states. The analysis is based on a fixed point iteration applying the next-step function of the considered model to a minimal and maximal topology. The outcome is a set of abstract traces, i.e. finite sequences of abstract topologies with appearance, disappearance, and evolution traced as precise as possible in the abstraction (cf. discussion in Chapter 5), on which the temporal expressions can be evaluated. The most obvious commonality between this and our model is the annotation of evolution, the most obvious difference that they don't trace identities.

The aim of [7, 12] is to determine a finite, abstract characterisation of all possibly reachable topologies of a given TTS (in our terms). They assume the system behaviour to be given in terms of a graph transformation system (GTS) with application conditions, there called partner constraints, in contrast to our high-level language from Chapter 9. The natural behavioural semantics of a GTS is a transition system where the states are labelled graphs, thus topologies in our terms. The main difference to our approach is that identities are not traced and dangling links are disregarded by assuming a vanish semantics for the modelled system.

A symbolic variant of shape analysis for Java programs is presented in [148, 181]. Instead of a rather explicit representation of system states in form of boolean structures [155], a predicative representation is chosen. In this framework, some of the properties from [7] can be established on a Java encoding of the Car Platooning application [147].

3.7.5. π Calculus

The π [137] calculus is a widely studied algebraic model of mobility and communication. Most importantly, π extends the CCS process algebra [136] by means to pass names between processes, that is, means to establish links in our terminology of ETTS. For example, the merge procedure of the Car Platooning application has successfully been modelled in the π calculus [131].

A major difference is that there is no natural notion of an identity in the structural operational term rewriting semantics. A notion of *structural equivalence* considers large classes of π terms equivalent, for example permutations of processes along commutative operators like parallel composition. Then we cannot refer to individual cars as easily as in our explicit setting, in particular from the specification logic side.

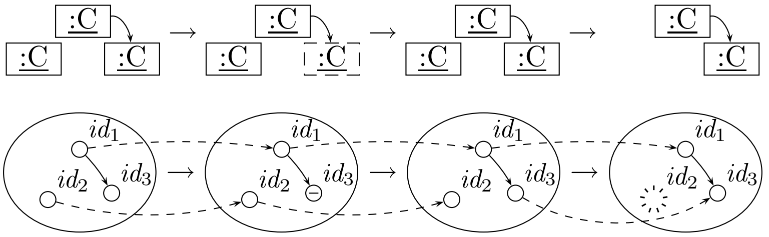


Figure 3.14.: **UML.** The first row shows a fragment of a run of a UML system as sequence of object diagrams. If the right object is destroyed, there is in general no overall garbage collection which removes all links, thus the link becomes dangling and we have to keep the object in a non-alive state (dashed outline). The space may be re-used for creation of a new object. Alternatively, objects may disappear completely, as demonstrated by the left object in the last step.
The second row shows a corresponding fragment of a run of a TTS/id model. In the second topology, individual 3 is there but no longer alive, the link from 1 is dangling. In the step to the third topology, identity 3 is re-used for a newly created individual. In the last step, individual 2 really disappears.

In addition, the π calculus considers synchronous, blocking communication while the examples in Chapter 10 are of asynchronous nature. The π model can be considered to be the most basic form of communication and can be employed to model asynchronous, non-blocking communication, but this again easily becomes hardly treatable.

The π calculus is able to encode TTS models but the encoding in pure π easily becomes awkward when data aspects should be considered. Applied π [156], which is often employed to obtain clearer encodings in such cases, is already similar to the DCS language discussed in Chapter 10.

There are also efforts in the π community aiming at formal verification, in particular model-checking of mobile communicating systems [133]. So there is a common goal despite the differences in computational model and description language which is approached from two different directions. The approach of the π community is to start from an established model and to develop specification and verification techniques for it, while our approach is to start from the established (finite state) specification and verification theory and to add coverage of typical π features like mobility and communication.

3.7.6. UML Semantics and UML Verification

The growing popularity of the employment of the Unified Modelling Language (UML) [138, 141, 140] for model driven development of safety or mission critical system, gave rise to numerous attempts to formally capture the precise the semantics of UML.

That these attempts are naturally related to our work becomes obvious in Chapters 9 and 10 where we discuss how ETTS are useful to encode a significant fragment of UML in order to obtain formal verification for UML, which has actually been one of the original starting points of this work. Anticipating Chapter 9, Figure 3.14 outlines the relation between a run of an instance of a UML model and a run of an ETTS.

The named attempts are not directly usable because they either take an even more abstract view (like [82, 86, 151, 152]) or try to capture too many aspects of UML, and thus remain too concrete, too close to full-fledged programming languages, or they don't employ transition systems.

The early approaches to UML model-checking mainly aimed at employing available finite-state model-checking technology. The difference to our approach is that they didn't treat (and thus needn't model) unbounded creation and destruction of objects but in the most developed cases demanded finite upper bound, or a set stage of finitely many objects, or even demonstrated their procedures only on pairs of state-charts, disregarding object creation and destruction at all. To achieve these results, there is basically no need for an elaborate formal semantics of UML, in contrast we can characterise these approaches as giving a formal UML semantics in terms of a (finite) fragment of the informal description of UML.

Our computational model, in contrast, could serve as a domain for a full UML semantics, as sketched in Chapters 9 and 10, and we discuss how to obtain an abstraction to a finite-state setting.

The computational model underlying our work et al. [159], was oriented on the array-based semantics of [45, 46] which prefers to exclude re-use of identities. It was the first to present the life-cycle of objects integrated with the run-to-completion semantics of UML in form of a state chart. That is, initially an object is the semantical state of being not yet created, after construction is completed, it is idle or in a run-to-completion step of its UML state-machine, and from both it may be destroyed, that is, complete a destructor, and then be dead.

Xie, Browne, and others [187, 185, 188, 186, 160, 184, 189] base their tool ObjectCheck on the automata of the COSPAN model-checker [106].

Shen and others [37, 38, 161, 162, 163] don't consider object creation and destruction for their ASM[6]-based tool VeriUML.

David, Möller, and Yi [52, 51] are mainly interested in the aspect of timed state-machines in UML, not in topologies in their Uppaal[7]-based tool UML RT/Uppaal.

Knapp, Merz et al. [157, 100, 101] discuss the verification of collaborations by their tools Hugo and Hugo/RT, that is, temporal behaviour in given fixed topologies, and are hence not concerned with the dynamic extension of the topology.

Eshuis and Wieringa [65], have an LTS and array-based semantics of UML, which is parameterised in many variation points like event-queue vs. event-set, i.e. with and without preservation of order. It is basically extension of the Statemate semantics of [44] to UML.

The UMLAUT/CADP approach of [74, 73, 93] uses UML object diagrams instead of topologies, which can be encoded into our topologies with the general notion of links.

[6] Abstract State Machines [75]

[7] Uppaal timed-automata model-checker [13, 109, 110]

Lilius and Porres [118, 117] basically consider state-machines in isolation for their tool vUML, which is based on the SPIN model-checker [83].

Similarly, Latella, Majzik, and Massink [111, 112] are basically interested in single state-machines and discuss an operational semantics which exhibits the same drawbacks as the π-calculus as discussed in Section 3.7.5.

Varro, Pataricza, and others [143, 172, 41] aim to support semantics-preserving transformations of UML models and to this end consider a graph transformation system-based view on the behaviour.

A similar model is employed by the USE tool of Richters and Gogolla [153], which aims at checking OCL constraints against class diagrams as such, thus is not interested in temporal properties and hence doesn't need an elaborate execution semantics. Both are, by their underlying formalism, related to [7, 12], thus the comments given there apply here as well.

3.7.7. Other Models

In the domain of hybrid systems, there are emerging efforts to provide means for modelling and analysis of what they call *reconfigurable systems*.

Some consider only finite technical systems where single components can be replaced during runtime, thus the system remains of finite extension in number of components but has to treat the — typically noncontinuous — reconfiguration operations. Others are sufficiently general to model the Car Platooning application in a hybrid setting. Most prominent among them is the language SHIFT [53], which basically obtains a semantics by a programming and simulation environment, the formal semantics remains on a sketchy level. The newer R-Charon [103] is a straight-forward dynamic topology extension of the Charon language [1], yet with less emphasis on the graph form than ours. Yet a straight-forward extension of ETTS with hybrid aspects would resemble R-Charon.

Semantically, both employ a similar, natural approach of evolving sets of individuals. In SHIFT, each world is finite; there's no explicit discussion of evolution or identities. In R-Charon, components from a fixed set can be added and removed during run-time, links are removed along ("vanish semantics"). Identities are not explicitly discussed in [103], the presentation may imply that identities are implicit, by name.

At their current states, they are at the stage of modelling, not yet property specification and all the issues concerning pre-mature disappearance of individuals.

3.7.8. Summary

For convenience, Table 3.2 summarises relations between ETTS and the computational models discussed in Section 3.7.1 and Section 3.7.3 in terms of expressive power. Note that some of the relations are established and some only conjectured; in this work, we won't answer the corresponding interesting and relevant questions but consider them to lie outside of the scope of this work.

infinite Parameterised Systems	$\subseteq$	infinite-Array Programs	$\subseteq$	ETTS	$\supsetneq$	TTS
$\rotatebox{90}{$\subsetneq$}$		$\rotatebox{90}{$\subsetneq$}$				
Parameterised Systems	$\subseteq$	Array Programs				

Table 3.2.: **Expressiveness of Computational models** in informal comparison.

Shape abstractions, in comparison, are employed under different assumptions and with different aims, so they typically abstract right-away from identities (cf. Section 3.7.4).

The π calculus can be used to model systems similar ETTS, but employs much more elementary means which makes it difficult to identify properties that are visible in higher-level description language but become invisible after an encoding in π. As discussed in Section 3.7.5 above, there are although common goals shared between this work and efforts aiming at formal verification, in particular model-checking, of mobile communicating systems given in the π calculus.

The existing approaches to UML verification merely choose finite-state verification tools and employ the semantic domains of the chosen tool to give ad hoc semantics to (finite) fragments of UML, thus typically require finite upper bounds on the number of objects simultaneously alive (cf. Section 3.7.6).

4. Property Specification Logic: EvoCTL*

Similarly to ETTS of Chapter 3 being a conservative generalisation of classical labelled transition systems, we'll introduce a formalism for requirements specifications of ETTS systems as a first-order extension of classical temporal logic [34].

This chapter is also structured similarly to Chapter 3. In Section 4.1, we begin with an extensive motivation and assessment of the features to expect from an adequate property specification logic for ETTS and lay out how we're approaching these issues.

We'll find the most intricate points to be the quantification over individuals *over time*, that is, tracing the evolution of individuals along computation paths to evaluate temporal properties for single individuals and then the treatment of pre-mature disappearance. That is, the semantics of formulae in case individuals bound to logical variables have already disappeared when they're needed to evaluate the formula.

For self-containedness, we introduce preliminaries like notions of signatures and structures in Section 4.2, together with a concept of compatibility to ETTS. Sections 4.3 and 4.4 first give the formal syntax and semantics of terms and formulae, and then turn to discussions of properties like monotonicity and definitiveness.

In Section 4.5, we discuss in particular which EvoCTL* formulae have equivalent prenex normal forms because the abstraction of Chapter 6 requires formulae in this normal form, so we give extended estimates for the range of properties Chapter 6 applies to.

We conclude the chapter with Section 4.6 where we discuss a number of different other approaches to treat topologies with dynamic appearance and disappearance of individuals and variable interconnection.

4.1. Requirements Specification for ETTS: The Intuition

In the following, we've got to assume some working knowledge in temporal logic in order to keep the discussion focused. The reader not familiar with temporal logic at all, is referred to a textbook like [34]. The reader with basic knowledge in this topic may wish to firstly skim through Sections 4.3 and 4.4, which give the formal syntax and semantics of terms and formulae, to recall the main operators and their semantics and then get back to Section 4.1 for an explanation of the differences between the ETTS case and classical temporal logic for Kripke structures.

4.1.1. State vs. Temporal Properties

Recall from the well understood case of requirements specification for Kripke structures by temporal logic that we can basically distinguish two aspects: Firstly, properties of a certain state, which boils down to logical expressions with atomic propositions as atoms. Secondly, behaviour of state-properties over time, for instance, whether certain state-properties hold in all next state, don't occur at all, hold globally, or until others, corresponding to the well-known modal operators of CTL*.

For ETTS, the first aspect of properties of states corresponds to properties of topologies which is again twofold:

- firstly, each individual in a state's topology has a local state on which requirements should be stated, for example, whether an individual in the car platooning example is a leader or a follower, or at which speed it is cruising, and
- secondly, individuals in a state's topology are interconnected, that is, it should be possible to express requirements like, for example, that a follower has a leader,

and of course combinations of both, for instance, that the leader of a follower is in a certain local state.

The second aspect, namely the behaviour of state-properties over time, is also twofold for ETTS:

- firstly, there is obviously a direct correspondence to Kripke structures in the sense that it should be possible to state requirements on the temporal evolution of the just named topology properties, for example, that there is no state where two cars mutually consider each other to be the leader,
- the second, more intricate thing with ETTS is the evolution of single individuals over time. That is, it should be possible to state that

  "each car, which is now a follower, remains in that role until a platoon split manoeuvre has been completed successfully". (4.1)

Table 4.1 summarises this first informal discussion to point out what we've got to expect from EvoCTL* as defined from Section 4.2 on.

4.1.2. Tracing Individuals over Time

The intricacy of properties referring to the evolution of single individuals over time, in contrast to the whole topology, is severely affected by the possible disappearance of individuals.

Firstly, for the evaluation of a requirement like (4.1) above, we have to take into account that the individual may (pre-maturely) disappear at any time due to an error in the platooning protocol.

We propose to employ three-valued logic with a third value "undefined" in addition to "true" and "false", in contrast to most other approaches which resort to "false" in

state properties			behaviour over time		
Kripke	ETTS		Kripke	ETTS	
	individuals' local state	inter-connection		topology	individual
propositional logic over atomic propositions AP	predicate logic over boolean terms with function σ accessing individuals' local state	link navigation accessing individuals' λ-link	CTL* with path quantifiers E, A, modalities X, F, G, U, over state properties	similar to Kripke structures	scope of quantified variables additionally covers multiple topologies

Table 4.1.: **Property specification** for ETTS in contrast to plain Kripke structures.

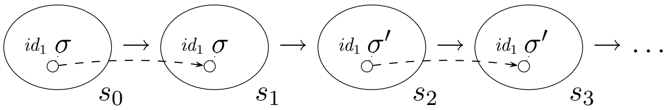


Figure 4.1.: **Identity vs. Instance** evolution.

case of disappearance. In the latter case, the evaluation of terms is called *biased* towards "false", as this value is ab-used to encode the case of pre-mature disappearance.

The reason for discouraging the use of a biased evaluation is basically that pre-mature disappearance may be shadowed when not sticking to negative normal forms. And that, if there are procedures to verify such properties, a "false" is strictly speaking misleading because there actually needn't be a counter-example where an individual violates the desired property.

Secondly, given a computation path, a requirement like (4.1) above, and an individual to establish the requirement for, we have to define which individual to consider in the *next* state. That is, given a computation path $\pi = s_0, s_1, s_2 \ldots$ of an ETTS M and an individual $u \in U(s_0)$, which individual is to be inspected in s_1? The answer is not as clear as it may seem at first sight, as we'll see in the following.

As an example consider the computation path in Figure 4.1 and assume that we want to evaluate the requirement

> *"for all x, if x is alive now, then the local state of x is σ until it disappears".* (4.2)

As there is only one individual in the topology of s_0, we identify x with the individual with identity id_1 and find the local state to be σ, thus the requirement possibly satisfied. All other identities trivially satisfy the implication because they're not alive.

On the transition to s_1, the individual with identity id_1 evolves, thus x denotes this individual and still the local state is σ; in addition, the individual disappears, so the overall requirement is satisfied here.

If id_1 is immediately re-used, as in s_2, we basically have two options to interpret the requirement. The first option is an *identity-based interpretation*, that is, considering x to bind to identities. Then evaluation of the quantified property starts over as the identity denoted by x is alive; according to the requirement, the individual should have local state σ, which is not the case in s_2, thus the overall property is not satisfied for the computation path shown in Figure 4.1.

The second option is an *evolution chain-based* interpretation, that is, it rather disregards identities and considers the quantification to be over evolution chains (cf. Chapter 3). Then in order to evaluate the above requirement in the given computation path starting at s_0, x binds to any evolution chain active in s_0. In the example, there is only one such evolution chain, the one of id_1. The evolution chain starting in s_2 is not considered because the property binds logical variables only in the first state of the given computation path, here in s_0. Thus in this interpretation, the requirement is satisfied by the computation path shown in Figure 4.1.

We now continue with a more elaborate discussion of both interpretations stemming from the literature (cf. Section 4.6) and then with an intuitive comparison.

To enable a more formal comparison in Section 4.4.7, we'll define two kinds of logical variables, or, as an alternative view, two kinds of quantification and treat them as "first-class citizens" throughout this chapter.

The Identity-based Interpretation

In case of a computational model with identities as ours, the most natural approach is quantification over identities. That is, logical variables are of the identity type and an assignment maps a logical variable x to a fixed identity.

This assignment in particular doesn't change over time; for example, consider a requirement of the form

$$\varphi := \forall\, x : T \,.\, \varphi_0(x), \tag{4.3}$$

i.e. outermost quantified (or in prenex normal form, cf. Section 4.5), where $\varphi_0(x)$ is a temporal logic formula with x of type T as free variable.

When evaluating φ for a given computation path $\pi = s_0, s_1, s_2 \ldots$, then x is bound to an identity from *Id*, independent from the topology of s_0, and this binding remains throughout π.

Note that formula (4.3) is of limited use if $\varphi_0(x)$ doesn't take into account that the identity to which x is bound needn't be in use.

For example, the inner formula

$$\varphi_0 := (\mathsf{G}(\sigma(x) \neq 0)) \tag{4.4}$$

("individual x's local state never is $0 \in \Sigma$")

becomes immediately indefinite (cf. Sections 4.3.4 and 4.4.6) in a topology where there is no individual with the identity bound to x, i.e. (by definition) where the partial node labelling function σ is not defined for this identity (cf. Def. 3.2.1).

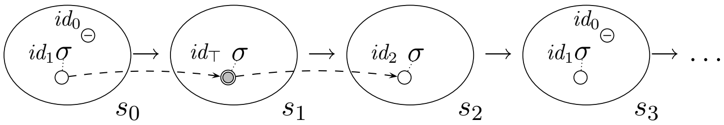


Figure 4.2.: **Evolution chain** covering different identities.

For instance, binding x to id_2 in the path shown in Figure 4.1 renders the formula indefinite; the formula doesn't definitively hold and isn't definitely violated for id_2, there just is no individual with identity id_2 to evaluate it for.

In contrast, with the inner formula

$$\varphi_0 := \mathsf{G}(\circledcirc\, x \rightarrow (\sigma(x) \neq 0)) \tag{4.5}$$

("x being alive implies that its local state isn't $0 \in \Sigma$")

(4.3) may hold in state s_0 of Figure 4.1 because the local state is only considered if there *is* an individual with the identity denoted by x, and this local state may be equal to 0.

Note that individuals in formula (4.3) are *effectively* anonymous, because in the formula we don't refer to particular individuals by identity, but the property applies to each individual independent from the identity.

The Evolution Chain-based Interpretation

The second approach is to follow evolution chains, disregarding identities. This is in particular natural when the computational model doesn't provide identities, because then individuals actually *are* anonymous, so there is not even a choice to refer to them by identity.

The idea than is to follow chains of the evolution relation, disregarding the identity. For example, consider the computation path shown in Figure 4.2 where the evolution relation has a non-standard form by merging id_1 into the summary node $id_\top$ and, possibly having lost the information which node merged into, evolves into id_2.

To indicate that a formula like (4.3) should be evaluated in this sense, we'll use boldface letters for logical variables, that is, the formula (4.3) becomes

$$\varphi := \forall \boldsymbol{x} : T \,.\, \varphi_0(\boldsymbol{x}). \tag{4.6}$$

Evaluating (4.6) for the computation path in Figure 4.2 with (4.4) as inner formula necessarily binds $\boldsymbol{x}$ to id_1, as this is the only alive individual in the topology of state s_0.

In state s_1, $\boldsymbol{x}$ denotes the individual with identity $id_\top$ as this is the individual id_1 evolves into according to the evolution relation. In state s_2, $\boldsymbol{x}$ denotes id_2, but afterwards the individual doesn't evolve so there is no binding for $\boldsymbol{x}$ and the overall formula turns indefinite for the computation path in Figure 4.2.

Note that s_3 is not considered at all. The reason is that the modal operator "G" (read: globally) appears under the quantifier, that is, the logical variable $\boldsymbol{x}$ is not bound anew in any of the states, only once in state s_0, the first state of the computation path for which the formula is considered.

In this interpretation, invariants like the one in (4.6) can only be satisfied if the affected individuals don't disappear at all, otherwise the outcome is either a violation or an indefinite result because of pre-mature disappearance.

Comparing the Two Interpretations

Anticipating the more elaborate discussion of the relation between these two interpretations in Section 4.4.7, we note that the evolution chain-based interpretation seems natural, but makes a couple of implicit assumptions. For example, from example (4.6) with different inner formulae we can tell that invariants for individuals have to be used with care as they are only *global* invariants if the individual under consideration doesn't disappear. On the other hand, this interpretation never misses the disappearance (and re-use) of an identity as the assignment of logical variables is only considered as long as the traced individual doesn't disappear and turns undefined afterwards.

In contrast, the identity-based interpretation may not notice a disappearance and re-use. For example, the computational path shown in Figure 4.1 satisfies the formula

$$\forall\, x : T\,.\, \mathsf{X}(\mathsf{X}(\circledcirc\, x)) \tag{4.7}$$

("x will be alive in two steps")

which on first sight seems to say that id_1 remains alive for the next two steps, but on second sight, all it requires is that the individual with identity id_1 has a certain local state, it isn't interested at all in the individual's fate in the meantime.

The first meaning would require a formula like the following

$$\forall\, x : T\,.\, \circledcirc\, x \to (\neg \otimes x \land \mathsf{X}(\neg \otimes x) \land \mathsf{X}(\mathsf{X}(\circledcirc\, x))). \tag{4.8}$$

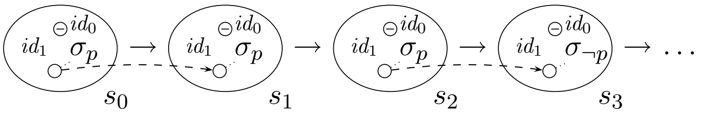
("x will be alive in two steps, and won't disappear before")

When defining the semantics in Section 4.4.2, we'll see that the main difference is that in the identity-based interpretation, a logical variable is bound to a single identity once (and for all), while in the evolution chain-based interpretation it may be bound to a different identity in each state of the considered trace.[1]

4.1.3. The Plan

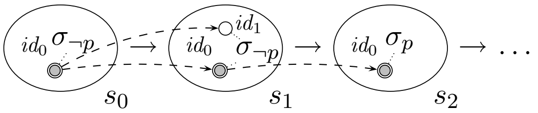
The fundamental difference between plain CTL* and EvoCTL* is that properties are finally evaluated for individuals, which are in most cases denoted by logical variables. This gives rise to three issues. Firstly, how to evaluate expressions in the topology of a given state? Secondly, and far less obvious, how does evolution of individuals affect the meaning of formulae? That is, given a binding of logical variables in one state of a computation path, how do these variables bind in the successor state? And

[1] As one of the referees pointed out, one can think of even more modes for binding logical variables (or: more kinds of logical variables), which to the best of our knowledge didn't appear in the literature yet. For instance, to initially bind to an identity, then await the creation of an individual with this identity, and trace only the first life-cycle, i.e. the first evolution chain.



		s_0	s_1	s_2	s_3
$\forall\, x : T \,.\, \mathsf{G}\, p(x.\sigma)$:	$\theta_1(x)$:	id_0	id_0	id_0	id_0
	$s, \theta_1 \models p(x.\sigma)$:	$1/2$	$1/2$	$1/2$	$1/2$
	$\theta_2(x)$:	id_1	id_1	id_1	id_1
	$s, \theta_2 \models p(x.\sigma)$:	1	1	1	0
$\forall\, \boldsymbol{x} : T \,.\, \mathsf{G}\, p(\boldsymbol{x}.\sigma)$:	$\theta(\boldsymbol{x})$:	id_1, id_1	id_1	n/d	n/d
	$s, \theta \models p(\boldsymbol{x}.\sigma)$:	1	1	n/a	n/a

(a) Unique evolution.



		s_0	s_1	s_2
$\exists\, x : T \,.\, \mathsf{F}\, p(x.\sigma)$:	$\theta_1(x)$:	id_0	id_0	id_0
	$s, \theta \models p(x.\sigma)$:	0	0	1
	$\theta_2(x)$:	id_1	id_1	id_1
	$s, \theta \models p(x.\sigma)$:	$1/2$	0	$1/2$
$\exists\, \boldsymbol{x} : T \,.\, \mathsf{F}\, p(\boldsymbol{x}.\sigma)$:	$\theta_1(\boldsymbol{x})$:	id_0, id_1	id_1	n/d
	$s, \theta \models p(\boldsymbol{x}.\sigma)$:	0	0	n/a
	$\theta_2(\boldsymbol{x})$:	id_0, id_0, id_0	id_0, id_0	id_0
	$s, \theta \models p(\boldsymbol{x}.\sigma)$:	0	0	1

(b) Multiple different chains of evolution.

Figure 4.3.: **Identity vs. Evolution Chain** quantification.

thirdly, assuming we know how to follow the evolution of individuals, how is pre-mature disappearance to be treated? That is, if an individual disappears before we are able to give a definite answer whether a formula holds or not, which truth value shall we evaluate the formula to?

The first issue is treated straightforwardly by relating logical variables to individuals in a topology and accessing their local state and link configuration. In case we need to access, for instance, the local state of an identity which is currently not in use, that is, which currently doesn't have a local state as it is not in the domain of σ, we'll revert to three-valued logic, which may cause whole formulae to evaluate to $1/2$, the indefinite value.

The second issue is more intricate. Consider the computation path prefix shown in Figure 4.3(a) and assume that topologies are over the finite set of identities $Id =$

$\{id_0, id_1\}$. In Figure 4.3(a), identity id_0 is unused while there is an individual with identity id_1 in all four states, but it is re-used once: it is disappearing in state s_1 and newly created in state s_3, as indicated by the dashed arrows for the evolution relation. Now if we want to assign a semantics to the EvoCTL* formula

$$\forall\, x : T \,.\, \mathsf{G}\, p(x.\sigma), \tag{4.9}$$

we're supposed to bind the logical variable x.

As x is of type T (for now considering only a single type), we'll bind it to an identity, that is, there are two possible assignments of x, namely θ_1 and θ_2 as shown in Figure 4.3(a) (assignments are formally introduced in Def. 4.2.6). Now evaluating the predicate p in each state for each of the assignments yields $1/2$ for θ_1 and a definite value for θ_2. The whole formula evaluates to 0 as the predicate is definitely not satisfied by the topology of state s_3 under assignment θ_2.

Note that the assignment itself is not affected by the temporary disappearance of individual id_1 in state s_1.

If we were only interested in the fate of the particular individuals alive in s_0, we'd to change the specification to

$$\forall\, x : T \,.\, \circledcirc\, x \rightarrow ((\neg \otimes x \wedge p(x.\sigma))\; \mathsf{W}\; (\otimes\, x \wedge p(x.\sigma))). \tag{4.10}$$

As (4.10) explicitly refers to the life-cycle of the individual bound to x by the operators $\circledcirc$ and $\otimes$, it is satisfied by both assignments in Figure 4.3(a). For θ_1, it is trivially satisfied as id_0 is not alive, and for θ_2 it is satisfied because predicate p holds during the whole life-cycle.

Alternatively, we could use a logical variable of the second kind, i.e. of type $\boldsymbol{T}$. Exchanging the type of the variable in (4.9) yields

$$\forall\, \boldsymbol{x} : T \,.\, \mathsf{G}\, p(\boldsymbol{x}.\sigma), \tag{4.11}$$

as $\boldsymbol{x}$ is of kind $\boldsymbol{T}$, we'll bind it to an evolution chain instead of simply an identity, in case of universal quantification to all possible evolution chains of all alive individuals in a given state.

For the computation path in Figure 4.3(a), there is only a single evolution chain as only id_1 is alive and evolves linearly. Evaluating, for instance, the boolean term $p(\boldsymbol{x}.\sigma)$ with assignment θ_1 from Figure 4.3(a) will be explained by evaluation for the first element of the evolution chain $\theta_1(\boldsymbol{x})$, in this case id_1. By considering evolution chains, this interpretation is implicitly sensitive for disappearance. Formula (4.11) holds in Figure 4.3(a) because id_1 satisfies the boolean term during the whole life-cycle from s_0 to s_1.

Figure 4.3(b) illustrates existential quantification. In case of logical variables of type T, quantification may bind to any of the identities from Id, independent from the question whether there is currently an alive individual with the bound identity or not. Hence there are the two possible assignments θ_1 and θ_2 shown in Figure 4.3(b) for the formula

$$\exists\, x : T \,.\, \mathsf{F}\, p(x.\sigma), \tag{4.12}$$

the whole formula holds because the inner boolean term finally holds for id_0. Thereby, we can express, for example, that behind a given state there will finally be a second individual appearing. In case of logical variables of type $\boldsymbol{T}$, that is, for the formula

$$\exists \boldsymbol{x} : T \,.\, \mathsf{F}\, p(\boldsymbol{x}.\sigma), \tag{4.13}$$

quantification may bind to any of the evolution chains starting in the state where the formula is evaluated. Thus there are two possibilities for the computation path shown in Figure 4.3(b). Firstly, evolution can follow the materialisation from id_1 out of id_0 and secondly, it can follow the summary node id_0. The whole formula (4.13) also holds because the latter evolution chain leads to a situation where the inner boolean term is finally satisfied.

This discussion of how we'll consider evolution in the semantics of EvoCTL* already indicated how we'll treat the third issue raised above, namely pre-mature disappearance. In case of logical variables of kind T it's treated not at all. If a requirement should be sensitive for disappearance, it has to be explicitly stated within the EvoCTL* formula by employing the life-cycle operators $\odot$, $\circledcirc$, and $\otimes$. In case of logical variables of kind $\boldsymbol{T}$ in contrast, it's treated implicitly. The semantics of a temporal operator like 'X' will be defined such that evaluation yields the indefinite value $1/2$ in case of pre-mature disappearance and definite values only if the formula can be evaluated for an alive individual.

Note that logical variables in [190, 191] are implicitly of the type $\boldsymbol{T}$ as they right-away abstract from identities (cf. Section 3.7). They provide the semantics by defining how an assignment evolves over time. We obtain the same semantics by binding these variables to evolution chains. When temporal logic is employed for classical parameterised systems, identities are typically considered so their semantics is more akin to our logical variables of type T. We'll discuss the relation to other approaches and between other approaches in more detail in Section 4.6.

4.2. Signature and Structure

4.2.1. Signature

Note that signatures as introduced in the following are not minimal as they, for example, provide types and a collection of function symbols which are strictly speaking not necessary for a discussion of requirements specification for dynamic topology systems.

The reason for having these features is that they allow us to develop the abstraction theory of Chapters 5 and 6 such that they directly apply to higher-level languages like UML or DCS as discussed in Chapters 9 and 10.

Definition 4.2.1 (Signature)**.** *A* signature *is a quadruple*

$$\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda) \tag{4.14}$$

of finite sets of types $\mathcal{T}$*, typed* logical variables $\mathcal{V} = \mathcal{V}^T \,\dot\cup\, \mathcal{V}^{\boldsymbol{T}}$*, typed* function symbols $\mathcal{F}$*, and* link names Λ*.*

The set of types comprises at least the following basic types*: the booleans* $\mathbb{B}$*, the type of local states* S*, the type of link sets* L*, the types of identities* $T_1, \ldots, T_n$*, and the types of evolution chains* $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_n$*,* $n \in \mathbb{N}^+$*, i.e.* $\mathcal{T} \supseteq \{\mathbb{B}, S, L, T_1, \ldots, T_n, \boldsymbol{T}_1, \ldots, \boldsymbol{T}_n\}$*. Non-basic types are also called* additional types*.*

Logical variables $x, y, \ldots \in \mathcal{V}^T$ *are of identity type, i.e. of* T_1*, ..., or* T_n*, and called* identity variables *as they denote identities. Logical variables* $\boldsymbol{x}, \boldsymbol{y}, \ldots \in \mathcal{V}^{\boldsymbol{T}}$ *are of evolution chain type, i.e.* $\boldsymbol{T}_1$*, ..., or* $\boldsymbol{T}_n$*, and called* destiny variables *as they denote evolution chains. Identity and evolution chain types* T_i *and* $\boldsymbol{T}_i$ *are said to* correspond*. If the type is not relevant in a particular context, we may use* $v, w, \dots \in \mathcal{V}$ *to denote logical variables of any of the two classes of types.*

A function symbol $f : \tau_1 \times \cdots \times \tau_k \to \tau$ *from* $\mathcal{F}$ *is said to have* arity k*,* argument type $\tau_1 \times \cdots \times \tau_k$*, and* (result) type τ*. Argument and result types may not be* $\boldsymbol{T}$*. Function symbols of type* $\mathbb{B}$ *are called* predicate symbols*. We use* $\mathcal{P}$ *and* $\mathcal{P}^k$ *to denote the subsets of* $\mathcal{F}$ *consisting of predicates and predicates of arity* k*.*

The components of $\mathcal{S}$ *are referred to as* $\mathcal{T}(\mathcal{S})$*,* $\mathcal{V}(\mathcal{S})$*,* $\mathcal{V}^T(\mathcal{S})$*,* $\mathcal{V}^{\boldsymbol{T}}(\mathcal{S})$*,* $\mathcal{F}(\mathcal{S})$*,* $\mathcal{P}(\mathcal{S})$*, and* $\Lambda(\mathcal{S})$*, respectively.* ◇

Correspondence between types anticipates that the domain of $\boldsymbol{T}_i$ will be the set of sequences of identities from the domain of T_i. In the semantics section, we'll see that the reason for not admitting $\boldsymbol{T}$ as the type of function argument or result types is that evolution chains are supposed to be used interchangeably with identity types, which is achieved by considering only their first element.

The set of legal formulae over a signature will depend on the function symbols and link names. In order to later evaluate a formula for an ETTS we will have to require that they are (syntactically) compatible. Syntactical compatibility concerns link names from Λ, because link names are used within terms (cf. Def. 4.3.1) to access the interconnection in a topology on the level of the logic. It only concerns link names because the local state of individuals will be accessed by a fixed operator of the logic and then may be the argument of function symbols from $\mathcal{F}$. There we will need a *semantical* notion of compatibility, requiring that the interpretations of function symbols operate on Σ, the set of local states in the ETTS.

Definition 4.2.2 (Compatibility)**.** *A* (Σ, Λ)*-topology over identities Id is called* compatible *with a signature* $\mathcal{S}$*, or* $\mathcal{S}$*-compatible, if and only if* $\Lambda(\mathcal{S})$ *comprises only link names from* Λ*, i.e. if* $\Lambda(\mathcal{S}) \subseteq \Lambda$*.*

Analogously, a (Σ, Λ)*-topology transition system* M *is called* compatible *with* $\mathcal{S}$*, or* $\mathcal{S}$*-compatible, if and only if* $\Lambda(\mathcal{S}) \subseteq \Lambda$*.* ◇

Note that in the following, we always assume that signatures considered in combination with topologies or ETTS are compatible, even if we don't explicitly mention this fact.

We will need the following relation in the context of simulation relations in Chapter 5.

Definition 4.2.3 (Subset-relation for Signatures)**.** *We say that a signature* $\mathcal{S}_1$ *is a* subset *of the signature* $\mathcal{S}_2$*, denoted by* $\mathcal{S}_1 \subseteq \mathcal{S}_2$*, if and only if* $\mathcal{S}_2$ *comprises at least all* *the types, (typed) logical variables, (typed) function symbols, and link names of $\mathcal{S}_2$, that is, if $\mathcal{T}(\mathcal{S}_1) \subseteq \mathcal{T}(\mathcal{S}_2)$, $\mathcal{V}(\mathcal{S}_1) \subseteq \mathcal{V}(\mathcal{S}_2)$, $\mathcal{F}(\mathcal{S}_1) \subseteq \mathcal{F}(\mathcal{S}_2)$, and $\Lambda(\mathcal{S}_1) \subseteq \Lambda(\mathcal{S}_2)$.*

Function Symbols

In this section, we shall briefly discuss what we imagine will belong into $\mathcal{F}$ and what not.

There is a clear separation between local-state dependent things and a global interpretation. This separation, as well as the choice of functions symbols discussed below, is driven by the needs, or by the intention to support, the discussion of abstractions in Chapters 5 and 6, and the connections to higher-level languages in Chapters 9 and 10.

The main difference between function symbols and operators in the logic is that function symbols obtain a global interpretation while operators depend on the topology. For example, accessing the local state, link configuration, and aliveness are clearly operators because an identity may obtain a different local state in each topology. As is comparison for equal identity, which is a part of the topology (in form of the function eq_{Id}) which is made visible in the logic. Then the semantics definition immediately applies to abstract topologies as discussed in Chapter 5, where in particular eq_{Id} is subject to abstraction.

Comparing local states, in contrast, is clearly a function symbol because whether two local states are considered equal doesn't depend on the topology they're found in. The reason is that we don't want to have an interpretation of function symbols *per* ETTS state, but a global one, because we feel that this yields a clearer separation of issues in Chapter 5 on abstractions. An example where variable interpretations are employed are the approaches that represent topologies by logical structures, which naturally provide the interpretation of predicates.

Constants. In Chapter 9 we'll see that there's some need for 0-ary function symbols, or constants, of type T denoting identities, and of type L denoting the empty multi-set.

Thus natural examples for 0-ary function symbols in $\mathcal{F}$ are:

- constant numbers etc. of S or any additional type,
- $id_0 : T$ for a particular identity, or
- $\emptyset : L$ for the empty multi-set

In Chapter 7 on query reduction, we'll see that constants of T have to be treated with care as they may break an otherwise given symmetry in identities.

Comparison for Equality. An equality relation symbol

$$=: \tau \times \tau \to \mathbb{B}, \tag{4.15}$$

typically written infix, belongs to the signature with the single exception of the type T for reasons discussed above.

As a consequence, the predicate "=" for L, the type of links, may be defined independently from T. This is sensible, because we have a rather abstract notion of link navigation. In a more restricted setting, "=" on L could well be defined in terms of the one on T.

Local State and Attributes. In case of typed (or many-sorted) ETTS (cf. Def. 3.2.5, Section 3.2), local states are by definition further structured into n components, i.e.

$$\Sigma = \Sigma_1 \times \ldots \Sigma_n, n \in \mathbb{N}_0. \tag{4.16}$$

Then a natural set of function symbols are of the form

$$.n_i : T \to \tau_i, 1 \leq i \leq n, \tag{4.17}$$

where τ_i obtains domain Σ_i in order to individually access the components of a local state σ by name n_i.

Navigation Operations

Link navigation, that is, accessing individuals indirectly via a link name and a given source individual, is not a function symbol but an operator for similar reasons as comparison of identities.

By a link name, one obtains a (possibly empty) multi-set comprising the individuals accessible via this link name. This is the natural representation because we may have multiple links of the same name to the same other node, and it will be used as the canonical domain of L. In order to further use multi-sets, we'll consider only an exemplary set of functions, namely

- a constant for the empty link (see above),
- a comparison for equality of links (see above), and
- $* : L \to T$ to access the only element in single-links, that is, in multi-sets of size 1.

This setting allows to state properties like

$$(x \rightarrowtail \lambda \neq \emptyset) \to \sigma(*(x \rightarrowtail \lambda)) \neq 0$$

("if x has a λ-link to somebody, then the local state of this guy won't be 0").

In the following sections and chapters, we'll often restrict ourselves to single links and *implicitly* assume a cast from multi-sets of at most size one to a single identity by the just introduced function $*$.

In general, one would have to consider general multi-set arithmetics. We consider it as far as we can with reasonable effort because it *is* an issue which comes up with the high-level language in Chapter 9 the latest, but we refrained from a full treatment, which is a topic in its own right, to limit the extension of this thesis.

A possible alternative to multi-sets would be to consider, as natural domain of the link type, structures on which iterators operate, similar to iteration concepts in UML's Object Constraint Language [139] (OCL).

4.2.2. Structure

Definition 4.2.4 (Structure). *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature. A* structure *of $\mathcal{S}$ is a pair*

$$\mathcal{M} = (\iota, \mathcal{D}) \tag{4.18}$$

of a (type-consistent) interpretation *ι of the function symbols in $\mathcal{F}$ and a set $\mathcal{D}$ of* domains *for the types in $\mathcal{T}$.*

A type-consistent interpretation ι of the function symbols in $\mathcal{F}$ assigns each function symbol $f \in \mathcal{F}$ of type $f : \tau_1 \times \cdots \times \tau_k \to \tau$ a (possibly total) function

$$\iota(f) : \mathcal{D}(\tau_1) \times \cdots \times \mathcal{D}(\tau_k) \nrightarrow \mathcal{D}(\tau) \tag{4.19}$$

where $\mathcal{D}(\tau)$ denotes the domain of τ as given by $\mathcal{D}$.

We also use $\mathcal{D}$ to denote the union of all domains $\mathcal{D}(\tau)$, $\tau \in \mathcal{T}$, that is, the set of all semantical values used by the semantical domain $\mathcal{D}$. ♢

Note that the interpretation of function symbols from $\mathcal{F}$ is global. The local things (current local state of an individual, for example, or being alive, newly created, or disappearing) is explicitly part of the syntax and obtains an explicit semantics (cf. Section 4.2.1).

Definition 4.2.5 (Canonical Structure). *Let $\mathcal{S}$ be a signature with n identity types and let G be a compatible (Σ, Λ)-topology over identities Id partitioned into $Id_1 \dot\cup \ldots \dot\cup Id_m$. A structure $\mathcal{M}$ of $\mathcal{S}$ is called* canonical structure wrt. *G if and only if the following domains are chosen as follows.*

- $\mathcal{D}(\mathbb{B}) = \mathbb{B}_3 = \{0, 1, 1/2\}$
- $\mathcal{D}(S) = \Sigma$
- $\mathcal{D}(L) = \bigcup_{\lambda \in \Lambda} \mathrm{dom}(\rightarrowtail_\lambda)$
- $\mathcal{D}(T_j) \in \{Id_1, \ldots, Id_m\}$, $1 \leq j \leq n$
- $\mathcal{D}(\boldsymbol{T}_j) = \mathcal{D}(T_j)^+ \cup \mathcal{D}(T_j)^\omega$, $1 \leq j \leq n$

A structure is called canonical wrt. an topology transition system M over Id if and only if it is canonical wrt. M's topologies. ♢

The canonical domain of the booleans $\mathbb{B}$ is three-valued, i.e. $\mathbb{B}_3$, in order to cater for all kinds of undefinedness; first of all, arising from pre-maturely disappearing individual, but also from navigating dangling links.

The canonical domain of the local states S is simply the set of local states, The canonical domain of the type of link sets is the set of multi-sets over identities in order to semantically represent the different notions of agent inter-connections (cf. Figure 3.5)

Concerning logical variables, there are two different kinds of domains as discussed in the introduction of this section.

The canonical domain of logical variables of type T is simply the set of identities, or one of the partitions in case of typed topologies, the canonical domain of logical variables of type $\boldsymbol{T}$ is the set of finite and infinite *sequences* of individuals.

Note that in the following, we only distinguish if necessary between having only one identity type T or multiple ones. In most cases, it'll be sufficient to discuss the single type case as it generalises straightforwardly to multiple types in our notion of typed topologies.

4.2.3. Assignment

The definition of assignment itself in terms of semantical domains is standard.

Definition 4.2.6 (Assignment and Modification)**.** *Let $\mathcal{M} = (\iota, \mathcal{D})$ be a structure of the signature $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$. An* assignment *of a set of variables $V \subseteq \mathcal{V}$ in $\mathcal{M}$ is a function $\theta : V \to \mathcal{D}$ assigning each $v \in V$ of type τ a value from $\mathcal{D}(\tau)$. We use $Assign_{\mathcal{M}}(V)$ to denote the set of all assignments of V in $\mathcal{M}$. The index may be omitted if it is determined by the context.*

The modification *of θ at v to d, that is, the assignment of V which coincides with θ on $V \setminus \{v\}$ and yields d for v, is denoted by $\theta[v \mapsto d]$.* ♢

Definition 4.2.7 (Assignment Information Order)**.** *Let $\mathcal{S}$ be a signature and G a compatible topology over Id. Let $\mathcal{M} = (\iota, \mathcal{D})$ be a canonical structure of $\mathcal{S}$ wrt. G and $\theta_1, \theta_2 \in Assign_{\mathcal{M}}(V)$ two assignments of a set of variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

We say, θ_2 is more indefinite *than θ_1, denoted by*

$$\theta_1 \sqsubseteq \theta_2, \tag{4.20}$$

if and only if they are identical up to length of evolution chains, that is, if $\theta_1(x) = \theta_2(x)$ for each $x \in V$ of type T and $\theta_1(v)/k = \theta_2(v)/k$ for $0 \leq k < n$ where n is the length of $\theta_2(v)$ for each $v \in V$ of type $\boldsymbol{T}$. ♢

Definition 4.2.8 (Assignment Evolution)**.** *Let $\mathcal{S}$ be a signature and G a compatible topology over Id. Let $\mathcal{M} = (\iota, \mathcal{D})$ be a canonical structure of $\mathcal{S}$ wrt. G and $\theta \in Assign_{\mathcal{M}}(V)$ an assignment of variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

The k-step evolution *of θ, denoted by θ/k, is the assignment of V which coincides with θ on all variables not of type $\boldsymbol{T}$ and yields the sequence suffix starting at the k-th element for all variables of type $\boldsymbol{T}$, i.e.*

$$(\theta/k)(v) = \begin{cases} \theta(v)/k & \text{, if } v \text{ is of type } \boldsymbol{T} \\ \theta(v) & \text{, otherwise} \end{cases} \tag{4.21}$$

♢

Note that the k-step evolution changes the valuation of $\boldsymbol{T}$-typed variables to the empty sequence if the original valuation is of finite length smaller than k.

Note 4.2.9 (Assignment Information Order Under Evolution)**.** *Let $\mathcal{S}$ be a signature and G a compatible topology over Id. Let $\mathcal{M} = (\iota, \mathcal{D})$ be a canonical structure of $\mathcal{S}$ wrt. G and $\theta_1, \theta_2 \in Assign_{\mathcal{M}}(V)$ two assignments of a set of variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

Information order is invariant under evolution, that is,

$$\theta_1 \sqsubseteq \theta_2 \implies \forall\, k \in \mathbb{N}_0 : \theta_1/k \sqsubseteq \theta_2/k. \tag{4.22}$$

♢

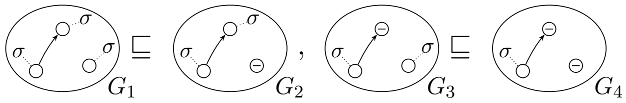


Figure 4.4.: **Information order on topologies.** Topology G_1 is less defined than both, G_2 and G_3, which are unordered with respect to each other but both are less defined than G_4.

Definition 4.2.10. *Let $\mathcal{S}$ be a signature, G a compatible topology, and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. G. Let $\theta \in Assign_{\mathcal{M}}(V)$ be an assignment of a set of variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

We call θ an assignment in G *if and only if θ maps every destiny variable to the empty evolution chain or an evolution chain beginning with an alive individual from the topology G, i.e. if*

$$\forall \boldsymbol{x} \in \mathcal{V}^{\boldsymbol{T}} : \theta(\boldsymbol{x}) = \varepsilon \vee \theta(\boldsymbol{x})/1 \in U^{\circledcirc}(G). \tag{4.23}$$

◇

Definition 4.2.11 (Assignment in State or Path)**.** *Let $\mathcal{S}$ be a signature, M a compatible evolving topology transition system, and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. G. Let $\theta \in Assign_{\mathcal{M}}(V)$ be an assignment of a set of variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

1. *Let $s \in S(M)$ be a state of M. We call θ an assignment* in s *if and only if θ is an assignment in the topology of s, i.e. in $\mathscr{L}(s)$.*

2. *Let $\pi \in \Pi_s(M)$ be a computation path in M starting at some state $s \in S(M)$. We call θ an assignment* in π *if and only if all evolution chains used by θ are evolution chains along π, i.e. if*

$$\forall \boldsymbol{x} \in V \cap \mathcal{V}^{\boldsymbol{T}} : \theta(\boldsymbol{x}) \in \Delta(U^{\circledcirc}(\pi^0), \pi) \tag{4.24}$$

The set of all assignments of V in a given state s and path π is denoted by $Assign_{\mathcal{M}}(V, s)$ and $Assign_{\mathcal{M}}(V, \pi)$. ◇

4.2.4. Topology and ETTS Information Order

One important property we expect from our EvoCTL* semantics is that it indicates unexpected disappearance of objects by yielding the indefinite value $1/2$.

As mentioned in Section 4.1, we intuitively consider disappearance of an individual to be unexpected if a logical variable refers to it and is not guarded by a life-cycle operator – like "◎" – which always evaluates definite in case of disappearance.

In order to formally discuss whether our semantics satisfies these expectations, we define a notion of information order on topologies. The expectation is then more precisely that the evaluation of an EvoCTL* tends to change to $1/2$ with growing undefinedness (cf. Sections 4.3.4 and 4.4.6).

Definition 4.2.12 (Topology Information Order)**.** *Let G_1 and G_2 be (Σ, Λ)-topologies over Id. We say, G_2 is* more undefined *than G_1, denoted by*

$$G_1 \sqsubseteq G_2, \tag{4.25}$$

if and only if G_1 and G_2 are identical up to local state labelling, that is,

1. *$U(G_1) = U(G_2)$, $L(G_1) = L(G_2)$, $\psi(G_1) = \psi(G_2)$, $\lambda(G_1) = \lambda(G_2)$,*
2. *$\rightarrowtail_\lambda^{G_1} = \rightarrowtail_\lambda^{G_2}$, for all $\lambda \in \Lambda$, and*
3. *$\forall\, id \in \mathrm{dom}(\sigma(G_2)) : id \in \mathrm{dom}(\sigma(G_1)) \wedge \sigma(G_2)(id) = \sigma(G_1)(id)$.* ♢

Note that the information order on topologies doesn't have the nice property of being a complete lattice. There is no single largest element, only subsets of the same size have a largest element.

The following definition extends information order to paths and states to prepare the above named sections on monotonicity.

Definition 4.2.13 (ETTS Information Order)**.** *Let M_1 and M_2 be two (Σ, Λ)-ETTS over Id. We say that M_1 is* less defined *than M_2, denoted by*

$$M_1 \sqsubseteq M_2, \tag{4.26}$$

if and only if

1. *they have identical state and initial state sets, that is, $S(M_1) = S(M_2)$, $S_{0_1}(M_1) = S_{0_2}(M_2)$,*
2. *they have identical transition sets and M_2 admits at least as much evolution as M_1, that is, given $R(M_1) = (R_1, e_1)$ and $R(M_2) = (R_2, e_2)$, $R_1 = R_2$ and $\forall\, r \in R_1 = R_2 : e_1\langle r\rangle \subseteq e_2\langle r\rangle$,*
3. *the corresponding labellings of states are in topology information order, that is, $\forall\, s \in S(M_1) = S(M_2) : \mathscr{L}(M_1)(s) \sqsubseteq \mathscr{L}(M_2)(s)$.* ♢

4.3. Terms

4.3.1. Syntax

Terms are used to express properties of a *single* topology (or world), for instance, logical connectives of the local states individuals are in or who has links to whom. A term is basically a predicate logic expression over predicates and functions.

Note that for completeness we introduce two flavours of logical variables, ranging over identities and over evolution chains as discussed in Section 4.1, although evolution is not relevant when evaluating terms for topologies. We'll discuss the relation between the two kinds of logical variables formally after having defined the semantics of terms.

Further note that quantification is limited to identities and individuals, that is, we don't admit direct quantification over links or local states. Accessing links is only indirectly possible by employing individuals. For example, the property of existence of at least one link labelled with λ is equivalent to the existence of two identities u_1 and u_2 such that u_1 has a λ-link to u_2.

Following [190, 191], it is also sensible to have a general transitive closure operator in order to express properties of reachability which is highly important when treating data-structures like linked lists, but not of *that* vital importance in our scope; thus, and because it is easily lost in our abstraction, we leave it out.

In the following, we distinguish between functional and logical terms. The intention is to avoid logical terms, like whole quantified expressions, as the argument of function or predicate symbols. Function or predicate symbols shall only depend on individuals, logical connections can always be expressed employing the explicitly given logical connectives.

Definition 4.3.1 (Term). *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature. A type-consistent word of the grammar*

$$a ::= x \mid \boldsymbol{x} \mid \circledcirc\, a_1 \mid \sigma(a_1) \mid \lambda(a_1) \mid a_1 = a_2 \mid f(a_1, \dots, a_k) \tag{4.27}$$

where $x, \boldsymbol{x} \in \mathcal{V}$ are logical variables, $\lambda \in \Lambda$ is a link name, and $f \in \mathcal{F}$ a function symbol, is called (typed) functional term *over $\mathcal{S}$, i.e.*

- *if x and $\boldsymbol{x}$ are logical variables from $\mathcal{V}$ denoting an identity or an evolution chain, i.e. of type T or $\boldsymbol{T}$, then x and $\boldsymbol{x}$ are functional terms over $\mathcal{S}$ of type T and $\boldsymbol{T}$, respectively,*
- *if a_1 is a functional term of type T or $\boldsymbol{T}$, then $\circledcirc\, a_1$, $\sigma(a_1)$, and $\lambda(a_1)$, are functional terms over $\mathcal{S}$ of type $\mathbb{B}$, S, and L, respectively,*
- *if a_1 and a_2 are functional terms, each of type T or $\boldsymbol{T}$, then $a_1 = a_2$ is a functional term over $\mathcal{S}$ of type $\mathbb{B}$,*
- *if $f : \tau_1 \times \cdots \times \tau_k \to \tau$ is a function symbol from $\mathcal{F}$ and $a_1, \dots, a_k$ are terms of types $\tau_1, \dots, \tau_k$, then $f(a_1, \dots, a_k)$ is a functional term over $\mathcal{S}$ of type τ, and*
- *nothing else is a functional term over $\mathcal{S}$.*

A type-consistent word of the grammar

$$\begin{aligned} t ::= {} & 0 \mid 1 \mid a \mid \neg(t_1) \mid (t_1 \vee t_2) \mid (t_1 \wedge t_2) \\ & \mid \forall\, x : T \,.\, t_1 \mid \exists\, x : T \,.\, t_1 \mid \forall\, \boldsymbol{x} : T \,.\, t_1 \mid \exists\, \boldsymbol{x} : T \,.\, t_1, \end{aligned} \tag{4.28}$$

where a is a functional term of type $\mathbb{B}$ and $x, \boldsymbol{x} \in \mathcal{V}$ are logical variables of the identity type $T \in \{T_1, \dots, T_n\}$ and the evolution chain type $\boldsymbol{T} \in \{\boldsymbol{T}_1, \dots, \boldsymbol{T}_n\}$, is called (typed) (logical) term *over $\mathcal{S}$, i.e.*

- 0 *and* 1 *are logical terms over $\mathcal{S}$,*

- *if a is a functional term of type $\mathbb{B}$ over $\mathcal{S}$, then a is a logical term over $\mathcal{S}$,*
- *if t_1 and t_2 are logical terms over $\mathcal{S}$, then $\neg(t_1)$, $(t_1 \vee t_2)$, $(t_1 \wedge t_2)$, $\forall\, x : T \,.\, t_1$, $\exists\, x : T \,.\, t_1$, $\forall\, \boldsymbol{x} : T \,.\, t_1$, and $\exists\, \boldsymbol{x} : T \,.\, t_1$ are logical terms over $\mathcal{S}$, and*
- *nothing else is a logical term over $\mathcal{S}$.*

We use Term$_\tau(\mathcal{S})$ and Term$(\mathcal{S})$ to denote the sets of all terms of type τ and of all terms over $\mathcal{S}$. The terms in Term$_\mathbb{B}(\mathcal{S})$ are called boolean terms. ◇

Definition 4.3.2 (Bound and Free Variables.)**.** *Let t be a logical term over signature $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$. A variable $v \in \mathcal{V}$ is said to occur* bound *in t if and only if each occurrence of v takes place in a sub-term of t of the forms $\forall\, v \,.\, t_1$ or $\exists\, v \,.\, t_1$. Otherwise v is called* free.

We use Free(t) to denote the set of free variables in t. ◇

Abbreviations, Alternative Notations, and Binding Priorities

We shall use the following common abbreviations and alternative notations.

1. $t.\sigma$ and $t.\lambda$ for $\sigma(t)$ and $\lambda(t)$,
2. $t.\lambda.\cdots.\lambda.\sigma$ for $\sigma(*(\lambda(\ldots * (\lambda(t))\ldots)))$ if $\mathcal{F}$ comprises the link-projection operator $*$ (see Section 4.2.1 above),
3. $(t_1 \dot{\vee}\, t_2)$ for $(t_1 \wedge \neg t_2) \vee (\neg t_1 \wedge t_2)$,
4. $(t_1 \rightarrow t_2)$ for $(\neg t_1) \vee t_2$,
5. $(t_1 \leftrightarrow t_2)$ for $(t_1 \rightarrow t_2) \wedge (t_2 \rightarrow t_1)$,
6. $\forall\, v_1 : \tau_1, v_2 : \tau_2, \ldots, v_n : \tau_n \,.\, t$ for $\forall\, v_1 : \tau_1 \,.\, \forall\, v_2 : \tau_2 \,.\, \ldots \forall\, v_n : \tau_n \,.\, t$, where $v_i \in \mathcal{V}$ of type τ_i, $1 \leq i \leq n$,
7. $\exists\, v_1 : \tau_1, v_2 : \tau_2, \ldots, v_n : \tau_n \,.\, t$ for $\exists\, v_1 : \tau_1 \,.\, \exists\, v_2 : \tau_2 \,.\, \ldots \exists\, v_n : \tau_n \,.\, t$ where $v_i \in \mathcal{V}$ of type τ_i, $1 \leq i \leq n$,
8. $\forall\, v_1, v_2, \ldots, v_n : \tau \,.\, t$ for $\forall\, v_1 : \tau \,.\, \forall\, v_2 : \tau \,.\, \ldots \forall\, v_n : \tau \,.\, t$, where $v_i \in \mathcal{V}$ of type τ, $1 \leq i \leq n$, and
9. $\exists\, v_1, v_2, \ldots, v_n : \tau \,.\, t$ for $\exists\, v_1 : \tau \,.\, \exists\, v_2 : \tau \,.\, \ldots \exists\, v_n : \tau \,.\, t$ where $v_i \in \mathcal{V}$ of type τ, $1 \leq i \leq n$.

We shall use the following common *binding priorities* to avoid parentheses.

1. Negation ($\neg$) binds most tightly.
2. Next in order is conjunction ($\wedge$).
3. Then come disjunctions with equal priority ($\vee$ and $\dot{\vee}$).

4. Finally implication and equivalence with equal priority (→ and ↔).

Using $\lll$ do denote lower binding priority, we thus have

$$\{\neg\} \lll \{\wedge\} \lll \{\vee, \dot{\vee}\} \lll \{\rightarrow, \leftrightarrow\}. \tag{4.29}$$

4.3.2. Semantics

Although we assume a complete assignment of the free variables in a term below, terms may be undefined because we may navigate empty links, or links comprising more than one individual, or dereference an identity for which there is currently no individual.

Furthermore, we admit that functions may be undefined for certain values, for example, the link-projection operator $*$ will typically be undefined for empty links, or, most common example, division by 0 if there is arithmetics on local states.

The whole term may of course still yield a definite value, even if some sub-term is undefined as it may be a tautology or a contradiction.

Definition 4.3.3 (Term Semantics)**.** *Let $\mathcal{S}$ be a signature, G a compatible topology over identities Id, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. G.*

Let t be a term over $\mathcal{S}$ and let θ be an assignment of the variables occurring free in t, i.e. an assignment of the set $\theta \in Assign_{\mathcal{M}}(Free(t))$. The valuation of t is inductively defined as follows.

1. $\iota[\![0]\!](G,\theta) = 0$
2. $\iota[\![1]\!](G,\theta) = 1$
3. $\iota[\![x]\!](G,\theta) = \theta(x)$
4. $\iota[\![\boldsymbol{x}]\!](G,\theta) = \theta(\boldsymbol{x})(0),$ *if $\theta(\boldsymbol{x}) \neq \varepsilon$, undefined otherwise*
5. $$\iota[\![\circledcirc\, t_1]\!](G,\theta) = \begin{cases} 0 & \textit{, if } t_1 \textit{ has a valuation and} \\ & (\iota[\![t_1]\!](G,\theta) \in U^{\not\circledcirc}(G) \setminus U^{\circledcirc}(G)) \\ 1 & \textit{, if } t_1 \textit{ has a valuation and} \\ & (\iota[\![t_1]\!](G,\theta) \in U^{\circledcirc}(G) \setminus U^{\not\circledcirc}(G)) \\ {}^{1}\!/_{2} & \textit{, otherwise} \end{cases}$$
6. $\iota[\![\sigma(t_1)]\!](G,\theta) = \sigma(G)(\iota[\![t_1]\!](G,\theta))$ *if t_1 has a valuation which is in* $\mathrm{dom}(\sigma(G))$*, undefined otherwise*
7. $\iota[\![\lambda(t_1)]\!](G,\theta) = \rightarrowtail_{\lambda}(\iota[\![t_1]\!](G,\theta))$ *if t_1 has a valuation, undefined otherwise*
8. $\iota[\![f(t_1,\ldots,t_k)]\!](G,\theta) = \iota(f)(\iota[\![t_1]\!](G,\theta),\ldots,\iota[\![t_k]\!](G,\theta)),$ *if $t_1,\ldots,t_k$ have a valuation and $\iota(f)$ is defined for these values, undefined otherwise*

9. $\iota[\![p(t_1,\dots,t_k)]\!](G,\theta) = \iota(p)(\iota[\![t_1]\!](G,\theta),\dots,\iota[\![t_k]\!](G,\theta))$, *if* $t_1,\dots,t_k$ *have a valuation and* $\iota(p)$ *is defined for these values,* $1/2$ *otherwise*

10. $\iota[\![t_1 = t_2]\!](G,\theta) = eq_{Id}(\iota[\![t_1]\!](G,\theta),\iota[\![t_2]\!](G,\theta))$, *if* t_1 *and* t_2 *have a valuation,* $1/2$ *otherwise*

11. $\iota[\![\neg(t_1)]\!](G,\theta) = 1 - \iota[\![t_1]\!](G,\theta)$

12. $\iota[\![(t_1 \vee t_2)]\!](G,\theta) = \max\{\iota[\![t_1]\!](G,\theta),\iota[\![t_2]\!](G,\theta)\}$

13. $\iota[\![(t_1 \wedge t_2)]\!](G,\theta) = \min\{\iota[\![t_1]\!](G,\theta),\iota[\![t_2]\!](G,\theta)\}$

14. $\iota[\![\forall\, x : T \,.\, t_1]\!](G,\theta) = \min\{\iota[\![t_1]\!](G,\theta[x \mapsto id]) \mid id \in Id\}$

15. $\iota[\![\exists\, x : T \,.\, t_1]\!](G,\theta) = \max\{\iota[\![t_1]\!](G,\theta[x \mapsto id]) \mid id \in Id\}$

16. $\iota[\![\forall\, \boldsymbol{x} : T \,.\, t_1]\!](G,\theta) = \min\{\iota[\![t_1]\!](G,\theta[\boldsymbol{x} \mapsto id]) \mid id \in U^{\circledcirc}(G)\}$

17. $\iota[\![\exists\, \boldsymbol{x} : T \,.\, t_1]\!](G,\theta) = \max\{\iota[\![t_1]\!](G,\theta[\boldsymbol{x} \mapsto id]) \mid id \in U^{\circledcirc}(G)\}$ ♢

Def. 4.3.3 gives rise to the following equivalences – which we recall here in order to point out differences between 3-valued Kleene logic and classical 2-valued Boolean logic. Note 4.3.4.4 exemplifies the significant consequence of using three-valued logic that many syntactical indications for tautologies or contradictions known for classical boolean logic are lost in three-valued logic.

Note 4.3.4 (Indefiniteness of Terms)**.** *Let* t_1 *and* t_2 *be logical terms over signature* $\mathcal{S}$*,* G *a topology over identities Id compatible with* $\mathcal{S}$*, and* $\mathcal{M}$ *a canonical structure of* $\mathcal{S}$ *wrt.* G*. Then*

1. $\iota[\![\neg t_1]\!](G,\theta) = 1/2$ *if and only if* $\iota[\![t_1]\!](G,\theta) = 1/2$,

2. $\iota[\![t_1 \vee t_2]\!](G,\theta) = 1/2$ *if* $\iota[\![t_1]\!](G,\theta) = 1/2$ *and* $\iota[\![t_2]\!](G,\theta) \neq 1$,

3. $\iota[\![t_1 \wedge t_2]\!](G,\theta) = 1/2$ *if* $\iota[\![t_1]\!](G,\theta) = 1/2$ *and* $\iota[\![t_2]\!](G,\theta) \neq 0$.

4. $t_1 \vee \neg t_1$ *is not a tautology,* $t_1 \wedge \neg t_1$ *is not a contradiction.* ♢

4.3.3. Semantical Equivalence

Definition 4.3.5 (Semantical Equivalence of Terms)**.** *Let* t_1 *and* t_2 *be terms over signature* $\mathcal{S} = (\mathcal{T},\mathcal{V},\mathcal{F},\Lambda)$*. We say* t_1 *and* t_2 *are* semantically equivalent*, denoted by*

$$t_1 \equiv t_2, \tag{4.30}$$

if and only if their valuation is independent from the employed structure, that is, if

$$\iota[\![t_1]\!](G,\theta) = \iota[\![t_2]\!](G,\theta) \tag{4.31}$$

for any canonical structure $\mathcal{M} = (\iota,\mathcal{D})$ *of* $\mathcal{S}$ *wrt. an* $\mathcal{S}$*-compatible topology* G *and any assignment* θ*.* ♢

4.3.4. Definiteness and Monotonicity

Given the three-valued semantics of terms from Section 4.3.2 we can investigate *definiteness* and *monotonicity* properties. The former denotes the wish to give syntactical criteria which ensure that a given term evaluates definite, that is, not to $1/2$, under all circumstances. As this is only a side-aspect of our work, we're content to only obtain some sufficient criteria and a characterisation of sources for indefiniteness, but no exact criteria.

The second denotes the desired property that a term evaluates rather to $1/2$ on a more undefined topology (cf. Section 4.2.4).

In this section, we assume concrete topologies, that is, $U^{\circledcirc}(G) \cap U^{\not\circledcirc}(G) = \emptyset$.

Monotonicity

The following lemma indicates that our semantics of terms (without life-cycle queries) is monotone in the employed topologies, that is, with the same assignment, a more defined topology in the sense of Def. 4.2.12 yields a more definite valuation.

The exclusion of "$\circledcirc$" in the lemma provides the following understanding. Aliveness queries allow us to distinguish topologies that are in information order.[2] For example, if a topology G_1 is more defined than a topology G_2 at identity *id*, then an aliveness query for *id* may yield 0 in G_2 and 1 in G_1, two logical values that are not in information order. The other way round, if life-cycle queries are present and *guard* certain variables, they express an awareness for disappearance. Yet this is different from saying that formulae with guarded variables *always* evaluate definite.

The crucial case is when they're left out. Then the formula is not aware of disappearance, and then the evaluation should become less definite in less defined topologies.

For a similar reason, quantification is excluded from the discussion of logical terms because quantification can also distinguish topologies that only differ in alive individuals. For example the query whether there exists an individual with a certain local state is 1 if one is present in G_1 and turns 0 if they are removed (or dead) in G_2. Our initial observations are still useful since they tell us what happens to the term under the quantifier, in particular, what happens with evolution over time, which we'll see in a later lemma.

Note that for terms, only the first element of sequences assigned to variables of type $\boldsymbol{T}$ is relevant. This will naturally change when considering temporal properties.

Lemma 4.3.6 (Monotonicity of Term Evaluation)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature, $G_1 \sqsubseteq G_2$ two $\mathcal{S}$-compatible topologies over Id, and $\mathcal{M}$ a canonical structure wrt. G_1 and G_2.*

1. *The evaluation of a boolean functional term a without sub-terms of the form $\circledcirc\, a_0$ is monotone in the topology, that is,*

$$\iota\llbracket a \rrbracket(G_1, \theta) \sqsubseteq \iota\llbracket a \rrbracket(G_2, \theta) \tag{4.32}$$

[2] Distinction in the formal sense that the formula evaluates to different definite values, or: that we know something about both cases, which is different for both cases.

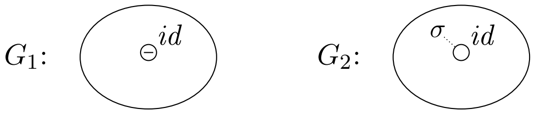


Figure 4.5.: **Example topologies** supporting the claim of Note 4.3.7.

for any assignments $\theta_1 \sqsubseteq \theta_2$ of a's free variables.

2. *The evaluation of logical terms where logical variables of type $\boldsymbol{T}$ only occur freely is monotone in the valuation of their boolean sub-terms.*

3. *The valuation of a boolean term a without sub-terms of the form $\circledcirc\, a_0$ and where logical variables of type $\boldsymbol{T}$ only occur freely is monotone in the topology, that is,*

$$\iota[\![t]\!](G_1, \theta) \sqsubseteq \iota[\![t]\!](G_2, \theta) \tag{4.33}$$

for any assignments $\theta_1 \sqsubseteq \theta_2$ of t's free variables. ♢

Proof. See Section A.1. □

The following note complements the lemma. Together we're close to an "if-and-only-if", the missing point are again possibly occurring tautologies.

Note 4.3.7. *Let $\mathcal{S}$ be a signature such that $\mathcal{V}(\mathcal{S})$ comprises at least one logical variable $\boldsymbol{x}$ of an evolution chain type $\boldsymbol{T}$ and one logical variable x of an identity type T.*

Let t denote a term of the form

1. *$\circledcirc\, v$, or*
2. *$\forall \boldsymbol{x} : T . \psi$.*

Then we can choose ψ such that there exist two $\mathcal{S}$-compatible topologies G_1 and G_2, a canonical structure $\mathcal{M}$ of $\mathcal{S}$ wrt. G_1 and G_2, and assignments θ_1, θ_2 with $G_1 \sqsubseteq G_2$ and $\theta_1 \sqsubseteq \theta_2$, but

$$\iota[\![\varphi]\!](G_1, \theta_1) \not\sqsubseteq \iota[\![\varphi]\!](G_2, \theta_2). \tag{4.34}$$

♢

Proof.

1. Consider the topologies G_1 and G_2 shown in Figure 4.5. We have $G_1 \sqsubseteq G_2$ because both are identical up to definedness of σ. Assuming θ assigns id to v, as sequence of length one in case v is of type $\boldsymbol{T}$ and plainly otherwise, we obtain

$$\iota[\![\circledcirc\, v]\!](G_1, \theta) = 0 \not\sqsubseteq 1 = \iota[\![\circledcirc\, v]\!](G_2, \theta). \tag{4.35}$$

2. With the same setting as in the previous case, we have

$$\iota[\![\forall \boldsymbol{x} : T . 0]\!](G_1, \theta) = 1 \not\sqsubseteq 0 = \iota[\![\forall \boldsymbol{x} : T . 0]\!](G_2, \theta) \tag{4.36}$$

because the first quantification ranges over the empty set and the second doesn't.

□

Definiteness

Lemma 4.3.8 (Definite Valuation of Terms)**.** *Let $\mathcal{S}$ be a signature, G a compatible topology over identities Id, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. G.*

Let t be a logical term such that

1. *all variables occurring in t are of type $\boldsymbol{T}$ and bound,*
2. *none of the operators $\lambda \in \Lambda(\mathcal{S})$ occurs in t,*
3. *the interpretation of all function and predicate symbols f and p occurring in t by ι is total and definite.*

Then t evaluates definite in G under $\mathcal{M}$, i.e., $\iota[\![t]\!](G, \theta) \neq 1/2$, for any assignment θ. ◇

Proof. See Section A.1. □

On second sight, the previous lemma has very strong premises, namely the exclusion of link navigation. Without them we'd have to impose semantical restrictions, which means giving up the aim of this section: to tell by the (syntactical) form of a formula whether it evaluates definite or not. We'd have to take into account the topology where the formula is evaluated.

A syntactical alternative lies in our restriction to single links instead of using full multisets. This doesn't directly ensure definite evaluation because, for instance, comparison for equality still evaluates indefinite on dangling links.

The assumption of complete absence of dangling links, that is, links don't even temporally dangle, is in general to strict. A remaining way to exclude this source for indefiniteness is to assume query links only in a guarded fashion, that is, to transform terms of the form

$$a := p(*(\lambda(v))) \tag{4.37}$$

to

$$\lambda(v) \neq \emptyset \wedge a \quad \text{or} \quad \lambda(v) \neq \emptyset \implies a \tag{4.38}$$

Note that this is obviously not semantics preserving in case of dangling links. One has to consider carefully whether (4.38) is admissible, that is, actually characterises the intended property.

Here the problem is that (4.38) is *biased*, that is, the former expression maps the undefined case to one to the definite value 0, to second one maps it to 1. In other words: these definite value 0 suddenly happens to mean two things, namely, that the link exists and p evaluates to 0 *and* that the link doesn't exist.

The following note complements Lemma 4.3.8 by listing cases in which we cannot assume definite valuations of terms. By the monotonicity lemma, Lemma 4.3.6, such indefinite valuations are likely to propagate through to the valuation of the whole term unless they appear explicitly guarded, for example by implications depending on aliveness or in tautologies or contradictions . That is, together they're still not complete, but provide a first indication of what we can and have to expect.

Firstly, logical variables of type $\boldsymbol{T}$ lead to indefiniteness of functional terms if they are assigned the empty sequence ε. Secondly, any logical variable denoting a non-alive individual immediately leads to indefinite valuation of functional terms by definition. And thirdly, function and predicate symbols may obtain an interpretation which yields indefinite values or is partially undefined, for instance division is often undefined when dividing by zero.

Note 4.3.9 (Indefinite Valuation of Terms)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature.*

- *Let t denote any of the functional terms*

  $$\circledcirc\, \boldsymbol{x}, \sigma(\boldsymbol{x}), \lambda(\boldsymbol{x}), f(\dots, \boldsymbol{x}, \dots), p(\dots, \boldsymbol{x}, \dots), \boldsymbol{x} = v \tag{4.39}$$

  $\boldsymbol{x} \in \mathcal{V}$ of type $\boldsymbol{T}$, $f, p \in \mathcal{F}$.

  Then for all $\mathcal{S}$-compatible topologies G and canonical structures $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. G, there is an assignment θ of the free variables in t such that the valuation of t is indefinite or undefined.

- *Let t denote any of the functional terms*

  $$\sigma(v_1), \lambda(v_1), f(\dots, v_1, \dots), p(\dots, v_1, \dots), v_1 = v_2 \tag{4.40}$$

  $v_1, v_2 \in \mathcal{V}$ of type T or $\boldsymbol{T}$, $f, p \in \mathcal{F}$.

  Then for all $\mathcal{S}$-compatible topologies G with at least one non-alive identity and all canonical structures $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. G, there is an assignment θ of the free variables in t such that the valuation of t is indefinite or undefined.

- *Let t denote any of the functional terms*

  $$f(\dots, v, \dots), p(\dots, v, \dots), \tag{4.41}$$

  $v \in \mathcal{V}$ of type T or $\boldsymbol{T}$, $f, p \in \mathcal{F}$.

  Then for all $\mathcal{S}$-compatible topologies G there exists a canonical structure $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. G such that the valuation of t is indefinite or undefined for any assignment θ of the free variables in t. ♢

Proof.

1. By assigning the empty evolution chain to $\boldsymbol{x}$, that is, $\theta(\boldsymbol{x}) = \varepsilon$, the valuation of t immediately turns indefinite by definition.

2. Let $id \in Id$ denote a non-alive identity. Assigning id to v, that is, $\theta(x) = id$ or $\theta(\boldsymbol{x}) = id$ (sequence of length 1), the valuation of t immediately turns indefinite by definition.

3. The interpretation of f and p can be chosen to be undefined or to constantly yield the indefinite value $1/2$. Then the valuation of the term is indefinite independent from topology and assignment.

□

Evolution Chain vs. Identity Quantification

The following note states that, within terms, the only difference between quantification of logical variables of types T and $\boldsymbol{T}$ is that for the latter only alive individuals are considered.

This can be expressed with quantification over variables of type T by employing the aliveness operator $\circledcirc$.

Treating logical variables of type $\boldsymbol{T}$ at all in terms is necessary because terms form the building blocks of temporal formulae where variables of this type may also be bound and then referred to in terms. In this case, of terms within temporal formulae, the difference between identities and evolution chains becomes visible.

Note 4.3.10 (Identities vs. Evolution Chains in Terms)**.** *Let t be a term over signature $\mathcal{S}$ and $x, \boldsymbol{x} \in \mathcal{V}(\mathcal{S})$ logical variables. Then:*

1. $\forall \boldsymbol{x} : T \,.\, t \equiv \forall x : T \,.\, (\circledcirc x) \rightarrow t$

2. $\exists \boldsymbol{x} : T \,.\, t \equiv \exists x : T \,.\, (\circledcirc x) \wedge t$ ◇

Proof. We have to show that for each compatible (concrete) topology G over identities *Id*, and canonical structure $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. G we have

1. $\iota[\![\forall \boldsymbol{x} : T \,.\, t]\!](G, \theta) = \iota[\![\forall x : T \,.\, (\circledcirc x) \rightarrow t]\!](G, \theta)$ and
2. $\iota[\![\exists \boldsymbol{x} : T \,.\, t]\!](G, \theta) = \iota[\![\exists x : T \,.\, (\circledcirc x) \wedge t]\!](G, \theta)$

for any assignment θ.

This is evident because the identities considered for $\boldsymbol{x}$ are, by definition, taken from the set $U^{\circledcirc}(G)$. Also by definition, $\circledcirc$ is interpreted as the characteristic function of $U^{\circledcirc}(G)$ (in concrete topologies, that is, $U^{\circledcirc}(G) \cap U^{\not\circledcirc}(G) = \emptyset$). The logical combinations of $\circledcirc x$ and t ensure coincidence in particular in the trivial case $U^{\circledcirc}(G) = \emptyset$. □

4.4. EvoCTL*

The logic introduced in the following is called EvoCTL* because it is a conservative first-order extension of CTL* which is in addition aware of evolution and has primitives to query life-cycle properties.[3]

Our aims are the same as, for instance [190, 190] or any other proposal of first-order temporal logic for the specification of dynamic topology systems, but we give a more adequate semantics, and we see that evolution can also be added to a branching time logic.

The semantics is more adequate because by employing a three-valued semantics, we preserve well-known invariants of LTL, which most biased (see above) proposals fail. Examples can be found in the joint work [11].

[3] The name ETL is already occupied, even more than once [105, 191], although the ETL of [191] has later been renamed VTL [190].

Furthermore, the inclusion of both, quantification over identities *and* destinies in the same formalism, allows to compare them for expressiveness, and strengths and weaknesses.

Note that EvoCTL* is basically first-order CTL* over dynamic domains, where "first-order" refers to the possibility to quantify over individuals and destinies in addition to the first-order quantification over paths already provided by CTL*.

4.4.1. Syntax

Definition 4.4.1 (Evolution CTL*). *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature.*

1. *A type-consistent word of the grammar*

$$\begin{aligned}\phi ::= t \mid \odot\, a \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \\ \mid \forall\, x : T \,.\, \phi_1 \mid \exists\, x : T \,.\, \phi_1 \mid \forall\, \boldsymbol{x} : T \,.\, \phi_1 \mid \exists\, \boldsymbol{x} : T \,.\, \phi_1 \mid \mathsf{A}\,\psi \mid \mathsf{E}\,\psi\end{aligned} \tag{4.42}$$

   where $x \in \mathcal{V}$, is called state formula *over $\mathcal{S}$, i.e.*

   - *if $t \in Term_{\mathbb{B}}(\mathcal{S})$ is a boolean term over $\mathcal{S}$, then t is a state formula over $\mathcal{S}$,*
   - *if $t \in Term_T(\mathcal{S})$ is a term over $\mathcal{S}$ of identity type, then $\odot\, a$ is a state formula over $\mathcal{S}$,*
   - *if ϕ_1 and ϕ_2 are state formulae over $\mathcal{S}$, then $\neg\phi_1$, $\phi_1 \vee \phi_2$, and $\phi_1 \wedge \phi_2$ are state formulae over $\mathcal{S}$,*
   - *if ϕ_1 is a state formula over $\mathcal{S}$ and $x, \boldsymbol{x} \in \mathcal{V}$ are logical variables of types T and $\boldsymbol{T}$, then $\forall\, x : T \,.\, \phi_1$ ("for all identities, ϕ_1"), $\exists\, x : T \,.\, \phi_1$ ("for some identity, ϕ_1"), $\forall\, \boldsymbol{x} : T \,.\, \phi_1$ ("for all individuals' fate, ϕ_1"), and $\exists\, \boldsymbol{x} : T \,.\, \phi_1$ ("for some individual's fate, ϕ_1") are state formulae over $\mathcal{S}$,*
   - *if ψ is a path formula over $\mathcal{S}$, then $\mathsf{A}\,\psi$ ("on all paths ψ") and $\mathsf{E}\,\psi$ ("on some path ψ") are state formulae over $\mathcal{S}$, and*
   - *nothing else is a state formula over $\mathcal{S}$.*

2. *A type-consistent word of the grammar*

$$\begin{aligned}\psi ::= \phi \mid \otimes\, a \mid \neg\psi_1 \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \forall\, x : T \,.\, \psi_1 \mid \exists\, x : T \,.\, \psi_1 \\ \mid \forall\, \boldsymbol{x} : T \,.\, \psi_1 \mid \exists\, \boldsymbol{x} : T \,.\, \psi_1 \mid \mathsf{X}\,\psi_1 \mid \mathsf{F}\,\psi_1 \mid \mathsf{G}\,\psi_1 \mid \psi_1\,\mathsf{U}\,\psi_2 \mid \psi_1\,\mathsf{R}\,\psi_2\end{aligned} \tag{4.43}$$

   is called path formula *over $\mathcal{S}$, i.e.*

   - *if ϕ is a state formula over $\mathcal{S}$, then ϕ is a path formula over $\mathcal{S}$,*
   - *if $t \in Term_T(\mathcal{S})$ is a term over $\mathcal{S}$ of an identity type, then $\otimes\, a$ is a path formula over $\mathcal{S}$,*
   - *if ψ_1 and ψ_2 are path formulae over $\mathcal{S}$, then $\neg\psi_1$, $\psi_1 \vee \psi_2$, and $\psi_1 \wedge \psi_2$ are path formulae over $\mathcal{S}$,*

- *if ψ_1 is a path formula over $\mathcal{S}$ and $x, \boldsymbol{x} \in \mathcal{V}$ are logical variable of an identity type T and an evolution chain type $\boldsymbol{T}$, then $\forall\, x : T \,.\, \psi_1$, ("for all identities, ψ_1"), $\exists\, x : T \,.\, \psi_1$ ("for some identity, ψ_1"), $\forall\, \boldsymbol{x} : T \,.\, \psi_1$, ("for all individuals' fate, ψ_1"), and $\exists\, \boldsymbol{x} : T . \psi_1$ ("for some individual's fate, ψ_1") are path formulae over $\mathcal{S}$,*
- *if ψ_1 and ψ_2 are path formulae over $\mathcal{S}$, then $\mathsf{X}\, \psi_1$ ("next ψ_1"), $\mathsf{F}\, \psi_1$ ("finally ψ_1"), $\mathsf{G}\, \psi_1$ ("globally ψ_1"), $\psi_1 \,\mathsf{U}\, \psi_2$ ("ψ_1 until ψ_2"), and $\psi_1 \,\mathsf{R}\, \psi_2$ ("ψ_1 releases ψ_2") are path formulae over $\mathcal{S}$, and*
- *nothing else is a path formula over $\mathcal{S}$.*

3. *Evolution CTL* (EvoCTL*) is the set of path formulae over $\mathcal{S}$.*

The symbols $\odot$, $\circledcirc$, and $\otimes$ are called life cycle queries. *The symbols A and E are called* universal *and* existential path quantifier. *The symbols X, F, G, U, and R are called* temporal modalities. ♢

Note that the use of terms (cf. Def. 4.3.3) in EvoCTL* is type consistent because we assumed $\mathbb{B}_3$ as the domain boolean terms evaluate to and boolean terms are the atoms of temporal formulae.

Note 4.4.2 (Formulae over Subset-related Signatures)**.** *Let $\mathcal{S}_1 \subseteq \mathcal{S}_2$ be two signatures. Then each EvoCTL* formula over $\mathcal{S}_1$ is also an EvoCTL* formula over $\mathcal{S}_2$.* ♢

Definition 4.4.3 (Bound and Free Variables.)**.** *Let φ be an EvoCTL* formula over signature $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$. A variable $v \in \mathcal{V}$ is said to occur* bound *in φ if and only if each occurrence of v takes place in a sub-formula of φ of the forms $\forall\, v \,.\, \phi$ or $\exists\, v \,.\, \phi$. Otherwise v is called* free.

We use Free(φ) to denote the set of free variables in φ. ♢

Definition 4.4.4 (Variables Under Temporal Operators.)**.** *Let φ be an EvoCTL* formula over signature $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$. A variable $v \in \mathcal{V}$ is said to occur* under *temporal operator X (F, ...) if and only if φ has a sub-formula of the form $\mathsf{X}\, \tilde{\varphi}$ and v occurs in $\tilde{\varphi}$.*

It is said to occur directly *under X if and only if there don't occur any temporal operators in $\tilde{\varphi}$.* ♢

Abbreviations and Binding Priorities

The abbreviations from Section 4.3.1 apply correspondingly to EvoCTL*.

In addition, we shall use the following common abbreviations.

1. $\psi_1 \,\mathsf{W}\, \psi_2$ for $(\mathsf{G}\, \psi_1) \vee (\psi_1 \,\mathsf{U}\, \psi_2)$ ("ψ_1 unless ψ_2")
2. $\mathsf{X}^k\, \psi_1$ for $\underbrace{\mathsf{X} \ldots \mathsf{X}}_{k \text{ times}} \psi_1$, $k \in \mathbb{N}_0$

Binding priorities for EvoCTL* are as follows.

1. Unary temporal connectives (X, G, F) bind equally strong to negation ($\neg$).
2. The binary temporal connectives (U, R, W) bind less tightly than the unary ones and more tightly than binary logical connectives like conjunction and disjunction.
3. The path quantifiers A and E bind least tightly.

Using the symbol from (4.29) for terms on page 89, for EvoCTL* we have

$$\{\neg, \mathsf{X}, \mathsf{G}, \mathsf{F}\} \lll \{\mathsf{U}, \mathsf{R}, \mathsf{W}\} \lll \{\wedge\} \lll \{\vee, \dot{\vee}\} \lll \{\rightarrow, \leftrightarrow\} \lll \{\mathsf{A}, \mathsf{E}\}. \tag{4.44}$$

Named Fragments of EvoCTL*

Actually, we're only interested in three fragments of EvoCTL*, namely the one that doesn't refer to evolution (by quantifying only locally, within terms) and the one that does (by accessing logical variables in the scope of different temporal operators).

Furthermore, as soon as we're studying abstractions, we're interested in the universally quantified (both, paths and logical variables) fragment. For completeness and to sort in the just named three, in the following we name the commonly used fragments of temporal logic in terms of EvoCTL*.

Definition 4.4.5 (Named Fragments of EvoCTL*)**.**

1. *An EvoCTL* formula only consisting of a term is called* state property *(or* state invariant*), otherwise it's called* temporal property*.*
2. *An EvoCTL* formula where quantification only occurs in terms is called* local topology property*, otherwise it's called* evolution property*.*
3. *The set of EvoCTL* formulae comprising only universal quantifiers, both path and other quantifiers, is called AEvoCTL*, the set of EvoCTL* formulae comprising only existential quantifiers is called EEvoCTL*.*
4. *An EvoCTL* formula which comprises only identity variables is called* identity property*, an EvoCTL* formula which comprises only destiny variables is called* destiny property*.*
5. *The set of EvoCTL* formulae of the form $\mathsf{A}\,\varphi$ where φ doesn't comprise further path quantifiers is called FO-LTL. The set of EvoCTL* formulae where each occurrence of the temporal modalities X, F, G, U, and R is immediately preceded by a path quantifier is called FO-CTL.*
6. *The set of EvoCTL*, FO-LTL, and FO-CTL where terms comprise only 0-ary predicates (also called atomic propositions) and no logical variables are called CTL*, LTL, and CTL, respectively.* ♢

4.4.2. Semantics

Definition 4.4.6 (EvoCTL* Semantics)**.** *Let $\mathcal{S}$ be a signature, M a compatible evolving topology transition system, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M.*

Let ϕ be an EvoCTL state formula over $\mathcal{S}$. Given a state $s \in S(M)$ and an assignment $\theta \in Assign(Free(\phi), s)$ of the free variables of ϕ in s, the valuation of ϕ in s under θ is inductively defined as follows.*

1. $\mathcal{M}[\![t]\!](s,\theta) \quad = \iota[\![t]\!](\mathscr{L}(s),\theta)$

2. $\mathcal{M}[\![\odot\, a]\!](s,\theta) =$

$$\begin{cases} 1 & , \textit{ if } id := \iota[\![a]\!](\mathscr{L}(s),\theta) \textit{ exists, is in } U^{\circledcirc}(s), \textit{ and} \\ & \quad \forall\, r = ('s, s) \in R(M) : \\ & \qquad \text{☼} \overset{r}{\rightsquigarrow}_e id \wedge \forall\, 'id \in Id :' id \overset{r}{\rightsquigarrow}_e id \implies 'id \notin U^{\circledcirc}('s) \\ 0 & , \textit{ if } id := \iota[\![a]\!](\mathscr{L}(s),\theta) \textit{ exists, is in } U^{\circledcirc}(s), \textit{ and} \\ & \quad \exists\, r = ('s, s) \in R(M)\ \exists\, 'id \in Id :' id \overset{r}{\rightsquigarrow}_e id \wedge' id \notin U^{\emptyset}('s) \\ 1/2 & , \textit{ otherwise} \end{cases}$$

3. $\mathcal{M}[\![\neg\phi]\!](s,\theta) \quad = 1 - \mathcal{M}[\![\phi]\!](s,\theta)$

4. $\mathcal{M}[\![\phi_1 \vee \phi_2]\!](s,\theta) \quad = \max(\mathcal{M}[\![\phi_1]\!](s,\theta), \mathcal{M}[\![\phi_2]\!](s,\theta))$

5. $\mathcal{M}[\![\phi_1 \wedge \phi_2]\!](s,\theta) \quad = \min(\mathcal{M}[\![\phi_1]\!](s,\theta), \mathcal{M}[\![\phi_2]\!](s,\theta))$

6. $\mathcal{M}[\![\forall\, x : T \,.\, \phi]\!](s,\theta) \quad = \min\{\mathcal{M}[\![\phi]\!](s, \theta[x \mapsto id]) \mid id \in Id\}$

7. $\mathcal{M}[\![\exists\, x : T \,.\, \phi]\!](s,\theta) \quad = \max\{\mathcal{M}[\![\phi]\!](s, \theta[x \mapsto id]) \mid id \in Id\}$

8. $\mathcal{M}[\![\forall\, \boldsymbol{x} : T \,.\, \psi_1]\!](s,\theta) \quad = \min\{\mathcal{M}[\![\psi_1]\!](\pi, \theta[\boldsymbol{x} \mapsto id]) \mid id \in U^{\circledcirc}(s)\}$

9. $\mathcal{M}[\![\exists\, \boldsymbol{x} : T \,.\, \psi_1]\!](s,\theta) \quad = \max\{\mathcal{M}[\![\psi_1]\!](\pi, \theta[\boldsymbol{x} \mapsto id]) \mid id \in U^{\circledcirc}(s)\}$

10. $\mathcal{M}[\![\mathsf{A}\, \psi]\!](s,\theta) \quad = \min\{\mathcal{M}[\![\psi]\!](\pi, \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n]) \mid$
    $\pi \in \Pi_s(M), \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), i = 0, \ldots, n\}$
    where $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are the destiny variables obtaining non-empty evolution chains from θ, i.e. $\theta(\boldsymbol{x}_i) \neq \varepsilon$, $1 \leq i \leq n$

11. $\mathcal{M}[\![\mathsf{E}\, \psi]\!](s,\theta) \quad = \max\{\mathcal{M}[\![\psi]\!](\pi, \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n]) \mid$
    $\pi \in \Pi_s(M), \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), i = 0, \ldots, n\}$
    with $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ as in the previous case

Let ψ be an EvoCTL path formula over $\mathcal{S}$. Given a path $\pi \in \Pi(M)$ and an assignment $\theta \in Assign(Free(\psi), \pi)$ of the free variables of ψ in π, the valuation of ψ on π under θ is inductively defined as follows.*

12. $\mathcal{M}[\![\phi]\!](\pi,\theta) = \mathcal{M}[\![\phi]\!](\pi^0,\theta)$

13. $\mathcal{M}\llbracket \otimes a \rrbracket(\pi, \theta) =$

$$\begin{cases} 1 & , \textit{ if } id := \iota\llbracket a \rrbracket(\mathscr{L}(\pi^0), \theta) \textit{ exists, is in } U^{\circledcirc}(\pi^0)\textit{, and is not in} \\ & \quad \mathrm{dom}(e\langle \pi^0, \pi^1 \rangle) \textit{ or } e\langle \pi^0, \pi^1 \rangle(id) \in U^{\not\circ}(\pi^0) \setminus U^{\circledcirc}(\pi^0) \\ 0 & , \textit{ if } id := \iota\llbracket a \rrbracket(\mathscr{L}(\pi^0), \theta) \textit{ exists, is in } U^{\circledcirc}(\pi^0)\textit{, and is in} \\ & \quad \mathrm{dom}(e\langle \pi^0, \pi^1 \rangle) \textit{ and } e\langle \pi^0, \pi^1 \rangle(id) \in U^{\circledcirc}(\pi^0) \setminus U^{\not\circ}(\pi^0) \\ 1/2 & , \textit{ otherwise} \end{cases}$$

14. $\mathcal{M}\llbracket \neg\psi_1 \rrbracket(\pi, \theta) = 1 - \mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta)$

15. $\mathcal{M}\llbracket \psi_1 \vee \psi_2 \rrbracket(\pi, \theta) = \max(\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta), \mathcal{M}\llbracket \psi_2 \rrbracket(\pi, \theta))$

16. $\mathcal{M}\llbracket \psi_1 \wedge \psi_2 \rrbracket(\pi, \theta) = \min(\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta), \mathcal{M}\llbracket \psi_2 \rrbracket(\pi, \theta))$

17. $\mathcal{M}\llbracket \forall\, x : T \,.\, \psi_1 \rrbracket(\pi, \theta) = \min\{\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta[x \mapsto id]) \mid id \in Id\}$

18. $\mathcal{M}\llbracket \exists\, x : T \,.\, \psi_1 \rrbracket(\pi, \theta) = \max\{\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta[x \mapsto id]) \mid id \in Id\}$

19. $\mathcal{M}\llbracket \forall\, \boldsymbol{x} : T \,.\, \psi_1 \rrbracket(\pi, \theta) = \min\{\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta[\boldsymbol{x} \mapsto \delta]) \mid \delta \in \Delta(U^{\circledcirc}(\pi^0), \pi)\}$

20. $\mathcal{M}\llbracket \exists\, \boldsymbol{x} : T \,.\, \psi_1 \rrbracket(\pi, \theta) = \max\{\mathcal{M}\llbracket \psi_1 \rrbracket(\pi, \theta[\boldsymbol{x} \mapsto \delta]) \mid \delta \in \Delta(U^{\circledcirc}(\pi^0), \pi)\}$

21. $\mathcal{M}\llbracket \mathsf{X}\, \psi_1 \rrbracket(\pi, \theta) =$

$$\begin{cases} 1 & , \textit{ if } \varepsilon \notin (\theta/1)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/1, \theta/1) = 1 \\ 0 & , \textit{ if } \varepsilon \notin (\theta/1)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/1, \theta/1) = 0 \\ 1/2 & , \textit{ otherwise} \end{cases}$$

22. $\mathcal{M}\llbracket \mathsf{F}\, \psi_1 \rrbracket(\pi, \theta) =$

$$\begin{cases} 1 & , \textit{ if there is a } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/k, \theta/k) = 1 \\ & \quad \textit{and for all } j < k\textit{, } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/j, \theta/k) = 0 \\ 0 & , \textit{ if for all } k \in \mathbb{N}_0\textit{,} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1)) \textit{ implies } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/j, \theta/k) = 0 \\ 1/2 & , \textit{ otherwise} \end{cases}$$

23. $\mathcal{M}\llbracket \mathsf{G}\, \psi_1 \rrbracket(\pi, \theta) =$

$$\begin{cases} 1 & , \textit{ if for all } k \in \mathbb{N}_0\textit{,} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1)) \textit{ implies } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/k, \theta/k) = 1 \\ 0 & , \textit{ if there is a } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/k, \theta/k) = 0 \\ & \quad \textit{and for all } j < k\textit{, } \mathcal{M}\llbracket \psi_1 \rrbracket(\pi/j, \theta/k) = 1 \\ 1/2 & , \textit{ otherwise} \end{cases}$$

24. $\mathcal{M}[\![\psi_1 \mathsf{U}\, \psi_2]\!](\pi, \theta) =$

$$\begin{cases} 1 & , \textit{if there is a } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_2)) \textit{ and } \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 1 \\ & \quad \textit{and for all } 0 \leq j < k, \\ & \quad \varepsilon \notin (\theta/j)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}[\![\psi_1]\!](\pi/j, \theta/j) = 1 \\ 0 & , \textit{if for all } k \in \mathbb{N}_0, \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1 \mathsf{U}\, \psi_2)) \textit{ and } \mathcal{M}[\![\psi_1]\!](\pi/k, \theta/k) = 1, \textit{ and} \\ & \quad \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 0, \\ & \quad \textit{or there is } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1 \mathsf{U}\, \psi_2)) \textit{ and} \\ & \quad \mathcal{M}[\![\psi_1]\!](\pi/k, \theta/k) = \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 0, \\ & \quad \textit{and for all } j < k, \\ & \quad \mathcal{M}[\![\psi_1]\!](\pi/j, \theta/j) = 1 \textit{ and } \mathcal{M}[\![\psi_2]\!](\pi/j, \theta/j) = 0, \\ 1/2 & , \textit{otherwise} \end{cases}$$

25. $\mathcal{M}[\![\psi_1 \mathsf{R}\, \psi_2]\!](\pi, \theta) =$

$$\begin{cases} 1 & , \textit{if there is a } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1) \cup \textit{Free}(\psi_2)) \textit{ and } \mathcal{M}[\![\psi_1]\!](\pi/k, \theta/k) = 1 \textit{ and} \\ & \quad \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 0 \\ & \quad \textit{and for all } 0 \leq j < k,\ \mathcal{M}[\![\psi_1]\!](\pi/j, \theta/j) = 0, \\ & \quad \textit{or for all } k \in \mathbb{N}_0, \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1) \cup \textit{Free}(\psi_2)) \textit{ and } \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 1, \\ 0 & , \textit{if there is a } k \in \mathbb{N}_0 \textit{ such that} \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_2)) \textit{ and } \mathcal{M}[\![\psi_2]\!](\pi/k, \theta/k) = 0, \\ & \quad \textit{and for all } 0 \leq j \leq k, \\ & \quad \varepsilon \notin (\theta/k)(\textit{Free}(\psi_1)) \textit{ and } \mathcal{M}[\![\psi_1]\!](\pi/j, \theta/j) = 0, \\ 1/2 & , \textit{otherwise} \end{cases}$$

Recall that, for instance, $(\theta/1)(\textit{Free}(\psi_1))$ denotes the set-extension of $\theta/1$, that is, the set $\{\theta/1(v) \mid v \in \textit{Free}(\psi_1)\}$ of all semantical values assigned by $\theta/1$ to any variable from $\textit{Free}(\psi_1)$. ♢

Note 4.4.7. *The definition of $\otimes\, t$ is stronger than the definition of disappearance along a transition from Chapter 3.*

The latter calls an individual already disappearing if it evolves into something non-alive, that is, into an individual from $U^{\not\circledcirc}$. The former in addition requires it not *to evolve into something alive, i.e. into an individual from $U^{\circledcirc}$. This difference will be relevant in the section on abstraction where abstract nodes may have the property to be both, alive and non-alive.* ♢

Satisfaction Relation

As we're dealing with three-valued logic, we define the notions of a systems satisfying or not satisfying a formula explicitly. Note that, in contrast to the typical setting with Kripke structures, there remains a case in the middle: we may neither be able to say that the system definitely satisfies or definitely doesn't satisfy a formula.

Definition 4.4.8 (Satisfaction Relation)**.** *Let M be an ETTS compatible with signature $\mathcal{S}$ and let $\mathcal{M}$ be a canonical structure of $\mathcal{S}$ wrt. M.*

1. *Given a state formula ϕ, a state $s \in S(M)$ of M, and an assignment θ of the free variables of ϕ in s, we say*
   - *s* satisfies *ϕ under θ,*
     denoted by $M, s, \theta \models_{\mathcal{M}} \phi$, iff $\mathcal{M}[\![\phi]\!](s, \theta) = 1$ and
   - *s* doesn't satisfy *ϕ under θ,*
     denoted by $M, s, \theta \not\models_{\mathcal{M}} \phi$, iff $\mathcal{M}[\![\phi]\!](s, \theta) = 0$.
2. *Given a path formula ψ, a path $\pi \in \Pi(M)$ of M, and an assignment θ of the free variables of ψ in π, we say*
   - *π* satisfies *ψ under θ,*
     denoted by $M, \pi, \theta \models_{\mathcal{M}} \psi$, iff $\mathcal{M}[\![\psi]\!](\pi, \theta) = 1$ and
   - *π* doesn't satisfy *ψ under θ,*
     denoted by $M, \pi, \theta \not\models_{\mathcal{M}} \psi$, iff $\mathcal{M}[\![\psi]\!](\pi, \theta) = 0$.

We say, the transition system M satisfies *a given EvoCTL* formula φ, denoted by*

$$M \models_{\mathcal{M}} \varphi, \tag{4.45}$$

if and only if $M, s, \theta \models_{\mathcal{M}} \varphi$ for all initial states $s \in S_0(M)$ and all assignments $\theta \in Assign(Free(\varphi))$ in s; we say, M doesn't satisfy *φ, denoted by*

$$M \not\models_{\mathcal{M}} \varphi, \tag{4.46}$$

if and only if there is an initial state $s \in S_0$ and an assignment $\theta \in Assign(Free(\varphi))$ in s such that $M, s, \theta \not\models_{\mathcal{M}} \varphi$.

The structure index can be omitted if it is clear from the context. ◇

A consequence of this separate definitions of '$\models$' and '$\not\models$' is

$$M, s, \theta \models \neg\phi \leftrightarrow M, s, \theta \not\models \phi, \tag{4.47}$$

which is not very surprising, but also

$$M, s, \theta \models \neg\phi \not\leftrightarrow \neg(M, s, \theta \models \phi), \tag{4.48}$$

which has to be considered carefully – the reason is that we can't say anything if an individual disappears while we still want to refer to it, so we're actually treating the unexpected disappearance of individuals.

This is in our opinion better than [190, 191], as it makes loss of individuals more explicit; in ETL one has to double-check all outcomes for whether the contrary holds, too.

The Model-Checking Problem of EvoCTL* and ETTS

The model-checking problem of EvoCTL* is, given an ETTS M and an EvoCTL* formula φ, decide whether $M \models \varphi$ holds.

The problem is in general considered undecidable, as other undecidability results [4, 5, 165] are likely to carry over, because it is easily imaginable how to encode a two-counter machine or a Turing machine tape in a topology.

An interesting question, is how the decidability problem depends on particular features of the higher-level languages (cf. Chapter 9) like, for instance, communication or the single-link property.

4.4.3. Fairness

The semantics of EvoCTL* with respect to fair ETTS is very similar to the just presented semantics with respect to an ordinary ETTS . The main difference is that path quantification only considers fair paths and in addition state propositions hold in a state only if there is a fair path starting at that state.

Definition 4.4.9 (Fair EvoCTL* Semantics)**.** *Let M be a fair ETTS compatible with signature $\mathcal{S}$ and let $\mathcal{M}$ be a canonical structure of $\mathcal{S}$ wrt. M. Let φ be an EvoCTL* formula over $\mathcal{S}$.*

Then the fair valuation *of φ under an assignment θ of the free variables in φ, denoted by $\mathcal{M}[\![\varphi]\!]_F(s,\theta)$ if φ is a state formula and by $\mathcal{M}[\![\varphi]\!]_F(\pi,\theta)$ if φ is a path formula, is defined inductively by the set of rules obtained from Def. 4.4.6 by replacing (1), (10), and (11) by*

$$1\ \mathcal{M}[\![t]\!]_F(s,\theta) = \begin{cases} s[\![t]\!](\theta) & \text{, } \textit{if } \Pi_s^F(M) \neq \emptyset \\ 1/2 & \text{, } \textit{otherwise} \end{cases}$$

$$10\ \mathcal{M}[\![\mathsf{A}\,\psi]\!]_F(s,\theta) = \max\big((\Pi_s^F(M) = \emptyset), \min_{\pi\in\Pi_s^F(M)} \mathcal{M}[\![\psi]\!](\pi,\theta)\big)$$

$$11\ \mathcal{M}[\![\mathsf{E}\,\psi]\!]_F(s,\theta) = \min\big((\Pi_s^F(M) \neq \emptyset), \max_{\pi\in\Pi_s^F(M)} \mathcal{M}[\![\psi]\!](\pi,\theta)\big) \qquad \diamondsuit$$

The definition of the satisfaction relation remains unchanged from Section 4.4.2. We write, for example, $M \models_F \varphi$ to denote that the fair ETTS M satisfies the EvoCTL* formula φ.

4.4.4. Semantical Equivalence

Definition 4.4.10 (Semantical Equivalence of Formulae)**.** *Let φ_1 and φ_2 be EvoCTL* formulae over signature $\mathcal{S}$, both state or both path formulae. We say φ_1 and φ_2 are* semantically equivalent*, denoted by*

$$\varphi_1 \equiv \varphi_2, \tag{4.49}$$

if and only if their valuation is independent from the employed ETTS and structure, that is, if

$$\mathcal{M}[\![\varphi_1]\!](s,\theta) = \mathcal{M}[\![\varphi_2]\!](s,\theta) \tag{4.50}$$

in case both are state formulae and

$$\mathcal{M}[\![\varphi_1]\!](\pi,\theta) = \mathcal{M}[\![\varphi_2]\!](\pi,\theta) \tag{4.51}$$

for any evolving topology transition system M compatible with $\mathcal{S}$, any canonical structure $\mathcal{M}$ of $\mathcal{S}$ wrt. M, any state or path of M, and any assignment θ.

We say φ_1 and φ_2 are semantically equivalent under property "prop"*, denoted by*

$$\varphi_1 \equiv_{prop} \varphi_2, \tag{4.52}$$

if and only if their valuation is independent from the employed ETTS and structure as long as they have property "prop". An example for such a property is linearity of the evolution relation of ETTS. ♢

Negative Normal Form

The following results will be useful for proofs of the behaviour of EvoCTL* formulae under abstraction.

Definition 4.4.11 (Negative Normal Form)**.** *An EvoCTL* formula φ is said to be in* negative normal form *if and only if the negation operator ("$\neg$") only appears in terms or in front of aliveness queries, that is, if $\neg\varphi_0$ is a sub-formula of φ, than either $\varphi_0 \in Term_{\mathbb{B}}(\mathcal{S})$, or $\varphi_0 = \odot\,\varphi_1$, or $\varphi_0 = \otimes\,\varphi_1$.* ♢

Lemma 4.4.12 (Negation-related Equivalences)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature. Let $x \in \mathcal{V}$ be a logical variable of an identity type T and $\boldsymbol{x} \in \mathcal{V}$ a logical variable of an evolution chain type $\boldsymbol{T}$. The we have the following equivalences for EvoCTL* formulae φ_1, φ_2, state formulae ϕ, and path formulae ψ_1, ψ_2 over $\mathcal{S}$.*

1. $\neg\neg\varphi_1 \equiv \varphi_1$, $\neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2$, $\neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2$.
2. $\neg\forall\, v : \tau \,.\, \varphi_1 \equiv \exists\, v : \tau \,.\, \neg\varphi_1$, $\neg\exists\, v : \tau \,.\, \varphi_1 \equiv \forall\, v : \tau \,.\, \neg\varphi_1$,
3. $\neg\,\mathsf{A}\,\psi \equiv \mathsf{E}\,\neg\psi$, $\neg\,\mathsf{E}\,\psi \equiv \mathsf{A}\,\neg\psi$,
4. $\neg\,\mathsf{X}\,\psi_1 \equiv \mathsf{X}\,\neg\psi_1$.
5. $\neg\,\mathsf{F}\,\psi_1 \equiv \mathsf{G}\,\neg\psi_1$, $\neg\,\mathsf{G}\,\psi_1 \equiv \mathsf{F}\,\neg\psi_1$,
   $\neg(\psi_1\,\mathsf{R}\,\psi_2) \equiv \neg\psi_1\,\mathsf{U}\,\neg\psi_2$, $\neg(\psi_1\,\mathsf{U}\,\psi_2) \equiv \neg\psi_1\,\mathsf{R}\,\neg\psi_2$. ♢

Proof. Cases 1., 2., and 3. are obvious by definition in terms of maximum and minimum over $\{0, 1, {}^1\!/\!_2\}$ and Note 2.5.1; for instance,

$$\begin{aligned}\neg(\varphi_1 \vee \varphi_2) &= 1 - \max(\mathcal{M}[\![\phi_1]\!](s,\theta), \mathcal{M}[\![\phi_2]\!](s,\theta)) \\ &= \min(1 - \mathcal{M}[\![\phi_1]\!](s,\theta), 1 - \mathcal{M}[\![\phi_2]\!](s,\theta)) = \neg\varphi_1 \wedge \neg\varphi_2.\end{aligned} \tag{4.53}$$

Cases 4. and 5. are clear by close inspection of Def. 4.4.6. In case 4., the two definite cases are the logical inverse of each other and the indefinite case remains indefinite under negation. Case 5. follows similarly but involves pairs of operators. Use that, for instance, the definition of the positive definite case of "F" is the logical inverse of the negative definite case of "G". □

Corollary 4.4.13 (Negative Normal Form)**.** *For each EvoCTL* formula φ there is a semantically equivalent EvoCTL* formula $\tilde{\varphi}$, which is in negative normal form.* ♢

Proof. Recursive application of the equivalences from Lemma 4.4.12 to largest subformulae comprising negation. The procedure terminates rule application moves negation operators inwards. □

Minimal Syntax

The following properties come in handy for proofs of monotonicity properties.

Lemma 4.4.14 (Other Equivalences)**.** *Let ψ_1 and ψ_2 be EvoCTL* path formulae over signature $\mathcal{S}$. Then*

1. $\mathsf{F}\,\psi_1 \equiv 1\,\mathsf{U}\,\psi_1$.
2. $\psi_1\,\mathsf{R}\,\psi_2 \equiv \mathsf{G}\,\psi_2 \vee ((\neg\psi_1)\,\mathsf{U}\,(\psi_1 \wedge \neg\psi_2))$, *in particular* $\mathsf{G}\,\psi \equiv 0\,\mathsf{R}\,\psi$.
3. $\mathsf{F}\,\psi_1 \equiv \psi_1 \vee \mathsf{X}\,\mathsf{F}\,\psi_1$, $\mathsf{G}\,\psi_1 \equiv \psi_1 \wedge \mathsf{X}\,\mathsf{G}\,\psi_1$.
4. $\psi_1\,\mathsf{U}\,\psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \mathsf{X}(\psi_1\,\mathsf{U}\,\psi_2))$. ♢

Proof. We only show case 1. Let M be an evolving topology transition system compatible with $\mathcal{S}$ and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M. Let $\pi \in \Pi(M)$ be a path of M and θ an assignment.

We have

$$\mathcal{M}[\![\mathsf{F}\,\psi_1]\!](\pi,\theta) = 1 \iff \mathcal{M}[\![1\,\mathsf{U}\,\psi_1]\!](\pi,\theta) = 1 \tag{4.54}$$

because both cases require existence of $k \in \mathbb{N}_0$ such that $\mathcal{M}[\![\psi_1]\!](\pi/k, \theta/k) = 1$ and require definite evaluation of ψ_1 for $j < k$. The additional requirement of definite (positive) evaluation of the left-hand side of U is trivially given.

Similarly, we have

$$\mathcal{M}[\![\mathsf{F}\,\psi_1]\!](\pi,\theta) = 0 \iff \mathcal{M}[\![1\,\mathsf{U}\,\psi_1]\!](\pi,\theta) = 0. \tag{4.55}$$

The remaining cases of both sides are the indefinite cases. □

Definition 4.4.15 (Normal EvoCTL*)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature.*

A type-consistent word of the grammar

$$\phi ::= t \mid \odot\, a \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid \forall\, v\,.\,\phi_1 \mid \mathsf{A}\,\psi \tag{4.56}$$

where t is a normal term and $v \in \mathcal{V}$, is called normal *state formula and a type-consistent word of the grammar*

$$\psi ::= \phi \mid \otimes a \mid \neg\psi_1 \mid \psi_1 \wedge \psi_2 \mid \forall\, v \,.\, \psi_1 \mid \mathsf{X}\,\psi_1 \mid \psi_1 \,\mathsf{U}\, \psi_2 \tag{4.57}$$

is called normal *path formula over $\mathcal{S}$.*

Normal EvoCTL is the set of normal path formulae over $\mathcal{S}$.* ♢

Corollary 4.4.16 (Normal EvoCTL*)**.** *For each EvoCTL* formula φ there is a semantically equivalent normal EvoCTL* formula $\tilde{\varphi}$.* ♢

Proof. Recursive application of the equivalences from Lemma 4.4.12 and Lemma 4.4.14 replaces all operators not occurring in Def. 4.4.15 by admitted operators. □

Note that being new (“⊙”) or disappearing (“⊗”) doesn’t reduce to being alive (“⊚”) because a process can disappear now and reappear in the subsequent state. Then it is constantly alive but in two different life-cycles. This is different in other proposed specification logics, for example, in ATL [57] (cf. Section 4.6).

4.4.5. Relation to CTL*

We note without a proof that Def. 4.4.6 is a conservative extension of CTL*. Recall that, by Def. 2.6.3, the class Kripke structures, is a proper sub-class of ETTS. Given an EvoCTL* formula adhering to the well-known syntax of CTL*, that is, in particular without quantification, function symbols, and non-0-ary predicates, Def. 4.4.6 yields the same value on a Kripke structure embedded casted to ETTS than in the classical semantics.

Which is good because it says that our definition is reasonable and a lot of intuitions carry over, at least as long as the intricacies of three-valued logics are kept in mind.

4.4.6. Definiteness and Monotonicity

Technically viewed, monotonicity means that a formula without life-cycle queries evaluates more indefinite in a more indefinite system. In other words, if a system is less defined than expected and the formula cares for less life-cycle issues than it should, then the outcome may be indefinite, which is better than a definite outcome contrary to the intention

Viewed more practical: can we tell from the syntax, which formulae evaluate definite and which in general don’t? Strictly speaking, we can not, at least not for an interesting class of formulae because predicates may evaluate to the indefinite value any time. If we exclude that issue semantically, we can ask what class of formulae is sensitive to indefiniteness due to pre-mature disappearance of actors.

Monotonicity

Note that for first results we use a rather pragmatic than strict definition when requiring that topologies are nearly identical in the sense that they only differ in local state labelling.

One can easily think of relaxations where in addition the absence of individuals is permitted, which is identical to not having a local state for those individuals which are not destinations of links. For now, we avoid the intricacies of this discussion involving links and stick to a stronger definition. It is sufficient to make our point, the nice property of the EvoCTL* semantics given by Lemma 4.4.17.

The following lemma corresponds to Lemma 4.3.6 for terms and is accordingly stronger than Note 4.4.20. It relates the findings of Lemma 4.3.6 for single topologies to sequences of topologies. That is, if two transition systems only differ by the aliveness of individuals, then any EvoCTL* formula evaluates more precise in the more definite transition system.

Similar to $\circledcirc$, life cycle operators are excluded because they of course distinguish different paths in the sense discussed before Lemma 4.3.6. In addition, usage of variables of type $\boldsymbol{T}$ is restricted because they, in combination with the temporal operators based on "U" provide an implicit "sensor" for disappearance because they only evaluate within evolution chains. Similarly, the valuation of quantification over $\boldsymbol{T}$ depends on the set of alive individuals in a state. The lemma is still useful as explained above – and not maximally strong as also explained above.

Lemma 4.4.17 (Monotonicity of Formula Evaluation)**.** *Let $\mathcal{S}$ be a signature, $M_1 \sqsubseteq M_2$ two compatible evolving topology transition systems, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M_1 and M_2.*

Let φ be an EvoCTL state formula over $\mathcal{S}$ without life cycle queries and such that all variables of type $\boldsymbol{T}$ in φ occur free and directly under X. Then*

$$\mathcal{M}_{M_1}[\![\varphi]\!](s,\theta_1) \sqsubseteq \mathcal{M}_{M_2}[\![\varphi]\!](s,\theta_2) \quad \text{or} \quad \mathcal{M}_{M_1}[\![\varphi]\!](\pi,\theta_1) \sqsubseteq \mathcal{M}_{M_2}[\![\varphi]\!](\pi,\theta_2) \tag{4.58}$$

for each state $s \in S(M_1) = S(M_2)$ or path π in M_1 and M_2 if φ is a state or path formula, and assignments $\theta_1 \sqsubseteq \theta_2$. ♢

Proof. See Section A.1. □

The following examples show that Lemma 4.4.17 is close to complete in the sense that *in general* we can't admit live cycle query operators, and neither quantification over variables of an evolution chain type $\boldsymbol{T}$ nor such variables under other temporal operators. They have in common that they have the power to definitely distinguish two ETTS in information order, that is, yield 0 for one and 1 for the other. Using only EvoCTL* constructs admitted in Lemma 4.4.17, the distinction is not definite.

Note 4.4.18 (Monotonicity of Formula Evaluation)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature such that $\mathcal{V}$ comprises at least one logical variable $\boldsymbol{x}$ of an evolution chain type $\boldsymbol{T}$ and x of an identity type T.*

Let φ denote one of the EvoCTL formulae*

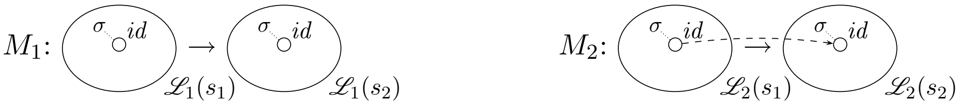


Figure 4.6.: **Example ETTS** supporting the claims of Note 4.4.18.

1. *$\odot\, v$,*
2. *$\otimes\, v$,*
3. *$\circledcirc\, v$,*
4. *$\forall \boldsymbol{x} : T \,.\, \psi$, or*
5. *$\psi_1 \mathsf{U}\, \psi_2$.*

Then there exist two $\mathcal{S}$-compatible ETTS M_1 and M_2, a canonical structure $\mathcal{M}$ of $\mathcal{S}$ wrt. M_1 and M_2, and assignments θ_1, θ_2 such that $M_1 \sqsubseteq M_2$ and $\theta_1 \sqsubseteq \theta_2$, but

$$\mathcal{M}_{M_1}[\![\varphi]\!](s, \theta_1) \not\sqsubseteq \mathcal{M}_{M_2}[\![\varphi]\!](s, \theta_2) \text{ or } \mathcal{M}_{M_1}[\![\varphi]\!](\pi, \theta_1) \not\sqsubseteq \mathcal{M}_{M_2}[\![\varphi]\!](\pi, \theta_2) \tag{4.59}$$

where $s \in S(M_1) = S(M_2)$ and π is a path in M_1 and M_2. ♢

Proof. See Section A.1. □

The gap to be closed for exactness lies in the prerequisites of Note 4.4.18. The pathological cases of no variables in the structure may be admitted in Lemma 4.4.17. We refrain from doing so since in our opinion, the gained insight wouldn't weigh up the technical complication added to the proof of Lemma 4.4.17.

Definiteness

Lemma 4.4.19 (Definite Evaluation of Formulae)**.** *Let $\mathcal{S}$ be a signature, M a compatible ETTS, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M.*

Let φ be an EvoCTL state formula over $\mathcal{S}$ such that*

1. *all variables occurring in t are of an evolution chain type $\boldsymbol{T}$ and bound,*
2. *the interpretation of all function and predicate symbols f and p occurring in φ by ι is total,*
3. *the temporal next operator ("X") doesn't occur in φ.*
4. *for each occurrence of the temporal until operator ("U"), the free variables of the left hand side are a subset of the ones of the right hand side, i.e. $Free\psi_1 \subseteq Free\psi_2$ for any a sub-formula of φ of the form $\psi_1 \mathsf{U}\, \psi_2$.*

Then φ evaluates definite at any state $s \in S(M)$ of M, that is, $\mathcal{M}[\![\varphi]\!](s, \theta) \neq 1/2$, for any assignment θ. ♢

Proof. See Section A.1. □

The strong premises are inherited from Lemma 4.3.8, that is, in particular the practical exclusion of dangling links . The notable finding is that logical variables of an evolution chain type $\boldsymbol{T}$ mostly ensure definite valuation, which is probably the motivation for their introduction in the literature .

Although there are two twists: firstly, the temporal next operator is not admitted (which may not be the greatest loss of all times in asynchronous systems), and secondly, the until operator may still turn indefinite if the left hand side disappear pre-maturely as we'll see in the following note.

Note that the corresponding premise in Lemma 4.4.19 is not necessary, but only sufficient; the necessary condition is absence of pre-mature disappearance, which is a too deeply semantical property to be appropriate in such a syntax oriented lemma as Lemma 4.4.19. Further note, that the premise is trivially satisfied for the globally and the finally operator, thus restriction to these operators yields a "definitely definite" fragment.

The following note is similar to Note 4.3.9.

Note 4.4.20 (Definite Evaluation of Formulae)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature.*

- *Let φ denote any of the EvoCTL* formulae*

  $$\odot \boldsymbol{x} \quad \text{or} \quad \otimes \boldsymbol{x}, \tag{4.60}$$

  $\boldsymbol{x} \in \mathcal{V}$ of an evolution chain type $\boldsymbol{T}$, or an EvoCTL formula the form*

  $$\mathsf{X}\, \psi_1 \tag{4.61}$$

  such that Freeψ_1 comprises at least one logical variable of an evolution chain type $\boldsymbol{T}$.

  Then for all $\mathcal{S}$-compatible ETTS M and canonical structures $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. M, there is an assignment θ of the free variables in φ such that the evaluation of φ is indefinite.

- *Let φ be an EvoCTL* formula of the form*

  $$\mathsf{X}\, \psi_1 \quad \text{or} \quad \psi_1 \,\mathsf{U}\, \psi_2. \tag{4.62}$$

  For certain EvoCTL formulae ψ_1 and ψ_2, there exists a $\mathcal{S}$-compatible ETTS M such that for all canonical structures $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. M, there is an assignment θ of the free variables in φ such that the evaluation of φ is indefinite.*

- *Let φ be an EvoCTL* formula of the form*

  $$\psi_1 \,\mathsf{U}\, \psi_2. \tag{4.63}$$

  For certain EvoCTL formulae ψ_2, there exists a $\mathcal{S}$-compatible ETTS M such that for all canonical structures $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ wrt. M, there is an assignment θ of the free variables in φ such that the evaluation of φ is indefinite.* ♢

Proof. See Section A.1. □

Again, the setting is not hopeless, because we have the possibility to guard subformulae by implications. Together with life cycle queries, it will be definite on the desired cases and indefinite on some border cases.

4.4.7. Destiny vs. Identity Quantification

One of the reasons for having both notions, destiny and identity quantification, included in EvoCTL* is to be able to compare them for expressive power. The basic finding is that we can reduce some destiny quantifications to identity quantification and thus also treat some properties with destiny quantification with the DTR abstraction, which depends on identity quantification as discussed in Chapter 6.

Note that we need a boolean constant denoting “half” in the following (which we intentionally don’t have natively in the logic). Without it, the lemma is still useful because we obtain a biased definition as often used in the literature.

Lemma 4.4.21 (Evolution Chain vs. Identity Quantification)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature such that for each logical variable $\boldsymbol{x} \in \mathcal{V}$ of an evolution chain type $\boldsymbol{T}$ there is a unique logical variable $x_{\boldsymbol{x}} \in \mathcal{V}$ a logical variable of the corresponding identity type T and such that there is a 0-ary predicate $indef \in \mathcal{F}$.*

The syntactical transformation $\mathfrak{f}$ on EvoCTL formulae is defined as follows:*

1. $\mathfrak{f}(\forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \psi_1) := \forall\, x_{\boldsymbol{x}} : T \,.\, \circledcirc\, x_{\boldsymbol{x}} \to \mathfrak{f}(\psi_1)$ *where T corresponds to $\boldsymbol{T}$*

2. $\mathfrak{f}(\psi_1 \,\mathsf{U}\, \psi_2) :=$

$$\begin{aligned}
&\Big(\big(\big((\neg disap_1 \vee \mathsf{X}(\neg disap_2 \wedge \mathfrak{f}(\psi_2))\big) \\
&\qquad \wedge \neg disap_2 \wedge \mathfrak{f}(\psi_1)\big) \,\mathsf{U}\, (\neg disap_2 \wedge \mathfrak{f}(\psi_2))\big) \to 1\Big) \\
&\wedge\Big(\big((\neg disap_1 \wedge \neg disap_2 \wedge \mathfrak{f}(\psi_1)) \,\mathsf{U}\, (disap_2 \wedge \neg\mathfrak{f}(\psi_2))\big) \to 0\Big) \\
&\wedge\Big(\big((\neg disap_1 \wedge \neg disap_2 \wedge \mathfrak{f}(\psi_1)) \,\mathsf{U}\, (disap_1 \wedge \neg\mathfrak{f}(\psi_2))\big) \to indef\Big)
\end{aligned} \tag{4.64}$$

where

$$disap_i := \bigvee_{\boldsymbol{x} \in Free(\psi_i)} \otimes\, x_{\boldsymbol{x}}. \tag{4.65}$$

3. $\mathfrak{f}(\psi_1) = \psi_1[x_{\boldsymbol{x}}/\boldsymbol{x}]$ *otherwise.*

Let φ be an EvoCTL formula over $\mathcal{S}$ without free occurrences of any of the variables $x_{\boldsymbol{x}}$. Then $\varphi \equiv_{lin.,id.\,presv.,indef} \mathfrak{f}(\varphi)$, that is, both sides are equivalent on any linear and identity preserving ETTS if indef is constantly interpreted as the indefinite value 1/2.* ◇

Proof. Definition 1 just rephrases the restriction to alive individuals. The formula 2 distinguishes the three cases: positive, negative, and indefinite by disappearance and explicates the aliveness (or non-disappearance) property of evolution chains (cf. Section B.1). If one of the two sub-formulae ψ_1 or ψ_2 evaluates indefinite pre-maturely, formula (4.64) also turns indefinite.

Equivalence is in general not given if the considered ETTS is not identity preserving, that is, if the evolution chain comprises different identities. Equivalence is not given if the considered ETTS is not linear but splitting or summarising, because variables of an identity type T cannot split to denote two different individuals. □

Corollary 4.4.22 (Identity EvoCTL*)**.** *For each EvoCTL* formula φ there is an EvoCTL* formula $\tilde{\varphi}$, which employs only variables of an identity type T, such that $\varphi \equiv_{lin.,id.\,presv.,indef} \tilde{\varphi}$.* ♢

Proof. Cor. 4.4.16 and Lemma 4.4.21. □

That is, in the case "linear, identity preserving", evolution chains are "only" more concise but can be eliminated, In other settings they are more expressive. All concrete models of our application domain that is, the models *before* applying any abstraction, that we've seen up to now are "linear, identity preserving" or can be treated as such (cf. discussion of fixed vs. variable universe in Chapter 3).

Together with Section 4.4.6, we also learn something about the definiteness of formulae from Cor. 4.4.22. Namely, Section 4.4.6 indicates that formulae employing the temporal globally and finally operators ("G" and "F") and quantified variables of an evolution chain type $\boldsymbol{T}$ evaluate definite as long as the interpretation of predicates or function symbols goes well.

Now Cor. 4.4.22 indicates how we can ensure definite evaluations for formulae over variables of an identity type T. The strategy is to guard functional terms and whole sub-formulae by life cycle queries, thereby making explicit what one expects from the quantified identities. What one expects could be that they are alive as long as the formula applies or that they may also satisfy the formula by disappearing, and for this reason not doing something bad. This backs up our claim from section Section 4.4.6 that the situation is not hopeless. Despite the numerous reasons for indefinite evaluation, there are reasonable EvoCTL* properties for reasonable ETTS.

In the disjoint universe model, identity quantification is only sensible with local topology properties, that is, where quantification only appears in terms. Otherwise it's hardly useful because formulae evaluate indefinite as soon as two or more topologies are involved: in the disjoint universe model, an individual exists in at most one topology, thus formulae evaluate indefinite in one or the other if two or more are involved. Destiny variables are particularly fine with disjoint universes.

4.5. Prenex Normal Forms, Case-Split, and Other Useful Equivalences

In the following, we provide rules to obtain a prenex normal form from some EvoCTL* formulae. This is important as the basic idea of the verification technique we're investigating in the following chapters Chapters 5 and 6 is

- given a formula
$$\varphi := \forall\, x_1 : T_1, \dots, x_n : T_n \,.\, \varphi_1 \tag{4.66}$$
over variables of identity types T_i in universal prenex normal form,

- construct a finite set of representative cases, that is, assignments $\theta_1, \dots, \theta_m$ of the quantified variables,

- then "$\varphi[x_1/\theta_i(x_1), \dots, x_n/\theta_i(x_n)]$" (in parentheses as this kind of substitution is not properly defined yet) is a propositional temporal logic formula and as such amenable to standard CTL*, LTL, or CTL model-checkers,

- given an ETTS M, we'll actually construct a finite-state abstraction $M^\sharp$ depending on θ_i, then the whole task is amenable to standard finite state model-checkers.

As a prerequisite is that formulae are in prenex normal form, the findings of this section gives an indication of the fragment of EvoCTL* that we can treat with the sketched verification technique.

Which is nice but not necessary – Chapter 10 indicates that even the plain fragment of formulae in prenex normal form are useful as they cover a wide range of so-called scenario properties.

The later Sections 4.5.3 and 4.5.4 discuss further equivalences and approaches to certain phenomena carrying over from higher-level languages which will turn out useful when applying the verification technique to higher-level languages like UML/LSC and DCS/METT in Chapters 9 and 10.

4.5.1. Identity Quantification

When quantifying identities, we often have a normal form. The intuitive prerequisite is that the evaluation time and extension of inner quantification is fixed and known beforehand. Which is the case for the modalities "next" and "globally", but not necessarily for "finally".

Lemma 4.5.1 (Prenex Normal Form, Identity)**.** *Let φ_1 and φ_2 be EvoCTL* formulae over signature $\mathcal{S}$. Then*

1. $(\forall\, x : T \,.\, \varphi_1) \land \varphi_2 \equiv \forall\, x : T \,.\, \varphi_1 \land \varphi_2$,

2. $(\forall\, x : T \,.\, \varphi_1) \lor \varphi_2 \equiv \forall\, x : T \,.\, \varphi_1 \lor \varphi_2$,

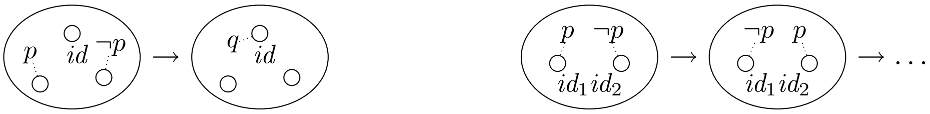
3. $\varphi_1 \to (\forall\, x : T \,.\, \varphi_2) \equiv \forall\, x : T \,.\, \varphi_1 \to \varphi_2$,

4. $\mathsf{X}\, \forall\, x : T \,.\, \varphi_1 \equiv \forall\, x : T \,.\, \mathsf{X}\, \varphi_1$,

5. $\mathsf{G}\, \forall\, x : T \,.\, \varphi_1 \equiv \forall\, x : T \,.\, \mathsf{G}\, \varphi_1$,

6. $(\forall\, x : T \,.\, \varphi_1)\, \mathsf{U}\, \varphi_2 \equiv \forall\, x : T \,.\, (\varphi_1\, \mathsf{U}\, \varphi_2)$, *and*

7. $\varphi_1\, \mathsf{R}\, (\forall\, x : T \,.\, \varphi_2) \equiv \forall\, x : T \,.\, (\varphi_1\, \mathsf{R}\, \varphi_2)$.



(a) M_1.

(b) M_2.

Figure 4.7.: **Example ETTS** supporting the claims of Note 4.5.2.

For M over a finite set Id of identities, we also have

8. $\mathsf{F}\,\mathsf{G}\,\forall\, x : T \,.\, \varphi_1 \equiv \forall\, x : T \,.\, \mathsf{F}\,\mathsf{G}\,\varphi_1$. ♢

Proof. See Section A.1. □

For completeness, we recall examples where a prenex normal form doesn't exist. What the examples have in common is that they address the whole set of identities *at once*, not individually, and where the point in time where the formula under quantification holds is not fixed as it is for the temporal next and globally operators ("X" and "G").

Note 4.5.2 (No Prenex Normal Form, Identity)**.** *Let φ_1 and φ_2 be EvoCTL* formulae over signature $\mathcal{S}$. Then (in general)*

1. $\neg\forall\, x : T \,.\, \varphi_1 \not\equiv \forall\, x : T \,.\, \neg\varphi_1$,

2. $(\forall\, x : T \,.\, \varphi_1) \to \varphi_2 \not\equiv \forall\, x : T \,.\, \varphi_1 \to \varphi_2$,

3. $\mathsf{F}\,\forall\, x : T \,.\, \varphi_1 \not\equiv \forall\, x : T \,.\, \mathsf{F}\,\varphi_1$,

4. $\varphi_1\,\mathsf{U}\,(\forall\ : Tx \,.\, \varphi_2) \not\equiv \forall\, x : T \,.\, (\varphi_1\,\mathsf{U}\,\varphi_2)$, *and*

5. $(\forall\, x : T \,.\, \varphi_1)\,\mathsf{R}\,\varphi_2 \not\equiv \forall\, x : T \,.\, (\varphi_1\,\mathsf{R}\,\varphi_2)$.

6. $\mathsf{F}\,\mathsf{G}\,\forall\, x : T \,.\, \varphi_1 \not\equiv \forall\, x : T \,.\, \mathsf{F}\,\mathsf{G}\,\varphi_1$. ♢

Proof. By counter-examples.

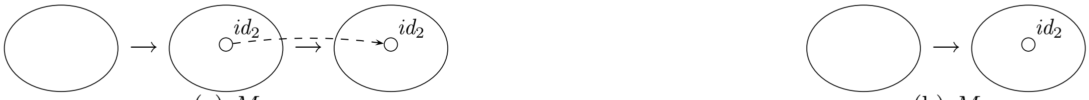
1. Lemma 4.4.12.

2. Consider $(\forall\, x : T \,.\, p(x)) \to q(y)$ vs. $(\forall\, x : T \,.\, p(x) \to q(y))$. The latter holds in M_1 from Figure 4.7 with $\theta = \{y \mapsto id\}$, the former doesn't hold.

3. Consider $\mathsf{F}\,\forall\, x : T \,.\, p(x)$ vs. $\forall\, x : T \,.\, \mathsf{F}\,p(x)$. The latter holds in M_2 from Figure 4.7, the former doesn't.

4. Similar to the previous case.

5. Similar to the previous case.



(a) M_1 (b) M_2

Figure 4.8.: **Example ETTS** supporting the claims of Lemma 4.5.3.

6. The right hand side in general doesn't imply the left hand side if there are infinitely many identities in *Id*. Choosing to enumerate identities from *Id* with natural numbers, we can construct an ETTS with a path where identity id_k satisfies

$$(\neg\varphi_1) \wedge \cdots \wedge (\mathsf{X}^k \neg\varphi_1) \wedge (\mathsf{X}^{k+1} \, \mathsf{G} \, \varphi_1) \tag{4.67}$$

for all $k \in \mathbb{N}_0$, that is, each identity finally globally satisfies φ_1 but there is no point in time where all identities have satisfy φ_1.

□

Note that one conclusion from the above is that *bounded discrete time* modalities have a normal form because they can be explained in terms of finally many "X's". This is in general good news, but it doesn't gain us much when turning to an interleaving semantics in Chapter 9.

4.5.2. Evolution Chain Quantification

Quantification over evolution chains, that is, logical variables of type $\boldsymbol{T}$, in general don't have a prenex normal form. The reason is simply that the domain of evolution chains depends on the state where the quantification evaluates.

This is different from identities, the set *Id* is constant. For example, consider M_2 in Figure 4.8(b). In the left state, there are no individuals thus quantifiers evaluate trivially in this state. In the right state, there is at least one individual. The following lemma states a weak claim in this direction and proves it by a simple example.

The situation would become more intricate if we allowed a prenex normal form including logical variables of an identity type T. We conjecture that the situation then is similar to the one discussed in the previous section on prenex normal forms with variables of type T. Namely that the existence of a normalisation depends on whether we can exactly characterise the point in time and extension of the set of individuals.

Lemma 4.5.3 (No Prenex Normal Form, Destiny)**.** *Let φ be an EvoCTL* formula over signature $\mathcal{S}$ such that all logical variables occurring in φ are of an evolution chain type $\boldsymbol{T}$ and at least one such variable does occur.*

Then there needn't be any EvoCTL formula $\tilde{\varphi}$ in prenex normal form which is equivalent to φ and only employs variables of type $\boldsymbol{T}$.*

Proof. For example, consider

$$\varphi := \mathsf{A}\,\mathsf{X}\,\forall\,\boldsymbol{x} : T \,.\, \otimes \boldsymbol{x}. \tag{4.68}$$

If there were an EvoCTL* formula $\tilde{\varphi}$ in prenex normal form equivalent to φ, then it either has the form

$$\tilde{\varphi}_1 = \forall\,\boldsymbol{x}_1 : T_1 \,.\, \varphi_1 \tag{4.69}$$

or

$$\tilde{\varphi}_2 = \exists\,\boldsymbol{x}_2 : T_2 \,.\, \varphi_2. \tag{4.70}$$

In the former case, consider the ETTS M_1 from Figure 4.8. Then we have

$$\mathcal{M}_{M_1}[\![\varphi]\!](s,\theta) = 0 \neq 1 = \mathcal{M}_{M_1}[\![\tilde{\varphi}_1]\!](s,\theta) \tag{4.71}$$

and

$$\mathcal{M}_{M_2}[\![\varphi]\!](s,\theta) = 1 \neq 0 = \mathcal{M}_{M_2}[\![\tilde{\varphi}_2]\!](s,\theta) \tag{4.72}$$

for canonical structures $\mathcal{M}_{M_1}$ and $\mathcal{M}_{M_2}$ of M_1 and M_2 and any assignment θ yielding a definite valuation of φ_1 and φ_2. Otherwise, the right hand sides in (4.71) and (4.72) evaluate to the indefinite value $1/2$, and as such also different from the left hand sides. □

4.5.3. Case-Split

Case-split in the sense of [127] is a simple quantifier *introduction* rule, which could be written as.

$$\frac{\varphi}{\forall\, d \in \mathcal{D}(x) \,.\, x = d \to \varphi}, x \text{ a program variable} \tag{4.73}$$

The basic idea is that

$$\forall\, x \,.\, \varphi \tag{4.74}$$

holds if and only if

$$\forall\, x, y \,.\, *(\lambda(x)) = y \to \varphi \tag{4.75}$$

holds, because whatever value $\lambda(x)$ may have when φ is evaluated in (4.74), we know from (4.75) that φ holds in that case.

In the other direction we can consider the cases in which φ is evaluated in (4.74), and find that the navigation expression has *some* value in each case. Maybe only few values are reachable, maybe only a single one, maybe all. For the values that are actually not assumed, (4.74) holds trivially and for the others, it holds because of (4.75).

The abstraction technique investigated in Chapters 6 and 8 employs a heuristics to derive the minimal reasonable abstraction from the formula, more precise, from the number of quantified variables. Consequently, the introduction of new quantified variables provides a way to refine the abstraction, and case splits are one particular way. The informal intuition behind this refinement is, that we refine

"for all x, property", (4.76)

where the concrete object denoted by x may have links into an abstract part of the model to

$$\text{“}\textit{for all } x,\ y,\ \textit{where } x \textit{ has a particular relation to } y,\ \textit{property”},\tag{4.77}$$

where we focus on cases in which x has links within the (now extended) concrete part of the abstract model.

The heuristics and the refinement strategy is introduced in [127].

Recall from Chapter 1 that [127] considers parameterised systems, or array programs (cf. Chapter 9), where each variable or array field always carries some value. In our setting of ETTS, we want to introduce case-splits on links, which needn't be present. So their domain (in case of single-links) is not only the set of identities *Id*, but also the additional case that they're not present at all.

For these reasons, we introduce case-splits for valid navigation expression. We'll discuss after Lemma 4.5.5 why this restriction doesn't necessarily render the approach useless.

Definition 4.5.4 (Valid Navigation Expression)**.** *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature, $M = (S, S_0, R)$ a $\mathcal{S}$-compatible ETTS, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M.*

Let a be a navigation expression over $\mathcal{S}$, i.e. of an identity type T. We call ‘a’ a valid navigation expression *in state $s \in S$ under assignment θ if and only if it is defined, i.e. if*

$$\iota[\![a]\!](\mathscr{L}(s), \theta) \in \mathit{Id}.\tag{4.78}$$

It is called valid *in state s or path π if and only if it is valid in s or π under all assignments $\theta \in \mathit{Assign}(\mathcal{V}, s)$, and* valid *in M if and only if it is valid in all states $s \in S$.* ♢

Lemma 4.5.5 (Case-Split)**.** *Let φ_1 and φ_2 be EvoCTL* formulae over signature $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$, let x be a logical variable of type T not occurring freely in φ_1 or φ_2, and let a denote a navigation expression over $\mathcal{S}$ of type T.*

Let M be a $\mathcal{S}$-compatible ETTS, $s \in S(M)$ a state, and θ an assignment in a structure $\mathcal{M}$ of $\mathcal{S}$ canonical wrt. M. Then

1. $\varphi_1 \equiv_{a\ \mathit{valid}} \forall\, x : T \,.\, a = x \rightarrow \varphi_1$,
2. $\varphi_1 \equiv^{s}_{a\ \mathit{valid}} \forall\, x : T \,.\, a = x \rightarrow \varphi_1$, *and*
3. $\varphi_1 \equiv^{s,\theta}_{a\ \mathit{valid}} \forall\, x : T \,.\, a = x \rightarrow \varphi_1$. ♢

Proof. See Section A.1. □

The premises of Lemma 4.5.5 pose harsh restrictions for full equivalence. The navigation expression the case-split should be based on has to be valid. This is easy in static or finite systems, but far from easy in the dynamic setting we consider. Yet there is hope.

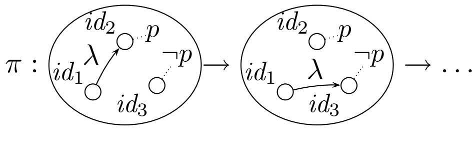


Figure 4.9.: **Temporal Evaluation** of navigation expressions.

First of all, we can sometimes know for particular situations of dynamic individuals whether there are links or not. Then in order to avoid indefinite valuation, our formulae will typically have the form of an implication where the quantified variables are tested for being alive and also to be in a certain configuration. That certain configurations guarantee existence of links is amenable to static analyses like [7, 12].

Secondly, the situation is friendlier when considering M with only single links where undefinedness is encoded by a particular designated value $id_0 \in Id$. Then is is sufficient to know aliveness of an individual denoted by a logical variable x, this then implies that a one-step navigation expression starting at x certainly has a value from Id . The following simple note will be of help in such situations, namely if t_1 characterises a certain configuration under which validity of navigation expression a is known, then we can apply a case-split to obtain the right hand side.

Note 4.5.6 (Case-Split). *Let $\mathcal{S}$ be a signature, let*

$$t_1 \rightarrow \varphi_1 \tag{4.79}$$

be an EvoCTL formula over $\mathcal{S}$. Let M be a compatible ETTS and let a be a navigation expression such that t_1 implies validity of a in M.*

Then we can apply a case-split, that is,

$$t_1 \rightarrow \varphi_1 \equiv \forall\, x : T \,.\, a = x \rightarrow t_1 \rightarrow \varphi_1. \tag{4.80}$$

♢

Proof. Via $t_1 \rightarrow a \rightarrow \varphi_1 \equiv t_1 \rightarrow a \rightarrow \varphi_1$. □

The following notes something trivial, just to have it recalled. Namely, when doing case-split for a term t on a navigation expression a, then we can replace already existing occurrences of a in t by the newly introduced logical variable. This may render t more readable.

In general, we cannot do this substitution below temporal operators because the navigation expression a is evaluated anew in each state – unless we know that it's crystallised [158] (cf. Section 7.5).

For example, consider the path shown in Figure 4.9. The navigation expression $x.\lambda$ evaluates to id_2 in state π^0 and to id_3 in state π^1 under $\theta = \{x \mapsto id_1\}$. Then

$$\mathsf{G}\, p(x.\lambda) \tag{4.81}$$

doesn't hold in π under θ while

$$\mathsf{G}\, p(y) \tag{4.82}$$

holds under $\theta' = \theta \cup \{y \mapsto id_2\}$.

In other words, by case-split, we don't crystallise [158], we don't fix links, but we consider particular starting situations case by case. These starting situations may evolve over time as they like, we can't influence that from the (observing) side of the formula.

Note 4.5.7. *Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature comprising equation of identities, that is, $= \in \mathcal{F}$. Let t be a logical term over $\mathcal{S}$ and let a denote a navigation expression over $\mathcal{S}$ of an identity type T.*

Then occurrences of a in t can be replaced by the logical variable x if x is introduced for case-split, that is,

$$\forall\, x : T \,.\, a = x \rightarrow t \equiv \forall\, x : T \,.\, a = x \rightarrow t[a/x]. \tag{4.83}$$

◇

4.5.4. Quantification Over Links

We know from Section 4.5 that some formulae have prenex normal forms and some doesn't. Unfortunately, formulae like

$$(\forall\, y : T \,.\, y \in x.\lambda \rightarrow p(y)) \rightarrow q(x) \tag{4.84}$$

where "$\in$" is a predicate symbol, don't have one. Recall that the reason is the dynamic extension of the left-hand side of the first implication.

This is unfortunate because it is actually a frequent or desirable pattern; it's not unreasonable to say something like this pattern instantiation, assuming a system from the rail domain:

"If all switches of track x are set, then x is signalled clear." (4.85)

Here, y would range over the switches belonging to track x, p would indicate "being set" for switches, and q would indicate "signal clear" for the track.

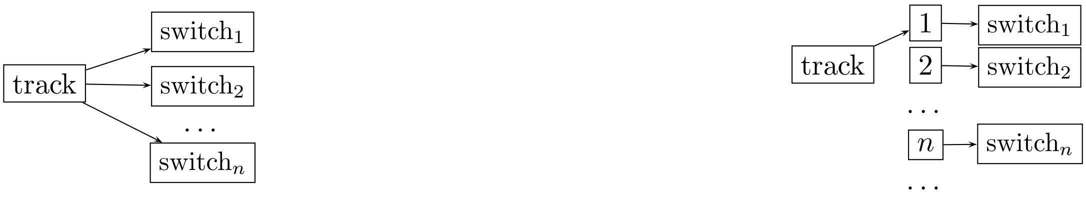
Another example is the following:[4]

$$\mathsf{G}\,((\forall\, x : T \,.\, \circledcirc(x) \wedge \mathit{switch}(x) \rightarrow p(x)) \implies \mathsf{X}\, q(y)) \tag{4.86}$$

(if all switches *in the system* (not of a particular track) are set, then something happens).

Yet we can do something about this (the former) case. In case we know that the link is finite, the following paragraph "The Finite Case" applies. In case the link is not bounded, the subsequent paragraph "The Infinite Case" may apply.

[4] the property needn't be stated in terms of a logical variable x, but may also be global



(a) Set-valued link.

(b) Array-valued link.

Figure 4.10.: **Set-valued** links with finite bounds.

The Finite Case

If a maximum size N of a link set is known, then the model can be changed to employ N single links named $\lambda_1, \dots, \lambda_N$. A designated identity id_0 would be used to indicate unused links, and we would have this name as a constant of type T in the set of function symbols (only a little bit more tricky for fixed universes). Rewriting the encoding is straight-forward, and it is also clear on the level of operations how to rewrite them to operate on the single links.

Equation (4.84) then becomes

$$(x.\lambda_1 \neq id_0 \rightarrow p(x.\lambda_1)) \wedge \cdots \wedge (x.\lambda_N \neq id_0 \rightarrow p(x.\lambda_N)) \rightarrow q(x) \tag{4.87}$$

This way, we can also have, for example, the sum over the local state of a set of identities; if it is formerly written as

$$f(x.\lambda) > 0, \tag{4.88}$$

then it may become

$$\begin{aligned}
&(x.\lambda_1 \neq id_0 \wedge \cdots \wedge x.\lambda_N \neq id_0) \rightarrow (f(x.\lambda_1, f(\dots)) > 0)\\
&\dots\\
&(x.\lambda_1 = id_0 \wedge \cdots \wedge x.\lambda_N = id_0) \rightarrow (f_0() > 0)
\end{aligned} \tag{4.89}$$

assuming a binary variant of f (and a commutative operation), and assuming a 0-ary constant f_0 yielding the value obtained for the empty set.

Note that this may require quite some rewriting, at least from functional to logical terms since we have implication only for logical terms. Alternatively one could have a (finite) set comprehension operator which takes N values and constructs a set of size (at most) N.

In the latter case from above we can also do something by changing the model such that there is a known global place to access the switches – and a known finite upper bound on their size.

The Infinite Case

Figure 4.10(a) depicts a track with a set of n switches, where switches may arbitrarily be removed or added. Conceptionally, we can view this set as an unbounded array as shown in Figure 4.10(b), possibly indicating unused entries by a designated identity id_0. Still, identities of switches may arbitrarily be removed or added, thus in general we cannot obtain a prenex form as we do not know the number of links beforehand, but only at binding time .

Despite this general fact, we'll see in Chapter 9 an approach to also treat ETTS with unbounded links under certain preconditions. These situations are characterised by formulae that refer to finite destination vertex, for example that for all two different switches at a track, a certain property holds. In the terms of Chapter 10, this is introducing a kind of sub-scenario, similar to the top-level scenario.

The preconditions are roughly that the set of links is symmetric in identities, that there is no order on the destination vertices and that system behaviour doesn't depend on the exact number of destination vertices. Then we can apply the same abstraction to links that we apply to reduce ETTS to finite-state systems.

4.6. Discussion

In the domain of parameterised systems, it is often sufficient to consider *indexed* variants of temporal logic for the requirements specification. The employed quantification can be unfolded into an ordinary formula without quantification because each instance of a parameterised system is finite.

For this reason, most recent proposals for property specification languages related to EvoCTL* stem from the domain of UML [138, 141, 140] or object orientation in general, of which ETTS also provide an adequate semantical domain (cf. Chapter 10).

The main novelty of EvoCTL* is that it employs a three-valued domain to handle unexpected disappearance of individuals adequately in the domain of critical systems specification. Other approaches either exclude the topic or employ a biased semantics (cf. Section 4.1.2).

Existing approaches cover a broad range. The Bandera Specification Logic (BSL) [39], a collection of LTL-based patterns for Java verification, simply restricts the discussion to Java without disappearance.

Others, as the Allocational Temporal Logic (ATL) [57] (and its variant NallTL) address appearance and disappearance (or: memory allocation and deallocation) *exclusively.* Its life-cycle queries referring to whether an object is *newborn* ("$\odot$"), *alive* ("$\circledcirc$"), or *disappearing* ("$\otimes$") inspired VTL/ETL [190, 191] (cf. footnote on page 95) and our syntax.

There is a large body of literature proposing to extend UML's (non-temporal) Object Constraint Language (OCL) [139] with temporal operators [57, 58, 30, 193, 20]. Most of them don't discuss the problem of disappearance explicitly, and all of them implicitly opt for a biased semantics. That is, they explicitly define the positive case and map both, a negative witness disproving a formula *and* pre-mature disappearance of objects to "*false*", although OCL provides the third logical value *oclUndefined.* Only Flake et al. [69, 70] employ this value in their proposal, OCL as term language of LTL with past operators, yet with issues in the formal definition.

Our proposal can be seen as an extension of VTL [190, 191], which is positioned in the context of the abstract interpretation-based analysis of Java-like, heap-manipulating programs. The logic METT [9] is basically VTL without transitive closure and additional operators to refer to event-based communication (cf. Chapter 10), but in the cited version also with a biased semantics in mind.

Our earlier proposal [99, 48, 49] of to adopt the visual scenario language LSCs [43] to systems with dynamic appearance and disappearance of objects didn't consider the issue of disappearance in the depth covered here. They're related to EvoCTL* because their expressive power is equivalent to a strict fragment of CTL* [47, 168]. As we'll see in Chapter 10, their life-lines express requirements on aliveness.

We refer to [11] for a slightly more elaborate discussion.

Another aspect to classify the named other works is the kind of quantification. Most proposals choose a quantification over identities, only few, like VTL [190, 191] and [69, 70] employ something similar to our destiny quantifiers. EvoCTL* is, to our knowledge, the first proposal to include both, in order to be able to discuss them in a common framework. A discussion, we've started with Section 4.4.7.

These topics as such have long been discussed for first-order modal logic in philosophical logic, for example by Barcan [6], Lewis [116], and Kripke [104]. A good overview is given in the excellent textbook [68]. As a side note, interestingly none of the above cited approaches, with the only exception of [190, 191], refers to these fundamental sources.

The main questions are those of disjoint vs. fixed universes (cf. Chapter 3), the former being embeddable in the latter, and the treatment of predicate evaluation for no-longer or not-yet existing individuals.

The choices are to accept that there will be *some* value, from which source ever, and to consider that predicates may hold and not hold at the same time [68]. The latter corresponds to a three-valued logic. We argue for this choice because in the domain of critical systems, we cannot assume that some evaluation of the predicate "being in critical state" obtains an evaluation from *somewhere.*

These issues shall best be made explicit, with a certain monotonicity to err on the safe side.

Furthermore, the only modalities discussed in [68] are $\Box$ and $\Diamond$. In our domain, there is typically a clear need for richer languages, including X and U and means to refer to the appearance and disappearance of objects, our life-cycle queries. This is not discussed in the literature on (philosophical) first-order modal logic.

In the discussion whether branching or linear time temporal logics are the best choice [170, 171], the race seems to be run in favour of linear time. For example, all of the approaches named above are linear.

Our EvoCTL* is a generalisation of branching and linear time for two reasons. Firstly, we're interested in the applicability of certain abstractions (cf. Chapters 5 and 6) and this applicability is, as we show, not limited by a certain flavour of logic.

From the proofs in Chapters 5 and 6 we can actually tell that it'll apply to even more general formalisms like the μ calculus.

It is arguable that the semantics of EvoCTL* is good, as it has certain nice properties in favour of existing proposals, but still not optimal. It is, for instance, too pessimistic in the sense that it evaluates formulae to the indefinite value $1/2$ as soon as certain *possibly* relevant variables turn undefined. The twist is that relevance cannot be exactly captured by syntactical means as long as tautologies or contradictions may occur within a given EvoCTL* formula. For example with the two formulae $\psi_1 := p(\boldsymbol{x}) \vee 1$ and $\psi_2 := 1$ we have $\psi_1 \equiv \psi_2$ but not $\mathsf{X}\,\psi_1 \not\equiv \mathsf{X}\,\psi_2$, which is surprising at first sight, but seems to be an inherent issue when striving for a completely definite fragment which *includes* the "next" ("X") modality. Yet the discussion in Chapter 9 clarifies that this operator is, in the clasical interpretation, hardly usable in interleaving semantics.

Furthermore, there are quite a few issues with EvoCTL*. For example our characterisations of monotinicity and definiteness are good first steps, but not exact. Similarly, the exact relation between the two flavours of logical variables is not completely understood, the same holds for the question of prenex normal forms. And we currently don't know anything of the relation between EvoCTL* and ETTS system, similar to results on temporal logic and Kripke structures.

Part III.

Query and Data-Type Reduction for ETTS

5. Abstraction

In the following, we introduce a notion of simulation between ETTS which is a conservative extension of the common definition for Kripke structures (cf. [34] for details). We first informally – but, due to its importance, in quite some detail – recall the definition for Kripke structures to point out in how far it is a natural extension of the classical definition.

The formal definitions in Section 5.4 then concentrate only on ETTS as we will exclusively focus this class of transition systems in the subsequent chapters. Note that the definitions in the following sections don't depend *formally* on the introductory text in Section 5.1, it only provides the motivations behind the definitions of Section 5.2 and later.

5.1. From Simulation Relations for Kripke Structures to Topology Embeddings and Corresponding Evolution

Given two Kripke structures M and $M^\sharp$, the classical definition of simulation goes back to [135]. Intuitively, $M^\sharp$ should be called simulating M if each path of M has a corresponding path in $M^\sharp$, while there may be much more paths in $M^\sharp$.

Two paths $\pi^\sharp$ and π of $M^\sharp$ and M correspond, if the i-th state in path $\pi^\sharp$ is labelled with a safe over-approximation of the label of the i-th state of π.

And intuitively, a label of a state is a safe over-approximation of another state's label if the information isn't contradictory. That is, the approximating label may carry the same or less precise information as long as it doesn't preclude the information in the approximated label. In addition, concerning additional information that is not related to the approximated label, the approximating label is free to carry any information.

$$\begin{tikzcd}
s \arrow[rr, "\mathscr{L}(M)"] \arrow[d, dash, dotted] & & d \arrow[d, dash, dotted] \\
H \arrow[d, dash, dotted] & \Longrightarrow & =/\sqsubseteq \gamma(\cdot) \arrow[d, dash, dotted] \\
s^\sharp \arrow[rr, "\mathscr{L}(M^\sharp)"] & & d^\sharp
\end{tikzcd}$$

(a) If two states are in simulation relation, then their labels correspond according to "$\sqsubseteq \gamma(\cdot)$".

$$\begin{tikzcd}
s \arrow[r, "R(M)"] \arrow[d, dash, dotted] & s' & & & s' \arrow[d, dash, dotted] \\
H \arrow[d, dash, dotted] & & \Longrightarrow \exists\, s^{\sharp\prime} : & & H \arrow[d, dash, dotted] \\
s^\sharp & & & s^\sharp \arrow[r, "R(M^\sharp)"] & s^{\sharp\prime}
\end{tikzcd}$$

(b) If two states are in simulation relation H and the simulated system can take a transition, then the simulating system can take a transition to a H-related state.

Figure 5.1.: **Simulation Relation.** Common definition for Kripke structures.

This intuition is formalised for Kripke structures using the notion of a so-called simulation relation, a relation H between the state sets of the two Kripke structures, i.e. $H \subseteq S(M) \times S(M^\sharp)$.

Given two Kripke structures M and $M^\sharp$ over the same set of atomic propositions, a relation H between the state sets is called simulation relation if, given a pair $(s, s^\sharp)$ of H-related states,

1. the label of $s^\sharp$ is equal to the label of s, i.e.

$$(\mathscr{L}(M))(s) = (\mathscr{L}(M^\sharp))(s^\sharp) \tag{5.1}$$

   and

2. if M can take a transition from s to some state s', then $M^\sharp$ can take a corresponding transition, that is, it can take a transition to an H-related state, i.e.

$$\begin{gathered}\forall\, (s, s') \in R(M), (s, s^\sharp) \in H \;\exists\, s^{\sharp\prime} \in S(M^\sharp) : \\ (s^\sharp, s^{\sharp\prime}) \in R(M^\sharp) \wedge (s', s^{\sharp\prime}) \in H\end{gathered} \tag{5.2}$$

(cf. Figure 5.1(a) and Figure 5.1(b)).

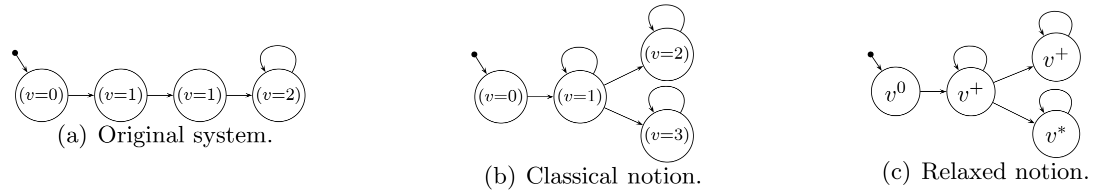
Concerning requirement 1 on states, there is also a slightly generalised definition in use which admits $M^\sharp$ to use a subset of M's atomic proposition. Then 1 requires that they agree on the common atomic propositions.

Note that the above definition — which we will follow — is sometimes called *strong simulation relation.* In particular in the domain of process algebra, which distinguishes invisible τ-transitions and visible transitions, it is sensible to define a *weak simulation relation* which permits the simulating system to take an arbitrary number of τ-transitions before and after the visible transition required in item 2.

The classical definition as given above requires that the labelling of related states is identical. This notion is primarily concerned with the abstraction of the *temporal behaviour* of a Kripke structure, that is, with its transitions. Instead, we also want to consider an abstraction of state labels.

Anticipating Section 5.4, it is obvious that a formula using the set of atomic proposition of a Kripke structure as atoms can directly be evaluated on both, the simulated and the simulating system in the sense of the classical definition. The reason is that the classical definition, even in the slight generalisation, enforces that both Kripke structures have the relevant atomic propositions in common.

This is too strict for the kind of safe over-approximation we informally envisaged in the introductory paragraph. Our primary aim, in contrast, is an abstraction of the state labelling because with ETTS, the labelling domain is typically unbounded as there are no necessary bounds on the number of individuals in a topology. For example, in an ETTS modeling car platooning using topologies as shown in Figure 3.4 there is a priori no bound on the number of cars present in each topology as they may freely enter and leave the highway.



(a) Original system.

(b) Classical notion.

(c) Relaxed notion.

Figure 5.2.: **Simulation Relation.** The Kripke structure shown in Figure 5.2(b) simulates the one given by Figure 5.2(a) in the strict classical notion, the Kripke structure given by Figure 5.2(c) simulates both others in the relaxed notion.
Notation: half-connected arrows point to initial states, states are labelled with atomic propositions as given by the labelling.

To this end, we adopt from the domain of abstract interpretation [40] the usage of Galois connections to denote the relation between concrete and abstract domains (cf. Section 2.4), in our case, these are the domains $\mathscr{D}$ and $\mathscr{D}^\sharp$ of the labellings of the simulated and the simulating system M and $M^\sharp$. As Galois connections as introduced in Section 2.4 assume complete join semi-lattices as concrete and abstract domain, we shall from now on assume that $\mathscr{D}$ and $\mathscr{D}^\sharp$ are complete join semi-lattices.

Given a Galois connection $\left(\mathscr{D}, \alpha, \gamma, \mathscr{D}^\sharp\right)$ between the labelling domains, we relax the condition (5.1) on H-related states s and $s^\sharp$ to

$$(\mathscr{L}(M))(s) \sqsubseteq \gamma((\mathscr{L}(M^\sharp))(s^\sharp)), \tag{5.3}$$

that is, we require that the concretisation of the label of $s^\sharp$ over-approximates the label of s, i.e. it is the same value or something less precise, but covering the original value.

Note that every simulation relation in the original definition is also a simulation relation in the relaxed definition (5.3), but in general not vice versa because even when using the minimal lattice, the relaxed version necessarily admits to label all states of $M^\sharp$ with the greatest element. Then the labellings may be non-equal as required by the original definition.

As an example, consider the Kripke structures shown in Figure 5.2(a) and Figure 5.2(b), both over the set

$$AP = \{(v = -3), (v = -2), (v = -1), (v = 0), (v = 1), (v = 2), (v = 3)\}, \tag{5.4}$$

of atomic propositions.[1]

The Kripke structure shown in Figure 5.2(b) simulates the former in the classical definition as it only adds additional states and additional transitions. Our aim is to in addition abstract from the labelling of states, for example, to encode instead of the actual value of v only the sign of the value. For example, we want to consider the Kripke structure over atomic propositons "v is positive", "v is 0", "v is negative", and "don't

[1] Note that with Kripke structures, the elements of AP are not expressions but only names. Though, the particular elements of AP may have been chosen as an easily readable representation of the encoding that a variable v has a certain integer value.

know", i.e.

$$AP^\sharp = \{v^+, v^0, v^-, v^*\}, \tag{5.5}$$

as given by Figure 5.2(c) to simulate both others because the labelling of the states is a safe approximation of the original labelling. In order to see that property (5.3) is satisfied, we have to view AP as a complete join semi-lattice (by taking the canonical completion) and order $AP^\sharp$ canonically.

This straightforward step of generalising the requirements imposed on states in simulation relation is principally sufficient to treat the requirement 1 on states in our case of ETTS. Recall that states are labelled with topologies, that is, we have to define a notion of abstractions of topologies. In general, anything can be an abstraction of a topology as long as it is linked via a Galois connection to the original domain. In order to support a good understanding of the particular abstraction technique we're discussing, we take a particular approach where we aim at an abstract domain which itself as closely as possible resembles topologies. The elements actually *are* topologies in the sense of the original definition in Section 3.3. In the course of this chapter it will become clear why some definitions, like equality function, alives and non-alives set, and navigation function, have been chosen as presented here.

Our approach follows [155] in that we define in Section 5.2 the embedding of on topology into another, more abstract one. The idea is that multiple nodes of the concrete topology can be mapped to, and thus represented by, a single (abstract) node in the abstract topology, yet the abstract node has to carry a suitable local state which doesn't contradict the concrete ones and links have to be considered carefully. In addition, abstract nodes have to respect life-cycle properties of represented nodes and the equality relation on identities.

The fundamental difference to [155] is that they consider uninterpreted (up to the summary predicate) logical structures while we address topology notions like node label, links, and evolution. We discuss this relation and the advantages and disadvantages in more detail in Section 5.7.

Yet the requirement 1 on transitions needs another extension of the original notion because ETTS are "less memory-less" than Kripke structure by the evolution relation. Namely, whether an individual is newly created or about to disappear in a state depends on the transition, it is not a sole property of the state but depends on the transition.

With Kripke structures, there is only the labelling of states, each property of a state depends solely on the atomic propositions assigned by the labelling. But in order to evaluate life-cycle properties in the simulating transition system, we introduce a notion of corresponding evolution and disappearance in Section 5.4. Then, in addition to 1, we not only require that there is some transition in the simulating system, but we require that there is one along which individuals evolve and disappear accordingly.

Having established a suitable relation between states independent from the choice of (5.1) or (5.3), the Kripke structure $M^\sharp$ is then said to *simulate* M if and only if each initial state of M is H-related to an initial state of $M^\sharp$.

In the context of model-checking of Kripke structures and temporal logic, the simulating system $M^\sharp$ is also commonly called an *abstraction* (or, more precise, a *safe* abstraction) of M [34]. Note that this usage of the name is different from the abstract interpretation community where typically an abstraction function, or the pair of abstraction and concretisation function, is called abstraction (cf. Section 2.4). As we make use of both domains, we will use the term seldom, and if, the meaning should be clear from the context.

Given a simulation relation, a Corresponding Path Lemma establishes the original intuition, that for each path in M there is a corresponding path in $M^\sharp$. This Corresponding Path Lemma can then be used to establish the theorem that for certain temporal logic properties (in our case the temporal logic EvoCTL* from cf. Chapter 4), if $M^\sharp$ satisfies the property, then so does M (but not necessarily vice versa). In the slightly generalised definition of the simulation relation where $M^\sharp$ uses a subset of the atomic propositions of M, only formulae over the common atomic propositions are considered.

In other words, a simulating system in the classical sense preserves all violations of a temporal property exactly, while in the generalised sense there is for each violation an abstract path which concretises to the violation. Intuitively, this claim is immediately clear: if there is a path in M violating the property, then there is, by the simulation property, a corresponding path in $M^\sharp$ that also violates the property. It is formally established in the following sections.

5.2. Topology Embedding

Definition 5.2.1 (Embedding)**.** *Let G be a (Σ, Λ)-topology over Id and $G^\sharp$ a $(\Sigma^\sharp, \Lambda)$-topology over $Id^\sharp$, both identities and local states sets partitioned into n partitions. Let Σ_i and $\Sigma_i^\sharp$, and* $\mathrm{dom}(\rightarrowtail_\lambda)$ *and* $\mathrm{dom}(\rightarrowtail_\lambda^\sharp)$ *form complete join semi-lattices for $1 \leq i \leq n$ and each $\lambda \in \Lambda$.*

Assuming there are Galois connections

$$\left(\Sigma_i, \alpha_{\Sigma_i}, \gamma_{\Sigma_i}, \Sigma_i^\sharp\right) \;\; \textit{and} \;\; \left(\mathrm{dom}(\rightarrowtail_\lambda), \alpha_\lambda, \gamma_\lambda, \mathrm{dom}(\rightarrowtail_\lambda^\sharp)\right), \tag{5.6}$$

a pair (f, g) of functions

$$f : U(G) \to U(G^\sharp) \tag{5.7}$$

and

$$g : L(G) \to L(G^\sharp) \tag{5.8}$$

is called embedding *of G into $G^\sharp$ if and only if*

1. *f assigns each identity $id \in U(G) \cap Id_i$ an identity $id^\sharp \in U(G^\sharp) \cap Id_i^\sharp$ such that*
    a) *the local state of $id^\sharp$ is a* safe over-approximation *of the local state of id, that is, if*

    $$\forall\, id \in U(G) : \sigma(id) \sqsubseteq \gamma_{\Sigma_i}(\sigma^\sharp(f(id))), \tag{5.9}$$

    b) *aliveness is respected, i.e.*

    $$f(U^{\circledcirc}(G)) \subseteq U^{\circledcirc}(G^\sharp) \;\; \textit{and} \;\; f(U^{\not\circledcirc}(G)) \subseteq U^{\not\circledcirc}(G^\sharp), \tag{5.10}$$

c) equality is respected, i.e.

$$eq_{Id}(id_1, id_2) \sqsubseteq eq_{Id^\sharp}(f(id_1), f(id_2)), \tag{5.11}$$

2. g assigns each link $\ell \in L(G)$ a link $\ell^\sharp \in L(G^\sharp)$ such that the link names of ℓ and $\ell^\sharp$ coincide, that is, if

$$\forall\, \ell \in L(G) : \lambda(\ell) = \lambda^\sharp(g(\ell)), \tag{5.12}$$

3. identity assignments and link navigation are consistent, that is,

$$\forall\, u \in U(G)\ \forall\, \lambda \in \Lambda : \alpha_\lambda(\rightarrowtail_\lambda(u)) \sqsubseteq_{\mathrm{dom}(\rightarrowtail^\sharp_\lambda)} \rightarrowtail^\sharp_\lambda(f(u)) \tag{5.13}$$

The two components of an embedding (f, g) are called individual and link embedding function, respectively.

We say (f, g) embeds *G into $G^\sharp$, denoted by $G \sqsubseteq^{(f,g)} G^\sharp$, if (f, g) is an embedding, and we say a topology G* can be embedded *into $G^\sharp$, if there exists an embedding of G into $G^\sharp$. To explicate the two topologies for which an embedding is provided, we shall write $(f_{G,G^\sharp}, g_{G,G^\sharp})$, in the context of ETTS we will alternatively refer to states, that is, we may write $(f_{s,s^\sharp}, g_{s,s^\sharp})$ as an abbreviation for $(f_{\mathscr{L}(s),\mathscr{L}^\sharp(s^\sharp)}, g_{\mathscr{L}(s),\mathscr{L}^\sharp(s^\sharp)})$.* ♢

Definition 5.2.2 (Tight Embedding)**.** *Let G and $G^\sharp$ be topologies such that (f, g)* embeds *G into $G^\sharp$. The embedding (f, g) is called* tight *if and only if*

1. f assigns each identity $id \in U(G)$ an identity $id^\sharp \in U(G^\sharp)$ such that

a) the local state is most precisely represented, i.e.

$$\forall\, id \in U(G) \cap Id_i : \sigma^\sharp(id^\sharp) = \bigsqcup\{\alpha_{\Sigma_i}(\sigma(id)) \mid f(id) = id^\sharp\}, \tag{5.14}$$

b) aliveness and equality are most precisely respected, i.e.

$$\{id \mid f(id) = id^\sharp\} \subseteq U^{\circledcirc}(G) \implies id^\sharp \notin U^{\not\circledcirc}(G^\sharp), \tag{5.15}$$

$$\{id \mid f(id) = id^\sharp\} \subseteq U^{\not\circledcirc}(G) \implies id^\sharp \notin U^{\circledcirc}(G^\sharp), \tag{5.16}$$

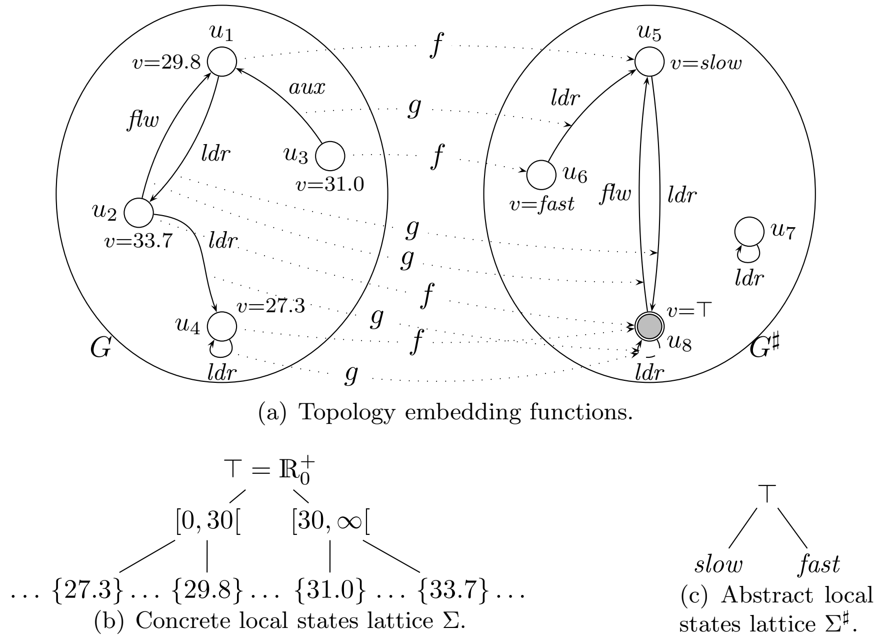
c) equality is most precisely respected, i.e.

$$eq_{Id^\sharp}(id^\sharp_1, id^\sharp_2) = \bigsqcup\{eq_{Id}(id_1, id_2) \mid id^\sharp_1 = f(id_1), id^\sharp_2 = f(id_2)\} \tag{5.17}$$

2. navigation is most precisely represented, i.e.

$$\rightarrowtail^\sharp_\lambda(id^\sharp) = \bigsqcup\{\alpha_\lambda(\rightarrowtail_\lambda(id)) \mid id^\sharp = f(id)\}. \tag{5.18}$$

♢



(a) Topology embedding functions.

(b) Concrete local states lattice Σ.

(c) Abstract local states lattice $\Sigma^\sharp$.

Figure 5.3.: **Topology Embedding.** Topology s can be embedded into $s^\sharp$ given the lattices for abstract and concrete local state and the shared lattice for link names.

We could continue to discuss best abstract structures and the effect of best abstract structures on tightly embedded topologies on term evaluations. We don't, because the embedding we'll discuss later is far from tight, and most probably won't become tight either, because the high level of abstraction is the key to a good implementability.

Figure 5.3 gives an example for a topology embedding. Consider G and $G^\sharp$ to be topologies representing car platoons with nodes representing cars and the typical links *ldr* (leader), *flw* (follower), and *aux* (auxiliary). For the example, we in addition consider that the local state of a car is a non-negative, denoted by v and a value from $\mathbb{R}_0^+$ in Figure 5.3(a).

We chose to use the lattices shown in Figure 5.3(b) and Figure 5.3(c) as concrete and abstract local state. The Galois connection between the local states maps velocities strictly below 30.0 to '*slow*' and others to '*fast*'. That is, in this example we want to abstract from the actual velocity and only distinguish between slow and fast.

The embedding functions f and g are given by Figure 5.3(a). We can already identify some characteristic examples:

- All individuals in G have a representative in $G^\sharp$, but not every individual in $G^\sharp$ needs to represent one, for example u_7 is not a representative.
- An individual in $G^\sharp$ may represent one, like u_5, or more individuals from G, like u_8, then called *summary individual.*
- Similarly, all links in G have a representative in $G^\sharp$, but not every link has to be

a representative, and there are *summary links.*

- For example, the loop at u_7 is not a representative, and the loop at u_8 represents two links, thus is a summary link (shown dashed in Figure 5.3(a)).
- The meaning of navigating a summary link is provided by the navigation functions of $G^\sharp$ (not shown in the picture). For instance, $\rightarrowtail_{ldr}$ could yield the multi-set with two occurrences of u_8 at u_8 corresponding to the two edges it represents. Alternatively, it could yield an indefinite value $\top$ indicating complete uncertainty about represented individuals.

Dashed lines and double outlines in Figure 5.3(a) give additional information which is not part of the embedding, but can be derived from it. The double outline of individual u_8, for instance, points out that this (abstract) individual represents multiple (concrete) ones. The dashed arrow to and from u_8 indicates, that in G, this edge exists only for *some* of the nodes represented by u_8. Note that this is similar to a summary edge[2] in [155], but not quite the same. The difference is that the dashed line in Figure 5.3(a) has to be read as part of the link name, i.e. $(\mathit{ldr}, \mathit{dashed}) \in \Lambda^\sharp$.

Requirement 1 of Def. 5.2.1 is satisfied because the local states of the individuals in $G^\sharp$ concretise to supersets of the ones in G. The velocity of u_8 is indefinite as it represents cars at both, a slow and a fast velocity, thereby trivially satisfying 1. The velocity of u_7 is irrelevant because it is not representing a concrete node.

Requirement 2 of Def. 5.2.1 is satisfied because the link names in $G^\sharp$ are faithful to the ones in G. Finally, it's easily verified that requirement 3 of Def. 5.2.1 is satisfied because each link between individuals in G has a corresponding link between corresponding individuals in $G^\sharp$.

5.3. Abstract Structure and Term Evaluation

In order to evaluate EvoCTL* terms on abstract topologies, in particular terms comprising function symbols on local states from $\Sigma^\sharp$, we employ an abstract structure. To this end, we slightly extend the definition of canonical structure.

Definition 5.3.1 (Canonical *Id*-Lattice Structure)**.** *Let $\mathcal{S}$ be a signature and G a compatible topology over identities $Id = Id_1 \dot\cup \ldots \dot\cup Id_n$. A structure $\mathcal{M} = (\iota, \mathcal{D})$ of $\mathcal{S}$ with m identity types $T_1, \ldots, T_m \in \mathcal{T}(\mathcal{S})$ is called* canonical *Id*-lattice structure *of $\mathcal{S}$ wrt. G if and only if*

1. *the domain $\mathcal{D}(T_i)$ of identity type T_i, $1 \leq i \leq m$, a complete join semi-lattice*

$$Id_j \dot\cup \{\top_{T_i}\} \tag{5.19}$$

with the reflexive closure of

$$\{id \sqsubseteq \top_{T_i} \mid id \in \mathcal{D}(T_i)\} \tag{5.20}$$

as $\sqsubseteq$ and $\top_{T_i}$ as top element, and

[2] a binary predicate with indefinite valuation

2. *the structure $\mathcal{M}' = (\iota', \mathcal{D}')$ which coincides with $\mathcal{M}$ except for $\mathcal{D}'(T_i) := \mathcal{D}(T_i) \setminus \{\top_{T_i}\}$ is a canonical structure of $\mathcal{S}$ wrt. G.* ♢

Definition 5.3.2 (Abstract Structure wrt. Embedding)**.** *Let $\mathcal{S}$ be a signature, G a compatible (Σ, Λ)-topology over Id and $G^\sharp$ a T a compatible $(\Sigma^\sharp, \Lambda)$-topology over $Id^\sharp$. Let $\mathcal{M} = (\iota, \mathcal{D})$ be a canonical structure of $\mathcal{S}$ wrt. G and $\mathcal{M}^\sharp = (\iota^\sharp, \mathcal{D}^\sharp)$ a canonical $Id^\sharp$-lattice structure of $\mathcal{S}$ wrt. $G^\sharp$.*

Let (f, g) be an embedding of G into $G^\sharp$. We call $\mathcal{M}^\sharp$ an abstract structure *of $\mathcal{S}$ in $G^\sharp$ wrt. the* concrete structure *$\mathcal{M}$ and the embedding (f, g), denoted by*

$$\mathcal{M} \sqsubseteq^{(f,g)} \mathcal{M}^\sharp, \tag{5.21}$$

if and only if

1. *there are Galois connections between corresponding domains, i.e.*

$$\Big(\mathcal{D}(\tau), \alpha_\tau, \gamma_\tau, \mathcal{D}^\sharp(\tau)\Big) \tag{5.22}$$

for all types τ except for identity types T, evolution chain types $\boldsymbol{T}$, and links L. where the (identical) boolean domains are connected by the identical functions on $\mathcal{D}(\mathbb{B})$, i.e., $\alpha_\mathbb{B} = \gamma_\mathbb{B} = id_\mathbb{B}$. To simplify notation, we use α_T to denote f and α_L to denote the point-wise application of f to multi-sets in the following.

2. *link navigation is safe, i.e. given $id \in \mathcal{D}(T)$ and $id^\sharp \in \mathcal{D}^\sharp(T)$ such that $f(id) \sqsubseteq id^\sharp$, we have*

$$\alpha(\iota(\lambda)(id)) \sqsubseteq_{\mathcal{D}^\sharp(L)} \iota^\sharp(\lambda)(id^\sharp). \tag{5.23}$$

3. *the interpretation of function symbols is safe, i.e. given a function symbol $f : \tau_1 \times \cdots \times \tau_k \to \tau$ of arity k and type τ and given k pairs of semantical values $d_i \in \mathcal{D}(\tau_i)$ and $d_i^\sharp \in \mathcal{D}^\sharp(\tau_i)$ such that*

$$\alpha_{\tau_i}(d_i) \sqsubseteq_{\mathcal{D}^\sharp(\tau_i)} d_i^\sharp \tag{5.24}$$

for $i = 1, \ldots, k$, we have

$$\alpha(\iota(f)(d_1, \ldots, d_k)) \sqsubseteq_{\mathcal{D}^\sharp(\tau)} \iota^\sharp(f)(d_1^\sharp, \ldots, d_k^\sharp). \tag{5.25}$$

♢

Figure 5.4 illustrates the relations between domains for abstract structures. If the abstract domain is fixed, there is a notion of a best abstract structure similar to the tight embedding above.

Definition 5.3.3 (Corresponding Assignment)**.** *Let $\mathcal{S}$ be a signature and G and $G^\sharp$ two $\mathcal{S}$-compatible topologies such that (f, g) embeds G into $G^\sharp$. Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. G, $G^\sharp$, and (f, g).*

Let $\theta^\sharp \in Assign_{\mathcal{M}^\sharp}(V)$ and $\theta \in Assign_{\mathcal{M}}(V)$ be assignments of some logical variables $V \subseteq \mathcal{V}^T$ in the structures $\mathcal{M}^\sharp$ and $\mathcal{M}$. We say θ and $\theta^\sharp$ correspond *wrt. G and $G^\sharp$, denoted by*

$$(\theta, G) \sim_{(f,g)} (\theta^\sharp, G^\sharp), \tag{5.26}$$

if and only if

$$\begin{array}{ccccccc}
\mathcal{D}(\boldsymbol{T}_i) & = & \mathcal{D}(T_i)^+ \cup \mathcal{D}(T_i)^\omega & \not\leftrightarrow & \mathcal{D}^\sharp(T_i)^+ \cup \mathcal{D}(T_i)^\omega & = & \mathcal{D}^\sharp(\boldsymbol{T}) \\
\mathcal{D}(T_i) & = & Id_j & \overset{\alpha(=f)}{\longrightarrow} & Id^\sharp_j \mathbin{\dot\cup} \{\top_{T_i}\} & = & \mathcal{D}^\sharp(T_j) \\
\mathcal{D}(L) & = & \bigcup_{\lambda\in\Lambda} \mathrm{dom}(\rightarrowtail_\lambda) & \overset{\alpha_\lambda,\lambda\in\Lambda}{\leftrightarrow} & \bigcup_{\lambda\in\Lambda} \mathrm{dom}(\rightarrowtail^\sharp_\lambda) & = & \mathcal{D}^\sharp(L) \\
\mathcal{D}(\mathbb{B}) & = & \mathbb{B}_3 & \overset{(id,id)}{\leftrightarrow} & \mathbb{B}_3 & = & \mathcal{D}^\sharp(\mathbb{B}) \\
\mathcal{D}(S) & = & \Sigma & \overset{(\alpha,\gamma)}{\leftrightarrow} & \Sigma^\sharp & = & \mathcal{D}^\sharp(S) \\
\mathcal{D}(\tau_1) & & & \overset{(\alpha,\gamma)}{\leftrightarrow} & & & \mathcal{D}^\sharp(\tau_1) \\
\dots & & & \dots & & & \dots
\end{array}$$

Figure 5.4.: **An abstract structure** wrt. a given embedding and a concrete structure provides certain connections between semantical domains except for $\boldsymbol{T}$. The domains of T and L are related by the individual embedding function f. The boolean domains shall be identical, and all other types shall be Galois connected. For S, this is a consequence of Def. 5.2.1, for all additional types, like τ_1 above, it is a plain requirement.

1. *θ and $\theta^\sharp$ are assignments in G and $G^\sharp$,*
2. *$\theta^\sharp$ is $f \circ \theta$ in the sense that*
   a) *for each identity variable $x \in V$, if $\theta(x) \in U(G)$ then $\theta^\sharp(x) = f(\theta(x))$, and otherwise, i.e. if $\theta(x) \notin U(G)$, $\theta^\sharp(x) =: id^\sharp_0$ is also from the complement of $U(G^\sharp)$ in $Id^\sharp$ or it is from $U^{\not\circledcirc}(G^\sharp)$ with*

   $$\sigma(G^\sharp)(id^\sharp_0) = \top \in \mathcal{D}^\sharp(S) \tag{5.27}$$

   and

   $$\rightarrowtail^\sharp_\lambda(id^\sharp_0) = \top \in \mathcal{D}^\sharp(L) \tag{5.28}$$

   for each link name $\lambda \in \Lambda(G) = \Lambda(G^\sharp)$,

   b) *for each pair of identity variables $x_1, x_2 \in V$ equality is preserved,[3] that is*

   $$eq_{Id}(\theta(x_1), \theta(x_2)) \sqsubseteq eq_{Id^\sharp}(\theta^\sharp(x_1), \theta^\sharp(x_2)), \tag{5.29}$$

   and

   c) *for each destiny variable $\boldsymbol{x} \in V$, if $\theta(\boldsymbol{x})(0) \in U^{\circledcirc}(G)$ then $\theta^\sharp(\boldsymbol{x})(0) = f(\theta(\boldsymbol{x})(0))$, and otherwise, i.e. if $\theta(\boldsymbol{x}) = \varepsilon$, then also $\theta^\sharp(\boldsymbol{x}) = \varepsilon$.* ♢

Note 5.3.4 (Aliveness in Corresponding Assignments)**.** *Let $\mathcal{S}$ be a signature and G and $G^\sharp$ two $\mathcal{S}$-compatible topologies such that (f, g) embeds G into $G^\sharp$. Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. G, $G^\sharp$, and (f, g).*

Let θ and $\theta^\sharp$ be corresponding assignments of some logical variables $V \subseteq \mathcal{V}^T$ in the structures $\mathcal{M}^\sharp$ and $\mathcal{M}$, i.e. $(\theta, G) \sim_{(f,g)} (\theta^\sharp, G^\sharp)$.

Then $\theta^\sharp$ assigns (non-)alive individuals if θ does. ♢

[3] Note that in case both, $\theta(x_1)$ and $\theta(x_2)$, are from $U(G)$, then is this property is already a consequence of the previous requirement by Def. 5.2.1.1c.

Proof. Given the premises, we have to show that

$$\theta(x) \in U^{\circledcirc}(G) \implies \theta^{\sharp}(x) \in U^{\circledcirc}(G^{\sharp}) \tag{5.30}$$

and

$$\theta(\boldsymbol{x})(0) \in U^{\circledcirc}(G) \implies \theta^{\sharp}(\boldsymbol{x})(0) \in U^{\circledcirc}(G^{\sharp}) \tag{5.31}$$

where $x \in \mathcal{V}^{T}$ and $\boldsymbol{x} \in \mathcal{V}^{\boldsymbol{T}}$, and analogously for $U^{\not\circledcirc}(G)$ and $U^{\not\circledcirc}(G^{\sharp})$.

This is a direct consequence of the fact that the identities employed by θ and $\theta^{\sharp}$ are related by (f, g) and that embeddings preserve aliveness, cf. Def. 5.2.1.1b. □

Lemma 5.3.5 (Embedding). *Let $\mathcal{S}$ be a signature and G and $G^{\sharp}$ two $\mathcal{S}$-compatible topologies such that (f, g) embeds G into $G^{\sharp}$. Let $\mathcal{M}^{\sharp}$ be an abstract structure of $\mathcal{S}$ in $G^{\sharp}$ wrt. a structure $\mathcal{M}$ of $\mathcal{S}$ and (f, g).*

Let t be a quantifier free logical term over $\mathcal{S}$ and θ and $\theta^{\sharp}$ corresponding assignments of the free variables of t in $\mathcal{M}$. Then

$$\iota[\![t]\!](G, \theta) \sqsubseteq \iota^{\sharp}[\![t]\!](G^{\sharp}, \theta^{\sharp}). \tag{5.32}$$

◊

Proof. See Section A.2. □

Corollary 5.3.6 (Embedding). *Let $\mathcal{S}$ be a signature and G and $G^{\sharp}$ two $\mathcal{S}$-compatible topologies such that (f, g) embeds G into $G^{\sharp}$. Let $\mathcal{M}$ and $\mathcal{M}^{\sharp}$ be canonical structures of $\mathcal{S}$ wrt. G, $G^{\sharp}$, and (f, g).*

Let t be a logical term over $\mathcal{S}$ and θ an assigment of the free variables of t in $\mathcal{M}$.

1. *If t is universally quantified, then*

$$\iota^{\sharp}[\![t]\!](G^{\sharp}, f(\theta)) = 1 \implies \iota[\![t]\!](G, \theta) = 1 \tag{5.33}$$

and

$$\iota^{\sharp}[\![t]\!](G^{\sharp}, f(\theta)) = 0 \implies \iota[\![t]\!](G, \theta) = 0. \tag{5.34}$$

2. *If t is existentially quantified, then $\iota[\![t]\!](G, \theta) \sqsubseteq \iota^{\sharp}[\![t]\!](G^{\sharp}, f(\theta))$.* ◊

Proof. By pointwise application of Lemma 5.3.5 considering the definition of the semantics of both quantifiers. □

Until this point in our presentation, we've also covered abstraction of links, that is, the situation shown in Figure 5.5 where more than one link in the concrete topology G maps to a single link in $G^{\sharp}$. In order to meet the premises of Def. 5.3.2, functional symbols with arguments of type L have to consider such a fact. Whatever they do, they're supposed to do something sound, that is, if we had an operator

$$|\cdot| : L \to \mathbb{N} \tag{5.35}$$

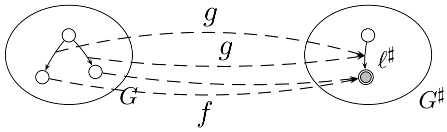


Figure 5.5.: **Abstract link.** In order for (f, g) to be an embedding, the navigation function $\rightarrowtail_{\ell^\sharp}$ along link $\ell^\sharp$ has to be consistent with f. Note that it needn't be the default multi-set comprehension.

which counts the elements in a link, than it has to obtain a sound interpretation on $G^\sharp$ in order to satisfy Def. 5.3.2.

This applies in particular if f maps to an identity type T, that is, functions like projection must do "the right thing". It's not immediately clear whether this is possible for links in general; For example:

- a "pick" operator (cf. Section 9.2.5) $\mathit{pick}(\cdot) : L \to T$ can clearly be treated;
- a comparison for equality, $= \emptyset(\cdot) : L \to \mathbb{B}$, also;
- link projection $*(\cdot) : L \to T$ has to turn undefined if the size of the argument is not 1, i.e. $*(\lambda(\mathit{id}))$ is not defined if $|\lambda(\mathit{id})| \neq 1$, in particular if $|\lambda(\mathit{id})| > 1$;
- fancy operators like summing up the local states of all individuals in a link multi-set, $\mathit{sum}(\cdot) : L \to \mathbb{N}$, has to yield $\top_{\mathbb{N}}$ if one (or more) individual(s) are labelled with $\top_\Sigma$;
- counting the elements in a link is more difficult, as it can no longer be defined point-wise, but has to depend on the content, the collected identities – or $\mathcal{D}^\sharp(L)$ cannot remain canonical but needs to carry additional information to suit all operators;

In Chapter 6, we'll provide a solution for a particular special case, but we'll limit the operations with L arguments to a minimum. The topic is also revisited in Chapter 9.

Note that there is an apparent discrepancy between the indentities appearing in topologies and the domain of the process type T in the abstract structure.

Namely that the domain of T comprises a top-element $\top$, which is not used in topologies. The reason is that C is different from $\top$. The latter is covering all identities, the concrete and the non-concrete ones, thus it wouldn't be correct to have $\top$ in a topology.

For the evaluation of terms we need this element, though. Namely if we navigate via C, for instance in $x \rightarrowtail \lambda$ with x bound to C. Then λ can point to anything, both concrete and non-concrete individuals, thus $\top$ is only correct value.

5.4. Simulation Relation for ETTS

The following definitions formalises the discussion of Section 5.1.

Definition 5.4.1 (Corresponding Evolution and Disappearance)**.** *Let M be an ETTS over Id and $M^\sharp$ an ETTS over $Id^\sharp$. Let $r = (s, s') \in R(M)$ and $r^\sharp = (s^\sharp, s^{\sharp\prime}) \in R(M^\sharp)$*

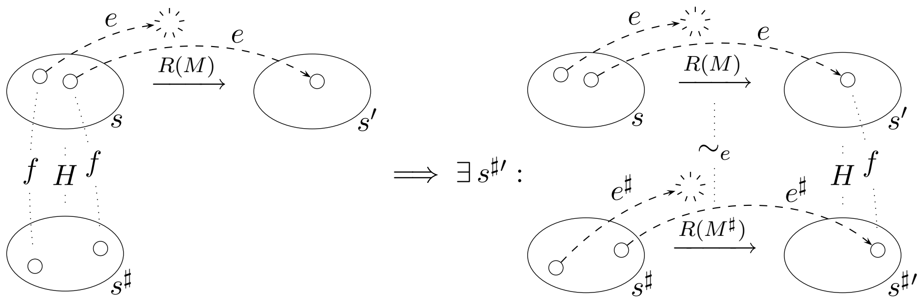


Figure 5.6.: **Individuals Evolve Accordingly** if disappearance or evolution in M implies disappearance or evolution, respectively, in $M^\sharp$.
Note that disappearing individuals are actually not in the domain of the evolution function. To make the disappearance more prominently visible, we though show dashed arrows resembling the notation for disappearance from Section 3.4.

be transitions such that the states of r *can be embedded into the corresponding states of* $r^\sharp$*, i.e.*

$$s \sqsubseteq^{(f_1,g_1)} s^\sharp \text{ and } s' \sqsubseteq^{(f_2,g_2)} s^{\sharp\prime}. \tag{5.36}$$

We say that individuals evolve and disappear correspondingly *along both transitions under* (f_1, g_1) *and* (f_2, g_2)*, denoted by*

$$r \sim_e r^\sharp \tag{5.37}$$

if the embeddings are clear by context, if and only if, given an identity $u \in U(s)$*,*

- *if* u *disappears along* r*, then the corresponding identity* $f_1(u) \in U(s^\sharp)$ *does not* not *disappear along* $r^\sharp$*, and*
- *if* u *evolves into* u' *along* r*, then the corresponding identity* $f_1(u) \in U(s^\sharp)$ *evolves along* $r^\sharp$ *into the corresponding identity* $f_2(u') \in U(s^{\sharp\prime})$*,*

that is, if

$$\forall\, u \in U(s) : u \overset{r}{\rightsquigarrow}_e ⁂ \implies f_1(u) \overset{r^\sharp}{\rightsquigarrow}_e ⁂ \tag{5.38}$$

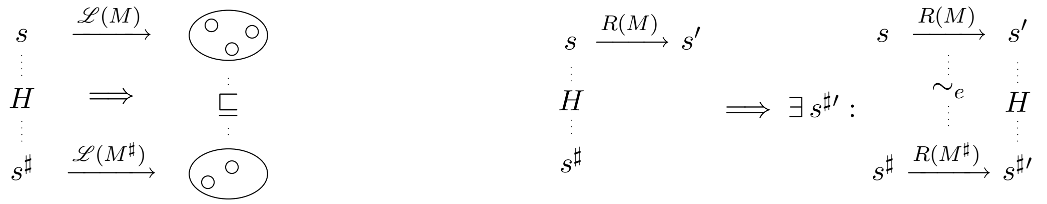
and

$$\forall\, u \in U(s), u' \in U(s') : u \overset{r}{\rightsquigarrow}_e u' \implies f_1(u) \overset{r^\sharp}{\rightsquigarrow}_e f_2(u'). \tag{5.39}$$

◇

Figure 5.6 illustrates Def. 5.4.1, Figure 5.7 illustrates Def. 5.4.2 and is best read in comparison to Figure 5.1.

Definition 5.4.2 (Simulation Relation)**.** *Let* $M = (S, S_0, R, \mathscr{L}, e)$ *be a* (Σ, Λ)*-ETTS over Id and let* $M^\sharp = (S^\sharp, {S_0}^\sharp, R^\sharp, \mathscr{L}^\sharp, e^\sharp)$ *be a* $(\Sigma^\sharp, \Lambda)$*-ETTS over Id*$^\sharp$*.*



(a) If two states are in simulation relation, then their labels correspond according to "$\sqsubseteq \gamma(\cdot)$".

(b) In addition to the common definition, there should exist a transition along which individuals evolve according to the original definition.

Figure 5.7.: **Simulation relation** for ETTS.

A relation

$$H \subseteq S \times S^\sharp \tag{5.40}$$

between the states of M and $M^\sharp$ is called simulation relation *between M and $M^\sharp$ if and only if for each pair of states $h = (s, s^\sharp) \in H$,*

1. *the pair determines an embedding (f_h, g_h) of the topology of s into the topology of s', i.e.*

$$\mathscr{L}(s) \sqsubseteq^{(f_h,g_h)} \mathscr{L}^\sharp(s^\sharp) \tag{5.41}$$

and if $U(s) \subsetneq Id$ then $U(s^\sharp) \subsetneq Id^\sharp$ or there is at least one $id_0^\sharp \in U(s^\sharp)$ as in Def. 5.3.3.2a and

2. *if there is a transition in M with state s from the pair as source and another state $s' \in S$ as destination, then there is a state $s^{\sharp\prime} \in S^\sharp$ which is in H-relation with s' and which is the destination of a transition in $M^\sharp$ starting at $s^\sharp$ such that individuals evolve and disappear accordingly along these two transitions under the embeddings determined by pairs h and $(s', s^{\sharp\prime})$, i.e.*

$$\begin{aligned} &\forall\, (s, s') \in R, (s, s^\sharp) \in H \;\exists\, s^{\sharp\prime} \in S^\sharp : \\ &\quad (s^\sharp, s^{\sharp\prime}) \in R^\sharp \wedge (s', s^{\sharp\prime}) \in H \wedge (s, s') \sim_e (s^\sharp, s^{\sharp\prime}). \end{aligned} \tag{5.42}$$

We say that $M^\sharp$ simulates *M, denoted by $M \preceq M^\sharp$, if there is a simulation relation H between M and $M^\sharp$ such that each initial state of M is in H-relation with an initial state of $M^\sharp$. To indicate a particular simulation relation, we may write $M \preceq_H M^\sharp$.* $\diamondsuit$

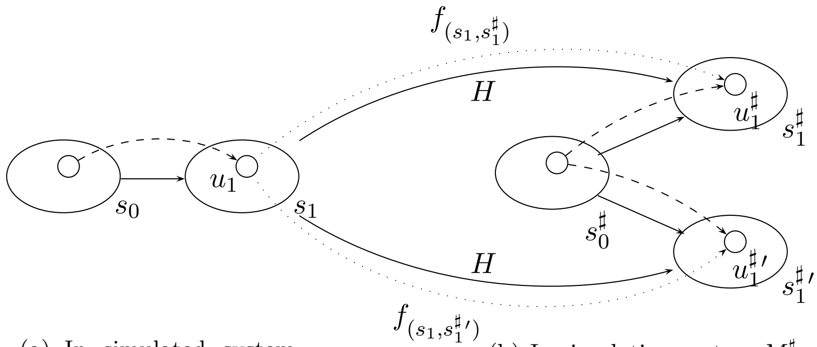
Definition 5.4.3 (Simulation with Corresponding Appearance). *Let M be a (Σ, Λ)-ETTS over Id and let $M^\sharp$ be a $(\Sigma^\sharp, \Lambda)$-ETTS over $Id^\sharp$ such that $M \preceq M^\sharp$.*

We say that $M^\sharp$ simulates M preserving appearance*, denoted by*

$$M \preceq^{\odot} M^\sharp \tag{5.43}$$

if and only if for each pair of related states $h = (s, s^\sharp)$ in the simulation relation, if an individual newly appears in s then it newly appears in $s^\sharp$, i.e. if

$$\forall\, ('s, s) = r \in R(M) : \divideontimes \overset{r}{\rightsquigarrow}_e id \tag{5.44}$$



(a) In simulated system M.

(b) In simulating system $M^\sharp$.

Figure 5.8.: **Embedding Function.** There is an embedding function per pair of related states since one state in the simulated system may be related to multiple different states with different individuals in the simulating system.

for id ∈ Id implies

$$\forall\,('s^\sharp, s^\sharp) = r^\sharp \in R(M^\sharp) : \circledast \overset{r^\sharp}{\rightsquigarrow}_e f_h(id). \tag{5.45}$$

♢

Note 5.4.4 (Partial Order)**.** *The relation '$\preceq$' as given by Definition 5.4.2 is a partial order on evolving topology transition systems.* ♢

Figure 5.8 illustrates why it is necessary to give an embedding function per *pair* of related states, that is, why it is not sufficient to have an embedding function per state of M. The reason is that a state in M may be related to different states in $M^\sharp$, like s_1, and $s_1^\sharp$ and $s_1^\sharp{}'$, and an individual u of s_1 may well evolve into different individuals $u^\sharp$ and $u^\sharp{}'$ in the different states $s^\sharp$ and $s^\sharp{}'$.

5.4.1. Fairness

The notion of simulation relation as defined in Def. 5.4.2 extends naturally to fair ETTS. Namely by changing the requirement on paths as given by formula (5.42) in point 2 to

$$\begin{aligned}&\forall\,(s, s^\sharp) \in H\ \forall\,\pi \in \Pi_s^F(M)\ \exists\,\pi^\sharp \in \Pi_{s^\sharp}^F(M^\sharp)\ \forall\,i \in \mathbb{N}_0 :\\ &\quad(\pi^i, \pi^i) \in H \wedge (\pi^i, \pi^i) \sim_e (\pi^{\sharp i}, \pi^{\sharp i}),\end{aligned} \tag{5.46}$$

that is, instead of only requiring that the simulating system $M^\sharp$ can take corresponding transitions we require that it has corresponding fair paths.

We say that the fair ETTS $M^\sharp$ *simulates* the fair ETTS M, denoted by $M \preceq^F M^\sharp$, if there is a simulation relation H between M and $M^\sharp$ such that each initial state of M is in H-relation with at least one initial state of $M^\sharp$. To indicate a particular simulation relation, we may write $M \preceq_H^F M^\sharp$.

5.5. AEvoCTL* under Simulation

This section is the core of this chapter, and its core in turn is the soundness theorem, Theorem 5.5.11, stating that a simulation system is a sound abstraction wrt. AEvoCTL*.

Definition 5.5.1 (Abstract Structure wrt. Simulation)**.** *Let $\mathcal{S}$ be a signature and M and $M^\sharp$ compatible ETTS such that $M^\sharp$ simulates M, i.e. $M \preceq_H M^\sharp$. Let $\mathcal{M}$ be a canonical structure of $\mathcal{S}$ wrt. M and $\mathcal{M}^\sharp$ a canonical structure of $\mathcal{S}$ wrt. $M^\sharp$.*

We call $\mathcal{M}^\sharp$ an abstract structure *of $\mathcal{S}$ in $M^\sharp$ wrt. the* concrete structure *$\mathcal{M}$, denoted by*

$$\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp, \tag{5.47}$$

if and only if $\mathcal{M}^\sharp$ is an abstract structure *of $\mathcal{S}$ in each topology of $M^\sharp$ wrt. the* concrete structure *$\mathcal{M}$ in the corresponding topologies of M, that is, if*

$$\forall\, h = (s, s^\sharp) \in H, \mathscr{L}(s) \sqsubseteq^{(f_h, g_h)} \mathscr{L}^\sharp(s^\sharp) : \mathcal{M} \sqsubseteq^{(f_h, g_h)} \mathcal{M}^\sharp. \tag{5.48}$$

◇

Definition 5.5.2 (Corresponding Assignment wrt. Simulation)**.** *Let $\mathcal{S}$ be a signature and G and $G^\sharp$ two $\mathcal{S}$-compatible topologies such that (f, g) embeds G into $G^\sharp$. Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. G, $G^\sharp$, and (f, g).*

Let $\theta^\sharp \in Assign_{\mathcal{M}^\sharp}(V)$ and $\theta \in Assign_{\mathcal{M}}(V)$ be assignments of some logical variables $V \subseteq \mathcal{V}^T$ in the structures $\mathcal{M}^\sharp$ and $\mathcal{M}$.

We say θ and $\theta^\sharp$ correspond *wrt. corresponding paths $\pi \sim_H \pi^\sharp$ in M and $M^\sharp$, denoted by*

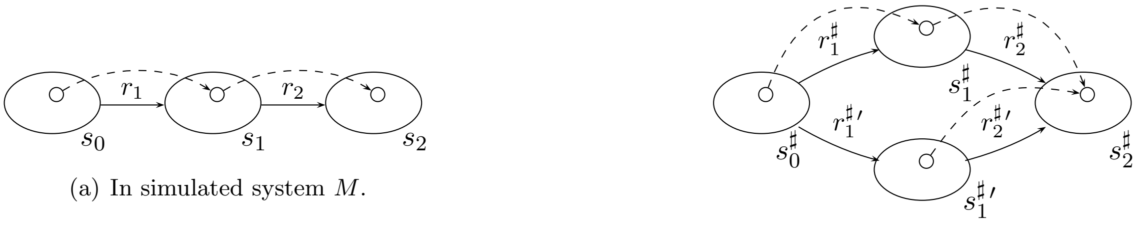
$$(\theta, \pi) \sim_H (\theta^\sharp, \pi^\sharp), \tag{5.49}$$

if and only if θ and $\theta^\sharp$ correspond wrt. to each pair of states in π and $\pi^\sharp$, that is, if $(\theta, \pi^k) \sim_H (\theta^\sharp, {\pi^\sharp}^k)$, for all $k \in \mathbb{N}_0$. ◇

Note that identity variables are bound to a particular identity . They cannot follow evolution, i.e. they cannot denote different things over time. Thus they cannot trace materialisation like destiny variables can. But, in the setting of abstract topologies, the precision of *the denoted thing* can change over time, that is, the denoted thing can be labelled with $\top$ at one time and carry fully precise information at another time. And the aliveness of the denoted thing can change over time.

The following is a direct consequence of the definition of simulation, abstract structure, corresponding assignments, and Lemma 5.3.5 – and the first step towards Theorem 5.5.11.

Corollary 5.5.3 (Simulation and Term Evaluation)**.** *Let M and $M^\sharp$ be two evolving topology transition systems over a signature $\mathcal{S}$ such that $M_1 \preceq_H M_2$, let $\mathcal{M} = (\iota, \mathcal{D})$ and $\mathcal{M}^\sharp = (\iota^\sharp, \mathcal{D}^\sharp)$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$, and let t be a universally quantified logical term over $\mathcal{S}$.*



(a) In simulated system M.

(b) In simulating system $M^\sharp$.

Figure 5.9.: **Corresponding Paths.** The path $s_0^\sharp, s_1^\sharp, s_2^\sharp$ in $M^\sharp$ corresponds to the path s_0, s_1, s_2 in M, while $s_0^\sharp, s_1^{\sharp\prime}, s_2^\sharp$ doesn't.

Then the evaluation of t in $s^\sharp$ is a safe over-approximation of the evaluation in s if these states are H-related, that is,

$$\forall\, h = (s, s^\sharp) \in H : \iota^\sharp[\![t]\!](\mathscr{L}(s^\sharp), f_h(\theta)). \implies \iota[\![t]\!](\mathscr{L}(s), \theta) \tag{5.50}$$

where θ is an assignment of the free variables in t in the topology $\mathscr{L}(s)$. ♢

Proof. Def. 5.4.2, Def. 5.3.2, and Lemma 5.3.5. □

Definition 5.5.4 (Corresponding States and Paths)**.** *Let H be a simulation relation between the evolving topology transition systems M and $M^\sharp$.*

1. *Let $s \in S(M)$ be a state of M and $s^\sharp \in S(M^\sharp)$ a state of $M^\sharp$.*

   We say s and $s^\sharp$ correspond*, denoted by $s \sim_H s^\sharp$, if and only if they are in simulation relation, i.e. $(s, s^\sharp) \in H$.*

2. *Let $\pi \in \Pi(M)$ be a path of M and $\pi^\sharp \in \Pi(M^\sharp)$ a path of $M^\sharp$.*

   We say π and $\pi^\sharp$ correspond, denoted by $\pi \sim_H \pi^\sharp$, if and only if for every $i \in \mathbb{N}_0$, states π^i and ${\pi^\sharp}^i$ correspond and individuals evolve and disappear correspondingly along the transitions $(\pi^i, \pi^{i+1}) \in R(M)$ and $({\pi^\sharp}^i, {\pi^\sharp}^{i+1}) \in R(M^\sharp)$. ♢

Note that this definition of corresponding paths is stronger compared to the notion used with simulation (or bisimulation) for Kripke structures; the difference is that it requires that *corresponding* transitions are taken, while the simulating system may have multiple transition from one state to others which not necessarily provide that individuals evolve sanely. This addition is again due to the fact that life-cycle properties in ETTS depend not only on the state, but also on the taken transition.

As an example consider Figure 5.9 and assume that states s_i and $s_i^\sharp$ are related by H for $0 \leq i \leq 2$ and in addition state s_1 to $s_1^{\sharp\prime}$. Note that the individual u evolves correspondingly to r_1 along $r_1^\sharp$ but not along $r_1^{\sharp\prime}$. This is no contradiction to s_0 being in H-relation to $s_0^\sharp$ because Def. 5.4.2.2 only requires that there *is* a transition along which individuals evolve correspondingly. Neither is it a contradiction with respect to ${s_1^\sharp}'$ because from there on the individual evolves correspondingly.

This absence of contradictions is legitimate: the criterion in the definition is sufficient to ensure that all bad paths of the simulated system have a correspondence in the simulating system. In the classical definition, s_0, s_1, s_2 and $s_0^\sharp, {s_1^\sharp}', s_2$ are segments of a corresponding path. It is sufficient that the states in the path are pairwise related as the transitions don't contribute anything. In our setting, these path fragments should not be considered corresponding as individuals don't evolve correspondingly along them. The following Lemma provides corresponding paths even for this stronger notion.

Lemma 5.5.5 (Corresponding Path and Assignment Suffix)**.** *Let M and $M^\sharp$ be two ETTS compatible with signature $\mathcal{S}$ and $M \preceq_H M^\sharp$.*

1. *Let π and $\pi^\sharp$ be corresponding paths in M and $M^\sharp$.*

   Then $\pi/k \sim_H \pi^\sharp/k$ for each $k \in \mathbb{N}_0$.

2. *Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$. Let θ and $\theta^\sharp$ be corresponding assignments of some logical variables $V \subseteq \mathcal{V}^T$ in $\mathcal{M}$ and $\mathcal{M}^\sharp$. Then*

   a) $\theta/k \sim_H \theta^\sharp/k$ for each $k \in \mathbb{N}_0$ and

   b) $\theta/k = \varepsilon$ if and only if $\theta^\sharp/k = \varepsilon$. ♢

Proof. See Section A.2. □

Lemma 5.5.6 (Corresponding Path)**.** *Let H be a simulation relation between the evolving topology transition systems M and $M^\sharp$.*

Let $s \in S(M)$ and $s^\sharp \in S(M^\sharp)$ be two corresponding states. Then for each path of M starting at s, there is a corresponding path of $M^\sharp$ starting at $s^\sharp$. ♢

Proof. See Section A.2. □

Definition 5.5.7 (Corresponding Evolution Chain)**.** *Let H be a simulation relation between the evolving topology transition systems M and $M^\sharp$.*

Let $\pi \in \Pi(M)$ and $\pi^\sharp \in \Pi(M^\sharp)$ be two corresponding paths, i.e. the pairs $h_i := (\pi^i, {\pi^\sharp}^i)$ are in simulation relation H. Let $id \in U^{\circledcirc}(\pi^0)$ be an individual in the first state of π and δ an evolution chain of id along π, i.e. $\delta \in \Delta(id, \pi)$.

Let $\delta^\sharp$ be an evolution chain of $f_{h_0}(id)$ along $\pi^\sharp$, i.e. $\delta^\sharp \in \Delta(f_{h_0}(id), \pi^\sharp)$. We say δ and $\delta^\sharp$ correspond *wrt. π and $\pi^\sharp$, denoted by*

$$(\delta, \pi) \sim_H (\delta^\sharp, \pi^\sharp), \tag{5.51}$$

if and only if $\delta^\sharp = f_{h_0}(\delta(0)), f_{h_1}(\delta(1)), \ldots$ ♢

Lemma 5.5.8 (Corresponding Evolution Chain). *Let H be a simulation relation between the evolving topology transition systems M and $M^\sharp$.*

Let $\pi \in \Pi(M)$ and $\pi^\sharp \in \Pi(M^\sharp)$ be two corresponding paths. Then for each evolution chain of an individual $id \in U^{\circledcirc}(\pi^0)$ along π, there is a corresponding evolution chain of $f(id)$ along $\pi^\sharp$. ♢

Proof. Similar to the proof of Lemma 5.5.6 by induction over the finite prefixes of δ. The evolution chain $\delta^\sharp$ is inductively constructed from δ employing the property of the simulation relation that individuals evolve and disappear accordingly along corresponding transitions. □

Lemma 5.5.9 (Corresponding Assignment). *Let M and $M^\sharp$ be two evolving topology transition systems over identities Id and $Id^\sharp$ compatible with signature $\mathcal{S}$ and $M_1 \preceq_H M_2$. Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$.*

Let $\theta^\sharp \in Assign_{\mathcal{M}^\sharp}(V)$ be an assignment of some logical variables $V \subseteq \mathcal{V}^T$ in the structure $\mathcal{M}^\sharp$.

1. *Let $x \in V \cap \mathcal{V}^T$ be an identity variable and let $(s, s^\sharp) \in H$ be a pair of corresponding states. Then the set of assignments θ corresponding to a modification of $\theta^\sharp$ comprises the modifications of assignments θ corresponding to the unmodified $\theta^\sharp$, that is, if*

$$\forall\, id^\sharp \in Id^\sharp\ \forall\, \theta \in Assign_{\mathcal{M}}(V), (\theta, s) \sim_H (\theta^\sharp[x \mapsto id^\sharp], s^\sharp) : \mathcal{M}[\![\phi]\!](s, \theta) = 1 \tag{5.52}$$

then

$$\forall\, \theta \in Assign_{\mathcal{M}}(V), (\theta, s) \sim_H (\theta^\sharp, s^\sharp)\ \forall\, id \in Id : \mathcal{M}[\![\phi]\!](s, \theta[x \mapsto id]) = 1. \tag{5.53}$$

2. *Let $(s, s^\sharp) \in H$ be a pair of corresponding states. Then*

$$\begin{aligned}&\forall\, \pi^\sharp \in \Pi_{s^\sharp}(M^\sharp)\\ &\quad \forall\, \theta^{\sharp\prime} = \theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta^\sharp(\boldsymbol{x}_i)(0), \pi^\sharp), 1 \le i \le n\\ &\quad\quad \forall\, \pi \in \Pi_s(M), \pi \sim_H \pi^\sharp\ \forall\, \theta' \in Assign(\pi), (\theta', \pi) \sim_H (\theta^{\sharp\prime}, \pi^\sharp) :\\ &\quad\quad\quad \mathcal{M}[\![\psi]\!](\pi, \theta') = 1\end{aligned} \tag{5.54}$$

implies

$$\begin{aligned}&\forall\, \theta \in Assign(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp)\ \forall\, \pi \in \Pi_s(M)\\ &\quad \forall\, \theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), 1 \le i \le n :\\ &\quad\quad \mathcal{M}[\![\psi]\!](\pi, \theta') = 1.\end{aligned} \tag{5.55}$$

♢

Proof. See Section A.2. □

Lemma 5.5.10 (Soundness)**.** *Let M and $M^\sharp$ be two evolving topology transition systems over a signature $\mathcal{S}$ such that $M \preceq_H M^\sharp$ and let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$.*

1. *Let ϕ be a $\odot$-free state formula of AEvoCTL* over $\mathcal{S}^\sharp$. Let $s \in S(M)$ and $s^\sharp \in S(M^\sharp)$ be corresponding states, i.e. $s \sim_H s^\sharp$, and let $\theta^\sharp \in Assign(s^\sharp)$ be an assignment of the free variables of ϕ in $s^\sharp$.*

   Then
   $$M^\sharp, s^\sharp, \theta^\sharp \models \phi \implies M, s, \theta \models \phi \tag{5.56}$$
   for each assignment $\theta \in Assign(s)$ of the free variables of ϕ in s corresponding to $\theta^\sharp$ wrt. $(s, s^\sharp)$.

2. *Let ψ be a $\odot$-free path formula of AEvoCTL* over $\mathcal{S}^\sharp$. Let $\pi \in \Pi(M)$ and $\pi^\sharp \in \Pi(M^\sharp)$ be corresponding paths in M and $M^\sharp$, i.e. $\pi \sim_H \pi^\sharp$, and let $\theta^\sharp \in Assign(\pi^\sharp)$ be an assignment of the free variables of ψ in $\pi^\sharp$.*

   Then
   $$M^\sharp, \pi^\sharp, \theta^\sharp \models \psi \implies M, \pi, \theta \models \psi \tag{5.57}$$
   for each assignment $\theta \in Assign(\pi)$ of the free variables of ψ in π corresponding to $\theta^\sharp$.

If $M^\sharp$ simulates M preserving appearance, i.e. if $M \preceq^\odot M^\sharp$, then (5.56) *and* (5.57) *also hold if ϕ and ψ are not $\odot$-free.*

Proof. See Section A.2. □

The following theorem is a direct consequence of Lemma 5.5.10.

Theorem 5.5.11 (Soundness)**.** *Let M and $M^\sharp$ be two evolving topology transition systems over a signature $\mathcal{S}$ such that $M_1 \preceq_H M_2$. Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$.*

Then for every $\odot$-free AEvoCTL formula φ over $\mathcal{S}$, $M^\sharp \models \varphi$ implies $M \models \varphi$. If $M \preceq^\odot M^\sharp$ then the implication also holds if φ is not $\odot$-free.* ◇

Theorem 5.5.11 ensures that the abstraction comprises all behvaiour of the original structure, but possibly more.

5.5.1. Fairness

With the notion of simulation between fair ETTSs from Section 5.4.1, a Corresponding Path Lemma similar to Lemma 5.5.6 can be established, the proof closely resembles that of Lemma 5.5.6.

A fair variant of the Soundness Lemma similar to Lemma 5.5.10 and based on the fair notion of corresponding paths then gives rise to a fair variant of Theorem 5.5.11.

5.6. Bisimulation

Similar to simulation, one can define bisimulation equivalence for evolving topology transition systems. For the scope of this work, simulation is most relevant as we are discussing abstractions of ETTS, in particular finite ones; due to the unbounded nature of ETTS, there is in general no hope for a bisimulation between an ETTS and a finite-state transition system.

The definition as such is similar to Def. 5.4.2, adding the complement of item 2, that is, considering the given pair of related states, each transition in $M^\sharp$ has to have a corresponding transition in M. Then the Corresponding Path Lemma 5.5.6 and the Soundness Lemma becomes bidirectional, such that the Soundness Theorem similar to Theorem 5.5.11 would state that the bisimilar ETTS satisfies the same AEvoCTL* and in addition EEvoCTL* properties as the simulated one.

5.7. Discussion

Our approach is basically standard for transition systems as laid out in Section 5.1. The major difference is that we've got to take evolution into account. We solve this by an introduction of according evolution in Def. 5.4.1, which shows to be sufficient to establish the Soundness Theorem.

A minor difference is that we consider abstract local states, in the classical setting, it is typically assumed that all (or at least the subset of the relevant) atomic propositions are kept. We employ the lattice theory (cf. Chapter 2) which is also employed in the domain which is commonly known under the name of abstract interpretation.

A particularity of our approach is that we, in a sense, separate abstraction in the ETTS from abstraction in the logic. On the ETTS side, the abstract transition system still has regular topologies as labelling, they're only abstract in their local state and link name labelling. This preserving of the topology form is one pre-requisite of the syntactical transformations discussed in Chapter 9, that is, obtaining a definition of the abstract transition system by a syntactical transformation of a higher-level language.

On the logic side, there is the abstract structure with abstract interpretation, in particular, of function symbols. In Chapters 6 and 9, we discuss how to canonically obtain a quite coarse variant heuristically, yet there are options for refinement by providing more "informed" definitions. In this point, this approach is fundamentally different from the above named abstract interpretation. We introduce the notion of a tight embedding, and by Chapter 2 we could directly have notions of best abstract structures wrt. given abstract domains, yet we don't provide a procedure to obtain these. It is one aspect of the abstraction discussed in the following Chapter 6 that it is just *not* the best one in many regards, instead precision is traded for easy computability of the abstract transition relation (cf. Chapter 9).

Our orientation on a notion of embedding is a direct descendent of the abstract interpretation-based approach of [155], with the only difference that our notion is on topologies (or graphs) while theirs is on representations of graphs in form of logical structures. The most closely related derivative of [155] is [190, 191], which, as discussed in Chapter 4, study the verification of the specification logic VTL/ETL which is similar to EvoCTL*. Their soundness proofs are not explicitly based on an elaborated theory of simulation, but they prove rather directly that VTL/ETL formulae evaluation is sound on abstract traces wrt. to concrete traces. Recall from Chapter 3 that their computational model is a set of states, instead of a transition system in our case.

Another difference is related to the above named separate considerations of the ETTS and the logic side. As the computational model of [190, 191] are logical structures, the logic doesn't have expressions like

$$\sigma(x \rightarrowtail \lambda) > 0 \tag{5.58}$$

but rather a predicate $p(x)$ denoting just (5.58). Thereby, the interpretation of formulae is directly determined by the states, the logical structures. The states of the abstract system then directly provide (abstract) interpretations of the predicate symbols. This setting makes certain things easier to handle than in our theoretical machinery, but it is a kind of pre-abstraction into predicates we wanted to avoid in particular for discussions in Chapters 6 and 9.

6. Data-Type Reduction

Having precisely defined what kind of systems we referred to in the introduction by Chapter 3 and a corresponding property specification logic in Chapter 4, we now investigate the applicability of the *Data-Type Reduction* [127] (DTR) abstraction in the context of ETTS. With the particular symmetry reduction *Query Reduction* [127] from Chapter 7, we obtain in Chapter 8 a finitary abstraction approach for ETTS and AEvoCTL* formulae in prenex normal form.

Recall from Chapter 3 that we introduced both, an untyped and a typed variant of ETTS. While in the previous sections, the presentation has been reduced to the untyped case, we assume a typed setting from now on.

That is, we assume that the set of identities *Id* is partitioned into n partitions, i.e. $\mathit{Id}_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} \mathit{Id}_n$. We think of each partition as providing the identities of one type of processes, that is, in the following we assume signatures $\mathcal{S}$ with identity types $T_1, \ldots, T_n \in \mathcal{T}(\mathcal{S})$ instead of just the single type T. In addition, we assume the set of local states Σ is partitioned into $\Sigma_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} \Sigma_n$, that is, each type of individuals has its own range of state labels, and that they're each complete lattices (which can always be achieved by adding a top element $\top$, cf. Chapter 2).

Types are important for DTR because we want to have one summary node for each type. It should over-approximate individuals of its type, but not more.

The chapter is structured as follows. In Section 6.1, we formally define DTR for our setting of ETTS. Section 6.2 establishes soundness employing the machinery of Chapter 5.

While soundness is easy insofar as a system with complete "chaos" is clearly sound, the more interesting question is which properties are preserved or *reflected*. That is, which EvoCTL* properties hold in the abstraction if they hold in the concrete system. We address this question in Section 6.3.

The outcome is that DTR is not exact, i.e. sound but not complete (which is not surprising). The next interesting questions then are options for refinement, and before that the decidability of whether a given (abstract) counter-example is spurious or not, i.e. whether it doesn't concretises to a computation path in the original system. We only briefly elaborate on this topic in Section 6.4 as it is one of the main subjects of [167].

The overall aim is to obtain a finite abstract transition system. DTR as such firstly only addresses the infiniteness stemming from the unbounded number of individuals in ETTS. In Section 6.5 we discuss first ideas to also treat the possibly unbounded number of links, even with finitely many individuals. We'll get back to this topic in the more application oriented setting of Chapter 9. Section 6.6 discusses related work.

6.1. Data-Type Reduction

We define the Data-Type Reduction (DTR) abstraction in three steps. Firstly, as a choice of individuals to represent concretely, secondly, its application to topologies, and thirdly its application to whole ETTS.

6.1.1. The Spotlight

Definition 6.1.1 (Data-Type Reduction (DTR))**.** *Let $Id = Id_1 \,\dot\cup\, \dots \,\dot\cup\, Id_n$ be a set of identities partitioned into n partitions.*

A DTR *of Id is a finite (possibly empty) set of pairs of partitions of Id and subsets of the partitions such that each partition appears at most once, i.e. a set*

$$D = \{(d_{j_1}, Id_{j_1}), \dots, (d_{j_m}, Id_{j_m})\} \subsetneq \bigcup_{1 \leq i \leq n} \mathfrak{P}(Id_i) \times \{Id_i\} \tag{6.1}$$

is called DTR if and only if

1. *$d_{j_i} \subseteq Id_{j_i}$, $1 \leq i \leq m$,*
2. *each index j_i, $1 \leq i \leq m$, is from $\{1, \dots, n\}$, and*
3. *indices are pairwise different, i.e. $j_i \neq j_k$ for each $1 \leq i \neq k \leq m$.*

We write $u \in D$ to denote that u is in one of D's subsets, i.e. as an abbreviation for $u \in d_{j_1} \cup \dots \cup d_{j_m}$. We say a partition Id_i of Id is considered *by a DTR D, denoted by $Id_i \in D$, if and only if there is a subset $d_i \subseteq Id_i$ such that $(d_i, Id_i) \in D$; and we write $Id_i \notin D$ if Id_i is not considered by D.*

A DTR D is called complete *if each partition of Id is considered.* ◇

In other words: D denotes a "spotlight" which lights the identities in the d_i such that the complement lie in shadows. Partitions not affected by D remain completely lighted (in this figurative intuition).

Given a DTR D on the identities, we set $Id = Id_1 \,\dot\cup\, \dots \,\dot\cup\, Id_n$, set

$$Id^\sharp := Id_1^\sharp \cup \dots \cup Id_n^\sharp \tag{6.2}$$

where

$$Id_i^\sharp := \begin{cases} d_i \,\dot\cup\, \{\mathsf{C}_i\} & \text{, if } (d_i, Id_i) \in D \\ Id_i & \text{, otherwise} \end{cases} \tag{6.3}$$

Then D induces the mapping $f_d : Id \to Id^\sharp$ as

$$id \mapsto \begin{cases} \mathsf{C}_i & \text{, if } (d_i, Id_i) \in D \text{ and } id \in Id_i \setminus d_i \\ id & \text{, otherwise} \end{cases} \tag{6.4}$$

that is, f_d summarises, per partition of Id, the shadows to a single auxiliary identity. As the identities in shadows are the complement of the lighted ones, the symbol for the auxiliary identity is the complement symbol. Note that this node is (by choice) added even if $d_i = Id_i$.

Definition 6.1.2 (Finite DTR)**.** *A DTR D on a set of identities*

$$Id = Id_1 \,\dot{\cup}\, \ldots \,\dot{\cup}\, Id_n \tag{6.5}$$

is called finite *if and only if the subset component of each pair $(d_j, Id_j) \in D$ is finite, i.e. if d_j is finite.* ♢

6.1.2. Application to Topology

Applying a DTR to a topology informally means completely dismissing all information about the individuals in the shadows, except for their type.

Definition 6.1.3 (DTR'ed Topology)**.** *Let $G = ((U^{\circledcirc}, U^{\not\circledcirc}), L, \psi, \sigma, \lambda)$ be a (Σ, Λ)-topology over Id, both Id and Σ partitioned into n partitions.*

Let D be a DTR of Id. Then $D(G)$ denotes the topology

$$((U^{\circledcirc\sharp}, U^{\not\circledcirc\sharp}), L^{\sharp}, \psi^{\sharp}, \sigma^{\sharp}, \lambda^{\sharp}) \tag{6.6}$$

over identities

$$Id^{\sharp} = d_{j_1} \,\dot{\cup}\, \{\mathtt{C}_{j_1}\} \,\dot{\cup}\, \ldots \,\dot{\cup}\, d_{j_m} \,\dot{\cup}\, \{\mathtt{C}_{j_m}\} \,\dot{\cup} \bigcup_{Id_i \notin D, 1 \leq i \leq n} Id_i \tag{6.7}$$

with fresh identities $\mathtt{C}_{j_i}$, $1 \leq i \leq n$, and

1. *the equality function on $Id^{\sharp}$ defined as*

$$eq_{Id}{}^{\sharp}(u_1^{\sharp}, u_2^{\sharp}) = \begin{cases} 1/2 & \text{, if } u_1^{\sharp} = u_2^{\sharp} = \mathtt{C}_i, 1 \leq i \leq n, \\ 1 & \text{, if } u_1^{\sharp} = u_2^{\sharp} \in Id^{\sharp} \cap Id, \\ 0 & \text{, otherwise,} \end{cases} \tag{6.8}$$

2. *aliveness is preserved precisely for lighted individuals and completely lost (in abstraction) for those in the shadows, i.e.*

$$U^{\circledcirc\sharp} = f_d(U^{\circledcirc}) \cup \{\mathtt{C}_{j_1}, \ldots, \mathtt{C}_{j_m}\}, \tag{6.9}$$

   and

$$U^{\not\circledcirc\sharp} = f_d(U^{\not\circledcirc}) \cup \{\mathtt{C}_{j_1}, \ldots, \mathtt{C}_{j_m}\}, \tag{6.10}$$

3. *local states are preserved precisely for lighted individuals and completely lost (in abstraction) for those in the shadows, i.e.*

$$\sigma^{\sharp}(u^{\sharp}) = \begin{cases} \top_{\Sigma_i} & \text{, if } u^{\sharp} = \mathtt{C}_i, \\ \sigma(u) & \text{, otherwise,} \end{cases} \tag{6.11}$$

4. *links are kept if they* originate *within the spotlight, with changed destination in case it happens to lie in the shadows, and additional self-links on the shadow individuals labelled with* $\top \in \Lambda^\sharp$*, i.e.*

$$L^\sharp = \{\ell \in L \mid ini(\ell) \in D\} \cup \{(\mathsf{C}_i, \mathsf{C}_i) \mid d_i \in D\} \tag{6.12}$$

$$\begin{aligned} \psi^\sharp = \{&\ell^\sharp \mapsto (f_d(u_1), f_d(u_2))) \mid \psi(\ell) = (u_1, u_2), \ell^\sharp \in L^\sharp\} \\ &\cup \{(\mathsf{C}_i, \mathsf{C}_i) \mapsto (\mathsf{C}_i, \mathsf{C}_i) \mid d_i \in D\}, \end{aligned} \tag{6.13}$$

and

$$\lambda^\sharp = \lambda|_{L^\sharp} \cup \{(\mathsf{C}_i, \mathsf{C}_i) \mapsto \top \mid d_i \in D\}, \tag{6.14}$$

5. *the links* from *the shadows are catered for in a changed definition of the navigation functions, namely*

$$\rightarrowtail^\sharp_\lambda : L^\sharp \to U^\sharp \,\dot\cup\, \{\top\} \tag{6.15}$$

with

$$\rightarrowtail^\sharp_\lambda(u^\sharp) = \top \tag{6.16}$$

if $u^\sharp \in U^\sharp \setminus d$ *is from the shadows and the regular multi-set comprehension if* $u^\sharp \in d$*, i.e.*

$$\rightarrowtail^\sharp_\lambda(u^\sharp) = \rightarrowtail_\lambda(u^\sharp). \tag{6.17}$$

With top element $\top$*,* $\mathrm{dom}(\rightarrowtail^\sharp_\lambda)$ *becomes a complete join semi-lattice.* ♢

Note that the definition of $eq_{Id}{}^\sharp$ is the place where we have one similarity to the summary nodes of, e.g. [155] (cf. Chapter 5).

Lemma 6.1.4 (Finite Topology). *Let G be a (Σ, Λ)-topology over Id, both Id and Σ partitioned into n subsets.*

Given a DTR D, the topology $D(G)$ is finite if and only if d is finite, all partitions of Id not considered by D are finite, and all identities $id \in D$ and in the partitions not considered by D have finite out-degree. ♢

Proof. The topology $G^\sharp := D(G)$ is called finite if both, the set of nodes $U(G^\sharp)$ and the set of edges $L(G^\sharp)$ is finite.

The set of nodes becomes finite because it is a subset of $Id^\sharp$ which by (6.7) comprises the subsets from D, which are finite as D is finite, and finitely many fresh identities. The rightmost union in (6.7) becomes finite by premises.

The set of edges becomes finite because the definition of $L^\sharp$ only considers nodes from $Id^\sharp$ and these have finite out-degree by premises. □

Corollary 6.1.5 (Finite and Complete Topology DTR). *Let G be a (Σ, Λ)-topology over Id with a finite upper bound on the out-degree of individuals, both Id and Σ partitioned into n subsets.*

Let D be a finite and complete DTR. Then $D(G)$ is finite.

Proof. Lemma 6.1.4 □

6.1.3. Application to ETTS

Given the application to topologies, the application to ETTS is straight-forward. Namely basically by state with the transition relation becoming an exists/exists abstraction.

Definition 6.1.6 (DTR of ETTS)**.** *Let $M = (S, S_0, R, \mathscr{L}, e)$ be an (Σ, Λ)-ETTS over Id, both Id and Σ partitioned into n partitions.*

Let D be a DTR of Id. Then $D(M)$ denotes the ETTS

$$(S^\sharp, {S_0}^\sharp, R^\sharp, \mathscr{L}^\sharp, e^\sharp) \tag{6.18}$$

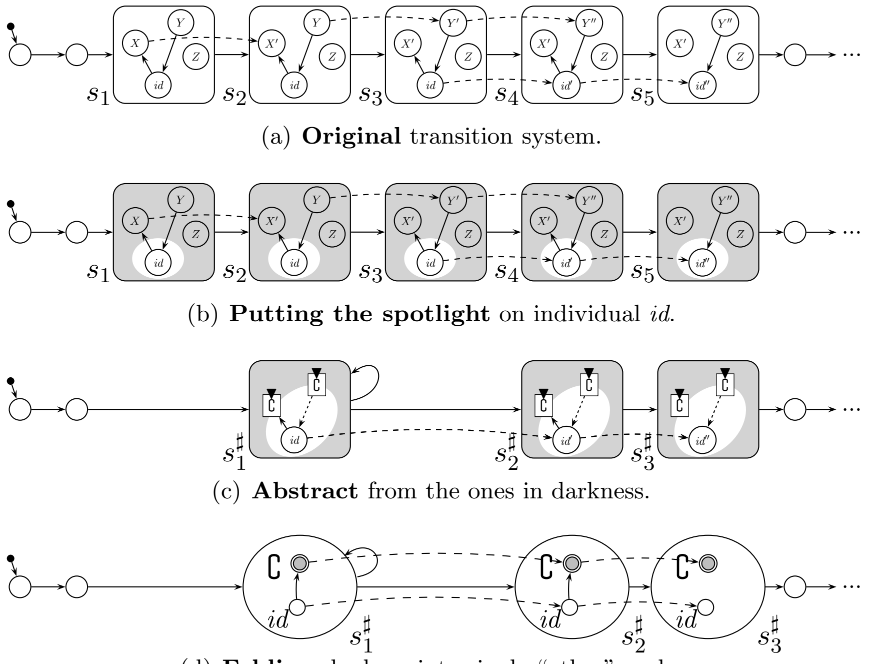
where

- *the sets of states and initial states comprises the topologies used to label S under D, i.e.*
$$S^\sharp = D(\mathscr{L}(S)) = \{D(\mathscr{L}(s)) \mid s \in S\}, \tag{6.19}$$
$${S_0}^\sharp = D(\mathscr{L}(S_0)), \tag{6.20}$$
- *the transition relation is an exists/exists abstraction of R, i.e.*
$$\begin{aligned} R^\sharp &= \{(D(s), D(s')) \mid (s, s') \in R\} \\ &\quad (= \{(s^\sharp, s^{\sharp\prime}) \in S^\sharp \times S^\sharp \mid \\ &\qquad \exists\, s \in S, s' \in S : (s, s') \in R \wedge s^\sharp = f_d(s) \wedge s^{\sharp\prime} = f_d(s')\}), \end{aligned} \tag{6.21}$$
- *as topologies are used as states (see above), states are labelled with itself, i.e.*
$$\mathscr{L}^\sharp = id_{S^\sharp} \ (= \{s^\sharp \mapsto s^\sharp \mid s^\sharp \in S^\sharp\}), \tag{6.22}$$
- *the evolution annotation*
$$e^\sharp = \{e^\sharp\langle r^\sharp\rangle \mid r^\sharp \in R^\sharp\} \tag{6.23}$$
preserves evolution of lighted individuals and considers other individuals to evolve constantly, i.e.
$$\begin{aligned} e^\sharp\langle r^\sharp\rangle &= \{(\mathsf{C}_{j_i}, \mathsf{C}_{j_i}) \mid 1 \leq i \leq m\} \\ &\qquad \cup \bigcup_{(D(s), D(s')) = r^\sharp} \{(f_d(u), f_d(u') \mid (u, u') \in e\langle(s, s')\rangle\} \end{aligned} \tag{6.24}$$

♢

Figure 6.1 illustrates the application of a DTR to a transition system on a prefix of a computation path (employing a graphical representation of topologies slightly different from our usual one).

Given the single-sorted topologies Figure 6.1(a), we choose the DTR $D = \{(\{id\}, Id)\}$ as shown in Figure 6.1(b). Applying D to Figure 6.1(b) yields Figure 6.1(c) where links



(a) **Original** transition system.

(b) **Putting the spotlight** on individual *id*.

(c) **Abstract** from the ones in darkness.

(d) **Folding** shadows into single “other” node.

Figure 6.1.: **DTR’d ETTS.**

into the shadows are preserved but point to the light/shadow border now, links *from* the shadows into the lights are not preserved. They’re shown dashed in Figure 6.1(c) because the abstraction does preserve (and over-approximates) interactions of the shadows with the concrete part.

Viewed as a proper topology, it can be depicted as in Figure 6.1(d). Note that the new self-loop on $s_1^\sharp$ is an effect of the exists/exists abstraction. Under DTR, states s_2 and s_3 are not distinguishable, but have the same abstract representation. As there is a transition between both, there is a self-transition at the abstract representation.

Two more illustrations of special cases can be found in Section B.2 (Figures B.1 and B.2).

Lemma 6.1.7 (Finite ETTS)**.** *Let M be an (Σ, Λ)-ETTS over Id, both Id and Σ partitioned into n subsets.*

Given a DTR D, the ETTS $D(M)$ is finite if and only if all topologies $G^\sharp \in \mathrm{dom}(\mathscr{L}(D(M)))$ are finite and if Σ is finite. ♢

Proof. Lemma 6.1.4. □

Corollary 6.1.8 (Finite and Complete DTR for ETTS)**.** *Let M be an (Σ, Λ)-ETTS over Id with a finite upper bound on the out-degree of individuals, Σ finite, and both Id and Σ partitioned into n subsets.*

Let D be a finite and complete DTR. Then $D(G)$ is finite for each $G \in \mathrm{dom}(\mathscr{L}(M))$.

Proof. Lemma 6.1.7 □

The most prominent example of immediately finite results are ETTS with finite local state and the single-link property (cf. Chapter 3).

6.2. Soundness

We're using the Soundness Theorem, Theorem 5.5.11, of Chapter 5 by showing that the DTR'd ETTS simulates the original.

Let M be an ETTS over *Id*, D a DTR, and $M^\sharp := D(M)$. Then the simulation relation is

$$H = \{(s, D(\mathscr{L}(M)(s))) \mid s \in S(M)\}. \tag{6.25}$$

Given a pair of related states $h = (s, s^\sharp) \in H$, the pair (f_h, g_h) with

- $f_h = f_d$ and
- $g_h = \{\ell \mapsto \ell \mid \mathit{ini}(\ell) \in D\} \cup \{\ell \mapsto (\mathsf{C}_i, \mathsf{C}_i) \mid d_i \in D, \mathit{ini}(\ell) \in \mathit{Id}_i, \mathit{ini}(\ell) \notin D\}$,

is an associated embedding in the sense of Def. 5.2.1 because

- we consider the identity function as Galois connection between the (sort related) partitions of Σ and $\Sigma^\sharp$, recall that we assume in this chapter that Σ is a complete lattice,
- similarly for Λ,
- the local state property Def. 5.2.1.1a is clearly satisfied for lighted individuals, and for the others by the assignment of $\top_{\Sigma_i}$ as local state,
- the aliveness property Def. 5.2.1.1b is clear for lighted ones and by having the other in both sets, alive and non-alive, i.e. $\mathsf{C} \in U^{\circledcirc} \cap U^{\not\circledcirc}$,
- the equality property Def. 5.2.1.1c is satisfied by definition of ${eq_{\mathit{Id}}}^\sharp$,
- the link embedding property Def. 5.2.1.2 is satisfied because links originating in the concrete part are kept and others map to the respective self-links (cf. Def. 6.1.3), and
- consistency as required by Def. 5.2.1.3 is given, because source and destination of abstract links are defined consistently with the individuals' embedding.

Thus we have $\mathscr{L}(s) \sqsubseteq^{(f_h, g_h)} \mathscr{L}^\sharp(s^\sharp)$ as required by Def. 5.4.2.1.

For the transition property, i.e. Def. 5.4.2.2 let $(s, s') \in R(M)$ be a transition of M. Then there is a corresponding transition in $M^\sharp$ by definition, namely $(s^\sharp, s^{\sharp\prime})$. That individuals appear and disappear accordingly, i.e. that (5.38) and (5.39) are satisfied, is clear for the lighted ones. They behave in $M^\sharp$ exactly as they do in M. There is in particular never a conflict: if *id* disappears along r_1 and evolves along r_2, then the destination states are never merged because they are different topologies.

For the shadow ones, note that the definitions of appearance and disappearance in Chapter 3 are carefully crafted such that the desired properties are obtained from C_i being both, alive and non-alive. Similarly, we even have corresponding appearance. It is clear for the lighted individuals (similar argument as above), and by being in both sets for the others.

Now theorem Theorem 5.5.11 yields that AEvoCTL* properties holding in $M^\sharp$ under an abstract structure, i.e. evaluating to 1, also evaluate to 1 in M.

An abstract structure can be obtained as follows if we assume that we only have the restricted set of function symbols operating and yielding T and L types (cf. Section 4.2.1).

For the domains not covered by the definition of canonical domain, use the minimal lattices, i.e. simply add a $\top$ to each and use the canonical embedding (cf. Chapter 2).

Assign each function symbol not operating on T by a monotone interpretation, i.e.

$$\iota^\sharp(f)(d_1^\sharp, \ldots, d_k^\sharp) = \begin{cases} \iota(f)(d_1^\sharp, \ldots, d_k^\sharp) & \text{, if } d_i^\sharp \neq \top, 1 \leq i \leq k \\ \top & \text{, otherwise} \end{cases} \tag{6.26}$$

Those having T parameters are defined to yield $\top$ for C_i, the identity representing the abstracted individuals.

The function symbols yielding T are actually only the link navigation functions and dereference. Their interpretation in the abstract structure yields $\top_T$ for $\top_L$.

Note that it is still possible to obtain one of the C_i from a link, for example by dereference, and to compare it against other links, but further navigation over C_i will then yield $\top_T$ (also cf. Chapter 9).

Theorem 6.2.1 (DTR Soundness). *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible (Σ, Λ)-ETTS over Id, and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. M.*

Let φ be an AEvoCTL formula over $\mathcal{S}$ and D a DTR. Then*

$$D(M) \models \varphi \implies M \models \varphi. \tag{6.27}$$

◇

Proof. Theorem 5.5.11 with simulation relation as given above. □

Note that the abstraction is (by far) not the best for most purposes. A trivial but useful observation is that it's best on the focused ones in terms of local state precision. The main benefit will become clear in Chapter 9. Namely that we're able to compute (or closely approximate) $D(M)$ from a higher-level language description by simple syntactical transformations. With respect to a higher-level language, there are additional reflected properties as we'll see in Chapter 9 when we have the high-level language. For example, the atomic execution of operations, the property that communication from the shadows is not complete chaos.

The following definition is useful for Section 6.4 on refinement.

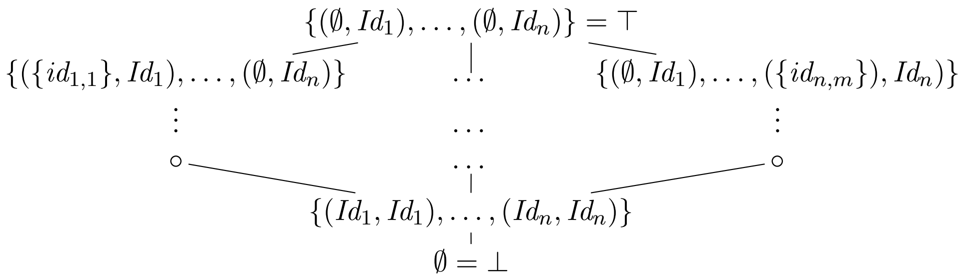

$\{(\emptyset, Id_1), \ldots, (\emptyset, Id_n)\} = \top$
$\{(\{id_{1,1}\}, Id_1), \ldots, (\emptyset, Id_n)\}$
$\ldots$
$\{(\emptyset, Id_1), \ldots, (\{id_{n,m}\}), Id_n)\}$
$\vdots$
$\ldots$
$\vdots$
$\circ$
$\ldots$
$\circ$
$\{(Id_1, Id_1), \ldots, (Id_n, Id_n)\}$
$\emptyset = \bot$

Figure 6.2.: **DTR Lattice.** $\top$ is the least and $\bot$ the most precise DTR of a set of identities $Id = Id_1 \,\dot\cup\, \ldots \,\dot\cup\, Id_n$.

Definition 6.2.2 (Precision Order on DTRs)**.** *Let*

$$D_1 = \{(d_{1,1}, Id_{j_{1,1}}), \ldots, (d_{1,n}, Id_{j_{1,n}})\} \tag{6.28}$$

and

$$D_2 = \{(d_{2,1}, Id_{j_{2,1}}), \ldots, (d_{2,m}, Id_{j_{2,m}})\} \tag{6.29}$$

be two of Id.

We say D_1 is more precise *than D_2, denoted by $D_1 \sqsubseteq D_2$, if and only if for each $1 \leq i \leq n$ there is a $1 \leq k \leq m$ such that $Id_{j_{1,i}} = Id_{j_{2,k}}$ and $d_{1,i} \supseteq d_{2,k}$.* ♢

Without proofs we note the following.

Note 6.2.3 (DTR Lattice)**.** *The precision order on DTRs of a set of identities Id is a complete lattice.*

The meet-operator on two non-empty DTRs is a two-step set-union, firstly joining both DTRs and then joining DTRs of the same partition of Id. If one of the operands is the empty DTR, the result is the empty DTR. The empty DTR is the bottom-element.

The join-operator on two (possibly) empty DTRs is a two-step set-intersection, firstly joining (sic!) both DTRs and then intersecting the DTRs of the same partition of Id. The DTR

$$\{(\emptyset, Id_1), \ldots, (\emptyset, Id_n)\} \tag{6.30}$$

considering all fragments of Id is the top-element, the least precise DTR (cf. Figure 6.2). ♢

Note 6.2.4 (DTR Precision and Simulation)**.** *Let $D_1 \sqsubseteq D_2$ be two DTRs of a set of identities Id in precision order. Then $D_1(M) \preceq D_2(M)$ for any ETTS M over Id.* ♢

The proof of Note 6.2.4 employs the property that the DTRs per partition of *Id* in D_2 are subsets of the ones in D_1, if they are considered by D_1. Recall that if they're not considered, there is no identity C added, but the original set of identities is retained. Thus in both cases, we obtain an embedding of topologies by mapping the identities in D_2 to those in D_1, if present, and to C otherwise.

6.3. Reflected Properties: Local Liveness

From the construction of the DTR'd system it is clear that quantified properties are in general not reflected. Whenever a logical variable binds to the identity C, most terms turn indefinite because of the abstract local state and modified navigation.

Properties that can be reflected are open formulae with each logical variable bound to a concrete individual. This is where Query Reduction comes into play. It provides a *finite* set of representative assignments for each given universal EvoCTL* formula φ in prenex normal form over an ETTS, which is symmetric in identities. Then the open sub-formula of φ together with the fixed assignment (heuristically) defines the DTR and can (in the good case) be established on the abstract system.

This is one major difference to many other approaches which try to obtain a single abstract system for the whole property, where in particular quantification is meaningful. Such as counter abstraction [119, 149, 71, 146] or the variants of shape analysis discussed in [191, 190] (cf. Section 6.6).

6.3.1. Local Liveness

The experiments reported in Chapter 10 show that the DTR abstraction is also able to reflect liveness properties (under the fairness assumption that each concrete individual is globally finally scheduled).

How can this be if we disregard a large part of the system nearly completely? The point is that DTR reflects *local liveness*, or local progress, which has also been studied in the context of compositional verification. This relates to DTR because we'll be able to point out in how far DTR can be seen as compositional verification in Chapter 9.

More precisely, DTR reflects computation path fragments where concrete individuals interact with each other *without* intervention or dependency of the non-concrete part.

This situation is in particular given when we consider scenarios (cf. Chapter 10): the property refers to a group of individuals in a certain configuration, which together follow a protocol to achieve some goal. They may be triggered from the non-concrete part, but the scenario is between *them*.

Furthermore, the non-concrete part may interfere. Identifying impossible interference, that is, interactions which would only be possible in the concrete system from individuals which are *participating* in the scenario is one of the major guidelines for refining the abstraction (cf. Section 6.4 below).

To approach this claim more formally, one would have to introduce a notion of computation paths per individual or per finite groups of individuals, i.e. a projection of topologies onto individuals. Then local liveness refers to the projection of liveness properties onto individuals.

In addition, there needs to be an annotation of causes for the transitions, a set of who interfered with whom to justify the transition. This information cannot be assumed for every ETTS, but it is reasonable to assume if the ETTS is the semantics of a higher-level language as in Chapter 9. There, we have a notion of which individual is scheduled and can access who interferes with whom to annotate it to transitions.

The aim is then to show that local computation paths present in the original transition system are found in the same form in the abstract system, that is, that self-loops introduced by the abstraction correspond to non-concrete behaviour and are eliminated by a fairness assumption.

From the discussion in Chapter 9, in particular Section 9.3.3, we claim that DTR is the coarsest abstraction which reflects local liveness under fairness assumptions, together with graph-properties like transitivity of comparison for equality, and which furthermore reflects interaction sequences determined by transition programs of a higher-level language (cf. Chapter 9). The abstract transition systems defined by syntactical transformations of the higher-level language descriptions can be even weaker (but still useful).

This seem to be harsh prerequisites to be hardly found in real system. But from the case studies considered in Chapters 1 and 10, the situation is possibly just the opposite.

First of all, we're considering scenarios to be completed in hostile environments. In particular in the Car Platooning case study it's clear that cars have to be prepared for unwanted interactions from other cars, such as split requests in the middle of merge manoeuvres or duplicate merge requests. If the protocols weren't highly robust, they wouldn't complete at all in the real world.

To this end, the protocols also employ a narrow interface and, in particular in case of the DCS language, only interact via message passing with the environment, instead of allowing direct read or write access to local states. This naturally further narrows down the possibilities for interference.

Note that the protocols though cannot be assumed to complete if surrounded with complete chaos, otherwise we wouldn't have cases where refinement is necessary.

6.4. Spurious Counter-Examples and Approaches to Refinement

The DTR abstraction is in general not exact for obvious reasons, but only an over-approximation. That is, there is behaviour in the abstract system which doesn't have a correspondence in the concrete system, and which may well contradict a given AEvoCTL* formula, then called spurious counter-example. Yet counter-examples obtained on the DTR abstraction, spurious or not, are not completely arbitrary but have certain properties we can derive from the construction principle.

To briefly enter this discussion, we'll first formally introduce counter-examples. Then we recall early speculations on the decidability of spuriousness, a complete treatment will be provided by [167]. Then we discuss certain patterns of reasons for spuriousness, and existing and new ideas on how to refine the abstraction. They're partly a generalisation of the approaches taken in [127], non-interference lemmata, and partly topology invariants from [10], or the observation that enriching the border will preserve the benefits, also employed in [166, 167].

6.4.1. Counter-Examples

In order to define what a counter-example is, we need to extend Def. 2.6.4 from paths to trees because some branching time logic formulae are disproved by computation trees.

Definition 6.4.1 (Computation Tree in ETTS)**.** *Let $\mathcal{S}$ be a signature, $M = (S, S_0, R, \mathscr{L}, F)$ be an ETTS, and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. M.*

Let T be a directed tree[1] *where vertices are labelled by pairs $(s, \theta) \in S(M) \times Assign(V)$ of a state and an assignment in that state of a set of logical variables $V \subseteq \mathcal{V}(\mathcal{S})$.*

The tree T is called infinite (computation) tree *or* tree *in M from state s_0 if and only if s_0 is the label of the tree's root, if labels of directly connected vertices are successive in the sense of Def. 2.6.4, and if assignments are assignments in all computation paths π (in the sense of Def. 4.2.11) obtained by taking the state labels of the paths in T originating at the considered vertex.* ♢

An alternative characterisation of computation trees in an ETTS M is that the sequence of state labels of each path in the tree is a finite computation path in M in the sense of Def. 2.6.4.

Given a computation tree T in an ETTS M, we straightforwardly obtain a definition of $\mathcal{M}[\![\varphi]\!](T)$ that is, the evaluation of EvoCTL* formula φ in the computation tree T.

The idea is to apply Def. 4.4.6 accordingly to the labels of vertices in T, that is, if φ is a term t, then

$$\mathcal{M}[\![\varphi]\!](T), := \mathcal{M}[\![\varphi]\!](s, \theta), \tag{6.31}$$

where (s, θ) is the label of the root of T.

Similarly for all other state formulae except for path quantifiers, which are restricted to range only over the paths and evolution chains present in T starting from its root, and except for identity or destiny quantifiers, for which the assignment from the root of T is taken; which is a legitimate choice by definition of computation trees. Here paths denotes paths in the tree, which are again trees yet linear ones. Thus if φ is a path formula ψ and tree a linear tree, then

$$\mathcal{M}[\![\varphi]\!](T) \tag{6.32}$$

is defined by inductively considering suffixes of T.

Definition 6.4.2 (Counter-Example)**.** *Let $\mathcal{S}$ be a signature, M a compatible ETTS, and $\mathcal{M}$ a canonical structure wrt. M.*

Let φ be an EvoCTL formula over $\mathcal{S}$. A computation tree T in M whose root is labelled with an initial state of M and an assignment of the free variables of φ is called* counter-example *for φ in M if and only if*

$$\mathcal{M}[\![\varphi]\!](T) = 0. \tag{6.33}$$

♢

[1] A *directed tree* is a directed graph, in which, disregarding orientation of edges, any two vertices are connected by exactly one path [54] and where each vertex is the terminal vertex of at most one edge; the unique vertex not serving as terminal vertex is called *root* of the tree.

6.4.2. Spurious Counter-Example

Recall from Section 5.1 that we call an ETTS $M^\sharp$ an abstraction of another ETTS M if $M \preceq_H M^\sharp$. In order to define spurious counter-examples, we need a notion of concretisation, not of whole ETTS, but of computation trees in $M^\sharp$.

Definition 6.4.3 (Counter-Example Concretisation)**.** *Let $\mathcal{S}$ be a signature and M and $M^\sharp$ be $\mathcal{S}$-compatible ETTS such that $M \preceq_H M^\sharp$.*

Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M}^\sharp$ is an abstract structure of $\mathcal{S}$ in $M^\sharp$, i.e. such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$.

Let $T^\sharp$ be a computation tree in $M^\sharp$. A computation tree T in M is called concretisation *of $T^\sharp$ if and only if there is a function*

$$f : V(T) \to V(T^\sharp) \tag{6.34}$$

from the vertices of T onto the vertices of $T^\sharp$ such that

- *$(s, s^\sharp) \in H$ if s is the label of $v \in V(T)$ and $s^\sharp$ the label of $f(v)$, and*
- *if v_1 and v_2 are initial and terminal vertices of a single edge in T, then $f(v_1) = f(v_2)$ or $f(v_1)$ and $f(v_2)$ are initial and terminal vertex of a single edge in $T^\sharp$.*

A concretisation T is called step-true *if and only if f is injective, that is, if T differs from $T^\sharp$ only in the labelling of nodes.* ♢

Note 6.4.4. *If $M^\sharp$ and M are bisimilar, then each counter-example in $M^\sharp$ has a step-true concretisation in M.* ♢

Definition 6.4.5 (Spurious Counter-Example)**.** *Let $\mathcal{S}$ be a signature and M and $M^\sharp$ be $\mathcal{S}$-compatible ETTS such that $M \preceq_H M^\sharp$.*

Let $\mathcal{M}$ and $\mathcal{M}^\sharp$ be canonical structures of $\mathcal{S}$ wrt. M and $M^\sharp$ such that $\mathcal{M}^\sharp$ is an abstract structure of $\mathcal{S}$ in $M^\sharp$, i.e. such that $\mathcal{M} \sqsubseteq^H \mathcal{M}^\sharp$.

Let φ be an EvoCTL formula over $\mathcal{S}$. A counter-example $T^\sharp$ for φ in $M^\sharp$ is called* spurious *(or* false-negative*) wrt. M if and only if there is no counter-example for φ in M which is a concretisation of $T^\sharp$.* ♢

Note that an abstract counter-example in $M^\sharp$ typically has many concretisations. By definition, it is only spurious if none of these concretisations is a counter-example in M.

For completeness, we call a confirmation of a formula in $M^\sharp$ *spurious* (or *false-positive*) if and only if the formula doesn't hold in M. An abstraction (in the sense of both, the abstract system and the abstraction function) is called *safe* (or sound) if there are no false-positives.

With this definitions, we can observe relations with the definitions of simulation from Chapter 5. Firstly, if $M^\sharp$ simulates M, then there are no false-positives. Secondly, if $M^\sharp$ is bisimulation equivalent to M, then there are neither false-positives nor false-negatives.

For example, consider an ETTS representing a system where communication is based on asynchronous message exchange as in the Car Platooning case-study. Then any *other*, non-concrete car as represented by identity C can send at *any* time *any* kind of message to *any* other car. Some of these messages sendings are correct; for instance if a concrete car recognises C in front and asks for a merge, then a positive or negative acknowledge from C is also possible in the concrete system.

But if C is not the leader of a given concrete car u, then in the concrete system it will never send a *new_leader* message to u. Sending a *new_leader* message in that situation would be a spurious behaviour, which could be the origin for violations of a given property. In this case we (informally) call the sending of the message a *source of spuriousness*. Now, given a counter-example computation path witnessing such a violation, it is desirable not only to exclude this single computation path but at best all computation paths which comprise the same source of spuriousness.

6.4.3. The Spuriousness Problem of Counter-Examples

If the original transition system is finite, it is decidable whether a counter-example is spurious. Given an abstract counter-example $T^\sharp$, one simply tries to construct a counter-example T by trying to obtain for each pair of adjacent vertices in $T^\sharp$ a pair of states in M and a connecting path. This is a finite (yet possibly large) set of finite-state model-checking problems.

Furthermore, if the original transition system is finite, then a counter-example is necessarily either a finite computation path or of the form for which people came up with the funny names *lasso* or *pan-handle*, that is, a finite prefix and then a finite suffix repeated ad infinitum. So, in particular the spuriousness of liveness counter-examples can be decided if the abstract counter-example is given in form of a lasso.

ETTS are infinite state in general, so the procedure given above needn't terminate. In addition, the particular abstraction DTR doesn't reflect the duration of computations of the shadows, that is, of individuals outside the spotlight. For example, consider a DTR of the car platooning model with non-empty spotlight. Then a car id_1 from the spotlight may send a request-to-merge message to C, that is, to a car not in the spotlight.

In the original system, it may take the addressee of the request a fixed number $N \in \mathbb{N}^+$ of steps to process the request and send an answer, or it may take a number of steps from an interval $[M, N]$. In the abstract transition system, all information about the state of non-concrete cars is lost, in particular information about how many steps have already been taken. Thus in the abstract transition system, the answer may appear within any finite number of steps or even never.

More formally, this can be established by an example; consider only two cars and assume an answer takes two steps, that is, a transition between two different local states. Consider the two global states where the requester is waiting while the answer is produced. The DTR will map both to the same abstract global state $s^\sharp$ and introduce a self-loop, that is, a transition from $s^\sharp$ to $s^\sharp$ by definition of $R^\sharp$. This self-loop can be taken any number of times.

For a given abstract counter-example this means that we cannot conclude from the number of steps it takes between a request from the spotlight and an answer from the shadows to the number of steps a corresponding operation takes in the concrete transition system. Consequently, we can also not conclude to the number of individuals in the shadows participating in the operation (as each step may involve a fresh individual). For instance, the merge protocol may be such that the requested car has to ask one or more of its followers for their opinion on a merge which takes a certain number of transitions.

We'll see in Chapter 9 on implementation issues, that is, on efficient procedures to obtain $D(M)$ for a given ETTS M in a higher-level description language and a DTR D, that we can augment $M^\sharp$ such that it becomes visible *which part* of the higher-level description under what (partial) circumstances justifies a transition in the abstract transition system.

For example, when the higher-level description language is UML state-machines, which are basically input/output automata where transitions are triggered by received events and may update local variables or output events (cf. Chapter 10), then there is a procedure to obtain a semantically equivalent ETTS M where local variables become local states of individuals. And in an abstract transition system $M^\sharp$, obtained from M by DTR, the abstraction from the shadows still executes transitions of the state-machine: each transition in $M^\sharp$ can be related to one or more transitions of the UML state-machine executed by concrete individuals or the abstract representation of the shadows by $\mathtt{C}$ individuals. This gives at least some hints when trying to concretise counter-examples, and it can be used to exclude spurious ones when they are caused by executing transitions in $M^\sharp$ which relate to UML state-machine transitions in impossible circumstances, like topologies known to be unreachable from topology analysis (cf. Section 6.4.6 below).

Note that, in general, the situation doesn't depend on the means of communication. In the above example, the requesting car has to wait for the answer independent of whether the communication is synchronous or asynchronous.

For a more formal an in depth treatment of these issues, we refer to [167].

6.4.4. Refining the Abstraction

If $M^\sharp \models \varphi$ has a spurious counter-example, one wants to *refine* the abstraction. That is, wants to construct $M^{\sharp'}$ and φ' such that $M^{\sharp'} \models \varphi'$ still implies $M \models \varphi$ but doesn't exhibit that counter-example.

One distinguishes counter-example *driven* and *guided* refinement. The former is the simpler one; if a counter-example is found, it just tries a different pair of $M^{\sharp'}$ and φ' satisfying the requirements and being known to be more precise than the former pair. It doesn't ensure that the particular counter-example is excluded. The latter is more informed and tries to construct $M^{\sharp'}$ and φ' such that at least the particular counter-example is excluded. At best, it excludes not only the single counter-example, but the whole source of spuriousness (see above).

There are basically two approaches:

- *assumption-based*, that is, by modifying the formula φ to φ' which basically says "under the assumption that the counter-example doesn't happen, φ is true", or, in other words, “disregarding the paths where counter-example occurs, φ holds”.

The abstract model $M^\sharp$ is typically left intact or only extended by auxiliary variables which make certain spuriousness observable, that is, $M^{\sharp'} = M^\sharp$ up to auxiliary variables.

- *model-based*, that is, significantly changing $M^\sharp$ into $M^{\sharp'}$ such that $M \preceq M^{\sharp'} \preceq M^\sharp$.

  The formula is basically left unchanged or, in some cases, is only rewritten to a case-split (cf. Section 4.5.3) which drives the construction of $M^{\sharp'}$ (cf. Chapter 8).

As they are orthogonal, combinations are possible.

In the following, we'll first discuss the latter approach, in particular why the only currently known strategy is to extend the spotlight, and then the assumption-based one because there exist differently elaborate proposals. Note that the refinement topic is mostly out of the scope of this work, there are promising results by other people, e.g. [166, 167]. We only present the thoughts we've contributed to the discussion of the refinement topic and indicate in how far our results may support other approaches.

To understand the vagueness of the following sections we have to keep two facts in mind

- spuriousness is undecidable [167] (cf. Section 6.4.3 above), and
- the model-checking problem of ETTS and EvoCTL* is undecidable (cf. Chapter 4),

that is, we inevitably *have to* rely on heuristics, and incomplete methods to get at least some results.

6.4.5. Refining the Abstraction: Model-based

Recall that in the model-based refinement, the aim is to obtain an abstract transition system $M^{\sharp'}$, which is more precise than $M^\sharp$ but still a sound over-approximation of M.

When the intention is to apply the DTR abstraction for its benefits, there are few known options preserving these benefits for the following reasons. When we discuss in Chapter 9 how to effectively (and efficiently) compute $M^\sharp$ from a given higher-level language system description, we'll see that the computation procedure is also highly efficient.

Based on our understanding of the characteristics of the DTR approach, namely “focus on a finite set of actors, abstract from all the rest”, we also understand that the reason for efficiency is that the abstract transition relation doesn't depend on that rest. In more precise abstractions, one typically either strives to preserve at least some information about the rest or to keep imprecise information about all individuals (cf. Section 6.6). There, the abstract transition relation has to soundly update the information about the rest, or the imprecise information about all individuals.

Computing the abstract transition relation is often computationally expensive. As the efficiently computable abstract transition relation of DTR is one of the main motivations to employ it at all (next to it being able to verify general liveness properties under fairness), the primary goal in refinement is to preserve it.

For example, tagging abstract nodes with definite information, as done for example in shape analysis, wouldn't preserve this efficiency. Then employing DTR can hardly be justified. Thus any modification of $M^\sharp$ into $M^{\sharp\prime}$ needs to be carefully crafted.

Case-Split

One class of sources for spuriousness (cf. Section 6.4.2) is a too small DTR, that is, a too limited spotlight.

For example, assume a more elaborate car platooning model which considers a car to consist of a control and a communication module, that is, there were two different sorts Id_1 and Id_2 of identities;. The control module would be responsible for granting or rejecting merge requests and would *employ* the communication module to actually send the answers. Then given a property

$$\varphi = \forall\, x_1, x_2 : T_1 \,.\, \varphi_0 \tag{6.35}$$

on two control modules, the heuristics of Chapter 8 uses the DTR

$$D = \{(\{id_{1,1}, id_{1,2}\}, Id_1), (\emptyset, Id_2)\} \tag{6.36}$$

comprising two control modules and no communication modules. That is, there were no concrete communication modules, thus chances are low that any reasonable property can be established because there are plenty of possibilities for spurious counter-examples by the abstracted communication modules.

A modification which is always sound is an enlargement of D, in the example, by adding communication modules. A practical way to indirectly obtain an enlarged D proposed already in [127] is to employ the case-split rule from Section 4.5.3

$$\frac{\forall\, x, y \,.\, *(\lambda(x)) = y \to \varphi}{\forall\, x \,.\, \varphi}. \tag{6.37}$$

Recall from Section 4.5.3 that we've got to consider whether the link we're splitting cases on is actually defined. In the particular example, this is evident because the communication module is created along with the car and later not changed.

In the example, one would isolate the cases where control module x_i employs a certain communication module y_i, i.e.

$$\varphi' = \forall\, x_1, x_2 : T_1, y_1, y_2 : T_2 \,.\, (*(\lambda(x_1)) = y_1 \land *(\lambda(x_2)) = y_2) \to \varphi_0 \tag{6.38}$$

The heuristically obtained DTR is then

$$D' = \{(\{id_{1,1}, id_{1,2}\}, Id_1), (\{id_{2,1}, id_{2,2}\}, Id_2)\} \tag{6.39}$$

and the property may succeed. We have $M \preceq D'(M) \preceq D(M)$ by Note 6.2.4 and $\varphi' \implies \varphi$ by Section 4.5.3.

Note that the choice of the case-split above is the result of insight into the model, and possibly an obtained counter-example. There is yet no mechanic procedure to choose a case-split that is guaranteed to exclude a given spurious counter-example.

On the other hand, if the model is *derived* from a higher-level description, for instance UML, and if the translation from UML into ETTS is known to be correct, then the class diagram could give hints on sensible link-names to split cases on (cf. also the discussion on crystallisation in Section 7.5).

Auxiliary Variables

In addition, we may always add so-called *auxiliary components* (or variables, cf. Chapter 9) to the local state of objects, that is, variables which are only written, but never read. This property ensures that they don't influence the original behaviour, but only *observe* it, and thereby preserve soundness.

Auxiliary variables can be used, for example, we could count how many messages we've sent without obtaining an answer or the kind of message just sent. Similarly, we may add auxiliary links or link-names, for example, in order to trace whom we last sent a message, an information which is not necessarily visible in the original system. This augmentation of an ETTS with auxiliary variables is particularly useful for the second, assumption-based, way of refining the abstraction as discussed in the following section and has already been employed in [127].

6.4.6. Refining the Abstraction: Assumption-based

To briefly recall the idea of classical assumption-based counter-example exclusion, let $T^\sharp$ be a spurious abstract counterexample. The goal is to obtain a formula φ' such that

- it evaluates to 1 for the spurious counter-example, i.e. $\mathcal{M}[\![\varphi']\!](T^\sharp) = 1$, and
- for every abstract computation tree $T^{\sharp\prime}$ which has a concretisation, or: which is not spurious, its evaluation coincides with φ, i.e.

$$\mathcal{M}[\![\varphi']\!](T^{\sharp\prime}) = \mathcal{M}[\![\varphi]\!](T^{\sharp\prime}). \tag{6.40}$$

In other words, the spurious trace $T^\sharp$ is trivially accepted, as it trivially satisfies φ'.[2]

Then proving $M^\sharp \models \varphi'$ intuitively verifies φ *under the assumption* (in the sense of [94]) that spurious behaviour doesn't take place. Successful verification implies $M \models \varphi$ because only the absence of spurious paths is assumed.

For example, if φ is from the linear fragment of AEvoCTL*, that is of the form $\mathsf{A}\,\varphi_0$, and if the source for spuriousness of $T^\sharp$ is indicated by an auxiliary individual id_a whose local state is a single boolean flag, then a simple example for φ' is

$$\mathsf{A}(\varphi_a \rightarrow \varphi_0) \tag{6.41}$$

[2] in terms of LSCs (cf. Chapter 10), this strategy is related to *cold conditions* and their legal exit semantics, and in particular to internal assumption treatment [97], where an LSC's symbolic Büchi automaton is modified such that it simply accepts all runs that don't adhere to the environment assumption stated within the LSC

because in the linear fragment, prove under assumption coincides with implication.

The absence of spurious behaviour φ_a could be stated as

$$\mathsf{G}\,\sigma(a) = \mathit{false} \tag{6.42}$$

assuming that the constant 'a' denotes id_a (when combining with QR, treated as a singularity, cf. Section 7.4) and that the constant '*false*' denotes the logical value 0.

In the branching fragment of EvoCTL*, matters are seriously more complicated. We refer to [167, 94, 182, 98] for a more elaborated discussion.

The oldest strategy to obtain characterisations of spurious counter-examples are so-called *non-interference lemmata* as already proposed in [127]. They can be seen as a dual characterisation, that is, they actually characterising *good behaviour*, which in particular the C individuals have to adhere to. The drawback is that these lemmata have to be stated in a creative act and that they've to be proven separately, where it's not always clear that this prove succeeds. In [31], this issue is discussed for their particular case study. They are even able to establish such lemmata on the abstract system with an elaborate reasoning addressing the apparent circularity.

In the following Section 6.4.6, in addition to [127], we provide a pattern generalising the lemmata chosen in [127].

Newer strategies are more sophisticated. On the one hand, they care more about *how* to establish the validity of employed invariants. They may rely on orthogonal techniques, like abstract interpretation, which may yield a characterisation of the legal topologies, for instance [7]. Then any deviation from the legal topologies is a *generic* source of spuriousness. Or they may actually consider the counter-example and try to identify *particular* sources of spuriousness. We'll briefly discuss this in the following Section 6.4.6.

Non-Interference Lemmata

Firstly, there is the strategy employed in [127], namely to manually analyse obtained counter-example, understand sources of spuriousness, and, in a creative act, hypothetically state an invariant of the system.

In [127], these invariants are called *non-interference lemma*, because it typically characterises in which situations individuals are supposed to interact with others

> "In general, such a lemma is needed whenever the state of one system component might be corrupted by a spurious message from other components that have been abstracted away. [125]"

Then try to prove the original property under the assumption that all participants, including the C individual(s), adhere to the assumption. A serious drawback of this approach is that the stated invariant is a priori not proven, it is in general not known whether it actually *is* an invariant. It has to be verified, too. In [127], this succeeds employing the same abstraction technique but only together with the ingenious application of a kind of temporal induction (cf. [127]). It's not clear to us whether this reasoning can be automated; possibly only by casting it into a pattern and requiring an annotation of the model which maps parts of the model onto the pattern.

In [48, 49], we characterised and generalised the strategy (implicitly) employed in [127] for finding non-interference lemmata. Namely the strategy is to look for instances of the pattern

> "If somebody sends something to me, then it is allowed to do so."

The pattern is intentionally stated in a subjective way, that is, from the point of view of a single individual. Thereby it can be imposed on the abstract system by looking onto the individuals in the spotlight: they're supposed to know who else is allowed to send which message in what situations. Then without changing the information kept for the shadows, spurious interaction of the shadows with the spotlight can be identified. Note that individuals don't necessarily keep track of this information in the original ETTS but it may need additional auxiliary local state components or auxiliary links (cf. Section 6.4.5).

For example, consider the Tomasulo algorithm [169] briefly introduced in the introductory Chapter 1. There, tagged results of functional units are returned on a bus and reservation stations are waiting for particular tags. After DTR, the functional unit `C` may put any tag on the bus because it doesn't keep track on which tag it is working. To keep the `C` functional unit from putting spurious tags on the bus, [125] adds the lemma

> "[...], at all times, if a result is returning on the `pout` bus, with a given tag `pout.tag`, then the unit returning the result [...] must be the unit that the indicated reservation station is waiting for [...]."

As mentioned above, the prove succeeds employing the same abstraction technique but only together with the ingenious application of a kind of temporal induction (cf. [127]).

As another example, consider the arrival/departure procedure for the automated rail cars system (ARCS) (cf. Chapter 10). If a car already has a link to the car handler it is guided by, so a useful invariant is that only the guiding car handler sends the permission to enter the station. The car handlers in the shadows don't adhere to this invariant in the abstract transition system, so they may trick the car into entering the station if it is actually not yet allowed to. Taking the invariant as an assumption disregards the runs with spurious interactions. In [178], the employed non-interference lemma is manually established by considering the UML class diagram and assuming that the translation from UML to ETTS is correct.

As a third example, consider the Car Platooning application. An invariant is that only the current leader may announce a leader change. The cars in the shadows again freely send messages, they may in particular trick a car in the spotlight to change its leader link to any other car. Imposing the invariant as an assumption excludes this source of spuriousness. This invariant can alternatively be stated as a structural invariant obtained by static analysis of higher-level language definitions of ETTS as discussed in the following paragraph.

Orthogonally Obtained Invariants

One proposal to address the two drawbacks of the non-interference lemmata approach, namely that there exact proposition has to be created and they've got to be proven separately, is to employ orthogonal (often static) analysis of a higher-level language definition of ETTS which *yield* invariant properties of the system.

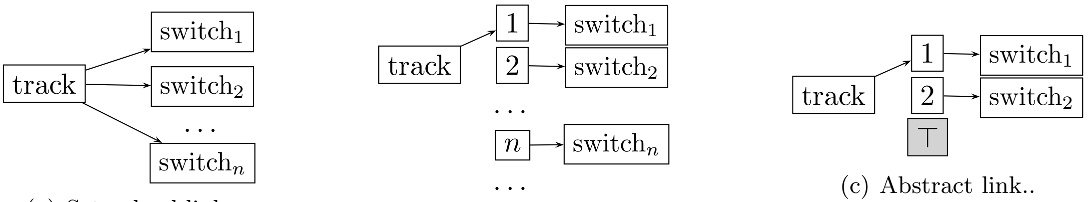
This solves both problems at once, there is (at best) a set of invariants to choose from and they're all valid by properties of the analysis.

One approach is to employ *topology invariants* in form of so-called *abstract clusters*. The abstract interpretation-based procedure of [7] yields for a given graph grammar a finite, abstract description of all possibly reachable topologies. This links well to ETTS as their transition relation can alternatively be given as a graph grammar on topologies. Although it is in general over-approximating, that is, not each described topology need be reachable in the original system, it still serves to characterise the *definitely* unreachable topologies.

In [10], we participated in a demonstration how to employ topology invariants to refine a DTR abstraction in the context of DCS/METT verification (cf. Chapter 10). In order to fit into the scheme of Section 6.4.6, we give a logical characterisation of topologies being legal. Furthermore we use that we can, as discussed in Section 6.4.4 and Chapter 10, augment the abstract transition system with auxiliary variables that keep track of the corresponding high-level description construct and the assumed partial concretisation of the current abstract state. If the partial concretisation contradicts the topology invariant, then this is a possible source of spuriousness which can be excluded.

In [10], a suitable subset of the topology invariant is manually chosen, so in its simplest form, this approach is similar to non-interference lemmata, with the difference that the validity of invariants is given. It discusses different ideas to further automatisation. The weakest one is to vary the precision of the topology analysis and make the whole procedure counter-example *driven*. That is, whenever an abstract counter-example $T^\sharp$ is obtained, a more precise topology invariant is tried, which may or may not exclude $T^\sharp$. A more elaborate approach would be counter-example *guided*. It would check whether $T^\sharp$ adheres to the topology invariant and, if not, identify the violated abstract clusters. Then only these abstract clusters would be employed in the next iteration as indicators for spuriousness, and φ' (see above) would refer to this spuriousness indicator.

The main criticism on [10] is that disregards temporal aspects. As it employs a collecting semantics, a topology invariant in the sense of [7] is a finite, abstract characterisation of all possibly reachable topologies, but it yields no information on temporal ordering of topologies, for example, which ones are reachable from which other ones. Furthermore, it is as such not driven by a counter-example, but completely separate. To this end, [166] proposes to analyse the higher-level description for dependencies between message sendings, in particular orders of messages exchanged between an individual and particular partners known via links. Then an augmentation in the sense of Section 6.4.5 with auxiliary, finite counters allows to identify and exclude violations of such *communication invariants*. For even more elaborate approaches we refer to [167].



(a) Set-valued link.

(b) Array-valued link.

(c) Abstract link..

Figure 6.3.: **Abstraction** of set-valued links.

6.5. Treating ETTS with Unbounded Link

While studying ETTS with the single-link property is relevant and difficult, an unbounded number of links remains a serious issue. For example, when designing the Car Platooning application, one doesn't want to think about upper bounds on the size of platoons, although in the final implementation there will be such a bound.

Recall from Section 4.5.4 the discussion of quantification over links (cf. Figure 6.3(a)) like

$$\mathsf{G}\,((\forall\, x : T\,.\, \circledcirc(x) \wedge \mathit{switch}(x) \rightarrow p(x)) \implies \mathsf{X}\, q(y)) \tag{6.43}$$

(if all switches *in the system* (not of a particular track) are set, then something happens). In case the link is not bounded, we again apply abstraction.

Figure 6.3(a) depicts a track with a set of n switches, where switches may arbitrarily be removed or added. Conceptionally, we can view this set as an unbounded array as shown in Figure 6.3(b), possibly indicating unused entries by a designated identity id_0. Still, identities of switches may arbitrarily be removed or added. Now we can apply to the index type of this array another data-type reduction. That is, we can choose any subset of array indices, add an element $\top$, and define what it means to access the array at $\top$. By default, this should yield an over-approximation of a switch.

What we obtain is principally a link set with a finite upper bound, that is, we enter the scope of Section **??**.

But this has to be applied with care as it is possibly unsound. Whenever the model behaviour depends on the *number* of elements in a link, then this has to be taken into account. For example, if individuals can *count* the number of tracks and base their choices on that number. And one has to be aware that counting may be indirect. One may send messages to all linked individuals and they may send messages on, and the recipient may count, etc.. This is possibly treatable by fairness, i.e. iterate as long as one wants but finally terminate.

For example, there may be iterators in the high-level language (cf. Chapter 9) and a transition program iterates over all elements in the link and collects the sum of some component of the local state. If this happens in zero-time, that is, in one step of the system, then we can treat it, but we explicitly have to treat it. The situation is then similar to logical "reduce" operations, which have already been treated by [89] and, by being based on that work, by [127].

A new example in our setting of Chapters 9 and 10 is given when we send each linked individual an event. If each receiver keeps his event, it's fine, if they pass it on to someone who's counting, we may be lost. In particular if we do refinement, in particular if that's got to do with counting events, we've got to take care to only count a "minimum", we can't count exactly then.

In Chapter 9 we also address the case with iteration taking multiple step. Then we've got to take care that iteration may take any number of rounds, but not loop forever.

6.6. Discussion

In this section, we shall discuss the following question. Firstly, how this chapter relates to [127], whether we possibly gained something. Secondly, how does it classify in the notions of data and control abstraction, and to some amount also why it is labelled *compositional* verification in [127]. And thirdly, how does it compare to other abstractions addressing dynamic topologies, in particular in terms of precision and reflection.

6.6.1. Relation to the Original Definition

Firstly, our definition of DTR is meant to be exactly the one of [124, 127]. A difference is the form. The original definition applies to the rather technical level of array programs, thus is rather comparable to our presentation in Chapter 9.

For this reason, it also doesn't have explicit notions of topologies, individuals, links, or evolution. The *effects* in terms of these concepts are thus only implicitly given by considering their encoding in array programs. Our choice for the presentation is the other way round. We firstly discuss DTR in the setting of ETTS, and then discuss an encoding in array programs.

What we gain from the description in terms of ETTS is that we get a good understanding of what is preserved and what is lost in the abstraction, and why certain properties can be proven on the abstract system. One intuition is the spotlight view, that a set of individuals is kept precise and surrounded by a rather hostile environment. From this conclusion we understand why the syntactical construction of the abstraction in Chapter 9 is possible, namely because no state information is kept for the "other" individual representing all individuals in the shadows. It is not necessary to have an abstract transition relation based on different abstract states of this representation of others, like in, for instance the shape analyses (see below).

In a second step, this understanding supports improvements and refinements of the DTR abstraction, e.g. in [167]. If it is desired to preserve the efficient computability of the abstract system, then additional information about the non-concrete individuals has to rest with the concrete ones, that is, added to their local state. The whole system state is and has to be viewed *from the perspective* or *through the eyes* of the concrete individuals. This has in a form already been employed in [127], now we can tell why this is the most reasonable choice.

And of course we establish that this abstraction is suitable for a setting of higher level

$$\begin{tikzcd}
M \arrow[d, "(1)"'] & \models^? & \varphi \arrow[d, "(2)"'] & & \checkmark \\
M^\alpha & \models^? & \varphi^\alpha \arrow[rr, "(3)"] \arrow[rrd] & & \checkmark \arrow[u, "(4)"'] \\
 & & & & \text{✗}
\end{tikzcd}$$

(a) Procedure of [96].

$$\begin{tikzcd}
M \arrow[d] & \models^? & \varphi & & \checkmark \\
D(M) & \models^?_{\mathcal{M}^\sharp} & \varphi \arrow[rr] \arrow[rrd] & & \checkmark \arrow[u] \\
 & & & & \text{✗}
\end{tikzcd}$$

(b) DTR.

Figure 6.4.: **Data Abstraction** and DTR.

concepts like topologies, individuals, life-cycles, and evolution, while the original discussion is targeted at parameterised systems and indexed temporal logic (cf. Chapter 4).

6.6.2. Data and Control Abstraction

In [96], the abstractions employed in formal verification of reactive systems are generalised and separated into two categories, namely data and control abstraction. We'll briefly discuss how DTR classifies according to these two categories, a discussion which is also supported by our different presentation of DTR (see previous section).

According to [96], *verification by data abstraction* summarises as follows (cf. Figure 6.4(a)).

1. devise a *(finitary) abstraction mapping* α to abstract the *concrete* transition system M into a (finite) abstract one M^α by an exists/exists abstraction (cf. Chapter 5),

2. abstract the *concrete* temporal logic property φ into a *(finitary) abstract* temporal property φ^α,[3]

3. verify $M^\sharp \models \psi^\sharp$,

4. infer $M \models \psi$ in the affirmative case.

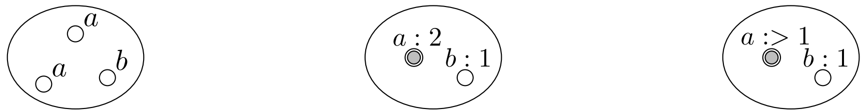
Here, an abstraction mapping is a function

$$\alpha : S(M) \to S^\alpha \tag{6.44}$$

mapping *concrete* states from $S(M)$ to a set of *abstract* states S^α. Such an abstraction mapping is called *finitary*, if S^α is finite. Note that the transformation of the formula to φ^α is determined by S^α, and thus by α.

Following [96], an implementation of this general strategy, that is, a recipe for defining the abstract transition system $M^\sharp$ and the abstract temporal formula $\psi^\sharp$, is called a *data abstraction method.* A data abstraction method is said to be *safe* (or *sound*), if for every transition system M, temporal property ψ, and abstraction mapping α, $M^\alpha \models \varphi^\alpha$ implies $M \models \varphi$.

[3] for this general discussion, it is again not relevant, *which* temporal logic we consider. For a concrete instance, one may consider Linear Temporal Logic (LTL), which is used in [96].



(a) Concrete snapshot. (b) Abstract snapshot, $c = 2$. (c) Abstract snapshot, $c = 1$.

Figure 6.5.: **Counter Abstraction.**

In these terms, DTR can technically be seen as a data abstraction method if we consider topologies to be data and the application of a DTR to an ETTS as the mapping.

Then only concerning the temporal property our approach seems different than proposed by [96] in that we aim to leave the formula as such unchanged, but change the *interpretation* in form of an abstract structure. In Chapter 9 we'll see that this can technically be achieved as an (indirect) transformation of the functional terms.

Verification by *control abstraction* applies to modular system. Single modules are checked in the context of abstractions of their environment, which includes the other modules.

As we'll see much clearer in Chapter 9, the DTR approach fits better into this scheme because we set the individuals from the spotlight into an abstraction of the environment consisting of all other individuals. In this view, the structure of topologies is considered. This alternative view of [124, 127] is also given in [144].

Data abstraction would then rather refer to an abstraction of the local state per individual. Note that is well covered by our definition of embedding in Chapter 5, which doesn't assume that the embedding is either to the precise concrete local state of the top element. This is a characteristic of DTR only.

6.6.3. Summarising Abstractions

There is a whole group of abstractions for parameterised systems or dynamic topologies which can be called summarising. The underlying idea is to only keep track of how many individuals are in a certain state, instead of considering all concrete states. With a finite cutoff, that is, with a finite number c which indicates that there are more than c such individuals, one can also represent infinite structures.

In the following, we recall the groups of counter abstraction variants and shape abstractions for data structures, and state how DTR relates to these approaches. The most prominent difference is the absence of identity blurring in DTR, so identity blurring obtains an own paragraph below. For a more formal and more extensive discussion we refer to our joint work [174].

Counter Abstraction

The most prominent approach is counter-abstraction [119, 149, 71, 146] known from the domain of parameterised systems in the sense of Section 3.7.1.

For example, if each process has as local state a label from a finite set, i.e. $\Sigma = \{a, b, c\}$, then Figure 6.5(a) shows a snapshot of a parameterised systems with two processes in state a and one in state b. The links of the full interconnection (cf. Section 3.7.1) are omitted.

Figure 6.5(b) shows an abstract topology with cutoff $c = 2$, that is, the labels indicate that this abstract topology represents a concrete one with on process in state b and two in state a. With cutoff $c = 1$, the exact number of individuals in state a is already lost (cf. Figure 6.5(c)). Note that this concretisation relation does not refer to identities, that is, it doesn't tell *which* process is in which state. We'll get back to this issue under the name of *identity blurring* [173] in a dedicated paragraph below.

Translating into our terms, the abstract topologies employ the local states as (finite set of) identities $U^\sharp = \Sigma$ labelled with

$$\Sigma^\sharp = \Sigma \times \{1, \dots, c, > c\}. \tag{6.45}$$

The embedding function is basically $f(u) = \sigma(u)$ with appropriate counting.

The DTR abstraction can be viewed as a counter abstraction, yet with a very uncommon cutoff counter. Namely, it only considers the two counter values 0, 1 and > 0, the former ones for the individuals in the spotlight and the latter for the representatives of the shadows. In difference to counter abstractions, the DTR counting takes into account the identity, thus a particular combination of local state and identity is either alive or not (0 or 1) and the extension of the shadows is not traced (always > 0). This is the reason why DTR is not identity blurring.

An interesting extension of counter abstraction is the environment abstraction of [35], which in addition keeps track of the relation between a certain set of processes in a kind of spotlight and their environment. They assume shared variables of continuous domain and only have operators for comparison of these variables in their description language so it is sufficient to recall how variables in the focused individuals relate to the environment.

Canonical Abstraction for Shapes

The idea of the abstractions employed for shape analysis in the sense of Section 3.7.4 is in a sense similar to counter abstraction. Global states (or: topologies) are represented by logical structures of finitely many unary and binary predicates. In the canonical abstraction approach of [155], a subset of the unary predicates on individuals, called *abstraction predicates*, define an equivalence relation on individuals. Two individuals are equivalent if and only if they're undistinguishable by the abstraction predicates.

The set of individuals in the abstract topology is basically determined by the valuations of the abstraction predicates. A notion of counting comes into play as it is distinguished, technically by an additional "is a summary" predicate, whether an individual represents exactly one or strictly more than one individual. Other predicates, in particular binary ones indicating relations between individuals obtain a most precise abstract interpretation, which can be $1/2$ (unknown) if the summarised concrete individuals have different valuations. Precision can be gained by employing a second set of predicates, called *instrumentation predicates*, which can be expressed in terms of others, but get more precise if their value in the abstraction is computed on the concretisation instead of from the (possibly less precise) values of the defining predicates in the abstract topology.

In has in particular shown how to state DTR in the framework of canonical abstraction in the joint work [174], which originates in [173].

Links are encoded as binary predicates in the computational model of [155] (cf. Chapters 3 and 4), and thus they're naturally abstracted along yielding a notion of "half edges" corresponding to the binary link predicate evaluating to the third truth value $1/2$. Intuitively, presence of a half-edges between summary nodes means "at least one individual from the source summary has a link to an individual at the destination summary", while a one-edge means "all individuals from here have a link to every one at the other end", and zero-edges mean absence of any links (of this name) between individuals represented by both summaries.

In the setting of DTR, summarising of edges takes place only with the links originating at the shadows. In terms of canonical abstraction, there is a half-edge between each C node and any other node in the topology. So DTR use only a very limited amount of the capabilities of Chapter 5, which are far less uniform and elegant than in canonical abstraction. For example in the explicit modification of the link navigation functions (cf. Def. 3.2.1) which has a good and a bad aspect.

This principle has been primarily applied to all kinds of data-structures on a dynamic heap [155], like single or doubly-linked lists, or trees, and in more explicit variants using a tool operating on three-valued logical structures [115], and symbolically [180, 148, 181]. As noted in Section 3.7.4, the aim of these usages are not temporal properties, but invariants (or safety properties), i.e. the fragment of EvoCTL* $\mathsf{G}\, t$ where t is a term, identity blurring doesn't hurt in this case.

A particular advantage of the pre-abstraction into predicates is that these predicates already indicate all possibly relevant parts of the system, while abstraction predicates choose the relevant aspects for a given property. These predicates furthermore provide best starting points for refinement of the abstraction. On the other hand, a good choice of these predicates is said to be kind of a "black art" requiring particular experience. This choice is not necessary in the QR/DTR approach.

In [192], the technique is applied to establish safety properties of concurrent Java programs. It tries to overcome the identity blurring problem by principles exactly matching that of the DTR/QR combination of [124, 127], interestingly without appreciating this relation. The procedure is to, in a sense, partially re-introducing identities as an augmentation of certain processes. Then the canonical abstraction doesn't summarise the augmented processes, if the augmentation is part of the abstraction predicates.

The ideas underlying the refinement language proposed in [192] again exactly match the DTR/QR refinement approach to the extent it is present in [124, 127]. In addition, there is few discussion of the applicability and limits of case-splits in comparison to our Section 6.4 and Chapter 4.

In [191, 190], the technique is finally applied to liveness properties of concurrent Java programs. Interestingly, it summarises on two levels: individuals per topology and whole topologies in abstract traces. The procedure is to compute a finite set of abstract computation paths, in a sense shapes of computation paths, which are in turn sequences of abstract topologies. Formulae of VTL/ETL, which is clearly an ancestor of EvoCTL*, are then evaluated on abstract traces. It explicitly keeps track of evolution similar to our evolution relation from Chapter 3, which is actually inspired from there as discussed in Section 3.7. Yet it is not free from the danger of identity blurring as multiple individuals may evolve into and from a single summary node (cf. Figure 3.6 in Section 3.4). Furthermore, summarising of states in computation paths is likely to destroy liveness, in particular local liveness, properties.

The partner abstraction of [7, 12] generalises the canonical abstraction of [155] to also consider binary predicates as abstraction predicates, also abstracting in two steps, and explicitly addresses the class of Evolving Topology Systems modelled with labelled graphs. Although it elaborately introduces a variant of first-order temporal logic, the discussion is immediately focused on the fragment of invariants to match the capabilities of the analysis. The analysis computes a finite abstract description of all reachable topologies in terms of individuals and their direct neighbours, their *partners*.

In the literature, the capabilities of partner abstraction, in particular wrt. dynamic creation and destruction is compared to [49, 48], a presentation of early results of this work concealed in terms of UML and LSC (cf. Chapter 10), as

> "[...reminding] only vaguely of the setting in this thesis." ([7], p. 133)

Actually compared to DTR, there are clearly many connection points that are less vague than the author of [7] admits. For instance, the DTR/QR approach is of course able to establish some of the properties obtained via partner abstraction. In our opinion, there is no need to deny this connection. Partner abstraction keeps far more precise information on the environment of processes and yields, in form of the abstract cluster far more than a simple yes/no answer, as, for instance, pointed out in the joint work [8]. On the other hand, it also suffers from identity blurring and only applies to safety properties.

As outlined in Section 6.4, the procedure is one highly relevant way to add precision to DTR (cf. the joint work [10]).

Identity Blurring

Identity blur is a name first appearing in [173]. It refers to the property of, for example counter abstraction, that it doesn't explicitly keep track of identities. Recall from the paragraph on counter abstraction above that an abstract state may say that there are two individuals in state a and one in state b. Now if there is a transition to an abstract state with two individuals in state b and one in state a, we can in general not tell *which* individual took a local transition. It could be that only one of the a's took a local transition to b, it could be that each individual took a local transition. So we cannot, for example, establish properties as shown in Figure 6.6.

That is, the counter abstraction is able to reflect *global* properties like mutual exclusion but in general not scenarios (cf. Chapter 10) where one is interested in the fate of particular individuals.

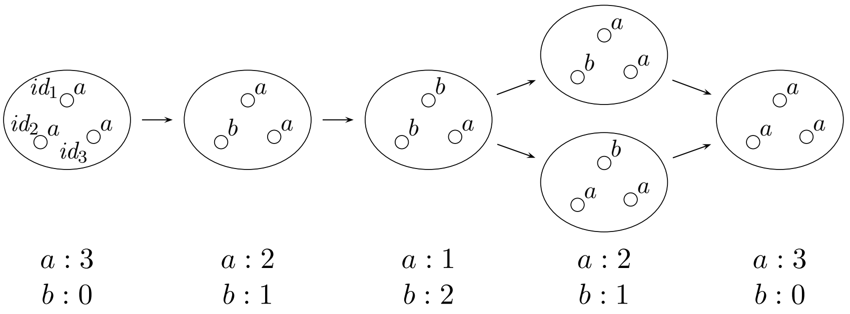


Figure 6.6.: **Identity Blurring** with counter abstraction. Counter abstraction is safe, but in general loses properties like "each individual is for at most two steps in state b" which is the case in this model but in the abstraction there is a sequence of three states with b different from 0.

In some cases, this can be implicitly concluded. For example, in a strict interleaving semantics with one local transition per global system step, one can identify who took the transition.

Similarly, shape analysis of list programs is not interested in *which* memory cell leaks; it doesn't care if the heap management re-arranges the heap (consistently) in each system step, it only cares about whether there is a memory leak, which is often reflected by that abstraction. Notably, [192] is not affected by blurring (as is our encoding of DTR in [174]) because there are discriminating unary predicates, which explicitly denote an identity.

This is a problem of most graph-based approaches and also of the attempts to establish verification for the π calculus, whose notion of structural equivalence neglects identities (cf. Chapter 3).

Getting back to the properties, note that *global* properties like mutual exclusion or leader election are easily lost with DTR. The latter is clear because it comprises an existential quantification, the former is typically spoiled by the abstracted others which simply don't keep track whether they're allowed to enter the critical section. Yet with suitable refinements it may be possible to approximate the property by reducing it to the local property of two individuals.

7. Query Reduction

The name Query Reduction (QR) has been coined by [185] for the kind of symmetry reduction employed in [124, 127]. As discussed in more detail in Chapter 8 on the combination of QR and DTR, the main difference between the classical approach aiming at a quotient-model by the equivalence relation induced by symmetry [89, 62] and QR is that the latter reduces the number of cases. It proves a finite set of representative cases, which are representative because any other case in the system is symmetric to one of these.

The benefit of the latter is that there are far less rigid restrictions on the considered temporal properties. Most quotient-based approaches suffer from the identity blurring problem discussed in Section 6.6.3, that is, the properties have to be symmetric themselves, which is a serious restriction because this is not the case for the scenarios discussed in Chapter 10. More recent developments are surveyed in [134].

Both approaches share a common theory of symmetry, which we recall minimally in Section 7.1. In Section 7.2 we define a notion of an ETTS being *symmetric in identities*, which is a special case of symmetry in scalarset types [89]. This instantiates [124, 127] in our context of EvoCTL* and ETTS.

In Section 7.4, we revisit the discussion of treating singularities, now from the perspective of the property to be verified. Section 7.5 discusses how QR interacts with a particular optimisation employed in the verification of Dynamic Topology Systems, namely *crystallisation* [158, 159].

7.1. Symmetry Theory

7.1.1. Automorphism

An automorphism is in general a structure preserving mapping, in case of transition systems, it is the following.

Definition 7.1.1 (Automorphism)**.** *Let $M = (S, S_0, R)$ be a transition system.*
A function $a : S \to S$ is called automorphism *of M if and only if*

1. a is bijective,

2. $s \in S_0$ if and only if $a(s) \in S_0$, and

3. $(s, s') \in R$ if and only if $(a(s), a(s')) \in R$. ♢

Note that this definition is not standard, for example the variant of [89] is already different, fitted for the employed computational model.

An automorphism can be seen as structural symmetry of M. It points out similar parts of the transition graph, when viewing states as nodes and transitions as edges, by mapping them onto each other. It is not (yet) useful as it disregards state labellings.

Example 7.1.2. *Given a transition system M, the identity function on $S(id)$, i.e. $id_{S(M)}$, is the trivial automorphism of M.* ♢

For completeness, and for better synchronisation with works on the classical, quotient-based symmetry reduction, we give the following notes. They are strictly speaking not necessary for our discussion because we don't employ the quotient-based approach (cf. comparison in Chapter 8).

Note 7.1.3 (Automorphism Group [89])**.** *If A is a set of automorphisms of a transition system M, then $G(A)$, the closure of $A \cup \{id\}$ under inverse and function composition, is a group as these operations preserve the automorphism property.* ♢

The following note is useful in our later proofs.

Note 7.1.4. *Let M be a transition system and $A \supseteq \{id\}$ be a set of functions $a : S(M) \to S(M)$ which comprises for each function a its inverse a^{-1}.*

Then it is sufficient to show implication in 2 and 3 of Def. 7.1.1 to establish that A are automorphisms of M, i.e.

$$\begin{aligned}\forall\, a \in A : a \text{ bijective } &\wedge (\forall\, s \in S_0(M) : a(s) \in S_0(M)) \\ &\wedge (\forall\, (s, s') \in R(M) : (a(s), a(s')) \in R(M))\end{aligned} \tag{7.1}$$

implies that A is an automorphism group, that is, that each $a \in A$ is an automorphism of M. ♢

Proof. Given $a \in A$, we can use (7.1) for the inverse a^{-1} to establish the equivalences in Def. 7.1.1. □

7.1.2. Permutation

The name permutation is typically reserved for bijective functions on a finite set. As it is so common in the context of combinatorics, we widen the notion of permutation to infinite sets because the set of identities *Id*, the target of our permutations, is not finite.

Definition 7.1.5 (Permutation)**.** *Let $M = (S, S_0, R, \mathscr{L}, e)$ be an ETTS over identities $Id = Id_1 \dot\cup \ldots \dot\cup Id_n$.*

We call a bijection $p : Id \to Id$ on Id a (partitioning consistent) permutation *of Id if and only if $p|_{Id_i}$ is a bijection on Id_i, $1 \leq i \leq n$.*

The set of all (partitioning consistent) permutations of Id is denoted by $perm(Id)$. ♢

Note that, when writing *perm*(*Id*), we don't indicate with *which* partitioning of *Id* the permutations in *perm*(*Id*) are supposed to be consistent.

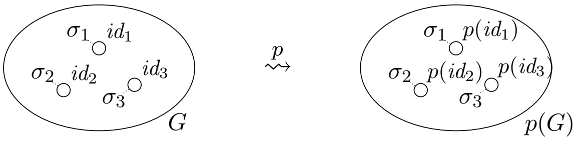


Figure 7.1.: **Individuals under permutation.** Permutation doesn't change the extension of a topology, that is, it doesn't add or remove individuals, it only changes the set of identities that are alive.

For the moment, we only consider the partitioning corresponding to types. In later sections we'll see that the definitions and results are actually *independent* from the partitioning. This observation will allow us to treat singularities in the set of identities in Section 7.4.

Definition 7.1.6 (p-Permuted Topology)**.** *Given a topology G over Id and p a partitioning consistent permutation of Id, the* p-permuted topology*, denoted by $p(G)$, is defined as*

1. $eq_{Id}(p(G)) = \{(u_1, u_2) \mapsto eq_{Id}(G)(p^{-1}(u_1), p^{-1}(u_2)) \mid u_1, u_2 \in Id\}$,
2. $U^{\circledcirc}(p(G)) := p(U^{\circledcirc}(G))$,
3. $U^{\not\circledcirc}(p(G)) := p(U^{\not\circledcirc}(G))$,
4. $\sigma(p(G)) := \{u \mapsto \sigma(G)(p^{-1}(u)) \mid p^{-1}(u) \in \mathrm{dom}(\sigma(G))\}$,
5. $L(p(G)) := L(G)$,
6. $\lambda(p(G)) := \lambda(G)$,
7. $\psi(p(G)) := \{\ell \mapsto (p^{-1}(u_1), p^{-1}(u_2)) \mid L(G)(\ell) = (u_1, u_2), \ell \in L(G)\}$,
8. $\rightarrowtail_{\lambda}^{p(G)} := \{u \mapsto \rightarrowtail_{\lambda}^{G}(p^{-1}(u)) \mid u \in U\}$,

with point-wise application to multi-set values in case 8., if multi-sets are used for G. ♢

Note that additional cases other than multi-set may occur, in particular due to the DTR, in the abstraction. But it needn't be discussed here because the DTR is supposed to be applied *only after* QR, in particular because the result of DTR is certainly no longer symmetric in identities (see below and Chapter 9).

The following note records the observation that permutation doesn't change the set of individuals as defined by the local state function (cf. Figure 7.1).

Note 7.1.7. *Let G be a topology over Id and p a partitioning consistent permutation of Id. Then the local state function of $p(G)$ is defined exactly on the permutation of the set of (alive) individuals according to the local state function of topology G, i.e.*

$$\mathrm{dom}(\sigma(p(G))) = p(\mathrm{dom}(\sigma(G))). \tag{7.2}$$

♢

Proof. Def. 7.1.6.4. □

Definition 7.1.8 (p-Permutation of Evolution Chains)**.** *Let δ be a (finite or infinite) evolution chain and p a partitioning consistent permutation of Id. Then $p(\delta)$, the p-*permutation of δ*, is defined point-wise as*

$$p(\delta) := p(\delta(0)), p(\delta(1)), \dots \tag{7.3}$$

◇

That is, if we consider identities to be additional labels of nodes, instead of being the nodes itself, then applying a permutation of identities to a topology merely re-labels the nodes according to the permutation.

7.1.3. Permutation and Terms

Definition 7.1.9 (Symmetric Structure)**.** *Let $\mathcal{S}$ be a signature and M an ETTS over Id compatible with $\mathcal{S}$. Let $V \subseteq \mathcal{V}(\mathcal{S})$ be a set of logical variables and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. M.*

The structure $\mathcal{M}$ is called symmetric in identities *if and only if the interpretations of functions do not depend on particular identities, but only work qualitatively on identities, that is, if for any partitioning consistent permutation p of Id,*

$$\tilde{p}(\iota(f)(d_1, \dots, d_k)) = \iota(f)(\tilde{p}(d_1), \dots, \tilde{p}(d_k)) \tag{7.4}$$

for each function symbol f of arity k and values d_i from the corresponding semantical domain.

For notational convenience, we use $\tilde{p}$ defined as follows

$$\tilde{p}(d) = \begin{cases} p(d) & \text{, if } d \text{ is of identity or evolution chain type} \\ p(d) & \text{, if } d \text{ is of type } L \\ d & \text{, otherwise} \end{cases} \tag{7.5}$$

where the middle case denotes the point-wise application of p to the given multi-set. ◇

Definition 7.1.10 (p-Permutation of Assignments)**.** *Let $\mathcal{S}$ be a signature and M an ETTS over Id compatible with $\mathcal{S}$. Let $V \subseteq \mathcal{V}(\mathcal{S})$ be a set of logical variables and $\mathcal{M}$ a canonical structure of $\mathcal{S}$ wrt. M.*

*Given an assignment θ of V in $\mathcal{M}$ and a partitioning consistent permutation p of Id, the p-*permuted assignment*, denoted by $p(\theta)$, is defined as*

$$p(\theta)(v) := \begin{cases} p(\theta(x)) & \text{, if } v = x \in \mathcal{V}^T \\ p(\theta(\boldsymbol{x}))(0), p(\theta(\boldsymbol{x}))(1), \dots & \text{, if } v = \boldsymbol{x} \in \mathcal{V}^{\boldsymbol{T}}, \end{cases} \tag{7.6}$$

that is, the permutation of evolution chains is defined by point-wise application of p as above. ◇

Note 7.1.11. *Under the premises of Definition 7.1.10, $p(\theta)$ is a type-consistent assignment.* ♢

Lemma 7.1.12 (Terms and Symmetry)**.** *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over Id, and $\mathcal{M}$ a structure which is canonical wrt. G and symmetric in identities.*

Let t be a term over $\mathcal{S}$. Then for each partitioning consistent permutation p of Id and each assignment $\theta \in Assign_{\mathcal{M}}(Free(t))$ of the free variables of t the valuation of t is compatible with permutation, i.e.

$$\tilde{p}(\iota[\![t]\!](G,\theta)) = \iota[\![t]\!](p(G), p(\theta)). \tag{7.7}$$

with $\tilde{p}$ as introduced in Def. 7.1.9. ♢

Proof. See Section A.3. □

Corollary 7.1.13 (Boolean Terms and Symmetry)**.** *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over Id, and $\mathcal{M}$ a structure which is canonical wrt. G and symmetric in identities.*

Given a boolean term t over $\mathcal{S}$, we have

$$\forall\, \theta \in Assign_{\mathcal{M}}(Free(\varphi))\ \iota[\![t]\!](G,\theta) = \iota[\![t]\!](p(G), p(\theta)) \tag{7.8}$$

for each partitioning consistent permuation p of Id.

Proof. By Def. 7.1.9, i.e. by definition of $\tilde{p}$. □

7.2. AEvoCTL* and Symmetry in Identities

Definition 7.2.1 (Symmetric in Identities)**.** *An ETTS M over Id is called* symmetric in identities *if and only if*

1. *labels discriminate states, that is, given two states $s_1, s_2 \in S$, we have $\mathscr{L}(s_1) = \mathscr{L}(s_2)$ only if $s_1 = s_2$,*

2. *for each partitioning consistent permutation p of Id, the function $a_p : S \to S$ induced by p which maps each state $s \in S$ to the state $s' \in S$ which is labelled with the p-permutation of the label of s, i.e. such that*

$$\mathscr{L}(s') = p(\mathscr{L}(s)), \tag{7.9}$$

is well-defined and

a) *an automorphism on the transition system core of M and* *b) consistent with evolution, that is,*

$$u \xrightarrow{(s,s')}\!\!\rightsquigarrow_e u' \tag{7.10}$$

if and only if

$$p(u) \xrightarrow{(a_p(s),a_p(s'))}\!\!\rightsquigarrow_e p(u'). \tag{7.11}$$

♢

Note that the discrimination property is necessary for a_p to be well-defined..

The requirement on e is necessary, as an individual may disappear for its particular identity in one half of the transition system while another individual remains alive in a different half.

Checking whether an ETTS is symmetric in identities is already hard for finite systems [33], and expected to be undecidable in the more general case. To this end, we will focus on ETTS which are symmetric in identities by construction (cf. Chapter 9).

Lemma 7.2.2 (Symmetric Paths in Symmetric ETTS). *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over Id which is symmetric in identities. Let $s \in S(M)$ be a state of M, $u \in U(s)$ an individual in s, and p a partitioning consistent permutation of Id.*

Then for each path π from s in M and each evolution chain δ of u along π, the p-permuted path

$$p(\pi) := p(\pi^0), p(\pi^1), \ldots \tag{7.12}$$

is a path from $p(s)$ in M and the p-permuted evolution chain

$$p(\delta) := p(\delta(0)), p(\delta(1)), \ldots \tag{7.13}$$

is an evolution chain of $p(u)$ along $p(\pi)$, i.e.

$$\forall\, \pi \in \Pi_s(M), \delta \in \Delta(u,\pi) : p(\pi) \in \Pi_{p(s)}(M) \wedge p(\delta) \in \Delta(p(u), p(\pi)). \tag{7.14}$$

Proof. Let $\pi \in \Pi_s(M)$ and $\delta \in \Delta(u,\pi)$. Then $p(\pi)$ is in $\Pi_{p(s)}(M)$ because a_p is an automorphism of M by Def. 7.2.1.2a and $p(\delta)$ is an evolution chain along $p(\pi)$ because a_p is consistent with evolution by Def. 7.2.1.2b as M is symmetric in identities. □

The previous Lemma states that in any ETTS M which is symmetric in identities, for each path π in M and each consistent permutation p, the permuted path $p(\pi)$ is also a computation path in M. In other words: if there is a path leading to a state where an invidual u_1 is in local state σ, then there is also a path in M leading to a state where individual u_2 is in local state σ.

That is, scenarios, in particular errors, do not depend on the identity; if individual u satisfies a property, then any other individual does, and if individual u runs into error, then any other individual does.

Note 7.2.3 (Symmetric Paths and Assignments)**.** *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over Id, and $\mathcal{M}$ a structure canonical wrt. M such that M and $\mathcal{M}$ are symmetric in identities. Let π be a computation path in M and θ an assignment of some logical variables from $\mathcal{S}$ in $\mathcal{M}$ along π.*

A partitioning consistent permutation p of Id is compatible with finite evolution, i.e.

$$\forall\, k \in \mathbb{N}_0 : \varepsilon \in \theta/k \iff \varepsilon \in p(\theta)/k. \tag{7.15}$$

♢

Proof. As M is symmetric in identities, evolution is consistent. That is, permutation of the assignment θ lets individuals disappear neither earlier nor later. □

Lemma 7.2.4 (AEvoCTL* and Symmetry)**.** *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over Id, and $\mathcal{M}$ a structure canonical wrt. M such that M and $\mathcal{M}$ are symmetric in identities.*

Let φ be an EvoCTL formula over $\mathcal{S}$ in negative normal form. Then*

$$\forall\, \theta \in \mathit{Assign}_{\mathcal{M}}(\mathit{Free}(\varphi)) : \mathcal{M}[\![\varphi]\!](s, \theta) = \mathcal{M}[\![\varphi]\!](p(s), p(\theta)) \tag{7.16}$$

and

$$\forall\, \theta \in \mathit{Assign}_{\mathcal{M}}(\mathit{Free}(\varphi)) : \mathcal{M}[\![\varphi]\!](\pi, \theta) = \mathcal{M}[\![\varphi]\!](p(\pi), p(\theta)) \tag{7.17}$$

for states $s \in S(M)$ and paths π in M and any partition consistent permutation p. ♢

Proof. See Section A.3. □

Theorem 7.2.5 (AEvoCTL* and Symmetry)**.** *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over Id, and $\mathcal{M}$ a structure canonical wrt. M such that M and $\mathcal{M}$ are symmetric in identities.*

Then for every EvoCTL formula φ over $\mathcal{S}$, $s \in S(M)$ a state of M, θ an assignment of the free variables of φ, and each partitioning consistent permuation p of Id, we have*

$$M, s, \theta \models \varphi \iff M, s, p(\theta) \models \varphi \tag{7.18}$$

and

$$M, s, \theta \not\models \varphi \iff M, s, p(\theta) \not\models \varphi. \tag{7.19}$$

♢

Proof. Lemma 7.2.4. □

7.3. Query Reduction Theorem

Definition 7.3.1 (Assignment Basis)**.** *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over Id, and $\mathcal{M}$ a canonical structure wrt. G.*

Let $V \subseteq \mathcal{V}(\mathcal{S})$ be a set of logical variables. A (possibly infinite) set

$$\Theta \subseteq \mathit{Assign}_{\mathcal{M}}(V) \tag{7.20}$$

of assignments of V in $\mathcal{M}$ is called assignment basis *of V if and only if each possible assignment of V in $\mathcal{M}$ is a p-permutation of an assignment from Θ, that is, if*

$$\forall\, \theta \in \mathit{Assign}_{\mathcal{M}}(V)\ \exists\, \theta_0 \in \Theta, p \in \mathit{perm}(\mathit{Id}) : \theta = p(\theta_0). \tag{7.21}$$

An assignment basis is called finite *if and only if the set Θ is finite.* ♢

Lemma 7.3.2 (Finite Assignment Basis (Strict))**.** *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over $\mathit{Id} = \mathit{Id}_1 \,\dot{\cup}\, \dots \,\dot{\cup}\, \mathit{Id}_n$ partitioned according to types in $\mathcal{S}$, and let $\mathcal{M}$ be a canonical structure wrt. G.*

Let $V \subseteq \mathcal{V}^T$ be a finite set of identity variables of the same type. Then there exists a finite assignment basis of V. ♢

Proof. See Section A.3. □

Example 7.3.3. *Let $V = \{x, y\}$ be a set of logical identity variables and assume the identities are taken from the natural numbers, i.e. $\mathit{Id} = \mathbb{N}^+$.*

Then

$$\Theta = \{\theta_1, \theta_2\}, \quad \theta_1 = \{x \mapsto 1, y \mapsto 1\}, \theta_2 = \{x \mapsto 1, y \mapsto 2\} \tag{7.22}$$

is an assignment basis of V because given any assignment

$$\theta = \{x \mapsto \mathit{id}_1, y \mapsto \mathit{id}_2\} \tag{7.23}$$

of V, we can distinguish two cases:

- *either $\mathit{id}_1 = \mathit{id}_2$, then $\theta = p(\theta_1)$ with $p = \{\mathit{id}_1 \mapsto 1, 1 \mapsto \mathit{id}_1\}$*
- *or $\mathit{id}_1 \neq \mathit{id}_2$, then $\theta = p(\theta_2)$ with $p = \{\mathit{id}_1 \mapsto 1, \mathit{id}_2 \mapsto 2, 1 \mapsto \mathit{id}_1, 2 \mapsto \mathit{id}_2\}$.* ♢

Note that the assignment basis constructed in the proof of Lemma 7.3.2 is not minimal. Not all of the assignments are actually used. For example, if identities are natural numbers, i.e. $\mathit{Id} = \mathbb{N}^+$ and we consider three logical variables, i.e. $V = \{x_1, x_2, x_3\}$, then Θ comprises

$$\{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 2\} \tag{7.24}$$

as well as

$$\{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 3\}. \tag{7.25}$$

These two are equivalent under the criterion that the first two variables x_1 and x_2 map to the same identity and x_3 to a different one.

The source of redundancy is that the usage of identities is not *tight* in the sense that a larger identity is used (here "3") even if there is a smaller identity (here "2") which would serve the same purpose. As identities are in general not ordered, we can apply this argument to the indices of employed identities. We shall not use an identity id_{0_i} if there is an unused id_{0_j} with $j < i$ available. This gives rise to the following lemma.

Lemma 7.3.4 (Minimal Finite Assignment Basis (Strict))**.** *Let $\mathcal{S}$ be a signature, G an $\mathcal{S}$-compatible topology over $Id = Id_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} Id_n$ partitioned according to types in $\mathcal{S}$, and let $\mathcal{M}$ be a canonical structure wrt. G.*

Let $V \subseteq \mathcal{V}^T$ be a finite set of $n \in \mathbb{N}_0$ identity variables of the same type and let $id_{0_1}, \ldots, id_{0_n}$ be n different identities from the corresponding domain. Then $\Theta := \Theta_n$, recursively defined by

$$\Theta_0 := \{\emptyset\} \tag{7.26}$$

$$\begin{aligned}\Theta_{k+1} := \{&\theta \cup \{x_{k+1} \mapsto id_{0_1}\}, \ldots, \theta \cup \{x_{k+1} \mapsto id_{0_{k+1}}\} \mid \theta \in \Theta_k\} \\ &\setminus \{\theta \cup \{x_{k+1} \mapsto id_{0_i}\} \mid id_{0_{i-1}} \notin \operatorname{ran}(\theta)\},\end{aligned} \tag{7.27}$$

is a minimal finite assignment basis of V. ♢

Proof. See Section A.3. □

The impact of Lemma 7.3.4 is only visible for $n > 2$. For example, with $n = 4$, there are already more than $(n-1)! - 1$ redundant cases, namely all but the last row of the following table:

x_1	x_2	x_3	x_4
1	1	1	4
1	2	1	4
1	1	2	4
1	2	2	4
1	1	3	4
1	2	3	4

The exact size of the minimal finite assignment basis is currently unclear, but we conjecture that it remains in the order of $O(n!)$. Yet we will see in the following sections that each removed redundant case saves a (possibly costly) model checking task, thus removal is worthwhile.

Corollary 7.3.5 (Minimal Finite Assignment Basis of Multiple Types)**.** *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over Id, and $\mathcal{M}$ a canonical structure wrt. G.*

Let $V \subseteq \mathcal{V}^T$ be a finite set of identity variables (of possibly different types). Then there exists a minimal finite assignment basis of V. ♢

Proof. See Section A.3. □

Now we have sufficient instruments to observe that for ETTS which are symmetric in identities, it is sufficient to prove finitely many *representative* cases in order to verify a formula in prenex normal form; as we'll only use that later, we consider only universal quantification.

Theorem 7.3.6 (Query Reduction)**.** *Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over $Id = Id_1 \dot{\cup} \ldots \dot{\cup} Id_n$, and $\mathcal{M}$ a structure canonical wrt. M such that M and $\mathcal{M}$ are symmetric in identities. Let φ be a closed EvoCTL* formula over $\mathcal{S}$ of the form*

$$\forall x_{1,1}, \ldots, x_{1,n_1} : T_1, \ldots, x_{m,1}, \ldots, x_{m,n_m} : T_m \cdot \varphi_0 \tag{7.28}$$

such that φ_0 is quantifier-free and $T_1, \ldots, T_m \in \mathcal{T}(\mathcal{S})$ are different identity types.

Then $M \models \varphi$ can be verified by verifying

$$M, \theta \models \varphi_0. \tag{7.29}$$

for only finitely many assignments $\theta \in Assign_{\mathcal{M}}(Free(\varphi))$. ♢

Proof. By definition, $M \models \varphi$ holds if and only if $M, \theta \models \varphi$ holds for all assignments $\theta \in Assign_{\mathcal{M}}(Free(\varphi))$.

Let Θ be a finite assignemnt basis of *Free*(φ) which exists by Cor. 7.3.5 and assume

$$M, \theta_0 \models \varphi_0 \tag{7.30}$$

holds for all $\theta_0 \in \Theta$.

As Θ is an assignment basis, for each assignment $\theta \in Assign_{\mathcal{M}}(Free(\varphi))$ there is a permutation p and an assignment $\theta_0 \in \Theta$ such that $\theta = p(\theta_0)$. By Theorem 7.2.5, we have

$$M, \theta \models \varphi_0 \iff M, p(\theta_0) \models \varphi_0 \iff M, p^{-1}(p(\theta_0)) \models \varphi_0 \iff M, \theta_0 \models \varphi_0 \tag{7.31}$$

because p^{-1} is also a permuation. Thus $M \models \varphi$. □

So Query Reduction is based on two ingredients: firstly, that for each *finite* set of logical identity variables there is a finite assignment basis, and secondly that system and structure are symmetric in identities, that is, we can not only by permutation *reach* all possible assignments we've got to consider, but also *conclude* on the evaluation under this assignment based on the representative case.

7.4. Treating Singularities

The previous paragraphs assumed a "clean" setting in the sense that ETTS and structure are symmetric in all identities. Yet in object-oriented programming languages, which we'll consider as descriptions of ETTS, there is often a notion of a special identity "NULL" to indicate unused links, that is, links who intentionally point to no other individual are set to point to NULL. Systems employing NULL are no longer symmetric in identities: it obviously makes a difference whether a property holds for NULL (for which there may not even be an individual in the topologies) or for a different identity. The reason is that NULL obtains special treatment by the semantics of object-oriented programming languages, interaction with NULL often leads to exceptional program termination. Thus for the future of an individual it makes a difference whether it has links to the NULL individual or to individuals with regular identities.

There are basically two ways to treat *singularities* such as NULL:

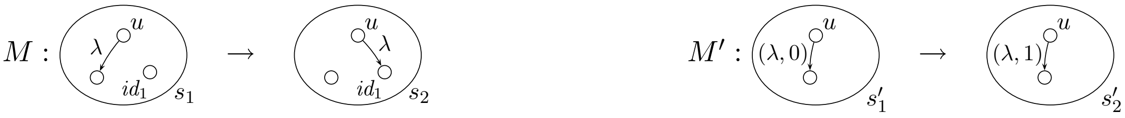


Figure 7.2.: **Symmetric encoding.** If link λ changes to the singularity id_1 in the transition from s_1 to s_2 in M, then the bisimilar ETTS M' will encode this change by labelling the link with $(\lambda, 1)$ in s'_2.
More concrete, if id_1 is NULL, then u setting its λ link to NULL in M is encoded in M' by marking the link as "pointing to NULL" using the new link name (λ, NULL).

1. symmetric encoding and
2. explicit, semantical treatment.

7.4.1. Symmetric Encoding

More formally, a singularity is an identity which has the effect that definitions Def. 7.1.9 and 7.2.1 are not satisfied. The most prominent source they stem from is the special identity NULL – individuals behave differently when having a link to NULL than when having a link to an ordinary individual, thereby breaking 7.2.1.

Furthermore, there may be a constant "NULL" in the signature denoting the special identity. Then the interpretation of the term $x = \text{NULL}$ depends on the value assigned to x. we have $\iota[\![x = \text{NULL}]\!](G, \theta) = 1$ if θ maps x to the special identity and $\iota[\![x = \text{NULL}]\!](G, \theta) = 0$ otherwise, hence we cannot freely permute thereby breaking 7.1.9.

As our aim is to faithfully model object-oriented programs, we've got to support such special identities.

We call *symmetric encoding* a transformation of an ETTS M over identities Id with finitely many singularities, i.e. where finitely many identities $id_1, \ldots, id_n \in Id$ obtain a non-symmetric treatment, into a bisimilar[1] ETTS M' over $Id' := Id \setminus \{id_1, \ldots, id_n\}$ which is symmetric in identities.

In case of only the single singularity NULL, the idea is to have M' use link names from $\Lambda \times \mathbb{B}$, that is, links are not only labelled by a name but by a pair of name and a boolean flag indicating whether the link is valid or NULL.

In general, M' would employ $\Lambda \times \{0, 1, \ldots, n\}$, where 0 indicates regular links and $1 \leq i \leq n$ indicates that a given link in M' represents a link to id_i in M. Independent from the *actual* terminal vertex (cf. Figure 7.2).

The resulting M' is bisimilar and symmetric in identities if the singular individuals have a constant local state, outgoing links, and behaviour. Otherwise there had to be an individual to represent that local state, outgoing links, and behaviour, which would lead us back into the non-symmetric situation. This is obviously the case with NULL,

[1] the definition is straightforward, similar to our definition of simulation; we didn't define bisimilarity in the section on simulation because it's only needed here

so this case can principally be treated by a symmetric encoding following these ideas. Being symmetric in identities has to be proven for each such transformation. Whenever the ETTS M is defined by a given program, one possibility to verify this property is to provide a defining *program* which yields M' and to apply the syntactic criteria of [89].

Yet we've only treated the *model* side. In order to employ this approach for EvoCTL* verification, the used structure would also need to be adjusted in order to interpret the new link labels correctly. For example with NULL, a navigation expression $x \rightarrowtail \lambda$ should evaluate "undefined" if the link is actually pointing to the NULL node. Here we see that canonical structures of non-symmetric ETTS are typically also not symmetric in identities.

In case of NULL, obtaining a canonical structure $\mathcal{M}'$ for M', given $\mathcal{M}$ and M, is straightforward. Namely, $x \rightarrowtail \lambda$ maps to the terminal vertex as seen from the individual denoted by x if the label is $(\lambda, 0)$, and is undefined in case the label is $(\lambda, 1)$. The example "NULL" has the additional nice property that it is typically not meant to be quantified, that is, there typically is a constant NULL in the signature and a property of the form

$$\forall\, x : T \,.\, \varphi_0 \tag{7.32}$$

has the further structure

$$\forall\, x : T \,.\, (x \neq \mathrm{NULL}) \rightarrow \tilde{\varphi}_0. \tag{7.33}$$

So, some cases can be treated by what we call symmetric encoding, but is has certain serious drawbacks

- we have to devise a procedure to obtain M' from M' which ensures that M' is bisimilar to M and that M' is symmetric in identities, and
- we have to devise a procedure to obtain canonical structures $\mathcal{M}'$ wrt. M', whose interpretations are sound wrt. $\mathcal{M}$

We have sketched this for the case of NULL, yet in general it may become arbitrarily intricate depending on the behaviour exhibited by the singular identities.

7.4.2. Semantical Treatment

The second approach – explicit, semantical treatment – is less awkward. It basically weakens the definition of symmetry in identities to symmetry in *a subset* of identities.

For example, in case of systems with a NULL, the behaviour is still symmetric in all other identities. Based on this observation, we can retain the definition of bases and obtain a finite basis lemma for partial symmetry.

For example, assume the simple case of NULL as the only singularity; in order to obtain type consistency, we'll have one NULL identity per type, that is $\mathrm{NULL}_i \in \mathit{Id}_i$ if $\mathit{Id} = \mathit{Id}_1 \,\dot{\cup}\, \ldots \,\dot{\cup}\, \mathit{Id}_n$. Then we consider the sub-partitioning $\mathit{Id}_i = \{\mathrm{NULL}_i\} \,\dot{\cup}\, \mathit{Id}_i \setminus \{\mathrm{NULL}_i\}$ and generalise the definition of symmetry in identities to sub-partitions. In this case, the system is certainly symmetric in the (singleton) partition $\{\mathrm{NULL}_i\}$, and if it is also symmetric in the complement, then we can apply the overall procedure in an unchanged fashion.

Recall that in order to represent types of individuals, we assumed that the set *Id* of identities is partitioned into $Id_1 \dot\cup \ldots \dot\cup Id_n$. In the following we consider each of these partitions to be further partitioned into $Id_{i,1} \dot\cup \ldots \dot\cup Id_{i,m_i}$ for $1 \leq i \leq n$ with $m_i \in \mathbb{N}^+$.

Note that Def. 7.1.5 doesn't require a particular partitioning of *Id*, thus it applies verbatim to the partitioning

$$Id_{1,1} \dot\cup \ldots \dot\cup Id_{1,m_1} \dot\cup \ldots Id_{n,1} \dot\cup \ldots \dot\cup Id_{n,m_n} \tag{7.34}$$

Similarly, Def. 7.1.6 is independent from the partitions the permutation operates on. Also similarly, by Def. 7.1.9 we'll call a structure symmetric in identities if and only if interpretations are invariant under permutations which are consistent with the finer partitioning. Lemma 7.3.2 on finite assignment basis is the first part which explicitly considers the partitioning corresponding to types. In the following, we extend it to sub-partitioning.

Lemma 7.4.1 (Finite Assignment Basis (Generic)). *Let $\mathcal{S}$ be a signature, G a $\mathcal{S}$-compatible topology over $Id = Id_1 \dot\cup \ldots \dot\cup Id_n$ partitioned according to types in $\mathcal{S}$, each partition Id_i sub-partitioned into $Id_{i,1} \dot\cup \ldots \dot\cup Id_{i,m_i}$, $m \in \mathbb{N}^+$, and let $\mathcal{M}$ be a canonical structure wrt. G.*

Let $V \subseteq \mathcal{V}^T$ be a finite set of identity variables of the same type. Then there exists a finite assignment basis of V. ♢

Proof. See Section A.3. □

Example 7.4.2. *Let $V = \{x, y\}$ be a set of logical identity variables and assume the identities are taken from the natural numbers with* 0*, i.e. $Id = \mathbb{N}_0$, where* 0 *is a singularity.*

Then

$$\begin{aligned} \Theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5\}, \quad & \theta_1 = \{x \mapsto 0, y \mapsto 0\}, \theta_2 = \{x \mapsto 1, y \mapsto 0\}, \\ & \theta_3 = \{x \mapsto 0, y \mapsto 1\}, \theta_4 = \{x \mapsto 1, y \mapsto 1\}, \\ & \theta_5 = \{x \mapsto 1, y \mapsto 2\} \end{aligned} \tag{7.35}$$

is an assignment basis of V. ♢

Note that the assignment basis given in the proof of Lemma 7.4.1 is again not minimal. We don't elaborate on a procedure to obtain a minimal assignment basis here because in the following, we'll only consider singularities with only a single element. Then a minimal assignment basis is straightforwardly obtained from a minimal assignment basis for the strict case following Lemma 7.3.4.

Further note that Lemma 7.4.1 is tight in the sense that if we admit an infinite sub-partitioning of identities (or: if the considered structure and ETTS only become symmetric in identities with an infinite sub-partitioning), then there is in general no finite set of representative assignments. In other words: the non-symmetric part of the considered systems have to be finite in order to apply the query reduction theorem.

Lemma 7.3.2 is a special case of Lemma 7.4.1 by considering the Id_i to be sub-partitioned into only the single partition Id_i. Cor. 7.3.5 applies directly: if there are variables of multiple types whose semantical domains are sub-partitioned, then the cross-product of given assignment bases is taken. Having extended all prerequisites of the Query Reduction Theorem (Theorem 7.3.6), we see that the we obtain it similarly with the relaxed definition of being symmetric in identities.

7.4.3. Sources of Singularities

Remaining questions are where the singularities stem from and how do we obtain good sub-partitionings.

The common case is simply knowing it, as in the case for NULL. If a high-level language with ETTS as semantical domain (cf. Chapter 9) has a concept of NULL and is otherwise independent from identities, then we're in the setting discussed above.

Furthermore, if the high-level language is checked with the criteria of [89], there may be constructs to declare parts of the description as *symmetry breaking* – then the syntactical check succeeds, but the user has to ensure that the parts declared to be symmetry breaking actually do preserve overall symmetry but are only not covered by the (too strict) criteria (this is the approach of [127]). Alternatively, affected regions are candidates for singularities.

Note that the syntactic criteria from [89] forbid literals (in the programming language sense), that is, code which compares identity variables (or pointers) to literal constants. For example in (the pseudo-code)

```
if (this == 0x0003) { /* A */ } else { /* B */ };
```

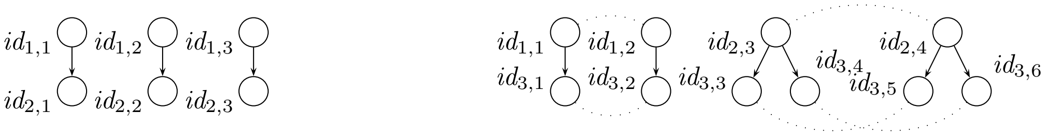
the constant `0x0003` would be a forbidden literal because the object behaves differently, depending on its address.

By the discussion of singularities, it seems that this prerequisite is not necessary as long as the description (the program) is finite. Then it can only comprise finitely many literals, thus there are always only finitely many singularities which can be treated by sub-partitioning, yielding symmetry in identities in the more generic sense. But in the worst case, there may be too many cases to consider to be practically treatable.

It's not clear whether this reasoning actually applies to the setting of [89]. They possibly actually need full symmetry in order for their procedure to work. Yet some things (like NULL) could be encoded with additional boolean variables and would pass their checks. It does apply to our approach (cf. Chapter 8).

7.5. Crystallisation

Another reason for an ETTS not being symmetric in identities can be *crystallisation*. In [158], crystallisation is used to reduce the variability in topologies by the observation that some links need not be dynamic.



(a) Individuals $id_{1,1}$, $id_{1,2}$, $id_{1,3} \in Id_1$ are wholes with fixed assigned parts $id_{2,1}$, $id_{2,2}$, $id_{2,3} \in Id_2$.

(b) Individuals $id_{1,1}, id_{1,2} \in Id_1$ and $id_{2,1}, id_{2,2} \in Id_2$ are wholes with fixed assigned parts $id_{3,1}$, …, $id_{3,6} \in Id_3$. Dotted lines indicate local symmetry.

Figure 7.3.: **Whole-part** relations and crystallisation.

An epitome is the concept of *aggregation* in UML, a whole-part relation where objects of one class comprise (or own) different objects of other classes. For instance, for car platooning a car may be modelled more realistically as having an engine and braking system and a set of front distance sensors. Then each time a car is created, the part-objects are created along and interlinked with the car.

In general, the identity of the parts will not depend on the identity of the whole, that is, for each combination of whole and part identities, there will be system run featuring that combination . The idea of crystallisation is to remove this variability by establishing a relation between whole and part identities, such that the identity of the whole uniquely determines the identities of the parts.

More formally, a crystallisation is a function $❄ : Id \to (\lambda \mathrel{+\!\!\!\!\to} Id)$ mapping identities from a sort Id_i, the whole, per link name to the determined part, for simplicity assuming single-link topologies. Then the crystallised system $M^{❄}$ only has topologies where the links affected by ❄ adhere to it, that is, have the individuals denoted by ❄ as terminal vertex.

A crystallisation is said to be well-formed, if $M^{❄}$ bisimulates M. Not every ETTS has a non-trivial crystallised correspondence, for instance, if there is no notion of whole/part in the ETTS. The intended effect is that links between whole and part need no longer be variable, but turn into constants, which is highly desirable and beneficial in finite-state symbolic model-checking. In other words, an encoding of the ETTS needs not store the links but can instead refer to the (constant) crystallisation function ❄ to look up links. This approach has been demonstrated in the joint work [159].

Obviously, this approach breaks symmetry in identities. For instance, if the λ-link of individual id_1 is constantly pointing to id_2, then any identity $id \neq id_2$ is no longer representative, there is no trace in the system where id is the part-object of id_1 denoted by λ. Yet the system may remain symmetric in the identities of the wholes – for each whole there is some part, and if the parts behave the same so do the wholes, independent from their both identities.

So we conjecture that we can further relax the definition of symmetry to only consider the partitions of Id related to wholes. Then any property *only* referring to wholes should be treatable by the approach presented in Chapter 8, the combination of QR and DTR.

But as soon as parts need to be considered, matters become more complicated. For example, consider Figure 7.3(a). It depicts some individuals from a topology where the wholes $id_{1,1}$, ..., $id_{1,3}$ are predetermined to employ the parts $id_{2,1}$, ..., $id_{2,3}$. The identities $id_{1,i}$ may still behave symmetric, thus in order to verify a requirement, in form of an EvoCTL* formulae, only referring to individuals of sort Id_1, considering individual $id_{1,1}$ is representative. Similarly, $id_{2,1}$ is representative for requirements only referring to sort Id_2. But for requirements referring to a whole/part the combination $id_{1,1}$ and $id_{2,3}$ is no longer representative because $id_{2,3}$ is never the part of $id_{1,1}$. Thus violations of the specification would go unnoticed.

In general, individuals of a type are not bound to be parts of individuals of a single type, there may be multiple whole types. For example in Figure 7.3(b) there are individuals of sorts Id_1 and Id_2 which both have parts of sort Id_3. In this case, $id_{3,1}$ is even no longer representative for its sort, we have to consider an individual which is part on an Id_2-sort whole, e.g. $id_{3,3}$, and possibly even also $id_{3,4}$ if these parts are treated differently by the whole.

This approach generalises as follows. We can still identify *local symmetries* within sorts, for example, we may be able to establish[2] that the system is symmetric in those individuals from Id_3 who are assigned as

- parts to wholes of sort Id_1,
- left parts to wholes of sort Id_2, and
- right parts to wholes of sort Id_2.

Thereby we obtain sub-partitions in the sense of Section 7.4 and with the theory of that section can treat requirements referring only to sort Id_3.

For mixed-sort requirements, Cor. 7.3.5 as such is not adequate because it doesn't respect the whole/part assignments from the crystallisation ❄. Thus the cross-product of assignment bases has to be adjusted such that it yields for two assignment bases a set of assignments where all cases are respected, the case where the part identities adhere to ❄, the case where they don't, and all (finitely many) combinations.

This can be achieved by permuting the affected assignments. The reason is that specification may also state properties on wholes and parts for the case that the denoted part is *not* a part of the denoted whole, e.g. "for any x of the whole-sort and y of the part-sort, if y is *not* the part of x, then ...". For example, in case of a property of the form

$$\forall x : T_1 y : T2 . \varphi_0, \tag{7.36}$$

for an ETTS with topologies similar to Figure 7.3(a), single bases are

$$\{\{x \mapsto id_{1,1}\}\} \text{ and } \{\{y \mapsto id_{2,1}\}\}. \tag{7.37}$$

[2] Detecting symmetries in a given ETTS is highly complex [62]. It seems like any practical application depends on "symmetry by design" or analysis on a higher-level language, or both. We discuss in Chapter 9 both cases for *global symmetry.* Firstly, the combination DCS/METT is symmetric by design, each high-level description denotes an ETTS which is symmetric in identities. Secondly, for the combination UML/LSC, or particularly C++/LSC, the syntactical characteristics of [89] can be applied to confirm that a given C++ description yields an ETTS which is symmetric in identities.

The cross-product has to yield, for example,

$$\{\{x \mapsto id_{1,1}\}, \{y \mapsto id_{2,1}\}\}, \{\{x \mapsto id_{1,1}\}, \{y \mapsto id_{2,2}\}\}, \tag{7.38}$$

in order to cover all cases.

This generalised cross-product obviously yields larger bases. That is, the price for crystallisation is that there are more cases to consider. Overall, it may pay off because, as discussed in Chapter 8, cases can be verified separately, in particular fully in parallel while there is no generally effective parallelisation of symbolic finite-state model-checking. That is, the *computation time*, the sum of seconds spent on all parallel processors to verify a property on many cases with crystallised systems, may be larger than without crystallisation, but the *wall clock time* may be smaller, given sufficiently many processors to work on the cases in parallel.

We don't pursue this path further as we don't employ crystallisation in our case-studies, yet there is a connection which needs proper elaboration in future work.

7.6. Discussion

The notion of symmetry presented here is taken from [89]. A new concept is symmetry in identities, although it is a special case of [89].

In comparison to [124, 127] we have a minimal set of representative cases, instead of only a finite sufficient set.

New is the discussion of the treatment of singularities and crystallisation. Both discussion are driven by the application domain UML, or, more precise, UML with C++ as action language as supported in the UVE tool [158, 159]. From C++, it inherits the certainly singular NULL identity, and it is an inherent optimisation to consider crystallisation.

8. Data Type and Query Reduction Combined

8.1. Finitary Abstraction by Heuristics

Recall that by Chapter 6, we have a procedure to obtain an abstract ETTS $M^\sharp$ when given an ETTS M and a DTR in the sense of Def. 6.1.1. If the individuals in M have finite local states, that is, if Σ is finite, and if there is a finite upper bound on the out-degree of individuals, then $M^\sharp$ is finite by Cor. 6.1.8.

With Chapter 7 we have a procedure to obtain a finite set of representative cases, in form of a finite assignment basis Θ, given an EvoCTL* specification φ of the form

$$\forall\, x_{1,1}, \dots, x_{1,n_1} : T_1, \dots, x_{m,1}, \dots, x_{m,n_m} : T_m \,.\, \varphi_0 \tag{8.1}$$

or an EvoCTL* formula that has an equivalent formula in the form of (8.1) (cf. Section 4.5).

This leads to the following principal strategy [127] for confirming

$$M \models \varphi \tag{8.2}$$

for an M over $\mathit{Id} = \mathit{Id}_1 \,\dot{\cup}\, \dots \,\dot{\cup}\, \mathit{Id}_n$ which is symmetric in identities (cf. Figure 8.1):[1]

1. obtain a finite assignment basis Θ, at best a minimal one,
2. for each assignment $\theta \in \Theta$ define the DTR

   $$D_\theta = \{(\mathrm{ran}(\theta) \cap \mathit{Id}_1, \mathit{Id}_1), \dots, (\mathrm{ran}(\theta) \cap \mathit{Id}_n, \mathit{Id}_n)\} \tag{8.3}$$

   that is, covering all partitions of *Id*, where subsets corresponding to types which doesn't occur in φ become empty,
3. construct the abstract transition system $D_\theta(M)$,
4. for all finitely many $\theta \in \Theta$, verify

   $$D_\theta(M), \theta \models \varphi_0. \tag{8.4}$$

   If verification does not confirm (8.4) but yields a counter-example, stop with "unknown".

[1] Footnote 2 from page 192 applies similarly. Namely, establishing whether a given ETTS is symmetric in identities is highly complex [62]. It seems like any practical application depends on "symmetry by design" or analysis on a higher-level language, or both. We discuss both cases in Chapter 9.

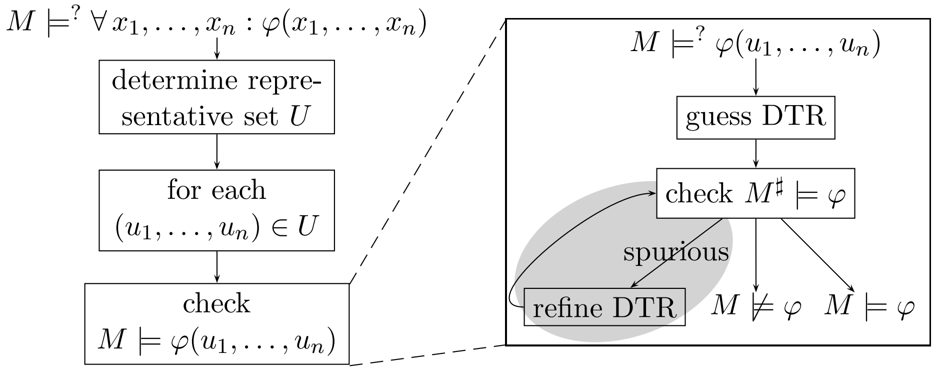

$M \models^? \forall x_1, \ldots, x_n : \varphi(x_1, \ldots, x_n)$
determine representative set U
for each $(u_1, \ldots, u_n) \in U$
check $M \models \varphi(u_1, \ldots, u_n)$
$M \models^? \varphi(u_1, \ldots, u_n)$
guess DTR
check $M^\sharp \models \varphi$
spurious
refine DTR
$M \not\models \varphi$
$M \models \varphi$

Figure 8.1.: **Overall procedure**. An AEvoCTL* property φ in prenex normal form is checked for an ETTS M by determining a representative set (which is finite if the system is sufficiently symmetric) and verifying each representative in isolation.
Each representative is checked by a heuristically obtained initial DTR abstraction $M^\sharp$ of the system. If φ holds for $M^\sharp$, then it holds for M. If the check yields a non-spurious counterexample, then the property doesn't hold for M. Otherwise the DTR is refined and another iteration started.

Otherwise conclude

$$M, \theta \models \varphi_0 \tag{8.5}$$

by Theorem 6.2.1.

5. If (8.5) can be established for all assignments of an assignment basis, conclude (8.2) by Theorem 7.3.6, thus stop with "holds".

Note that the procedure is not *deciding* (8.2), but only confirms it if step 4 succeeds. The reason is that it employs the abstraction DTR, which is not guaranteed to *reflect* (or preserve) the considered property. Instead, step 4 may fail for two reasons: either $M \not\models \varphi$, then the answer is "negative", or only $D_\theta(M), \theta \not\models \varphi$, then an obtained counter-example (cf. Section 6.4) is spurious, and the best answer we can give is "unknown".

Recall from Section 6.4 that it remains to identify whether a given counter-example is spurious or not and that this property is in general undecidable [167]. In Section 6.4.5 and Section 6.4.6, we've only briefly discussed possible approaches, model-based and assumption-based, as this lies out of our scope.

One option is to enlarge the DTR, that is, add additional concrete identities to the spotlight (cf. Section 6.4.5). Now enlarging the DTR fits into the overall procedure by adding so-called case-splits to (8.2) (cf. Section 4.5.3). That is, to quantify over more variables. Then the heuristic procedure above automatically constructs larger DTRs.

An example for a case-split would be that a car depends on its sensors, for example. Then the sensors wouldn't necessarily be named right-away in (8.2) thus the heuristic procedure wouldn't have any concrete sensor individual, they'd all end up in the shadows. That verification is bound to fail.

A case-split would add a logical variable x quantifying over sensor individuals and add the premise that the car in question has this concrete sensor x as its sensor. There are cases in [127] where this approach was sufficient to prove formulae of the form of (8.2).

8.2. Discussion

The procedure outline above is the original one from [124, 127]. It's sometimes considered a weakness that it is only finitary by heuristics, and that it employs a rather weak heuristics, cutting off everything not referred to by the property. On the other hand, given an implementation of the procedure, it's easy to refine the abstraction by quantifier introduction via case-splits (cf. Chapter 4).

Note that a similar reasoning based on QR can be applied for falsification by under-approximation. For example, [114] employs search techniques to find computation paths to certain configurations of object-oriented programs. It implicitly assumes a symmetry in identities as it doesn't try to allocate different identities, but is satisfied with any identity if the search shows that another individual is needed. Interestingly, there is no obvious under-approximating dual of DTR. The reason is that it's not clear with which local state to label the C individual, as there is no natural $\perp$ value which would carry over to Chapter 9. The only investigated under-approximation is a finite upper bound on the number of individuals.

Part IV.

Application

9. DTR/QR for Higher-Level Languages

Studying DTR and QR in the pure world of ETTS (cf. Chapters 6 and 7) helps understand its principles, but the description of DTR is not constructive because the starting point is an infinite state system. Furthermore, in order to apply QR we need to know whether the system is symmetric in identities, which is is computationally hard to establish on an ETTS [62]. Thus the practical approach is different: have a high-level language HLL whose semantics is an ETTS such that

1. we know that each ETTS obtained from an HLL description is symmetric in identities, or have easily checkable syntactic criteria to establish the symmetry property, and

2. there is a procedure to compute the finite abstraction directly from the HLL description.

In Section 9.1, we define syntax and semantics of a high-level modelling language we'll call HLL. The semantics of an HLL model $\mathscr{M}$ is an ETTS M (cf. Figure 9.1).

The main design objectives were that HLL should be as general as possible, at least generalising both application domains we aim at, namely UML and DCS (cf. Chapter 10), such that we need to discuss symmetry and computation of the abstraction only once. This goal is achieved by giving an HLL semantics which is parameterised in all aspects that are *orthogonal* to topologies, namely the scheduling of individuals, the communication medium, and object creation. That is, whenever a high-level language is similar to our HLL, or can be viewed as an instantiation, then we can learn from our results how to apply DTR and QR. In Chapter 10, we'll see that HLL is sufficiently general to embed both DCS [9] and a significant subset of UML [138, 45] into it, basically by instantiation; thereby the techniques of this and the previous chapters apply to the two contexts of UML and DCS (cf. Figure 9.1).

In Section 9.2, we discuss the topic of symmetry for HLL, that is, we discuss under which premises the ETTS semantics of a given HLL model is symmetric in identities. Existing applications of the idea to detect symmetry syntactically typically argue within the limits of a given formalism, that is, in a certain instantiation of HLL. From our results we can tell which premises these formalism limits (or these instantiations) have to satisfy to preserve symmetry.

In Section 9.3, we discuss procedures to obtain the DTR abstraction for a given HLL directly. That is, instead of obtaining the ETTS $M = [\![\mathscr{M}]\!]$ of an HLL model by applying the HLL semantics and then constructing $M^\sharp = D(M)$ from a given DTR D according to Chapter 6, we have (under certain premises) a procedure to construct $\mathscr{M}_D^\sharp$, which, by the standard HLL semantics maps to the ETTS $M^\sharp$, or to a mildly coarser abstraction

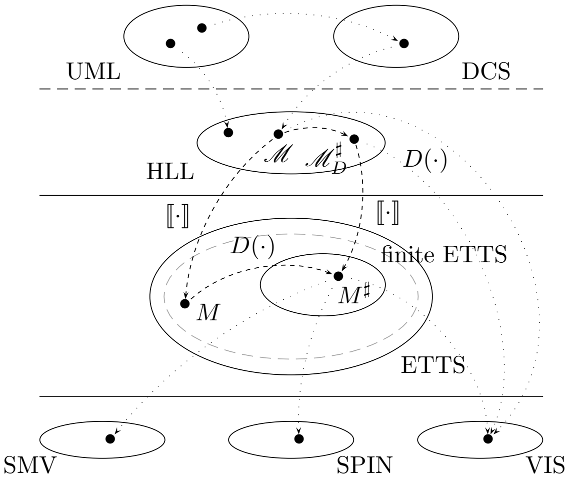


Figure 9.1.: **Schematic plan** for the current and the next chapter. The dashed line indicates that HLL is not able to describe all ETTS, yet the most relevant (cf. Section 9.1.4).

than $M^\sharp$, but in any case without constructing M first (cf. Figure 9.1). In other words, we want the part of Figure 9.1 connected by dashed arrows – read as a diagram – to commute.

Section 9.4 briefly discusses how to encode the resulting abstract and finite-state ETTS $M^\sharp$ in common input languages of finite-state model-checkers like VIS [154] or SMV [128] (cf. bottom of Figure 9.1). And, of most practical interests, we're discussing procedures to go with a given DTR D directly from the HLL model $\mathscr{M}$ (or from $\mathscr{M}_D^\sharp$) to a finite-state encoding in array programs (cf. Figure 9.1). Section 9.5 concludes.

Note that this chapter is to some amount technical and tedious. We have to define two languages, HLL and array programs, at a sufficient level of detail and we define two transformations, one into a normal-form and one into a representation of the abstract DTR'd system.

But the effort is not completely fruitless. We can discuss the approach for the first time in complete depth, which other works employing the abstraction also in form of a syntactical transformation get around [28, 29, 31].[1] What we obtain are, for instance, a clearer understanding of the relation between the syntactical transformation and Chapter 6. Certain choices do harm precision, even if they're semantics preserving in the original. Furthermore, we understand that a prerequisite is effective interleaving. Interestingly, this is given in all case-studies provided by the literature, but never explicitly discussed.

[1] The reasons for getting around it are now quite understandable to us.

9.1. High-Level Language

9.1.1. Syntax

HLL Model

Let $\mathscr{E}$ be a finite (possibly empty) set of events. Each $E \in \mathscr{E}$ has a number of parameters $p_1, \ldots, p_n$, each with either an object or a basic type (see below). We sometimes write the short form $E(\vec{p})$ for $E(p_1, \ldots, p_n)$.

Let $\{\tau_1, \ldots, \tau_n\}$, be a finite (possibly empty) set of basic types, for example integer or boolean types. Additional types are for each class C (see below) object types τ_C and the current state type $\tau_{C,st}$, and an events type $\tau_{\mathscr{E}}$, which is used to denote the type of event instances, that is, events with concrete parameters. The latter is a super-type of all event types τ_E, that is, the domain of the events type $\tau_{\mathscr{E}}$ will simply be the union of all event type domains.

Let $\mathscr{C}$ be a non-empty finite set of classes. Each $C \in \mathscr{C}$ has

1. a finite set of (local) variables
$$X_C = \{x_1, \ldots, x_n\} \cup \{x_{C,st}\}, i \in \mathbb{N}_0, \tag{9.1}$$
each $x \in X_C$ has a type $\tau(x)$, which is $\tau_{C,st}$ for x_{st} and any basic type otherwise,

2. possibly empty and disjoint sets of variables $X_{I,C} \subsetneq X_C$ and $X_{A,C} \subsetneq X_C$ denoting inputs and auxiliary variables,

3. a finite set of links $\Lambda_C \supseteq \{\mathit{this}\}$, each $\lambda \in \Lambda_C$ has a type $\tau(\lambda)$, which is τ_C for *this* and any object type otherwise.

   Here one can imagine any kind of additional annotations known for example from UML, like multiplicity, distinguishing association and composition, etc. We don't add these because adherence to multiplicity is not syntactically checkable (except for the single-link and the unrestricted case) and association and composition are syntactic sugar rather.

4. possibly empty and disjoint sets of link names $\Lambda_{I,C} \subsetneq \Lambda_C$ and $\Lambda_{A,C} \subsetneq \Lambda_C$ denoting input and auxiliary links, that is, links updated by the open environment, as we chose not to have local variables of an object type, and temporary links, and

5. a non-empty set of states S_C, a non-empty subset of initial states $S_{0_C} \subseteq S_C$, a (possibly empty) set of transitions R_C, where a transition $r_C \in R_C$ is a triple $r_C = (s_C, \ell, s'_C)$ with $s_C, s'_C \in S_C$, called source and destination state of r_C, and a label ℓ according to Section 9.1.1.

We use $R_{\mathscr{C}}$ to denote the set of local transitions of all classes in $\mathscr{C}$. If the class index is clear by context, we may omit it.

A condition cond_0 (cf. Section 9.1.1) characterises the initial system states. Every topology satisfying it is a legal initial state; we'll later actually consider a vastly restricted variant of initial state characterisations.

It's a first-order variant of the conditions from Section 9.1.1, namely

$$\mathit{init} ::= \langle \mathit{cond} \rangle \mid \forall\, p : \tau\,.\, \langle \mathit{init} \rangle \mid \exists\, p : \tau\,.\, \langle \mathit{init} \rangle \tag{9.2}$$

In contrast to regular conditions, it is not referring to *this* but only to p at the left-most position in navigation expressions, that is, without the second production of (9.9). And it is assumed to be closed, all parameters p occurring in the condition part have to be bound by an enclosing quantifier.

Given a language (or set) of transition labellings, the tuple

$$\mathscr{M} = (\mathscr{T}, \mathscr{E}, \mathscr{C}, \mathit{cond}_0) \tag{9.3}$$

is called *HLL model* over this transition labellings if all ℓ are from the given language and if cond_0 is a condition of the given language.

The rationale behind this approach is that we don't want to restrict the discussion to a particular action language, as is common with other approaches. We rather discuss what has to be required for the action language in order to obtain, for example, symmetry.

HLL Transition Programs

A transition label ℓ over events $\mathscr{E}$ is a word of the following grammar

$$\ell ::= \langle \mathit{ev} \rangle \, [\langle \mathit{cond} \rangle] / \langle \mathit{act}_1 \rangle ; \ldots ; \langle \mathit{act}_n \rangle \tag{9.4}$$

where $n \geq 0$ and

$$\mathit{ev} ::= \varepsilon \mid E(\vec{p}), \tag{9.5}$$

where $E \in \mathscr{E}$ is an event.

A condition

$$\mathit{cond} ::= \langle \mathit{term} \rangle \tag{9.6}$$

is a term over purely unprimed expressions of a boolean type, where terms are defined by the grammar

$$\mathit{term} ::= 1 \mid \langle \mathit{expr} \rangle \mid \langle \mathit{nav}_1 \rangle = \langle \mathit{nav}_2 \rangle \mid \neg \langle \mathit{term} \rangle \mid \langle \mathit{term}_1 \rangle \wedge \langle \mathit{term}_2 \rangle \tag{9.7}$$

that is,

1. 1 and expressions of a boolean type (see below) are boolean terms,
2. if $\langle \mathit{nav}_1 \rangle$ and $\langle \mathit{nav}_2 \rangle$ are navigation expressions of the same type, then $\langle \mathit{nav}_1 \rangle = \langle \mathit{nav}_2 \rangle$ is a boolean term,
3. negation and conjunction of boolean terms yields boolean terms, and
4. nothing else is a boolean term.

A functional term or expression is defined by the grammar

$$\begin{aligned} \mathit{expr} ::= p \mid \langle \mathit{nav} \rangle \mid \langle \mathit{nav} \rangle \rightarrowtail x \mid \langle \mathit{nav} \rangle \rightarrowtail x' \\ \mid f(\langle \mathit{expr}_1 \rangle, \dots, \langle \mathit{expr}_n \rangle) \end{aligned} \tag{9.8}$$

that is,

1. an event parameter p of type τ is a well-typed expression of type τ,
2. a (primed or unprimed) navigation expression of type τ_C is a well-typed expression of type τ_C,
3. if $\langle \mathit{nav} \rangle$ is a navigation expression of type τ_C and x is a variable of class C of type τ, then $\langle \mathit{nav} \rangle \rightarrowtail x$ (unprimed or primed) is a well-typed expression of type τ,
4. if $\langle \mathit{expr}_i \rangle$ is a well-typed expression of type τ_i, $1 \leq i \leq n$, and f is a function symbol (from a signature!) of type $\tau_1, \dots, \tau_n \to \tau$, then $f(\langle \mathit{expr}_1 \rangle, \dots, \langle \mathit{expr}_n \rangle)$ is a well-typed expression of type τ, and
5. nothing else is an expression.

An expression is called *(purely) unprimed* if none of the primed variants occurs, *(purely) primed* if all variables and link names appear primed, and *mixed primed/unprimed* otherwise.

A navigation expression is defined by the grammar

$$\mathit{nav} ::= p \mid \mathit{this}_C \mid \langle \mathit{nav} \rangle \rightarrowtail \lambda \mid \langle \mathit{nav} \rangle \rightarrowtail \lambda' \tag{9.9}$$

that is,

1. an event parameter p of an object type τ_C is a well-typed navigation expression of type τ_C,
2. this_C is a well-typed navigation expression of type τ_C,
3. if $\langle \mathit{nav} \rangle$ is a well-typed navigation expression of type τ_C and λ is a link name of class C and of type τ_D, i.e. $\lambda \in \Lambda_C$, then $\langle \mathit{nav} \rangle \rightarrowtail \lambda$ (unprimed or primed) is a well-typed navigation expression of type τ_D, and
4. nothing else is a well-typed navigation expression

Note that principally, `new` C can be considered to be a navigation expression of type τ_C. We prefer to consider `new` C only as the right-hand side of an assignment actions (cf. (9.17) below). Thereby we avoid to treat temporary objects in the semantics, which is principally possible but not having it isn't restricting the generality of our discussion because one can always introduce additional links, assign them a newly created object, and use the additional link in a particular expression.

In a compound navigation expression *nav* of the form $\lambda \rightarrowtail nav_0$, the sub-expression λ is called the *root* of *nav*. The root of non-compound navigation expressions like p and $this_C$ is the navigation expression itself,

Actions are defined by the grammar

$$\begin{aligned} act ::= {} & \texttt{skip} && (9.10)\\ & \mid \langle nav\rangle \rightarrowtail x' \texttt{ := } \langle expr\rangle && (9.11)\\ & \mid \langle nav\rangle \rightarrowtail \lambda' \texttt{ := } \langle expr\rangle && (9.12)\\ & \mid \langle nav\rangle \rightarrowtail \lambda' \texttt{ := new } C && (9.13)\\ & \mid \langle nav\rangle \,\texttt{!}\, E(\langle expr_1\rangle, \dots, \langle expr_n\rangle) && (9.14)\\ & \mid \texttt{delete } \langle nav\rangle && (9.15)\\ & \mid \texttt{if } \langle cond\rangle \texttt{ then } \langle act_1\rangle\texttt{; else } \langle act_2\rangle\texttt{; fi} && (9.16)\\ & \mid \langle nav_1\rangle; \langle nav_2\rangle && (9.17) \end{aligned}$$

where x is a variable but not x_{st}, λ a link name but not *this*, and E is an event, that is,

1. `skip` is a well-formed action,

2. if $\langle nav\rangle$ is a navigation expression of type τ_C and x or λ are a variable or a link name in C of type τ and if $\langle expr\rangle$ is an expression or creation of the same type, then assignment is a well-formed action,

3. if $\langle nav\rangle$ is a navigation expression, E is an event with parameters of types $\tau_1, \dots, \tau_n$, and if $\langle expr_i\rangle$, $1 \leq i \leq n$, are expressions of type τ_i, then event sending is a well-formed action,

4. if $\langle nav\rangle$ is a navigation expression, then object destruction is a well-formed action,

5. if $\langle cond\rangle$ is a condition, then conditional execution of actions is a well-formed action,

6. sequential composition is a well-formed action, and

7. nothing else is an action.

A transition label is well-formed only

1. if all parameter names occurring in expressions, either in conditions or in actions, appear in the event part *ev* of the label, and

2. if each auxiliary variable or link from X_A and Λ_I is assigned a value before its first use and always appears primed in expressions, that is, if an auxiliary variable or link appears in navigation expression *nav*, then *nav* appears only as *nav'* in expressions

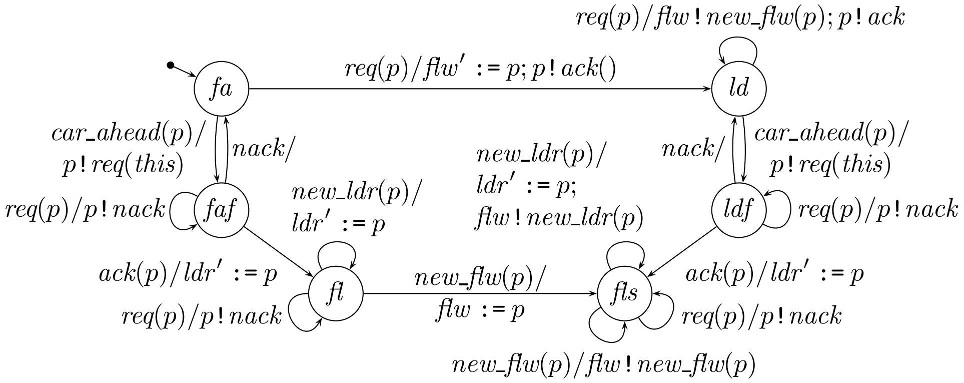


Figure 9.2.: **Platoon Merge.** Graphical representation of states and transitions.

We may use the following abbreviations:

$$\begin{aligned} [\langle cond\rangle]/\langle act_1\rangle;\ldots;\langle act_n\rangle \quad &\text{for} \quad \varepsilon\,[\langle cond\rangle]/\langle act_1\rangle;\ldots;\langle act_n\rangle \qquad (9.18)\\ \langle ev\rangle/\langle act_1\rangle;\ldots;\langle act_n\rangle \quad &\text{for} \quad \langle ev\rangle\,[1]/\langle act_1\rangle;\ldots;\langle act_n\rangle \qquad (9.19)\\ /\langle act_1\rangle;\ldots;\langle act_n\rangle \quad &\text{for} \quad \varepsilon\,[1]/\langle act_1\rangle;\ldots;\langle act_n\rangle \qquad (9.20) \end{aligned}$$

Note that terms are principally similar to the logic, but missing, for example, life-cycle queries. The same applies to Link expressions. In general, we've got to consider multiset arithmetics (cf. Chapter 4) with a select operator, but we'll focus on single-link case shortly.

Example: HLL Model of Car Platoon Merge

Example 9.1.1 (Car Platooning Merge)**.** *The merge protocol of the car platooning case study can be modelled directly as an HLL model as follows.*

To represent cars, there is a class C with no local variables in addition to x_{st} and with link names $\Lambda = \{\mathit{flw}, \mathit{ldr}\}$ to refer to the immediate follower and the leader in addition to this.

The set of states, initial states, and the transitions are given graphically by Figure 9.2. It reads as follows. States are labelled circles, i.e.

$$S = \{\mathit{fa}, \mathit{faf}, \mathit{ld}, \mathit{ldf}, \mathit{fl}, \mathit{fls}\}, \tag{9.21}$$

of which only fa is initial as indicated by the ingoing unconnected arrow. Transitions are indicated by labelled arrows between states.

The set of events in the system can be read from Figure 9.2 as

$$\mathscr{E} = \{\mathit{ack}, \mathit{car\_ahead}, \mathit{nack}, \mathit{new\_flw}, \mathit{new\_ldr}, \mathit{req}\}. \tag{9.22}$$

In order to create cars and notify them of each other's existence, we'd add a class E with only a single state with self-loops to create cars, and a single input link employed in sending car_ahead events. We'll study this in more detail in Chapter 10. ♢

Single Link HLL

In the general case below, link navigation is understood as multi-set comprehension, that is in an expression like

$$\lambda_1 \to \lambda_2 \to x := 0 \tag{9.23}$$

the prefix selects any number of individuals whose local state is then changed to have $x = 0$.

To limit it to single objects, there may be a select operator in the considered signature as discussed in Chapter 4; then

$$*(\lambda_1 \to \lambda_2) \to x := 0 \tag{9.24}$$

chooses a single individual to be updated.

In later sections on symmetry and DTR, for simplicity, we'll assume a single link setting which very closely resembles typical pointer programs, that is

- links are either assigned the empty set, or another link, or a set of size one (for example comprehended from an identity received via an event), and
- there is implicit selection such that a navigation over NULL turns the whole thing into undefined, and leads into the sink state.

This can be checked syntactically by typing rules.

Note that Example 9.1.1 is also an example for a single link HLL model in the sense of Section 9.1.1.

9.1.2. Semantics

The semantics of an HLL model

$$\mathscr{M} = (\mathscr{T}, \mathscr{E}, \mathscr{C}, cond_0) \tag{9.25}$$

is a typed (or: many-sorted) (Σ, Λ) ETTS over *Id*, i.e.

$$\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}) = (S, S_0, R, \mathscr{L}, e) \tag{9.26}$$

where ι is an interpretation of function symbols, $(\mathcal{E}, \oplus, \ominus)$ an *ether*, $\mathcal{S}$ a *scheduler*, and $\mathcal{O}$ an *input and creation oracle*, the latter are formally introduced in Section 9.1.2 below.

We define $\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})$ stepwise. Firstly, we construct the sets of local states Σ and link names Λ from the variables and local states and the link names of classes, in addition, the local state obtains an element of the ether to model event-based communication.

Then choosing sets of identities (with equation) for each class gives rise to the labelling domain $\mathscr{D}$, the set of many-sorted (Σ, Λ) topologies over *Id*. We obtain $S(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ by simply identifying states with the labelling domain, that is, we set $S = \mathscr{D}$ and $\mathscr{L} = id_S$, this is the subject of the paragraph on topologies and states of an HLL model below.

In the subsequent paragraph, we define the transition relation $R(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ and the evolution function. The basic idea is to assume a scheduler $\mathcal{S}$, which chooses for each state a list of pairs of individuals and local transitions that *may* be executed because the following requirements are satisfied

- the individual is in the right local state,
- the event required by the transition is ready to be consumed in the individual's ether,
- the guarding condition holds.

Then two global (or system) states s and s' are in transition relation if and only if s' is the result of applying the transition programs of the transitions from a scheduled list one after another to the corresponding individuals from the list. Thereby we can have true concurrency (yet we'll see in Section 9.3 on DTR that is makes matters complicated) without write/write races because the lists provided by the scheduler impose an order.

Topologies and States of an HLL Model

Assume we're given a domain $\mathcal{D}(\tau)$ for each basic type $\tau \in \mathscr{T}$ and a domain $\mathcal{D}(\tau_C)$ with an equality function eq_{Id}, if not otherwise noted, $\{C\} \times \mathbb{N}_0$ with normal equality.

The domain of the event type $\tau_{\mathscr{E}}$ is

$$\tau_{\mathscr{E}} = \bigcup_{E \in \mathscr{E}} \mathcal{D}(\tau_E) \tag{9.27}$$

where

$$\tau_E = \{E\} \times \mathcal{D}(\tau_1) \times \cdots \times \mathcal{D}(\tau_n) \tag{9.28}$$

if E has parameters $p_1, \ldots, p_n$ of types $\tau_1, \ldots, \tau_n$, $n \in \mathbb{N}_0$.

Then for each class $C \in \mathscr{C}$ with non-input and non-auxiliary local variables

$$X_C = \{x_1, \ldots, x_n, x_{C,st}\}, \tag{9.29}$$

we set

$$\Sigma_C := \mathcal{D}(\tau(x_1)) \times \cdots \times \mathcal{D}(\tau(x_n)) \times \mathcal{D}(\tau_{C,st}) \times \mathcal{E} \tag{9.30}$$

where the domain of the local state type $\tau_{C,st} \in \mathscr{T}$ is $\mathcal{D}(\tau_{C,st}) := S_C$ and $\mathcal{E}$ is the *ether* type (see below).

Let $\mathscr{C} = \{C_1, \ldots, C_n\}$, then

$$\Sigma := \Sigma_{C_1} \,\dot\cup\, \ldots \,\dot\cup\, \Sigma_{C_n} \tag{9.31}$$

$$\Lambda := \Lambda_{C_1} \,\dot\cup\, \ldots \,\dot\cup\, \Lambda_{C_n} \tag{9.32}$$

and

$$Id := Id_{C_1} \,\dot\cup\, \ldots \,\dot\cup\, Id_{C_n} \tag{9.33}$$

where $Id_{C_i} = \mathcal{D}(\tau_{C_i})$. Then the set of states $S(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ is the set of (Σ, Λ)-topologies over Id plus the designated *failure state* 0.

We use the following notation. Given a state $s \in S$ and an individual $u \in U^{\circledcirc}(s)$, we use

$$s(u).x \tag{9.34}$$

to denote the value of the x component of the local state of u in s, that is, of $\sigma(s)(u)$, $s(u).\epsilon$ to denote the ether component, and $s(u).\lambda$ to denote the value of the navigation function $\rightarrowtail_\lambda$ in s, i.e.

$$s(u).\lambda := \rightarrowtail_\lambda(u). \tag{9.35}$$

The initial states are those topologies satisfying the initial state condition $cond_0$ as defined in the following Section 9.1.2, i.e.

$$S_0(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})) = \{s \in (\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})) \mid \iota[\![cond_0]\!](s, id, \emptyset) = 1\} \tag{9.36}$$

where $id \in Id$ is any identity.[2]

Transitions and Evolution Function

We've briefly sketched the idea of the semantics definition above. To elaborate it a bit, recall that an HLL transition program has the form

$$\langle ev \rangle \; [\langle cond \rangle] / \langle act_1 \rangle; \ldots; \langle act_n \rangle \tag{9.37}$$

What we're after is to say that two global states s and s' are in transition relation iff

1. the scheduler $\mathcal{S}$ can choose a finite set of individual/transition program pairs, that is,
$$\mathcal{S}(s) = u_1/r_1, \ldots, u_n/r_n \tag{9.38}$$
where $u_i \in U^{\circledcirc}(s)$ is an alive object of class C and r_i is a transition of C,

2. the (local) state of u_i in s corresponds to the source state of r_i, that is, to the value of u_i's variable x_{st} in s, and the (local) state of u_i in s' corresponds to the destination state of r_i, i.e.
$$(s(u_i).x_{st}, \ell, s'(u_i).x_{st}) \in R_C, \tag{9.39}$$

3. there is an event ev in the ether of u_i,

4. the conditions $cond$ of the r_i hold for u_i in s, and

5. the local state of u_i in s' is the result of applying the action list of r_i to s.

[2] It's independent from id because *this* doesn't occur at the root of navigation expressions by definition.

To this end, we firstly define ethers and the operations for inserting and removing events. Intuitively, an ether is an abstract variant of a communication medium which may be instantiated by a single event queue, but as well by a set, or a collection of lossy priority queues.

Secondly, we give meaning to expressions, that is, explain how to evaluate an expression in a given global state wrt. to a particular individual. Correspondingly, we introduce a notion of applying transition programs to a global state. Together with a formal definition of a scheduler, everything is prepared to define the transition relation between global states following the intuition given above.

Ether We use something we call an *ether* as an abstract model for any kind of (asynchronous) communication. An ether $(\mathcal{E}, \oplus, \ominus)$ is a set $\mathcal{E}$ with two operations

$$\oplus : \mathcal{E} \times Id \times \mathcal{D}(\tau_{\mathscr{E}}) \times Id \to \mathcal{E} \tag{9.40}$$

$$\ominus : \mathcal{E} \times Id \times \mathcal{D}(\tau_{\mathscr{E}}) \times Id \to \mathcal{E} \tag{9.41}$$

which can be read as “insert” and “consume”.

Given an ether ϵ, an event $E \in \mathscr{E}$ with parameters $p_1, \dots, p_n$ of types $\tau_1, \dots, \tau_n$, $n \in \mathbb{N}_0$ and semantical values $d_i \in \mathcal{D}(\tau_i)$, $1 \leq i \leq n$, we use

$$\epsilon \oplus (u_1, E(d_1, \dots, d_n), u_2) \ (= \oplus(\epsilon, u_1, (E(d_1, \dots, d_n), u_2))) \tag{9.42}$$

to denote the insertion of the event E with parameter values $d_1, \dots, d_n$ into the (end of the) ether, and

$$\epsilon \ominus (u_1, E(d_1, \dots, d_n), u_2) \ (= \ominus(\epsilon, u_1, (E(d_1, \dots, d_n), u_2))) \tag{9.43}$$

to denote the removal of the event E with parameters $d_1, \dots, d_n$ from the (front of the) ether, if this event *is* in front of the ether.

The former can be read as

> *individual u_1 sends event E with parameters $\vec{d}$ to u_2 into ether ϵ*

and the latter as

> *individual u_2 consumes E with parameters $\vec{d}$ from u_1 from ether ϵ.*

We write

$$\epsilon = \epsilon_0.u_1 \xrightarrow{E(d_1,\dots,d_n)} u_2 \tag{9.44}$$

to denote that ϵ is in a configuration where we may consume E with parameters $d_i \in \mathcal{D}(\tau_i)$, where τ_i is the type of the i-th parameter, that is, if

$$\epsilon_0 = \epsilon \ominus (u_1, E(d_1, \dots, d_n), u_2). \tag{9.45}$$

The abbreviation

$$\epsilon = \epsilon_0.E(d_1, \dots, d_n) \tag{9.46}$$

is used if the receiving individual u_2 is clear by context, for instance, because ϵ is local to it, and if the identity of the sending individual u_1 is not relevant, i.e. if

$$\exists\, u_1 \in \mathit{Id} : \epsilon(u_2) = \epsilon_0.u_1 \xrightarrow{E(d_1,\dots,d_n)} u_2. \tag{9.47}$$

Note that the terms "end" and "front" best match if the ether is instantiated with a single event queue, possibly with priorities. But it may as well be instantiated with multiple queues, possibly one per sender, then (9.46) may hold true for multiple events at the same time.

Furthermore, an ether may simply be a set, thus it doesn't necessarily preserve the order of insertion, and it may well be lossy. We'll see that all of these aspects are orthogonal to our discussion as long as the ether is *symmetric in identities*, that is, as long as it doesn't lose messages depending on identities, or gives priority to events sent by a certain sender. The reason we're passing the sender and receiver identity is to leave open all choices, there could, for instance, be a single ether for the whole system.

Condition Evaluation: Terms and Expressions A condition *cond* is a boolean term. The evaluation of a condition for an individual u in system states s_0 and s under assignment θ, denoted by

$$\iota[\![\mathit{cond}]\!](s_0, s, u, \theta) \in \mathbb{B}_3 \tag{9.48}$$

is defined inductively as follows, assuming an interpretation ι of function symbols.

The intuition of having two states s_0 and s is that the former is used to evaluate unprimed expressions and the second one to evaluate primed ones. In case of purely primed or unprimed expressions, one of the two can be omitted. The definition follows the same philosophy than the one for EvoCTL* in Chapter 4.

1. $\iota[\![1]\!](s_0, s, u, \theta) \quad = 1$
2. $\iota[\![\mathit{nav}_1 = \mathit{nav}_2]\!](s_0, s, u, \theta) \quad = \mathit{eq}_{\mathit{Id}}(\iota[\![\mathit{nav}_1]\!](s_0, s, u, \theta), \iota[\![\mathit{nav}_2]\!](s_0, s, u, \theta))$ if both arguments are defined, $1/2$ otherwise
3. $\iota[\![\neg \mathit{term}]\!](s_0, s, u, \theta) \quad = 1 - \iota[\![\mathit{term}]\!](s_0, s, u, \theta)$
4. $\iota[\![\mathit{term}_1 \wedge \mathit{term}_2]\!](s_0, s, u, \theta) \quad = \min\{\iota[\![\mathit{term}_1]\!](s_0, s, u, \theta), \iota[\![\mathit{term}_2]\!](s_0, s, u, \theta)\}$

In addition, we use the common abbreviations for disjunction, implication, etc.

It extends naturally to the first-order extension from (9.2) by

$$\iota[\![\forall\, p : \tau \,.\, \mathit{init}]\!](s_0, s, u, \theta) = \min\{\iota[\![\mathit{init}]\!](s_0, s, u, \theta[p \mapsto \mathit{id}]) \mid \mathit{id} \in \mathit{Id}\} \tag{9.49}$$

$$\iota[\![\exists\, p : \tau \,.\, \mathit{init}]\!](s_0, s, u, \theta) = \max\{\iota[\![\mathit{init}]\!](s_0, s, u, \theta[p \mapsto \mathit{id}]) \mid \mathit{id} \in \mathit{Id}\}. \tag{9.50}$$

The semantics of functional expressions is inductively defined as follows.

1. $\iota[\![p]\!](s_0, s, u, \theta) \quad = \theta(p)$

2. $\iota[\![nav]\!](s_0, s, u, \theta) = \iota[\![nav]\!](s_0, s, u, \theta)$
3. $\iota[\![nav \rightarrowtail x]\!](s_0, s, u, \theta) = s(\iota[\![nav]\!](s_0, s, u, \theta)).x$
4. $\iota[\![nav \rightarrowtail x']\!](s_0, s, u, \theta) = s_0(\iota[\![nav]\!](s_0, s, u, \theta)).x$
5. $\iota[\![f(expr_1, \dots, expr_n)]\!](s_0, s, u, \theta) = \iota(f)(\iota[\![expr_1]\!](s_0, s, u, \theta), \dots, \iota[\![expr_n]\!](s_0, s, u, \theta))$
   if all $\iota[\![expr_i]\!](s_0, s, u, \theta)$, $1 \leq i \leq n$, are defined, otherwise undefined or $1/2$, if τ is boolean

The interpretation of navigation expressions operates on only a single system state, being primed or unprimed in the enclosing (functional) expression determines which state is used.

1. $\iota[\![p]\!](s_0, s, u, \theta) = \theta(p)$
2. $\iota[\![this_C]\!](s_0, s, u, \theta) = u,$ if $u \in U^{\circledcirc}(s)$ and undefined otherwise
3. $\iota[\![nav \rightarrowtail \lambda]\!](s_0, s, u, \theta) = s_0(\iota[\![nav]\!](s_0, s, u, \theta)).\lambda,$
   if $\iota[\![nav]\!](s_0, s, u, \theta) \in U^{\circledcirc}(s_0)$, undefined otherwise;
4. $\iota[\![nav \rightarrowtail \lambda']\!](s_0, s, u, \theta) = s(\iota[\![nav]\!](s_0, s, u, \theta)).\lambda,$
   if $\iota[\![nav]\!](s_0, s, u, \theta) \in U^{\circledcirc}(s)$, undefined otherwise;

Note that this definition does *the right thing*$^{\text{TM}}$ for *this* in the middle of definitions because this link is initialised for each individual to point back to itself and then never changed again. Furthermore, *this* (intentionally) turns undefined if *this* is deleted within a sequence of actions.

The semantics tolerates (to some amount) undefined valuations because the boolean value finally becomes $1/2$. For example, the expression

$$(\mathit{this} \rightarrowtail \lambda \neq \emptyset) \rightarrow (\mathit{this} \rightarrowtail \lambda \rightarrowtail x = 0) \tag{9.51}$$

is guaranteed to evaluate definite even if λ points into the void, and thus turns the right hand side of the implication indefinite.

The comparison for equality in (9.51) is a function from the signature, not the "=" from the syntax. The latter only operates on links and is tightly connected to the eq_{Id} function on *Id*.

The semantics of mixed primed/unprimed navigation expressions is a bit tricky but helpful at least for the definition of syntactic DTR. For example

$$\underbrace{\underbrace{\underbrace{\overbrace{\mathit{this} \rightarrowtail \lambda'}^{u_2=s(u).\lambda} \rightarrowtail \mu}_{u_1=s_0(u_2).\mu} \rightarrowtail \lambda'}_{u_0=s(u_1).\lambda} \rightarrowtail x}_{s_0(u_0).x} \tag{9.52}$$

denotes the pre-value of variable x in the individual denoted by the navigation expression, which, read out loud, is

- the individual u_0 denoted by the λ link of u_1 in current state, where
- the individual u_1 is denoted by the μ link of u_2 in the pre-state, where
- the individual u_2 is denoted by the λ link of *this* in the current state.

To avoid confusion, primed navigation expressions should be used rarely and with care, at best only at the left-hand side of assignments.

Creation and Input Oracle A creation oracle provides, for each state from a given set of states S and for each occurrence of a creation expression, a singleton set of identities to choose from for creation, that is, a function

$$\mathcal{O} : S \times \langle act \rangle \to \mathfrak{P}(Id) \tag{9.53}$$

and (admittedly awkwardly overloaded)

$$\mathcal{O} : S \to \mathfrak{P}(Id \times (\Lambda_I \cup X_I) \to \mathcal{D}) \tag{9.54}$$

That is, for each state $s \in S$, a set of functions mapping alive individuals and input variables to values, i.e. a set of functions of the form

$$\begin{aligned} \{(u, x) \mapsto d \mid u \in U^{\circledcirc}(s), x \in X_{I,C}, d \in \mathcal{D}(\tau)\} \\ \cup \{(u, \lambda) \mapsto d \mid u \in U^{\circledcirc}(s), \lambda \in \Lambda_{I,C}, d \in \mathcal{D}(\tau)\} \end{aligned} \tag{9.55}$$

where C is the class of u and τ denotes the type of x and λ.

Note that inputs of a type with finite domain can be simulated by non-determinism. If the domain is infinite, which is the general case for links, then HLL with inputs is strictly more expressive than HLL without.

Transition Program Application The third part in the transition labelling language as introduced in Section 9.1.1 are meant as actions, or modifications of objects' states. Semantically, these modifications become *applications* of transition program P to (global) system state s for individual u under parameter assignment θ and with input and creation oracle $\mathcal{O}$, denoted by

$$\iota[\![P]\!](s_0, u, \theta, \mathcal{O}) : S \to S \tag{9.56}$$

where s_0 is original source state in which, for example, conditions are evaluated, while s could've been updated by previous actions in a sequence.

They're defined inductively as follows.

- $\iota[\![\texttt{skip}]\!](s_0, u, \theta, \mathcal{O})(s) = s$

- $\iota[\![nav \rightarrowtail x' \;\texttt{:=}\; expr]\!](s_0, u, \theta, \mathcal{O})(s) = s'$,

  where s' differs from s only in that the x component of the local state of *each* individual denoted by *nav* has the value of *expr*, i.e.

$$s'(\iota[\![nav]\!](s_0, s, u, \theta)).x = \iota[\![expr]\!](s_0, s, u, \theta) \tag{9.57}$$

  if $u \in U^{\circledcirc}(s)$ and both, the navigation and the functional expression, are defined, undefined otherwise,

- $\iota[\![nav \rightarrowtail \lambda' \;\texttt{:=}\; expr]\!](s_0, u, \theta, \mathcal{O})(s) = s'$,

  where

$$s'(u_0).\lambda = \iota[\![expr]\!](s_0, s, u, \theta) \tag{9.58}$$

  with $u_0 \in \iota[\![nav]\!](s_0, s, u, \theta)$, and undefined if the navigation or the functional expression are undefined,

- $\iota[\![\underbrace{nav \rightarrowtail \lambda' \;\texttt{:=}\; \texttt{new}\; C}_{=:P}]\!](s_0, u, \theta, \mathcal{O})(s) = s'$,

  similar to the previous case yet instead of assigning the value of an expression, we assign an individual u as chosen by the creation scheduler for this particular action, that is, from the set $\mathcal{O}(s, P)$ setting $\sigma(u).\mathit{this} = u$ and $\sigma(u).x_{st} \in S_{0_C}$; further initialisation of the local state may take place on the first transition of u or by the creating individual,

- $\iota[\![nav \texttt{!} E(expr_1, \ldots, expr_n)]\!](s_0, u, \theta, \mathcal{O})(s) = s'$,

  where

$$s'(u_0).\epsilon = s(u_0).\epsilon \oplus (u, E(d_1, \ldots, d_n), u_0) \tag{9.59}$$

  where $u_0 := \iota[\![nav]\!](s_0, s, u, \theta)$ and $d_i := \iota[\![expr_1]\!](s_0, s, u, \theta)$, $1 \leq i \leq n$, and undefined if any of the navigation or the functional expressions are undefined,

- $\iota[\![\texttt{delete}\; nav]\!](s_0, u, \theta, \mathcal{O})(s) = s'$

  such that the only difference between s and s' is that the individual denoted by $\iota[\![nav]\!](s_0, s, u, \theta)$ is not alive [3] in s', and undefined if the valuation of the navigation expressions is undefined,

- $\iota[\![\texttt{if}\; cond\; \texttt{then}\; act_1\; \texttt{else}\; act_2\; \texttt{fi}]\!](s_0, u, \theta, \mathcal{O})(s) = s'$,

  where if $d := \iota[\![cond]\!](s_0, s, u, \theta) = 1$, then $s' = \iota[\![act_1]\!](s_0, u, \theta, \mathcal{O})(s)$, if $d = 0$, then $s' = \iota[\![act_2]\!](s_0, u, \theta, \mathcal{O})(s)$, and undefined otherwise,

[3] This is the easy way, actually requiring that a deleted object is gone for at least one step. The ETTS model is (intentionally) more powerful. By the evolution relation it can distinguish immediate re-uses (cf. Chapter 9). It is also possible to incorporate this behaviour into the HLL by keeping track of who's deleted and to allow the creation scheduler to also chose from these ones, but it would only complicate the definitions without contributing anything significant.

- $\iota[\![act_1; act_2]\!](s_0, u, \theta, \mathcal{O})(s) = \iota[\![act_2]\!](s_0, u, \theta, \mathcal{O})(\iota[\![act_1]\!](s_0, u, \theta, \mathcal{O})(s))$ if the intermediate step $\iota[\![act_1]\!](s_0, u, \theta, \mathcal{O})(s)$ is defined, and undefined otherwise.

Note that we don't have loops in transition programs in order to be sure that each transition program terminates. Further note that transition programs are deterministic, non-determinism is only on the level of transitions.

Scheduler Given a system state s from a set of states S, a scheduler basically chooses individuals which are ready to take a transition in s.

Formally, a scheduler is a function mapping system states to a finite, possibly empty set of finite sequences of pairs of identities and transitions, that is,

$$\mathcal{S} : S \to \mathfrak{P}((Id \times R_{\mathscr{C}})^+), \tag{9.60}$$

yielding sequences of the form

$$(u_1, r_{C_1,1}), \ldots, (u_n, r_{C_n,n}), n \in \mathbb{N}_0, \tag{9.61}$$

where identities are pairwise different, i.e. $u_i \neq u_j$, $1 \leq i \neq j \leq n$ and if id_i is of class C, then r_i is a transition of this class, i.e. $r_i \in R_C$.

A sequence

$$(u_1, r_{C_1,1}), \ldots, (u_n, r_{C_n,n}) \tag{9.62}$$

provided by a scheduler for state s_0 is called *ready* in state s_0 if and only if

1. the identity id_i is alive in s_0, i.e. $id_i \in U^{\circledcirc}(s_0)$,
2. if r_i is of the form
$$(s_{C_i}, ev_i\,[cond_i]/act_i, s'_{C_i}) \tag{9.63}$$
then
   - u_i is in local state s_{C_i} in s_0, i.e.
   $$s_0(u_i).x_{st} = s_{C_i}, \tag{9.64}$$
   - the required event is ready to be consumed in the ether of u_i, i.e., either $ev_i \neq \varepsilon$ or $ev_i = E(p_1, \ldots, p_m)$ and
   $$s_0(u_i).\epsilon = \epsilon_0.E(d_1, \ldots, d_m), \tag{9.65}$$
   - the input oracle $\mathcal{O}$ provides an input valuation $o \in \mathcal{O}(s_0)$ such that the guarding conditions hold in $o(s_0)$, i.e.
   $$\iota[\![cond_i]\!](o(s_0), o(s_0), u_i, \theta) = 1, 1 \leq i \leq n, \tag{9.66}$$
   with $\theta = \{p_1 \mapsto d_1, \ldots, p_m \mapsto d_m\}$ and where $o(s_0)$ coincides with s_0 outside the inputs and for the inputs with o, i.e.
   $$\begin{aligned}&\{(u, x) \mapsto o(s_0)(u).x \mid u \in Id_C, x \in X_{I,C}\}\\ &\quad \cup \{(u, x) \mapsto o(s_0)(u).\lambda \mid u \in Id_C, \lambda \in \Lambda_{I,C}\} = o.\end{aligned} \tag{9.67}$$

Transition Relation and Evolution Function Given a scheduler $\mathcal{S}$ and an input and creation oracle $\mathcal{O}$. Two system states $s, s' \in S(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ are in the transition relation $R(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ if and only if

1. there is a sequence of individuals and transitions in s according to the scheduler, i.e.
$$u_1/r_{C_1,1}, \ldots, u_n/r_{C_n,n} \in \mathcal{S}(s) \tag{9.68}$$
with
$$r_{C_i,i} = (s_{C_i}, ev_i\,[cond_i]/P_i, s'_{C_i}), \tag{9.69}$$
which is ready in s with input oracle element $o \in \mathcal{O}$ and

2. s' is the result of subsequently applying the transition programs to $o(s)$ and consuming the ready-to-consume events, i.e. s' is identical to
$$s'_0 := \iota[\![P_n]\!](o(s), u_n, \theta_n, \mathcal{O})(\ldots \iota[\![P_1]\!](o(s), u_1, \theta_1, \mathcal{O})(o(s)) \ldots) \tag{9.70}$$
where θ_i maps parameters to parameter values similar to the assignments in the previous section on schedulers if all applications of transition programs are defined, with the only exceptions that
$$s'(u_i).\epsilon = s'_0(u_i).\epsilon \ominus ev_i(\vec{d}), 1 \leq i \leq n \tag{9.71}$$
and
$$s'(u_i).x_{st} = s'_{C_i}, 1 \leq i \leq n \tag{9.72}$$
if u_i is still alive in s'_0,

or if there is no ready sequence of ready-to-execute individuals and transitions in s according to the scheduler and s' coincides with s.

In addition, there is a transition from s to the failure state 0 if and only if one of the condition evaluations or transition program applications are undefined, for instance navigation via a NULL pointer if such a thing is considered in the HLL instantiation.

The transition relation is denoted by $o(s) \xrightarrow{\{u_1/r_1,\ldots,u_n/r_n\}} s'$ when we want to indicate by *which* local transitions and which oracle value the global transition is justified.

For each transition as defined above the evolution function is simply the identity on the set of identities that are alive in both states, i.e. $e\langle r\rangle = id_{U^{\circledcirc}(s) \cap U^{\circledcirc}(s')}$. It's that simple because we've taken the easy way with destruction. It's a consequence of the semantics of the delete operation that an object is non-alive for at least one step, thus we don't have to treat the more complex case of immediate resurrection. Thereby evolution annotation is also immediately consistent.

This completes the component-wise definition of $\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}))$ in terms of an ether, a scheduler, and a creation scheduler.

9.1.3. Interleaving vs. Concurrent Semantics

We call $\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})$ an *interleaving* semantics of $\mathscr{M}$ if and only if the scheduler $\mathcal{S}$ selects for each state sequences of at most length one. Otherwise the semantics is called *concurrent.*

A simple example for a concurrent scheduler is the one who offers all alive individuals with all enabled transitions, that is, $\mathcal{S}(s)$ is the set of pairs u/r such that $u \in U^{\circledcirc}(s)$ and the event required by r is available in the ether and the condition holds.

9.1.4. Relation to other Formalisms

First of all, note that the expressive power of HLL is not sufficient to cover all ETTS, that is, there are ETTS M that are not the semantics of a finite HLL model.

For example, if each of the infinitely many individuals in M behaves differently. As HLL may only have a finite number of classes with finitely many states and transitions, such a M is not the ETTS of any HLL model (cf. Figure 9.1).

This limitation is not critical for our purposes because, in general, QR doesn't apply to these ETTS either because it is then not sufficient to consider a finite set of representative cases. The class covered by HLL instead is the one, to which QR applies, under premises we'll see in Section 9.2.

Parameterised Systems

With some overlap with the discussions from Chapter 3 and 4 note that HLL is similar to parameterised systems in the sense that a number of processes (here: individuals) executes one of finitely many finite programs, in most cases independent from the identity of the process (or individual). Then processes are clearly interchangeable with each other.

The differences are the following. Firstly, parameterised systems (cf. Chapter 3) have infinitely many *finite* instances, that is, in each instance there is a finite bound on the number of processes. In HLL we don't have this bound, topologies in the resulting ETTS may grow unboundedly.

Secondly, they typically don't consider creation and destruction of processes. One benefit is that first-order temporal logics for classical parameterised systems needn't consider premature disappearance (cf. Chapter 4).

Furthermore, classical parameterised systems don't have notions of links, some variants don't even provide means to store identities because the behaviour of processes depends on iterations over the global shared memory, that is, by considering either none or all of the other processes; in HLL, there is selective consideration of others via links.

Graph Transformation Systems

An HLL model could also naturally obtain a semantics via a graph transformation system as the states are topologies, which are graphs. In addition to the point that we established the theory of DTR and QR on ETTS, there are two issues. One is that graph transformation systems typically don't trace the identity of nodes, thus a priori we don't have an equivalent of the evolution function. A second one is that data, like local states and the ether, require the less elementary attributed graph transformation systems.

Object-Orientation

The relation from HLL to object oriented programming or modelling languages is different than the relation to parameterised systems or graph transformation systems, because the latter can be seen to belong to a similar conceptual level as HLL while we locate object oriented languages on a higher level.

In Chapter 10 we'll see how to embed a significant core of the UML into HLL and discuss this relation in more detail. In brief, what object oriented languages add are information hiding, inheritance, and *virtual* function calls (or: late binding). An example for the former are attributes like *private* or *public* local variables, which is basically syntactic sugar and could easily be added to HLL.

Concepts of inheritance aid structuring and re-use. They can also be seen as syntactic sugar as it can be encoded in HLL by splitting an object of a class D, which derives from C, as two objects with a particular connection (cf. Chapter 10). Virtual function calls can also be encoded in HLL as discussed in Chapter 10.

9.2. Symmetry for HLL

From Chapter 7, we have a definition of an ETTS being symmetric in identities. As already noted in Chapter 7, this property is undecidable for infinite ETTS and computationally expensive for finite ones [62]. Thus in order to effectively apply QR, we need other means to establish whether the ETTS of an HLL model $\mathscr{M}$ is symmetric in identities.

A widely used idea is to introduce in a higher-level language a new kind of type called *scalarset* [89] with only a limited set of operations defined for it, such that the resulting transition system is symmetric in the values of this type. In Section 9.2.1, we recall this idea and in Section 9.2.2 list a number of cases where it has been applied, including our own previous work. In Section 9.2.3, we generalise the approach slightly by drawing attention to additional circumstances that are taken for granted in the discussed applications. Namely, the scheduling, the communication medium, and the treatment of inputs and creation.

In Section 9.2.4, we show the generalised property that symmetry of these aspects is sufficient to conclude that the ETTS of an HLL model is symmetric in identities. In typical cases, the extended premises to apply QR are still easily established.

9.2.1. Scalarset Types

Ip & Dill [89] proposed an input language for their model-checker Murphi such that the semantics of each well-formed Murphi program has a symmetric Kripke structure as semantics. Yet they weren't after symmetry in identities but, more general, introduced a new data-type called *scalarset*.

Symmetry is ensured by limiting the set of operations on scalarset types and the possible uses. In a maximally restrictive setting, these following rules ensure these limits.

1. variables (in particular inputs or constants) may be of a scalarset type.
2. Constants obtain an arbitrary value once which is fixed for system execution, (in other words: multiple initial states).
3. Variables of the same scalarset type may be compared for equality.
4. Variables of the same scalarset type may be assigned to each other.
5. Arrays may have a scalarset index type, and scalarset variables may be used to access positions of such arrays.

In addition, one may admit the following rule.

6. Loop variables may range over the domain of a scalarset type, if the outcome of the loop is independent from execution order, a fact for which there are again sufficient syntactic criteria.

Particular instances of such loops are, for instance, the sum of all array fields of an array with scalarset index type.

Checking well-formedness for a Murphi program is then a simple type-checking tasks: check whether only the allowed operations are applied to scalarset variables.

By the following thoughts it is intuitively clear that these operations do not allow to break symmetry, that is, to treat a particular value of a scalarset type different to other values.

Firstly, variables only obtain their values by being inputs or constants or by being a loop variable. Both are symmetric as inputs may assume any legal value and loops are restricted to be independent from execution order, thus the first iteration may start with any value.

Secondly, only variables are compared for equality, there are no literals to denote particular elements of a scalarset type's domain. So it is not permitted to write

$$\texttt{if}\ x = 1\ \texttt{then}\ \ldots \tag{9.73}$$

to treat value "1" specially because x may not be compared to it.

Finally note that the rules above in particular disallow to *cast* a scalarset variable into any other type (and then compare it against literals) and they disallow to perform any kind of arithmetics on scalarset types. A more general and formal discussion follows in Section 9.2.3.

Note that this connection, or this existence of a type-checking procedure, gives rise to two slightly different applications.

The first is the one taken by [89]: Have a language where scalarset types can be declared, know that well-typed programs give rise to a symmetric Kripke structure, and apply the type check to verify that the programmer adhered to its declarations, that she didn't apply forbidden operations to variables of the scalarset type.

The second is the one taken, e.g., by [28, 29], [31], and by us in [178]. Namely, given a program in a language which *doesn't* have a notion of scalarset types, for each variable declaration, *try* to check whether it adheres to the restrictions of scalarset types and thus *could* be safely declared to be a scalarset type.

The second approach possibly *detects* symmetries in a program, while the first application only *confirms* declared symmetry.

Another related approach is *Data Independence* of [183], later generalised in [113]. It applies to programs with certain infinite-domain data type. If variables of this type are only assigned and compared for equality, both in the program and in a requirement, then it is sufficient to consider only a finite domain.

The typical example is a network buffer carrying certain infinite data. As long as the buffer doesn't *interpret* the data and as long as the only impact on the buffer's behaviour is based on equality between two data values. It is sufficient to consider a finite domain large enough to reach all possible equality/inequality pairs of variables.

For example, with two variables, a domain of size two is sufficient. This is similar to QR in that the obtained finite domain is sufficient to have representative cases for all possible cases in the infinite domain. It relates to symmetry detection as the limitation, namely only assignment and comparison for equality, are sub-criteria of scalarsets, in other words, satisfying the Wolper criteria is a pre-requisite for scalarsets [134].

We conjecture that Data Independence in the sense that there exists is a *finite* bisimulation doesn't extend to scalarsets in general because with scalarset-indexed arrays, the number of variables is no longer finite. Yet one may still apply [183] to single classes, for instance, if an HLL class is such a network buffer and buffer values of two different instances are never compared directly, then the domain of the buffer values can be reduced following [183].

9.2.2. Applications of Scalarset Types

This approach, not to detect symmetry in a given Kripke structure or to assume it given, but to have a programming language with symmetric Kripke structures as semantics, has been used in other settings than [89]. It is a building block of [127], which we employ and extend in [49, 48] (cf. [134] for a survey of symmetry detection and declaration in general).

In this section, we briefly discuss these applications of [89]. We'll notice that what they have in common is that the applications are tailored (and formally discussed) in certain, limited settings, like Murphi or Cadence SMV which, for example, have a certain fixed scheduling policy.

In Section 9.2.2, we generalise these results to the just introduced HLL with a *generic* scheduling and *generic* communication means.

In [127], the exact same approach of [89] is also used to ensure symmetry. That is, in particular checking declared symmetry, not detecting symmetry . It adds to the model-checker Cadence SMV a new type declarator to declare scalarset types and checks whether a given Cadence SMV program is well-typed in the above sense.

An original extension in [127] is that they admit scalarset domains to be ordered and to have *some* arithmetic operations on scalarset types, namely computing the predecessor and successor. The semantics of programs in the extended language is then not fully symmetric, that is, invariant under *any* permutation of the domain, but only under permutations which preserve the order, namely circular permutations. Thereby, Cadence SMV is able to model, for example, ring topologies of systems, for instance, a number N of network nodes each connected to its left and right neighbour, and to apply QR. Furthermore there is an example supporting inductive reasoning.

SymmSPIN [19] implements the quotient-based symmetry reduction for SPIN. They apply [89] for detection of symmetry because SPIN's input language Promela doesn't have means to explicitly declare symmetry. The same applies to TopSPIN [59], which supersedes SymmSPIN as it is able to exploit more general symmetries.

In [178], we discuss the model-checker input language SMI [24, 179], which is not tied to a particular model-checker but translates, among others, to VIS and SMV. SMI doesn't provide a scalarset type either, so we declare scalarset types in a separate file and check for adherence. The tools also implements the second application, that is, checks whether some of the existing types could alternatively be declared scalarset types.

9.2.3. Scalarset Types with Communication and Scheduling

The applications of [89] discussed in Section 9.2.2 have in common that if the crucial property, that every well-typed program has a symmetric Kripke structure as semantics, is proven, then the proof is conducted in the limited setting of the considered model-checker.

That is, in the setting of the particular model-checker's semantics of its input language. This approach is natural, and there's nothing wrong with it with the only small exception that it doesn't tell us, what the model-checker's semantics has to provide in order to ensure that the proof succeeds. In other words, the criteria listed in Section 9.2.1 don't apply *universally*, to any language, but only in the context of, for instance, the Murphi model-checker.

This section sets out to answer the question *what* Murphi provides such that, within that setting, the criteria of Section 9.2.1 work as intended. Employing the HLL as introduced above is adequate for this discussion because in the HLL all aspects of the semantics are parameterised, namely the scheduler and the communication medium. The bottom line is that inputs of scalarset type shall symmetrically be set, the scheduler shall treat symmetric states symmetrically, the communication medium shall treat symmetric situations symmetrically, and only particular operations are allowed.

Once identified, it seems of course evident. Our more general look may have applications when studying symmetric data types for other languages than the known ones.

Symmetric Inputs and Creation

An input and creation oracle $\mathcal{O}$ is symmetric in identities if and only if the identity chosen for creation at action *act* in state $p(s)$ is the permutation of the identity chosen in state s, i.e.

$$\forall\, s \in S : \mathcal{O}(p(s), act) = p(\mathcal{O}(s, act)), \tag{9.74}$$

and if the possible valuations of inputs in state $p(s)$ are the permutations of the possible input valuations in s, i.e.

$$\mathcal{O}(p(s)) = p(\mathcal{O}(s)). \tag{9.75}$$

Here, permutation is meant to be applied point-wise, leaving values of non-object types unchanged, i.e.

$$\begin{aligned} &p(\{(u_1, x_1, d_1), (u_2, x_2, d_2), \dots\}) \\ &= \{(p(u_1), x_1, p(d_1)), (p(u_2), x_2, p(d_2)), \dots\}). \end{aligned} \tag{9.76}$$

How do we establish this property? In all of the applications discussed in Section 9.2.2, symmetry of the input oracle is given. Inputs are completely free, that is, any legal value can be chosen in any state.

Creation is different. First of all, most of the tools Section 9.2.2 don't support creation and destruction natively, thus it has to be encoded. The encoding employed in [178], for example, is a symmetric modification on the one from [159] whose creation is *not* symmetric but employs a round-robin strategy with a fixed starting point. Thereby, the first created object obtains the same identity in all system runs, thus one cannot apply Query Reduction. The modification employs inputs to choose identities for creation.

The model-checker SPIN is one exception as it supports process creation natively. Interestingly, a closer investigation shows that the choice of process identities is not (!) symmetric but follows a certain algorithm. In [2] there is an example demonstrating non-symmetry, the consequently employed encoding of [2] is also based on inputs.

Symmetric Communication

The communication in the ETTS semantics M of an HLL model $\mathscr{M}$ is symmetric in identities if and only if whenever an event E from individual u_1 with parameters $\vec{d}$ is ready to be consumed by individual u_2 in state $s \in S(M)$, i.e. if

$$s(u_2).\epsilon = \epsilon_0.u_1 \xrightarrow{E(d_1,\dots,d_n)} u_2, \tag{9.77}$$

then the same event is available in $p(s)$ with permuted identities, i.e. then

$$p(s)(p(u_2)).\epsilon = \epsilon_0.p(u_1) \xrightarrow{E(p(d_1),\dots,p(d_n))} p(u_2) \tag{9.78}$$

Note that full symmetric communication is helpful, but not necessary to establish symmetry in identities for M. The reason is that M can be symmetric by having the scheduler to avoid states where communication goes wrong. That is, individuals of a certain class may never be scheduled; then it doesn't play a role whether communication *towards* them is symmetric or not because other objects cannot query the ether of a given other objects. Adding the possibility of such queries in the expression language would have as consequence a tighter link between symmetric communication and symmetry in identities in M.

How do we establish this property? There is typically no reason for making communication sensitive to identities, and thus non-symmetric. Yet for example the UML has rather weak requirements on the communication medium and allows to *defer* events and principally allows for priority queues. One could think of giving priority to certain identities, which would clearly break symmetry.

Symmetric Interpretation for Conditions and Actions

An interpretation ι of function symbols is called symmetric in identities if and only if it satisfies Def. 7.1.9 correspondingly. It yields the desired property

$$\iota[\![cond]\!](s, id, \theta) = \iota[\![cond]\!](p(s), p(id), p(\theta)) \tag{9.79}$$

where the assignment is permuted point-wise, as usual. This property ensures that conditions guarding transitions can be taken in permuted states if they can be taken in the original state.

The general case of the syntactic detection given above works as follows. Given a set of function symbols,

1. if all conditions and actions in the HLL model $\mathscr{M}$ are well-typed, and
2. if ι is an interpretation which is symmetric in identities,

then the semantics of HLL, the ETTS M, is symmetric in identities provided communication and scheduling don't spoil it.

Note that establishing (1) amounts to a syntactic type check. The reason for restriction to assignment, equality, and array access is that a symmetric interpretation of comparison for equality is easy to obtain: the natural comparison for equality *is* symmetric.

Furthermore, constants or literals denoting particular identities are not admitted as they are by nature not symmetric, arithmetics on identities are not admitted for similar reasons.

The second desired property is

$$\iota[\![P]\!](s_0, u, \theta, \mathcal{O})(s) = \iota[\![P]\!](p(s_0), p(u), p(\theta), \mathcal{O})(p(s)) \tag{9.80}$$

This property ensures that the application of a transition program to a permuted state $p(s)$ yields the permutation of the application to the original state s. In our simplified setting, that is, without arrays and without loops in the action language, it follows directly by induction over the syntactic structure of transition programs, given a symmetric interpretation, creation oracle, and communication.

We'll see in Section 9.2.5 that the theory of singularities from Section 7.4 allows us to treat HLL models which adhere to significantly weakened requirements.

Symmetric Scheduling

A scheduler $\mathcal{S}$ for an HLL model $\mathscr{M}$ is called symmetric in identities if and only if for each sequence

$$(u_1, r_{C_1,1}), \dots, (u_n, r_{C_n,n}), n \in \mathbb{N}_0, \tag{9.81}$$

scheduled in state s, the same sequence with permuted identities is scheduled in the permuted state $p(s)$, i.e. if

$$(p(u_1), r_{C_1,1}), \dots, (p(u_n), r_{C_n,n}) \in \mathcal{S}(p(s)) \tag{9.82}$$

How do we establish this property?. Similarly to the communication medium, there is in most cases no reason for making the scheduling sensitive to identities, and thus non-symmetric. The model-checkers considered in the applications from Section 9.2.2 are either fully concurrent, that is, each identity is scheduled in every step ([127]), explicitly encode an interleaving [178], or provide interleaving natively, like SPIN as employed in [2]. In particular explicitly encoded interleaving has to be designed properly to preserve symmetry.

9.2.4. Symmetric HLL Models

Lemma 9.2.1 (Symmetric HLL Model)**.** *Let $\mathscr{M}$ be an HLL model with symmetric conditions and actions in the sense of Section 9.2.3.*

If ι, $\mathcal{O}$, $\mathcal{S}$, and $\mathcal{E}$ are a symmetric interpretation, oracle, scheduler, and ether in the sense of Section 9.2.3 to 9.2.3, then

$$M := \iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}) \tag{9.83}$$

is symmetric in identities. ♢

Proof. See Section A.4. □

Note that the premises of Lemma 9.2.1 are sufficient but only in the following weak sense necessary. If we drop one of the four premises, namely symmetry of interpretation, oracle, scheduler, and ether, then we can easily construct an HLL model whose ETTS is not symmetric in identities.

For example, with a non-symmetric interpretation of some operator the result of this operator applied to the identity can be used to switch between two behaviours depending on the identity. A non-symmetric oracle would propose inputs depending on identities or prefer particular identities in case of creation. In both cases, we can construct an HLL model where individuals use this non-symmetry to show different behaviour depending on their identities.

A scheduler is non-symmetric if its choices depend on identities, that is, if there are states where an individual u is scheduled while $p(u)$ is not scheduled in the permuted state, thus breaking symmetry.

Finally, a non-symmetric ether would drop or prefer messages based on identities, then the individual for which messages are dropped clearly behaves different than an individual whose messages arrive, given the classes are sensitive for that message.

The latter case already indicates why the premises are only necessary in the following weak sense. Namely, if the non-symmetry doesn't have a visible effect, then the resulting ETTS can of course still be symmetric in identities.

For example, non-symmetric inputs may simply be ignored by individuals and creation may not take place. Similarly, a non-symmetric ether could delay messages non-symmetrically, resulting in a state where an event is consumable but in the permuted state not, but if the ether manages to have all messages ready when they actually are consumed, the delay is not visible.

9.2.5. Less Restricted Syntax: Singularities and Symmetric Loops

Singularities

For simplicity, we introduced a strict definition of symmetric interpretation in Section 9.2.3, which is also considered in [89]. It requires that functions on scalarset types, in our case on identities, are completely symmetric. According to Section 7.4, where we discussed how to treat singularities in QR, it is sufficient to know whether a function is invariant under permutations that are consistent with a finer partitioning.

In other words, we're interested in what we could call partitioning of a function's domain such that partitioning respecting permutations are symmetric.

For example, we could wish for an equivalent of "NULL" pointers in HLL, that is, have a designated individual u to which links are directed if they are not in use.

To this end, we could have a 0-ary function symbol NULL of a class type τ_C. If we choose $\mathbb{N}_0$ as domain of τ_C, and thus as set of identities, an interpretation ι could map NULL constantly to 0. Then the partitioning of the NULL function in ι would be $\{0\} \mathbin{\dot\cup} Id \setminus \{0\}$ because for each partitioning consistent permutation p we have

$$\iota[\![\mathrm{NULL}]\!](s, id, \theta) = \iota[\![\mathrm{NULL}]\!](p(s), p(id), p(\theta)) \tag{9.84}$$

because if p is consistent with the permutation, then it leaves 0 unchanged, i.e. $p(0) = 0$, and only permutes the elements of $Id \setminus \{0\}$.

Given the partitioning of each function symbol occurring in an HLL model, we can construct a finest partitioning of the set of identities such that if a permutation is consistent with the latter, then it is consistent with all of the original ones.

And if the resulting partitioning is finite, we can still apply QR to obtain finitely many representative cases by Section 7.4.

Note that this line of thoughts allows to overcome the strict exclusion of literals (like NULL) in [89]. Instead we can principally allow any number of literals (with a known interpretation and partitioning) and automatically determine a unified partitioning by simply identifying the literals *occurring* in the HLL model.

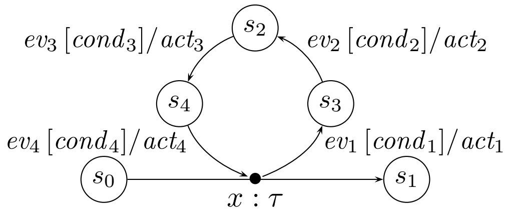


Figure 9.3.: **Exemplary Syntax for non-atomic loops.** The semantics could be to execute the loop s_2, s_3, s_4 for each possible binding of x and afterwards reach s_1.

Symmetric Loops

If we had arrays in our HLL language as possible types of local variables, there were a reason to have iteration in the action language, possibly looking like as follows.

$$\texttt{forall}\ x : \tau\ \texttt{do}\ \langle act \rangle\ \texttt{od} \tag{9.85}$$

where τ is a scalarset type.

The semantics would be the usual one, to execute *act* on different bindings for x in an arbitrary order. But we would have to require termination, which is possible for for-loops with explicit start and end value and defined loop variable update, or if the domain of τ in the example above is finite. This is the approach taken, for instance, by tools with simulation and code generation facilities like Statemate.

Alternatively, one can think of loops iterating over links, a topic we were only able to briefly address in earlier chapters for lack of time.

In both cases, one would be interested in having symmetric updates, that is, a preservation of property (9.80).

The approach of [89] is to have a particular syntax similar to the one given above and in addition to require that the outcome, that is, the valuation of variables affected by the loop, is independent from the order in which the domain of the loop variable is considered.

Natural examples are what are called *collective operations* or *reductions* in the parallel programming community, like

- all commutative logical operations on array elements, e.g., computing the conjunction or disjunction of all array elements,
- all commutative arithmetical operations on array elements, e.g., computing the sum of all array elements

This naturally extends to sending a message to all linked individuals when having loops over links.

Interestingly, we observe that we can extend this pattern from atomic loops, that is, loops executed atomically during a transition, to loops in the classes' transition systems if we invent a certain new syntax, namely a kind of for-all-loop as sketched in Figure 9.3.

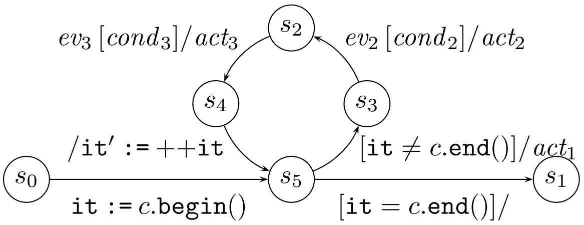


Figure 9.4.: **Alternative Syntax for symmetric, non-atomic loops.** Orienting on programming languages like C++.

In order to be symmetry preserving, the final local state should be independent from the order in which x is bound to the values from the domain of τ. Sufficient, but strong criteria directly carry over from the atomic case, for example iteratively updating a local variable to finally carry the sum of array entries. This is an interesting topic in its own right, but we chose not to elaborate on this here because it wouldn't contribute significantly to our overall aim.

But note that, while the ad-hoc syntax in Figure 9.3 can be considered as aesthetically awkward, there is a straightforward implementation if the considered transition programming language is something like C++ with its notion of iterators in the Standard Template Library (STL).

The general case of iteration over the elements of a container instance c of container class C, that is, C is a list or a vector or a set, is of the form

```
for (C::iterator it = c.begin(); it != c.end(); ++it)
   ... use iterator it ...;
```

where C :: `iterator` denotes an iterator type of container class C. The *it* is a new, local iterator object, which is initialised with c.`begin`() which denotes the first element in container c (where “first” is certainly understood differently for vectors (arrays) or sets). Before executing the loop body, the condition $it = c.end()$ is checked, that is, whether the end of container c is reached. After executing the loop body, the iterator is advanced to the next element (which is again understood differently for vectors and sets).

In HLL such an action would spread over multiple transitions forming a *non-atomic* loop (cf. Figure 9.4). Now in particular C++ allows to overload all participating classes and methods, so we could have a class of symmetric iterators where the *begin* method yields *any* element and the advancing operation ensures that all elements are visited in *any* order. Then we had *declared* in a common programming language setting that this loop is intended to be independent from iteration order, and could check whether the employed operations are sufficiently commutative to establish that the outcome actually *is* independent from order of iteration and conclude that symmetry is not broken.

An alternative syntax, for example considered in [9], could be the introduction of a *pick* operation on links which chooses one link with the given name randomly and removes that link from the considered individual until there are no links remaining. By

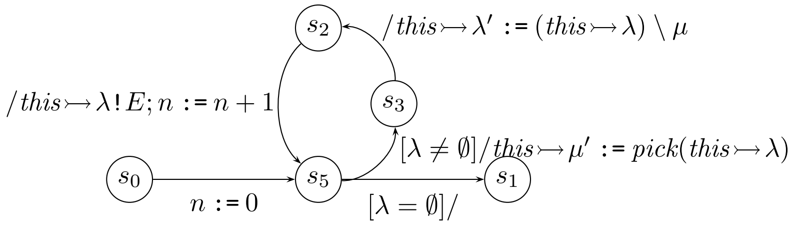

s_2
$/this{\rightarrowtail}\lambda' := (this{\rightarrowtail}\lambda) \setminus \mu$
$/this{\rightarrowtail}\lambda!E; n := n+1$
s_3
$[\lambda \neq \emptyset]/this{\rightarrowtail}\mu' := pick(this{\rightarrowtail}\lambda)$
s_0
s_5
s_1
$n := 0$
$[\lambda = \emptyset]/$

Figure 9.5.: **Alternative Syntax for symmetric, non-atomic loops.** A "random pick" function in combination with emptyness check for links.

$$\begin{tikzcd}
\mathscr{M} \arrow[rr, mapsto, "D_D"] \arrow[d, mapsto, "[\![\cdot]\!]"'] \arrow[drr, mapsto, "[\![\cdot]\!]^\sharp"] & & \mathscr{M}^\sharp_D \arrow[d, mapsto, "[\![\cdot]\!]"] \\
M \arrow[r, mapsto, "D", "\preceq"'] & M^\sharp \arrow[r, phantom, "\preceq"'] & M^\sharp_D
\end{tikzcd}$$

poss. infin. finite

Figure 9.6.: **Commuting Diagram Revisited (1).** The parts of Figure 9.1 relevant for Section 9.3. The shaded regions indicate the subject of Section 9.3.3, namely obtaining a DTR abstraction by a syntactic transformation of the HLL model and the regular HLL semantics.

the random choice, there is also no defined order in which the loop is executed. The example in Figure 9.5 sends a message E to all individuals known via λ and counts the number of messages in n.

9.3. DTR for HLL

By Section 9.2, we can see by looking at the syntax and semantics of an HLL model whether the resulting ETTS is symmetric in identities, we don't need to examine the ETTS itself. For the DTR abstraction we want to have something similar.

If M is the ETTS of an HLL model $\mathscr{M}$, and $M^\sharp$ its DTR abstraction under a certain DTR D, then we want to construct $M^\sharp$ directly by modifying $\mathscr{M}$ into $\mathscr{M}^\sharp_D$, such that the regular HLL semantics yields the finite ETTS $M^\sharp$ (cf. Figure 9.6).

After introducing a normal form of navigation expressions in Section 9.3.1 and conditional expressions Section 9.3.2 as technical aids, Section 9.3.3 is dedicated to define the mapping called D_D in Figure 9.6 and to establish the shaded simulation relation from Figure 9.6 for the case of an *interleaving* semantics of HLL. In Section 9.3.4 we'll see that matters become significantly harder with a truly concurrent semantics, and are only able to sketch some possible approaches.

9.3.1. Navigation Expression Normal Form

The presentation of the following sections becomes significantly easier if we assume a certain normal form of navigation expressions. The whole section can be done in the original, rather natural syntax but would become extremely cluttered and unreadable (cf. Example 9.3.5 where we transform (9.147) once directly in the original syntax and once via the normal form).

General Idea

We want to consider the grammar

$$nav ::= p \mid \mathit{this} \mid \mathit{this} \rightarrowtail h \mid \mathit{this} \rightarrowtail h \rightarrowtail \lambda, \tag{9.86}$$

where $h \in \Lambda_A$ is an auxiliary link, to define navigation expressions instead of (9.9).

This is not restricting generality because in the following we give a procedure to transform any given transition program into this normal form by adding auxiliary variables that are assigned before being used.

Definition 9.3.1 (Navigation Expression Normal Form)**.** *Let $\mathscr{M}$ be an HLL model. We say $\mathscr{M}$ is* navigation expression normal form *if and only if all navigation expressions appearing are particular instances of the form given by grammar* (9.86)*, namely*

1. *navigation expressions on the right-hand side of assignments or in conditions are of the form*

$$\begin{aligned} &\mathit{this} \rightarrowtail h' \\ &\mathit{this} \rightarrowtail h' \rightarrowtail x \\ &\mathit{this} \rightarrowtail h' \rightarrowtail x'. \end{aligned} \tag{9.87}$$

   In particular, parameters p don't occur outside the right-hand side of assignments.

2. *assignments are either of the forms*

$$\begin{aligned} &\mathit{this} \rightarrowtail h' := \mathit{this} && \mathit{this} \rightarrowtail h^{\mathtt{C}\prime} := \mathit{this}^{\mathtt{C}} \\ &\mathit{this} \rightarrowtail h' := p && \mathit{this} \rightarrowtail h^{\mathtt{C}\prime} := p^{\mathtt{C}} \\ &\mathit{this} \rightarrowtail h' := \mathit{this} \rightarrowtail h' \rightarrowtail \lambda && \mathit{this} \rightarrowtail h^{\mathtt{C}\prime} := \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}} \end{aligned} \tag{9.88}$$

   or of the forms

$$\begin{aligned} &\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' := \mathit{nav} \\ &\mathit{this} \rightarrowtail h' \rightarrowtail x' := \mathit{expr}, \end{aligned} \tag{9.89}$$

   where h is an auxiliary link and nav adheres to (9.87)*.* ♢

The general idea of the transformation procedure is to transform actions of the form

$$\mathit{this} \rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x \mathrel{\texttt{:=}} 0; \tag{9.90}$$

into

$$\begin{aligned} &\mathit{this} \rightarrowtail h_1' \mathrel{\texttt{:=}} \mathit{this} \rightarrowtail \lambda; \\ &\mathit{this} \rightarrowtail h_2' \mathrel{\texttt{:=}} \mathit{this} \rightarrowtail h_1' \rightarrowtail \mu'; \\ &\mathit{this} \rightarrowtail h_2' \rightarrowtail x' \mathrel{\texttt{:=}} 0; \end{aligned} \tag{9.91}$$

That is, a prefix of auxiliary variable assignments follows by the assignment in terms of previously set up auxiliary variables. If the original program is well-formed, then the transformation result is well-formed, too, because all auxiliary variables are assigned before use.

Let-Expressions for Guarding Conditions

The transformation sketched above is defined rather straight-forward for actions, because we can employ assignments, but for guarding conditions we'll extend the syntax and semantics because conditions as defined above don't have the possibility to update auxiliary variables.

The new syntax of guarding conditions is

$$\mathit{cond} ::= \texttt{let}\ \langle \mathit{act} \rangle\ \texttt{in}\ \langle \mathit{term} \rangle \tag{9.92}$$

instead of (9.92) with the following semantics.

$$\begin{aligned} &\iota[\![\texttt{let}\ \mathit{act}\ \texttt{in}\ \mathit{term}]\!](s_0, s, u, \theta) \\ &\quad = \iota[\![\mathit{term}]\!](s_0, \iota[\![\mathit{act}]\!](s_0, u, \theta, \mathcal{O})(s), u, \theta) \end{aligned} \tag{9.93}$$

That is, the term *term*, which refers to auxiliary variables, is evaluated in the (temporary) state obtained by applying the transition program *act*, which only updates auxiliary variables, to state s.

Recall that this is just to ease the presentation, even the modification of syntax and semantics doesn't restrict generality.

Transformation: Terms and Expressions

The transformation of guarding conditions into normal form is defined as

$$\mathit{term}^{NF} = \texttt{let}\ \mathit{term}^{L}\ \texttt{in}\ \mathit{term}^{E} \tag{9.94}$$

where term^L denotes a sequence of assignments of the form

$$\mathit{this} \rightarrowtail h' := \mathit{this} \text{ or } \mathit{this} \rightarrowtail h_2' := \mathit{this} \rightarrowtail h_1' \rightarrowtail \lambda' \tag{9.95}$$

and term^E the expression obtained from *term* by replacing all navigation expressions with auxiliary links.

The helper transformations $(\cdot)^L$ and $(\cdot)^E$ are inductively defined over the syntax of terms, expressions, and navigation expressions as follows.

1. $1^L = \epsilon, \qquad 1^E = 1$

2. $(nav_1 \rightarrowtail \lambda_1 = nav_2 \rightarrowtail \lambda_2)^L = nav_1{}^L; nav_2{}^L,$
   $(nav_1 \rightarrowtail \lambda_1 = nav_2 \rightarrowtail \lambda_2)^E$
   $= (this \rightarrowtail \mathrm{last}(nav_1{}^L) \rightarrowtail \lambda_1 = this \rightarrowtail \mathrm{last}(nav_2{}^L) \rightarrowtail \lambda_2)$

3. $\neg term^L = term^L, \qquad \neg term^E = \neg term^E$

4. $(term_1 \wedge term_2)^L = term_1{}^L; term_2{}^L,$
   $(term_1 \wedge term_2)^E = term_1{}^E \wedge term_2{}^E$

5. $p^L = \epsilon, \qquad p^E = p$
   if p is of basic type, otherwise by 11. below

6. $(nav \rightarrowtail x)^L = nav^L, \qquad (nav \rightarrowtail x)^E = this \rightarrowtail \mathrm{last}(nav^L) \rightarrowtail x$

7. $(nav \rightarrowtail x')^L = nav^L, \qquad (nav \rightarrowtail x')^E = this \rightarrowtail \mathrm{last}(nav^L) \rightarrowtail x'$

8. $(nav \rightarrowtail \lambda)^L = nav^L, \qquad (nav \rightarrowtail \lambda)^E = this \rightarrowtail \mathrm{last}(nav^L) \rightarrowtail \lambda$

9. $(nav \rightarrowtail \lambda')^L = nav^L, \qquad (nav \rightarrowtail \lambda')^E = this \rightarrowtail \mathrm{last}(nav^L) \rightarrowtail \lambda'$

10. $(f(expr_1, \ldots, expr_n))^L = expr_1{}^L; \ldots; expr_n{}^L,$
    $(f(expr_1, \ldots, expr_n))^E = f(expr_1{}^E, \ldots, expr_n{}^E)$

11. $p^L = (this \rightarrowtail h' := p), \qquad p^E = \mathrm{last}(p^L)$

12. $this^L = (this \rightarrowtail h' := this), \quad this^E = \mathrm{last}(this^L)$

13. $nav \rightarrowtail \lambda^L = nav^L, \qquad (nav \rightarrowtail \lambda)^E = this \rightarrowtail \mathrm{last}(nav^L) \rightarrowtail \lambda$

Here, h denotes a fresh auxiliary link and. $\mathrm{last}(\cdot^L)$ denotes the last auxiliary link of an expression, for example,

$$\mathrm{last}(this \rightarrowtail h'_1 := nav_1; \ldots; this \rightarrowtail h'_n := nav_n) = h_n \tag{9.96}$$

Well-definedness is ensured because this operator is never applied to an empty sequence of auxiliary assignments.

When concatenating let expressions, we assume that conflicts with names of auxiliary variables are solved by renaming.

Example 9.3.2 (Example: Navigation Expression Normal Form)**.** *The navigation expression normal form of the condition*

$$term = \neg(this \rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x > 0) \tag{9.97}$$

is

$$\mathit{term}^{NF} = \mathtt{let}\ \mathit{term}^{L}\ \mathtt{in}\ \mathit{term}^{E} = \begin{array}{l} \mathtt{let}\ \mathit{this}\rightarrowtail h'_3 := \mathit{this}; \\ \quad \mathit{this}\rightarrowtail h'_2 := \mathit{this}\rightarrowtail h'_3 \rightarrowtail \lambda; \\ \quad \mathit{this}\rightarrowtail h'_1 := \mathit{this}\rightarrowtail h'_2 \rightarrowtail \mu; \\ \mathtt{in}\ \neg(\mathit{this}\rightarrowtail h'_1 \rightarrowtail x > 0) \end{array} \tag{9.98}$$

because

$$\begin{aligned} \mathit{term}^{L} &= (\neg(\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x > 0))^{L} \\ &\overset{(3.)}{=} (\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x > 0)^{L} \\ &\overset{(10.)}{=} \mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x^{L}; 0^{L}; \\ &\overset{(6.)}{=} (\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu)^{L}; \\ &\overset{(13.)}{=} (\mathit{this}\rightarrowtail \lambda)^{L}; \mathit{this}\rightarrowtail h'_1 := \mathit{this}\rightarrowtail \operatorname{last}((\mathit{this}\rightarrowtail \lambda)^{L}) \rightarrowtail \mu; \\ &\overset{(13.)}{=} \mathit{this}^{L}; \mathit{this}\rightarrowtail h'_2 := \mathit{this}\rightarrowtail \operatorname{last}(\mathit{this}^{L}) \rightarrowtail \lambda; \\ &\qquad \mathit{this}\rightarrowtail h'_1 := \mathit{this}\rightarrowtail \operatorname{last}((\mathit{this}\rightarrowtail \lambda)^{L}) \rightarrowtail \mu \\ &\overset{(13.)}{=} \mathit{this}\rightarrowtail h'_3 := \mathit{this}; \mathit{this}\rightarrowtail h'_2 := \mathit{this}\rightarrowtail h'_3 \rightarrowtail \lambda; \\ &\qquad \mathit{this}\rightarrowtail h'_1 := \mathit{this}\rightarrowtail h'_2 \rightarrowtail \mu \\ \mathit{term}^{E} &= (\neg(\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x > 0))^{E} \\ &\overset{(3.)}{=} \neg((\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x > 0))^{E} \\ &\overset{(10.)}{=} \neg((\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x)^{E} > 0) \\ &\overset{(6.)}{=} \neg(\mathit{this}\rightarrowtail \operatorname{last}((\mathit{this}\rightarrowtail \lambda \rightarrowtail \mu)^{L}) \rightarrowtail x > 0) \\ &= \neg(\mathit{this}\rightarrowtail h'_1 \rightarrowtail x > 0) \end{aligned} \tag{9.99}$$

assuming that "$>$" is a binary function symbol from the signature.

Similarly, the navigation expression normal form of the action

$$\mathit{act} = \mathit{this}\rightarrowtail \lambda \rightarrowtail \mu \rightarrowtail x := 0; \tag{9.100}$$

is

$$\mathit{act}^{NF} = \mathit{act}^{L}; \mathit{act}^{E} = \begin{array}{l} \mathit{this}\rightarrowtail h'_3 := \mathit{this}; \\ \mathit{this}\rightarrowtail h'_2 := \mathit{this}\rightarrowtail h'_3 \rightarrowtail \lambda; \\ \mathit{this}\rightarrowtail h'_1 := \mathit{this}\rightarrowtail h'_2 \rightarrowtail \mu; \\ \mathit{this}\rightarrowtail h'_1 \rightarrowtail x' := 0 \end{array} \tag{9.101}$$

◊

Transformation: Actions

Based on the definitions for terms and expressions, the procedure can analogously be extended to actions as follows.

1. $(\mathit{nav} \rightarrowtail x' \texttt{ := } \mathit{expr})^{NF} = \mathit{nav}^L; \mathit{expr}^L; \mathit{this} \rightarrowtail \mathrm{last}(\mathit{nav}^L) \rightarrowtail x' \texttt{ := } \mathit{expr}^E$
2. $\mathit{nav} \rightarrowtail \lambda' \texttt{ := new } C^{NF} = \mathit{nav}^L; \mathit{this} \rightarrowtail \mathrm{last}(\mathit{nav}^L) \rightarrowtail \lambda' \texttt{ := new } C$
3. $\mathit{nav}\,\texttt{!}\,E(\mathit{expr}_1, \ldots, \mathit{expr}_n)^{NF}$
   $= \mathit{nav}^L; {\mathit{expr}_1}^L; \ldots\ {\mathit{expr}_2}^L; \mathit{this} \rightarrowtail \mathrm{last}(\mathit{nav}^L)\,\texttt{!}\,E({\mathit{expr}_1}^E, \ldots, {\mathit{expr}_n}^E)$
4. $\texttt{delete } \mathit{nav}^{NF} = \mathit{nav}^L; \texttt{delete } \mathit{this} \rightarrowtail \mathrm{last}(\mathit{nav}^L)$
5. $\texttt{if } \mathit{cond} \texttt{ then } \mathit{act}_1\texttt{; else } \mathit{act}_2\texttt{; } \texttt{fi}^{NF}$
   $= \mathit{cond}^L; \texttt{if } \mathit{cond}^E \texttt{ then } {\mathit{act}_1}^{NF}\texttt{; else } {\mathit{act}_2}^{NF}\texttt{; } \texttt{fi}^{NF}$

Assignments of links are treated similar to assignments to local variables, the remaining cases are trivial.

A given HLL model $\mathscr{M}$ is brought into normal form by applying the above transformations to all guarding conditions and all transition programs, plus adding the required fresh auxiliary variables to the corresponding classes.

Preservation of Semantics

The main and general property of the transformations is that they're semantics preserving.

Lemma 9.3.3. *Let $M = (S, S_0, R, \mathscr{L}, e)$ be the semantics of an HLL model $\mathscr{M}$, $s, s_0 \in S$ system states, u an individual of M and θ an assignment of event parameters. Then normal-form transformation as introduced above is semantics preserving for each expression term and each action act of $\mathscr{M}$, i.e.*

$$\iota[\![\mathit{term}^{NF}]\!](s_0, s, u, \theta) = \iota[\![\mathit{term}]\!](s_0, s, u, \theta) \tag{9.102}$$

and

$$\iota[\![\mathit{act}^{NF}]\!](s_0, u, \theta, \mathcal{O})(s) = \iota[\![\mathit{act}]\!](s_0, u, \theta, \mathcal{O})(s) \tag{9.103}$$

◇

Proof. By induction over the structure of terms and actions. □

Now we can assume, without loss of generality, that all conditions and actions are in the normal form of Def. 9.3.1.

Note that this straightforward transformation is in general far from minimal, that is, there will be many duplicate definitions of auxiliary variables. Classical static analyses employed in optimising compilers can be employed to eliminate these duplicates, basically doing a expression analysis . We'll see later that minimality indeed plays a role in the precision of the obtained abstract transition system; the more redundant auxiliary variables, the less precise the obtained abstraction.

9.3.2. Conditional Expressions

In the following transformations, we'll make extensive use of conditional execution in the style

> *"if we have a link to a non-concrete object, then behave like this, otherwise behave like that"*

This is certainly expressible with the `if-then-else-fi` action from Section 9.1. The action in particular doesn't require that all expressions in the `else`-branch are defined when the `then`-branch is taken, which will be the case in many of the added conditional executions.

For convenience, we'll use a similar construct in the expression language, namely we extend (9.8) to

$$\mathit{expr} ::= \cdots \mid (\langle \mathit{expr}_1 \rangle \; ? \; \langle \mathit{expr}_2 \rangle : \langle \mathit{expr}_3 \rangle) \tag{9.104}$$

with the semantics

$$\begin{aligned} &\iota[\![(\mathit{expr}_1 \; ? \; \mathit{expr}_2 : \mathit{expr}_3)]\!](s, u, \theta) \\ &= \begin{cases} \iota[\![\mathit{expr}_2]\!](s, u, \theta) & \text{, if } \iota[\![\mathit{expr}_1]\!](s, u, \theta) = 1 \\ & \quad \text{and } \iota[\![\mathit{expr}_2]\!](s, u, \theta) \text{ defined} \\ \iota[\![\mathit{expr}_3]\!](s, u, \theta) & \text{, if } \iota[\![\mathit{expr}_1]\!](s, u, \theta) = 0 \\ & \quad \text{and } \iota[\![\mathit{expr}_3]\!](s, u, \theta) \text{ defined} \\ \text{undefined} & \text{, otherwise} \end{cases} \end{aligned} \tag{9.105}$$

We can use this construct without loss of generality because the extended language can be expressed with the `if-then-else-fi` action, but this will typically be less concise and less readable.

9.3.3. Syntactical DTR: Interleaving

By Section 9.1.3, we distinguish interleaving and concurrent schedulers. In this section, we'll concentrate on interleaving semantics of HLL before we turn to the concurrent case in Section 9.3.4.

Common Requirements

In order to transform an HLL model $\mathscr{M}$ into an HLL model $\mathscr{M}^\sharp_D$ according to a given DTR D such that $M^\sharp$, the ETTS of $\mathscr{M}^\sharp_D$ (bi)simulates $D(M)$, we'll employ the following features.

Firstly, we require support for inputs, both for local variables and for links (which is, cf. Section 9.1, somehow uncommon, and may be neglected by HLL variants and instantiations).

Secondly, we'll use the ability to choose a finite upper limit on the number of individuals per class, that is, we need to be able to modify the creation oracle, which may not always be possible in concrete HLL instantiations.

We'll assume to be notified by the `new` operation if the limit is reached; most naturally by `new` C yielding $\emptyset$, then we can write an HLL transition program corresponding to

$$\begin{aligned} &\mathit{this}\rightarrowtail h' \rightarrowtail \lambda' := \texttt{new}\ C; \\ &\mathit{this}\rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}'} := (\mathit{this}\rightarrowtail h' \rightarrowtail \lambda' = \emptyset); \end{aligned} \tag{9.106}$$

which sets the other-flag (see below) whenever creation fails. In addition, we'll employ an input which controls whether we try the `new` operation or directly employ another identity. The reason is that the identity used in the first creation needn't always be a concrete individual, even if creation could still succeed.

Furthermore, there has to be a boolean basic type $\tau_{\mathbb{B}}$.

Note that the requirements named here are only *sufficient* to provide a syntactic transformation of HLL models, we don't address the question whether they're necessary.

Possibly one or the other is expressible in terms of other HLL features, like having inputs vs. having non-determinism. For example, being able to modify the creation oracle could be circumvented by creating the finite number of individuals in advance and adding a "manual" management of aliveness, but thereby one would lose any, possibly given, native support for querying aliveness in HLL models.

Transformation: Types, Events, Classes

If we consider an HLL variant which is symmetric by nature (cf. Section 9.2), there we hit an obvious problem. Namely, we then cannot consider $\mathtt{C}_C$ from Chapter 6 to be an identity of class C, because this identity is obviously not symmetric to the other identities of class C.

In this section, we demonstrate how to solve this problem by adding new classes and local variables to $\mathscr{M}_D^\sharp$. In Section 9.4.3, where we demonstrate the syntactical transformation on less strictly, and in particular not natively symmetric, array programs, matters become slightly easier.

Given the premises of the preceding Section 9.3.3, that is, having support for basic and object type inputs, let $\mathscr{M}$ be an HLL model with navigation expressions in normal form according to Section 9.3.1.

Let

$$D = \{(d_{C_1}, \mathit{Id}_{C_1}), \dots, (d_{C_m}, \mathit{Id}_{C_m})\} \tag{9.107}$$

be a DTR in the sense of Chapter 6 where $C_1, \dots, C_m$ are disjoint classes from $\mathscr{M}$. Then $\mathscr{M}_D^\sharp = (\mathscr{T}^\sharp, \mathscr{E}^\sharp, \mathscr{C}^\sharp, \mathit{cond}_0{}^\sharp)$ is obtained from $\mathscr{M}$ as follows.

Types The set of types $\mathscr{T}^\sharp$ is obtained from $\mathscr{T}$ by adding object types for the new classes (see below).

Events The set of events $\mathscr{E}^\sharp$ is obtained from $\mathscr{E}$ by changing the events' parameters as follows. Let $E \in \mathscr{E}$ be an event in $\mathscr{M}$ with parameters $p_{1,1}, \dots, p_{1,n}$ of types $\tau_{1,1}, \dots, \tau_{1,n}$.

Then $E \in \mathscr{E}^\sharp$ has parameters $p_{2,1}, \dots, p_{2,m}$ of types $\tau_{2,1}, \dots, \tau_{2,m}$ such that the first parameters correspond, i.e.

$$p_{2,1} = p_{1,1}, \tau_{2,1} = \tau_{1,1} \tag{9.108}$$

and the i-th parameter, $i > 1$, is boolean if the parameter $p_{1,j}$ corresponding to it is of object type, i.e.

$$\tau_{2,i} = \tau_{\mathbb{B}}. \tag{9.109}$$

If $\tau_{1,j} = \tau_C$, for some class $C \in \mathscr{C}$, and the corresponding parameter otherwise, i.e.

$$p_{2,i} = p_{1,j}, \tau_{2,i} = \tau_{1,j} \tag{9.110}$$

Example 9.3.4. *Let $E \in \mathscr{E}$ be an event in $\mathscr{M}$ with parameters $p_{1,1}, p_{1,2}, p_{1,3}$ of types $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$, such that only $\tau_{1,2}$ is a basic type and the other two are object types.*

Then E has parameter $p_{2,1}, \dots, p_{2,5}$ of types

$$\tau_{2,1} = \tau_{1,1}, \qquad \tau_{2,2} = \tau_{\mathbb{B}}, \qquad \tau_{2,3} = \tau_{1,2}, \qquad \tau_{2,4} = \tau_{1,3}, \qquad \tau_{2,5} = \tau_{\mathbb{B}} \tag{9.111}$$

in $\mathscr{E}^\sharp$. ♢

Original Classes The sets of states and initial states of class C remains untouched. Expressions and actions of transition labels are modified according to Section 9.3.3 below, that is,

$$R_C = \{(s, \ell^\sim, s') \mid (s, \ell, s') \in R_C\}, \tag{9.112}$$

where

$$\ell^\sim = ev[term^\sim]/act_1^\sim; \dots; act_n^\sim \tag{9.113}$$

following Section 9.3.3 below if

$$\ell = ev[term]/act_1; \dots; act_n. \tag{9.114}$$

To support that translation, the set of local variables is extended as follows

1. For each link name λ pointing to a DTR'ed class, that is, a class covered by D, a local, non-input variable λ^{C} of boolean type is added to X_C, keeping track whether λ is pointing to a regular individual or to another one. That is, to C_C in terms of Chapter 6.

   We call the boolean variable λ^{C} the *other-flag* of λ (cf. Figure 9.7).

2. For each occurrence of a term of the form

   $$nav_1 = nav_2, \tag{9.115}$$

   where both sides are either a parameter of an auxiliary variable as we consider $\mathscr{M}$ to be in normal form, there is a fresh[4] input variable i in $X_{I,C}{}^\sharp$.

[4] not occurring in $X_{I,C}$

3. For each occurrence of an expression (in the sense of (9.8)) of the form
$$nav \rightarrowtail x, \tag{9.116}$$
where nav is either a parameter p or an auxiliary variable h because $\mathscr{M}$ is in normal form, a fresh input i of the type of x is added to $X_{I,C^\sharp}$; in particular in the definition part of let-expressions.[5]

4. Similarly, input links are added to $\Lambda_{I,C^\sharp}$ for each occurrence of an expression of the form
$$nav \rightarrowtail \lambda. \tag{9.117}$$

5. For each occurrence of an action of the form
$$this \rightarrowtail h' \rightarrowtail \lambda' \texttt{ := new } C \tag{9.118}$$
there is a fresh boolean input variable i in $X_{I,C^\sharp}$.

In Section 9.3.3 we'll assume that can access the input variables and links introduced for particular occurrences of expressions.

Additional Classes For each variant of a class C in $\mathscr{C}^\sharp$ as defined in previous paragraph, if C occurs in the DTR D, we add an additional class C^{C} to $\mathscr{C}^\sharp$ (cf. Figure 9.7).

The set of states S^{C} of C^{C} comprises only a single state s_0, which is also initial.

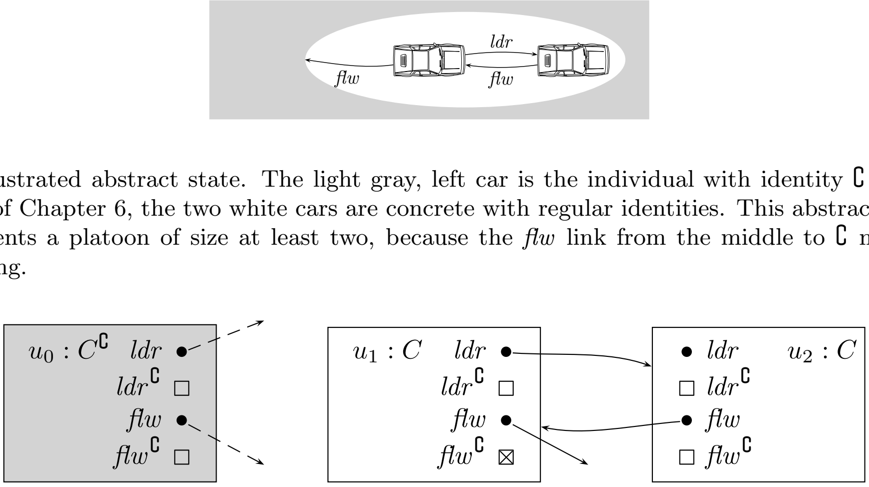
The local variables and links of C, with the only exceptions of $this$ and x_{st}, become inputs in C^{C}, i.e.
$$X_I^{\mathsf{C}} = X_C \setminus \{x_{st}\}, \Lambda_I^{\mathsf{C}} = \Lambda_C \setminus \{this\}, \tag{9.119}$$
including the fresh inputs added in the treatment of C in the paragraph above, plus fresh inputs added by the following transformation of transitions' events.

Then for each transition $(s, \ell^\sim, s')$ of C in $\mathscr{M}_D^\sharp$, we have a transition $(s_0, \ell_0^\sim, s_0)$ in C^{C} where $\ell_0^\sim$ differs from $\ell^\sim$ only in that the event component is always ev and parameter p (including the other-flag parameters p^{C} where applicable) is replaced by a fresh input i of the same type. In addition, there is a stutter transition $(s_0, /\texttt{skip}, s_0)$ for idle steps.

The intuition is that any of these transitions is always enabled, assuming to have received any event with any combination of parameters, independent from the state of the ether.

Initial State Condition Adjusting the initial state condition such that it meets the DTR'd initial states can turn arbitrarily complicated. For example, if the initial state condition $cond$ of $\mathscr{M}$ describes states where a particular number of individuals is alive and connected in a certain topology, then the initial states of $\mathscr{M}_D^\sharp$ have to comprise all combinations of concrete and other (in the sense of C from Chapter 6) individuals satisfying $cond$.

[5] we discuss the issue of precision and optimality in Section 9.3.3, the definition here only aims at convenient definitions later in this section



(a) Illustrated abstract state. The light gray, left car is the individual with identity $\mathtt{C}$ in the sense of Chapter 6, the two white cars are concrete with regular identities. This abstract state represents a platoon of size at least two, because the *flw* link from the middle to $\mathtt{C}$ may be dangling.

(b) Representation in $\mathscr{M}_D^\sharp$. The identity $\mathtt{C}$ is represented by an instance of a dedicated class $C^{\mathtt{C}}$. It is shown light gray because it doesn't have local state, all variables and links are inputs. The two instances of C show the additional boolean other-flag, which is set for the middle individual indicating that its *flw* link points to $\mathtt{C}$, independent from the actual destination, the link may even be absent.

Figure 9.7.: **Syntactical DTR.** Figure 9.7(b) illustrates how an abstract state as shown in Figure 9.7(a) is represented in a state of $\mathscr{M}_D^\sharp$.

To stay focused, we don't discuss this in general, but assume that the initial state is the empty topology, that is, *cond* is of the form

$$\bigwedge_{C \in \mathscr{C}(\mathscr{M})} \forall\, p : \tau_C \,.\, 0. \tag{9.120}$$

Then $\mathit{cond}^\sharp$ is

$$\mathit{cond} \wedge \bigwedge_{C \in \mathscr{C}(\mathscr{M})} \exists\, p_0 : \tau_{C^{\mathtt{C}}} \,.\, \forall\, p : \tau_{C^{\mathtt{C}}} \,.\, p = p_0. \tag{9.121}$$

That is, we require that there is exactly one instance of each $C^{\mathtt{C}}$ alive in the initial state.[6]

A solution for the general case may follow the ideas underlying the above transformations, yet the difference is that there is no "*this*" in *cond* to start from.

Transformation: Transitions

The intuition is as follows. In each transition modify the actions such that if a link is not concrete (somewhere in a link expression), then "guess" the outcome employing a fresh input, skip sending messages and deletion, and when creating, firstly decide

[6] note that this is actually more restrictive than necessary; in fact we only need that there is *at least one* instance per $C^{\mathtt{C}}$ class, having multiple instances doesn't hurt

	1.	2.	3.	4.
this	u_1		u_0	
h_1	u_2	u_0	u_2	u_0
h_1^{C}	0	1	0	1
$\mathit{this}.\epsilon'$	$\oplus\ E(\llbracket \vec{\mathit{expr}} \rrbracket)$	n/a	$\oplus\ E(\llbracket \vec{i} \rrbracket)$	n/a

Table 9.1.: **Effect of Send Action** (9.122). In all four combinations of concrete and non-concrete individuals. We use $\llbracket \vec{\mathit{expr}} \rrbracket$ to denote the valuations of $\mathit{expr}_1, \dots, \mathit{expr}_n$ in the current state and $\llbracket \vec{i} \rrbracket$ to denote the values of the corresponding inputs.

	1.	2.	3.	4.
this, h_0	u_1		u_0	
h_0^{C}	0		1	
h_1	u_2	u_0	u_2	u_0
h_1^{C}	0	1	0	1
$h_0 \rightarrowtail x'$	$\llbracket h_1 \rightarrowtail y \rrbracket$	$\llbracket i \rrbracket$	n/a	n/a

Table 9.2.: **Effect of Local Variable Update** (9.123). In all four combinations of concrete and non-concrete individuals. We use $\llbracket h_1 \rightarrowtail y \rrbracket$ to denote the valuations of the navigation expression in the current state, and similarly the corresponding input.

whether to set the link to non-concrete anyway, otherwise try creation, and set the link to non-concrete if creation fails because the finite limit is hit.

Treat events similarly, that is, guard each link parameter with a boolean flag indication whether it's concrete or shadowy, and leave the rest alone.

The effect of this transformation is best understood on an example. From Chapter 6 we can distinguish four cases

1. action executed in a concrete individual, depending on or affecting another concrete individual,
2. action executed in a concrete individual, depending on or affecting a non-concrete individual,
3. action executed in a non-concrete individual, depending on or affecting another concrete individual, and
4. action executed in a non-concrete individual, depending on or affecting a non-concrete individual.

We'll consider each case for the following actions

- sending an event, i.e. a transition program of the form

$$\mathit{act} = \mathit{this} \rightarrowtail h_1' \,!\, E(\mathit{expr}_1, \dots, \mathit{expr}_n), \tag{9.122}$$

	1.	2.	3.	4.
this, h_0	u_1		u_0	
$h_0^{\mathtt{C}}$	0		1	
h_1	u_2	u_0	u_2	u_0
$h_1^{\mathtt{C}}$	0	1	0	1
$h_0 \rightarrowtail \lambda'$	$[\![h_1 \rightarrowtail \mu]\!]$	$[\![\, i \,]\!]$	n/a	n/a
$h_1 \rightarrowtail \lambda^{\mathtt{C}\prime}$	$[\![h_1 \rightarrowtail \mu^{\mathtt{C}}]\!]$	$[\![\, i^{\mathtt{C}} \,]\!]$	n/a	n/a

Table 9.3.: **Effect of Link Update** (9.124). In all four combinations of concrete and non-concrete individuals. We use $[\![h_1 \rightarrowtail \mu]\!]$ to denote the valuations of the navigation expression in the current state, and similarly the corresponding input.

- reading a local variable, i.e.

$$\mathit{act} = \mathit{this} \rightarrowtail h_0' \rightarrowtail x' := \mathit{this} \rightarrowtail h_1' \rightarrowtail y \qquad (9.123)$$

- writing a local variable, i.e.

$$\mathit{act} = \mathit{this} \rightarrowtail h_0' \rightarrowtail \lambda' := \mathit{this} \rightarrowtail h_1' \rightarrowtail \mu \qquad (9.124)$$

where h_1 may denote a concrete or non-concrete individual.

For simplicity we assume in the following discussion that h_0 denotes *this*, that is, we don't distinguish the case where the individual for which the action is executed, always *this*, is different from the modified one, here h_0.

The effects of these actions in the four different situations are given by Tables 9.1 – 9.3, where we omit the "*this*$\rightarrowtail$" prefix in front of auxiliary links.

First of all, we can observe that neither sending events to the non-concrete individual u_0 nor modifying its local state or links have actually change the local state of u_0 (cases 2. and 4. in Table 9.1 and cases 3. and 4. in Tables 9.2 and 9.3).

The technical reason is, that the flag $h_1^{\mathtt{C}}$ (in case of sending) and $h_0^{\mathtt{C}}$ (in case of assignments) is set, which leads us into the `skip`-branch of (9.137), (9.134), and (9.135).

The intuitive reason is that there is no sense in sending to u_0 because all transitions in $C^{\mathtt{C}}$ are guarded by ε. Similarly, there is no sense in writing any local variable of u_0, because they're all inputs and whenever they're accessed later by any other individual, a local input of the accessing individual will be considered, not the local state of u_0.

Four cases remain, two for sending and two for manipulation of variables and links. When sending to a concrete individual (cases 1. and 3. in Table 9.1), a message is sent even if the sender is non-concrete. The difference is only in the parameters. In a concrete sender, the parameters are taken from the sender's local state[7] while the

[7] assuming for the moment that navigation to other individuals is not involved, the case with further navigation is similar to the modification of variables and links as discussed below

parameter values are determined by the corresponding inputs in a non-concrete sender (cf. (9.137)).

When updating a local variable or link of a concrete individual (cases 1. and 2. in Tables 9.2 and 9.2), the variable or link is indeed modified, yet the assigned value can be different.

If the value for the right-hand side of the assignment is provided by a concrete individual, than it is assigned as it is.

If a non-concrete individual is navigated, that is, if h_2^{C} is set in the example, then the corresponding input is considered. The intuitive reason is that link and other-flag together encode a link to the C identity (in terms of Chapter 6), which has local state $\top$, thus the upper bound of all possible values. In $\mathscr{M}_D^\sharp$, this is represented by the non-determinism of the considered corresponding input, thus in $\mathscr{M}_D^\sharp$, we simply *try* all possible values (cf. (9.134) and (9.135).

Auxiliary Links and Let-Expressions In addition to the auxiliary links from Section 9.3.1, we'll now employ boolean auxiliary variables to track whether the auxiliary links refer to something sensible.

That is, we transform each assignment of an auxiliary link of the form $\mathit{this} \rightarrowtail h' := p$ to

$$\mathit{this} \rightarrowtail h' := p; \mathit{this} \rightarrowtail h^{\mathsf{C}'} := p^{\mathsf{C}}, \tag{9.125}$$

each assignment of the form $\mathit{this} \rightarrowtail h' := \mathit{this}$ to

$$\mathit{this} \rightarrowtail h' := \mathit{this}; \mathit{this} \rightarrowtail h^{\mathsf{C}'} := 1, \tag{9.126}$$

and each assignment of the form $\mathit{this} \rightarrowtail h' := \mathit{this} \rightarrowtail h' \rightarrowtail \lambda$ to

$$\begin{aligned} &\mathit{this} \rightarrowtail h' := (\mathit{this} \rightarrowtail h^{\mathsf{C}'} \mathrel{?} i : \mathit{this} \rightarrowtail h' \rightarrowtail \lambda); \\ &\mathit{this} \rightarrowtail h^{\mathsf{C}'} := (\mathit{this} \rightarrowtail h^{\mathsf{C}'} \mathrel{?} i^{\mathsf{C}} : \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathsf{C}}) \end{aligned} \tag{9.127}$$

where i and i^{C} are the inputs corresponding to this occurrence of the navigation expression. Similarly for the primed case.

By Section 9.3.1, i.e. by navigation expression normal form, there are only these three cases to consider. This applies to both, assignments in let-expressions and assignments in regular action sequences. The intuition is again that, when navigating via a link to another individuals, we may reach any regular individual of this type and even again another individual, while when navigating via a legal link, we have to consider the local state of that regular individual. The other-flag has to be updated accordingly in both cases.

Terms A term *term* of the form $\mathit{nav}_1 = \mathit{nav}_2$ is transformed as follows. By Def. 9.3.1, the navigation expressions nav_i are of the forms $\mathit{this}\rightarrowtail h'_i$, $i = 1, 2$, and we set

$$\begin{aligned}&(\mathit{this}\rightarrowtail h'_1 = \mathit{this}\rightarrowtail h'_2)^{\sim}\\ &\quad = \Big(\mathit{this}\rightarrowtail h_1^{\mathsf{C}'} \wedge \mathit{this}\rightarrowtail h_2^{\mathsf{C}'} \;?\; i_{\mathit{term}}\\ &\qquad : \big(\neg\mathit{this}\rightarrowtail h_1^{\mathsf{C}'} \wedge \neg\mathit{this}\rightarrowtail h_2^{\mathsf{C}'} \;?\; \mathit{this}\rightarrowtail h'_1 = \mathit{this}\rightarrowtail h'_2 : 0\big)\Big)\end{aligned} \tag{9.128}$$

where i_{term} is the input from X_I responsible for this occurrence of *term*. Similarly for parameters, where we assume a pair of p and p^{C} for parameters of object type. If *term* is of any other form, then $\mathit{term}^{\sim} = \mathit{term}$.

Expressions The transformation of expressions is defined inductively following the grammar given in (9.8).

1. $p^{\sim} = p$, note that p is of basic type by Def. 9.3.1.

2. Navigation expression are in normal form by assumption, thus navigation expressions of object type are of the form $\mathit{this}\rightarrowtail h'$ (primed or unprimed) and left alone, that is, we set
$$(\mathit{this}\rightarrowtail h')^{\sim} = \mathit{this}\rightarrowtail h'. \tag{9.129}$$

3. For the same reason, navigation expressions of basic type are of the form $\mathit{this}\rightarrowtail h'\rightarrowtail x$ (primed and unprimed) and we set
$$(\mathit{this}\rightarrowtail h'\rightarrowtail x)^{\sim} = \Big(\mathit{this}\rightarrowtail h^{\mathsf{C}'} \;?\; i_{h,x} : \mathit{this}\rightarrowtail h'\rightarrowtail x\Big) \tag{9.130}$$
where $i_{h,x}$ is the input responsible for this occurrence of navigation to x via h.

4. A function symbol evaluates to an input whenever at least one of the arguments is a link to another individual, that is,
$$f(\mathit{expr}_1, \ldots, \mathit{expr}_n)^{\sim} = \Big(\bigvee_{1\leq i\leq n} \mathit{expr}_i^{\mathsf{C}} \;?\; i : f(\mathit{expr}_1^{\sim}, \ldots, \mathit{expr}_n^{\sim})\Big) \tag{9.131}$$
where $\mathit{expr}_i^{\mathsf{C}}$ is $\mathit{this}\rightarrowtail h^{\mathsf{C}'}$ if expr_i is of an object type, that is, by Def. 9.3.1 a navigation expression of the form $\mathit{this}\rightarrowtail h'$, otherwise it is 0.

Note that the transformation of function symbols also affects the precision of the ETTS resulting from $\mathscr{M}_D^{\sharp}$ compared with DTR applied to the ETTS of $\mathscr{M}$.

The definition above is rather coarse, in particular if singularities and functions operating on identities are permitted. For example, checking whether a given navigation expression denotes a particular identity can distinguish between C and regular identities, so there shouldn't be an input involved in (9.131), instead we could provide a particular transformation per function symbol.

We'll get back to this issue when discussing precision in more general in Section 9.3.3.

Actions Navigation expressions are in normal form (9.86) also in actions, we exemplary show the case h, the case p is defined similarly. The transformation of actions is defined inductively following the grammar given in (9.17).

1. $\mathtt{skip}^{\sim} = \mathtt{skip}$
2. By Def. 9.3.1 we only have to consider the cases of assignments given by (9.88) and (9.89). Assignments to auxiliary links of the form (9.88) are left unchanged except for the following two cases, where only the right-hand side changes.

   $(\mathit{this} \rightarrowtail h' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h' \rightarrowtail \lambda)^{\sim}$

   $$= \mathit{this} \rightarrowtail h' \mathrel{\mathtt{:=}} \left(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \mathbin{?} i_{h,\lambda} : \mathit{this} \rightarrowtail h' \rightarrowtail \lambda\right) \tag{9.132}$$

3. $(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}})^{\sim}$

   $$= \mathit{this} \rightarrowtail h^{\mathtt{C}'} \mathrel{\mathtt{:=}} \left(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \mathbin{?} i_{h,\lambda^{\mathtt{C}}} : \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}}\right) \tag{9.133}$$

4. In assignments to variables and links via auxiliary links of the form (9.89), the left-hand side changes as well; namely if the modified objects is a non-concrete one, a skip statement is executed instead.

   $(\mathit{this} \rightarrowtail h' \rightarrowtail x' \mathrel{\mathtt{:=}} \mathit{expr})^{\sim}$

   $$= \mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then\ skip;\ else}\ \mathit{this} \rightarrowtail h' \rightarrowtail x' \mathrel{\mathtt{:=}} \mathit{expr}^{\sim};\ \mathtt{fi} \tag{9.134}$$

5. $(\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0)^{\sim}$

   $$\begin{aligned} =&\mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then\ skip;} \\ &\mathtt{else}\ \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h'_0; \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}'} \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0^{\mathtt{C}'};\ \mathtt{fi} \end{aligned} \tag{9.135}$$

   This is the only form we need to consider by Def. 9.3.1.

6. $(\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathtt{new}\ C)^{\sim}$

   $$\begin{aligned} = &\mathtt{if}\ i\ \mathtt{then}\ \mathit{this} \rightarrowtail h_0^{\mathtt{C}'} \mathrel{\mathtt{:=}} 1; \\ &\mathtt{else}\ \mathit{this} \rightarrowtail h'_0 \mathrel{\mathtt{:=}} \mathtt{new}\ C; \mathit{this} \rightarrowtail h_0^{\mathtt{C}'} \mathrel{\mathtt{:=}} (\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' = \emptyset);\ \mathtt{fi;} \\ &\mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then\ skip;} \\ &\mathtt{else}\ \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h'_0; \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}'} \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0^{\mathtt{C}'};\ \mathtt{fi} \end{aligned} \tag{9.136}$$

   where i is the boolean input and h_0 and $h_0^{\mathtt{C}}$ form an auxiliary link corresponding to this occurrence of the creation action.

7. $(\mathit{this} \rightarrowtail h' \,\texttt{!}\, E(\mathit{expr}_1, \dots, \mathit{expr}_n))^\sim$

$$\begin{aligned} &= \texttt{if } \mathit{this} \rightarrowtail h^{\mathtt{C}'} \texttt{ then skip;} \\ &\quad \texttt{else } \mathit{this} \rightarrowtail h' \,\texttt{!}\, E(\mathit{expr}_1^{\mathtt{C}}, \dots, \mathit{expr}_m^{\mathtt{C}});\ \texttt{fi} \end{aligned} \tag{9.137}$$

where the parameter values d_i are determined as follows.

Let $1 \leq i \leq n$ denote a parameter of E in $\mathscr{M}$ and let $1 \leq j \leq m$ denote the corresponding parameter of E in $\mathscr{M}_D^\sharp$. Then

$$\mathit{expr}_j^{\mathtt{C}} = \mathit{expr}_i^\sim \tag{9.138}$$

and, if the i-th parameter of E in $\mathscr{M}$ is of an object type, then

$$\mathit{expr}_{j+1}^{\mathtt{C}} = \mathit{this} \rightarrowtail h^{\mathtt{C}'}. \tag{9.139}$$

This is the only possible form of expr_j according to Def. 9.3.1.

8. $(\texttt{delete } \mathit{this} \rightarrowtail h')^\sim$

$$= \texttt{if } \mathit{this} \rightarrowtail h^{\mathtt{C}'} \texttt{ then skip; else delete } \mathit{this} \rightarrowtail h';\ \texttt{fi} \tag{9.140}$$

9. $(\texttt{if } \mathit{cond} \texttt{ then } \mathit{act}_1 \texttt{ else } \mathit{act}_2 \texttt{ fi})^\sim$

$$= \texttt{if } \mathit{cond}^\sim \texttt{ then } \mathit{act}_1^\sim \texttt{ else } \mathit{act}_2^\sim \texttt{ fi} \tag{9.141}$$

10. $(\mathit{nav}_1; \mathit{nav}_2)^\sim = \mathit{nav}_1^\sim; \mathit{nav}_2^\sim$

Abbreviations Note that the combination of, for instance, h and $h^{\mathtt{C}}$ keeps track whether h points to $\mathtt{C}$ (in the sense of Chapter 6) or to a concrete individual.

We can introduce the following syntactical (!) abbreviations (or macros) to ease the consistent writing of transformed HLL models.[8]

We write $\mathit{this} \rightarrowtail h' = \mathtt{C}$ and $\mathit{this} \rightarrowtail h' \neq \mathtt{C}$ as an abbreviation for

$$\mathit{this} \rightarrowtail h^{\mathtt{C}'} = 1 \quad \text{and} \quad \mathit{this} \rightarrowtail h^{\mathtt{C}'} = 0. \tag{9.142}$$

Furthermore, we write $\mathit{this} \rightarrowtail h_1' \texttt{ := } \mathit{this} \rightarrowtail h_2'$ as an abbreviation for the sequence

$$\mathit{this} \rightarrowtail h_1' \texttt{ := } \mathit{this} \rightarrowtail h_2'; \mathit{this} \rightarrowtail h_1^{\mathtt{C}'} \texttt{ := } \mathit{this} \rightarrowtail h_2^{\mathtt{C}'}; \tag{9.143}$$

and similarly for the other possible combinations of object type expressions on the left and right-hand side of the assignment (cf. Def. 9.3.1).

We write

$$\mathit{nav} \rightarrowtail \lambda' \texttt{ := } (\mathit{this} \rightarrowtail h' = \mathtt{C}\ ?\ (i_\mu, i_\mu^{\mathtt{C}}) : \mathit{this} \rightarrowtail h' \rightarrowtail \mu) \tag{9.144}$$

[8] And, of course, to further the reader's confusion.

for

$$\begin{aligned} \mathit{nav}\rightarrowtail\lambda' &\mathrel{:=} (\mathit{this}\rightarrowtail h' = \mathtt{C} \mathbin{?} i_\mu : \mathit{this}\rightarrowtail h'\rightarrowtail\mu)\\ \mathit{nav}\rightarrowtail\lambda^{\mathtt{C}'} &\mathrel{:=} (\mathit{this}\rightarrowtail h' = \mathtt{C} \mathbin{?} i^{\mathtt{C}}_\mu : \mathit{this}\rightarrowtail h'\rightarrowtail\mu^{\mathtt{C}}) \end{aligned} \tag{9.145}$$

And we use the common abbreviation `if` *expr* `then` *act*; `fi` for

$$\texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{act};\ \texttt{else skip};\ \texttt{fi}. \tag{9.146}$$

Example 9.3.5 (Artificial)**.** *The transformation of the term*

$$\mathit{this}\rightarrowtail\lambda\rightarrowtail\mu\rightarrowtail x > 0 \tag{9.147}$$

is

$$\begin{aligned} \big(\mathit{this}\rightarrowtail\lambda^{\mathtt{C}} \mathbin{?} (i^{\mathtt{C}}_\mu \mathbin{?} i_x : i_\mu\rightarrowtail x)& \\ : (\mathit{this}\rightarrowtail\lambda\rightarrowtail\mu^{\mathtt{C}} \mathbin{?} i_x : \mathit{this}\rightarrowtail\lambda\rightarrowtail\mu\rightarrowtail x)\big) &> 0 \end{aligned} \tag{9.148}$$

where $i_\mu, i^{\mathtt{C}}_\mu$ is a pair of input link and boolean input variable corresponding to μ and i_x is an input corresponding to x.

Equation (9.147) *reads out loud roughly as follows. If this$\rightarrowtail\lambda$ points to a non-concrete individual, as indicated by this$\rightarrowtail\lambda^{\mathtt{C}}$ being set, then navigating μ from there on may yield anything, because a non-concrete individual doesn't have concrete information. Thus we consider the input pair $i_\mu, i^{\mathtt{C}}_\mu$ as providing the result of the navigation.*

The result may again be a non-concrete individual, as indicated by $i^{\mathtt{C}}_\mu$, then we're using any value x could possibly have, provided by the input i_x. Only the last branch does regular navigation as it guarantees that both λ and μ point to concrete individuals.

In navigation expression normal form this becomes

$$\begin{aligned} &\textbf{\textit{let}}\ \mathit{this}\rightarrowtail h'_3 \mathrel{:=} \mathit{this}; \mathit{this}\rightarrowtail h^{\mathtt{C}'}_3 \mathrel{:=} 0;\\ &\quad \mathit{this}\rightarrowtail h'_2 \mathrel{:=} (\mathit{this}\rightarrowtail h^{\mathtt{C}'}_3 \mathbin{?} i_\lambda : \mathit{this}\rightarrowtail h'_3\rightarrowtail\lambda;);\\ &\quad \mathit{this}\rightarrowtail h^{\mathtt{C}'}_2 \mathrel{:=} (\mathit{this}\rightarrowtail h^{\mathtt{C}'}_3 \mathbin{?} i^{\mathtt{C}}_\lambda : \mathit{this}\rightarrowtail h'_3\rightarrowtail\lambda^{\mathtt{C}});\\ &\quad \mathit{this}\rightarrowtail h'_1 \mathrel{:=} (\mathit{this}\rightarrowtail h^{\mathtt{C}'}_2 \mathbin{?} i_\mu : \mathit{this}\rightarrowtail h'_2\rightarrowtail\mu);\\ &\quad \mathit{this}\rightarrowtail h^{\mathtt{C}'}_1 \mathrel{:=} (\mathit{this}\rightarrowtail h^{\mathtt{C}'}_2 \mathbin{?} i^{\mathtt{C}}_\mu : \mathit{this}\rightarrowtail h'_2\rightarrowtail\mu^{\mathtt{C}});\\ &\textbf{\textit{in}}\ \neg(\mathit{this}\rightarrowtail h^{\mathtt{C}'}_1 \mathbin{?} i_x : \mathit{this}\rightarrowtail h^{\mathtt{C}}_1\rightarrowtail x > 0) \end{aligned} \tag{9.149}$$

which significantly longer than (9.147)*, but we consider it to be more readable because each input occurs exactly once and each pair of lines conducts only a single navigation step. In addition, independent from the readability of the result, we consider it significantly easier to* define *the transformation for the normal form.*

With the just introduced abbreviations, it turns into the slightly more readable form

$$\begin{aligned}
&\texttt{let}\ \mathit{this}\rightarrowtail h_3'\ := \mathit{this}; \mathit{this}\rightarrowtail h_3^{\mathsf{C}'}\ := 0;\\
&\quad \mathit{this}\rightarrowtail h_2'\ := (\mathit{this}\rightarrowtail h_3{}' = \mathsf{C}\ ?\ (i_\lambda, i_\lambda^{\mathsf{C}}) : \mathit{this}\rightarrowtail h_3'\rightarrowtail \lambda;);\\
&\quad \mathit{this}\rightarrowtail h_1'\ := (\mathit{this}\rightarrowtail h_2{}' = \mathsf{C}\ ?\ (i_\mu, i_\mu^{\mathsf{C}}) : \mathit{this}\rightarrowtail h_2'\rightarrowtail \mu);\\
&\texttt{in}\ \neg(\mathit{this}\rightarrowtail h_1^{\mathsf{C}'}\ ?\ i_x : \mathit{this}\rightarrowtail h_1^{\mathsf{C}}\rightarrowtail x > 0)
\end{aligned} \tag{9.150}$$

$\diamondsuit$

Example 9.3.6. *DTR for Car Platooning Merge For example consider the label*

$$\mathit{req}(p)/\mathit{flw}'\ := p; p\,!\,\mathit{ack}() \tag{9.151}$$

of the transition from local state fa to ld (cf. Figure 9.2).

Its navigation expression normal form is

$$\begin{aligned}
&\mathit{req}(p)/\mathit{this}\rightarrowtail h_1'\ := p; \mathit{this}\rightarrowtail h_2'\ := \mathit{this};\\
&\qquad \mathit{this}\rightarrowtail h_2'\rightarrowtail \mathit{flw}'\ := \mathit{this}\rightarrowtail h_1'; \mathit{this}\rightarrowtail h_1'\,!\,\mathit{ack}()
\end{aligned} \tag{9.152}$$

according to Sections 9.3.1 and 9.3.1.

The DTR transformation turns it into

$$\begin{aligned}
&\mathit{req}(p, p^{\mathsf{C}})/\mathit{this}\rightarrowtail h_1'\ := p; \mathit{this}\rightarrowtail h_1^{\mathsf{C}'}\ := p^{\mathsf{C}};\\
&\qquad \mathit{this}\rightarrowtail h_2'\ := \mathit{this}; \mathit{this}\rightarrowtail h_2^{\mathsf{C}'}\ := 0;\\
&\qquad \texttt{if}\ \mathit{this}\rightarrowtail h_2^{\mathsf{C}'}\ \texttt{then}\ \texttt{skip};\ \texttt{else}\\
&\qquad\quad \mathit{this}\rightarrowtail h_2'\rightarrowtail \mathit{flw}'\ := \mathit{this}\rightarrowtail h_1'; \mathit{this}\rightarrowtail h_2'\rightarrowtail \mathit{flw}^{\mathsf{C}'}\ := \mathit{this}\rightarrowtail h_1^{\mathsf{C}'};\\
&\qquad \texttt{fi};\\
&\qquad \texttt{if}\ \mathit{this}\rightarrowtail h_1^{\mathsf{C}'}\ \texttt{then}\ \texttt{skip};\ \texttt{else}\ \mathit{this}\rightarrowtail h_1'\,!\,\mathit{ack}();\ \texttt{fi}
\end{aligned} \tag{9.153}$$

because all navigation expressions of object type, in particular event parameters, obtain a companion p^{C}*. The assignment is modified according to* (9.135)*.*

With the just introduced abbreviations, it turns into the slightly more readable form

$$\begin{aligned}
&\mathit{req}(p, p^{\mathsf{C}})/\mathit{this}\rightarrowtail h_1'\ := p;\\
&\qquad \mathit{this}\rightarrowtail h_2'\ := \mathit{this};\\
&\qquad \texttt{if}\ \mathit{this}\rightarrowtail h_2 \neq \mathsf{C}\ \texttt{then}\ \mathit{this}\rightarrowtail h_2'\rightarrowtail \mathit{flw}'\ := \mathit{this}\rightarrowtail h_1';\ \texttt{fi};\\
&\qquad \texttt{if}\ \mathit{this}\rightarrowtail h_1 \neq \mathsf{C}\ \texttt{then}\ \mathit{this}\rightarrowtail h_1'\,!\,\mathit{ack}();\ \texttt{fi}
\end{aligned} \tag{9.154}$$

$\diamondsuit$

Note that, as we'll see in Chapter 10, the Car Platooning protocol of Figure 9.2 is a typical example for the DCS language, which chooses a very closed view on class instances in the sense that class instances are not allowed to read or modify each other's local variables or links. This is achieved by exchanging identities exclusively via event parameters and having updates only for own local states and links.

That is, DCS doesn't make use of assignments like this one

$$\mathit{this} \rightarrowtail \lambda \rightarrowtail x_1' := \mathit{this} \rightarrowtail \mu \rightarrowtail x_2, \tag{9.155}$$

which sets local variable x_1 of the individual denoted by $\mathit{this} \rightarrowtail \lambda$, which may well be different from *this*, to the value of x_2 in $\mathit{this} \rightarrowtail \mu$.

For this reason, there is enormous potential for optimisation in the transformation, for example, the first condition in (9.154) always evaluates to 1 in C and to 0 in $C^{\mathtt{C}}$. We'll discuss this briefly in Section 9.3.3.

Technical Optimisations

Note that the there's large room for optimisation in the above transformation. The first aim here is to support the presentation and the discussion of soundness in Section 9.3.3, instead of being optimal in any way. Although this comes for the price of precision as discussed in Section 9.3.3.

The most obvious issue is the inflation of duplicate auxiliary variables and inputs. For example, we have for simplicity a dedicated auxiliary variable per appearance of a navigation expression and we have the redundant rewriting of

$$\mathit{this} \qquad\qquad \text{to} \qquad\qquad \mathit{this} \rightarrowtail h' := \mathit{this}; \mathit{this} \rightarrowtail h' \tag{9.156}$$

in order to have fewer cases to distinguish.

The latter can easily be removed by re-substituting *this* for h after the transformation. The former can be addressed by a more sophisticated normalisation scheme taking care of common sub-expressions and sharing auxiliary variables between multiple different occurrences of expressions, which would also improve precision (cf. Section 9.3.3).

Similarly, if it's beneficial to minimise the number of employed inputs, they can often be shared between different transition programs, and even within the same one if they're independent. For example, the transition program

$$\begin{aligned} &\mathtt{if}\ \mathit{expr}\ \mathtt{then}\ \mathit{this} \rightarrowtail x' := \mathit{this} \rightarrowtail h_0' \rightarrowtail x; \\ &\mathtt{else}\ \mathit{this} \rightarrowtail y' := \mathit{this} \rightarrowtail h_1' \rightarrowtail y;\ \mathtt{fi} \end{aligned} \tag{9.157}$$

is transformed into

$$\begin{aligned} &\mathtt{if}\ \mathit{expr}\ \mathtt{then}\ \mathit{this} \rightarrowtail x' := \left(\mathit{this} \rightarrowtail h_0^{\mathtt{C}'}\ ?\ i_0 : \mathit{this} \rightarrowtail h_0' \rightarrowtail x;\right); \\ &\quad \mathtt{else}\ \mathit{this} \rightarrowtail y' := \left(\mathit{this} \rightarrowtail h_1^{\mathtt{C}'}\ ?\ i_1 : \mathit{this} \rightarrowtail h_1' \rightarrowtail y;\right);\ \mathtt{fi} \end{aligned} \tag{9.158}$$

Now if x and y are of the same type τ, then a single input i of type τ can be employed replacing both, i_0 and i_1, because they're read in independent branches of the conditional

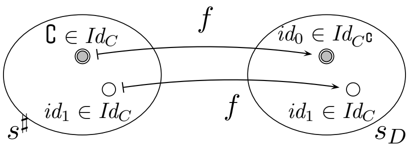


Figure 9.8.: **Simulation Relation.** State (and topology) $s^\sharp$ is in simulation relation with s_D. The differences are that $\mathtt{C}$ in $s^\sharp$ is labelled with local state $\top$ by Chapter 6 while id_0 has a regular local state and that $\mathtt{C}$ is an identity from sort Id_C while id_0 is an identity from the sort corresponding to class $C^{\mathtt{C}}$ as added in the transformation procedure. Note that the *this* links are omitted in the pictures.

statement. This independence is necessary for soundness, otherwise we may lack abstract transitions in M_D that are actually possible in the concrete M.

An even more sophisticated optimisation would consider as few inputs as possible and of as limited domain as possible. For example, instead of turning

$$\mathit{this} \rightarrowtail h' \rightarrowtail x > 0 \tag{9.159}$$

into

$$\Big(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \ ?\ i_0 : \mathit{this} \rightarrowtail h' \rightarrowtail x\Big) > 0 \tag{9.160}$$

with i_0 of the same integer type as x, it can be turned into

$$\mathit{this} \rightarrowtail h^{\mathtt{C}'} \ ?\ i_1 : (\mathit{this} \rightarrowtail h' \rightarrowtail x > 0) \tag{9.161}$$

with a boolean i_1.

The general principle is to shift consideration of inputs outwards in the parameters of function symbols or logical expressions .

Soundness

Lemma 9.3.7 (Soundness of Syntactic DTR)**.** *Let $\mathscr{M}$ be an HLL model in navigation expression normal form and let $\mathcal{E}$ be an ether, $\mathcal{S}$ an interleaving scheduler, $\mathcal{O}$ an input and creation scheduler, and ι an interpretation of function symbols.*

Let D be a DTR and $\mathscr{M}_D^\sharp$ the HLL model obtained by the transformation from Sections 9.3.3 and 9.3.3. Then

$$D(\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})) \preceq \iota[\![\mathscr{M}_D^\sharp]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}^\sharp)) \tag{9.162}$$

where the latter uses the identities provided by D as the domain $\mathcal{D}(\tau_C)$ of classes C and where $\mathcal{O}^\sharp$ corresponds to $\mathcal{O}$ except that it, in each state, permits to create any individual at any time and provides any value for the inputs introduced in the transformation to $\mathscr{M}_D^\sharp$. ♢

Proof. See Section A.4. □

Note that Lemma 9.3.7 doesn't require the scheduler, oracle, etc. and HLL model to be symmetric. Recall from Chapter 8 that DTR can well be applied independent from QR. But only in combination do we reach a finite set of representative cases for which we have finite abstractions.

Thus, the workflow from Chapter 8 remains unchanged:

1. Verify that the HLL model $\mathscr{M}$ is symmetric.
2. Choose (finitely many) representative cases depending on considered property or by other heuristics.
3. For each case, verify a DTR abstraction $\mathscr{M}_D^\sharp$.

Finiteness

Recall from Chapters 6 and 8 that the DTR abstraction of an ETTS is finite if there is an upper bound on the number of outgoing links and if the set of local states per individual is finite.

The first requirement is satisfied as we consider single link HLL models since Section 9.1.1. But the second requirement is a bit intricate because the local state of an individual is with HLL not only determined by the local variables, but also by the ether. We'll briefly re-consider this issue in Section 9.4.3 below.

The basic conclusion is that we have to require that the ether is somehow bounded, that is, either we know that the HLL model doesn't "flood" queues or we assume a finite ether which dismisses events after the queue is filled. Or that we have to apply another, orthogonal, abstraction to treat unbounded ethers, for example in case the ether behaves like ordinary FIFO message queues, there are numerous proposals of abstractions, for instance, representing queue content by regular expressions.

The bottom line is to note (again) that the QR/DTR approach only treats the unboundedness stemming from the topology structure, that is, from unbounded creation and destruction of individuals, but not other sources of unboundedness like infinite-domain local variables, or hybrid behaviour, or real-time.

Levels of Precision

By Lemma 9.3.7, we have that the ETTS M_D of the syntactically transformed HLL model $\mathscr{M}_D^\sharp$ simulates the DTR abstraction $D(M)$ of the ETTS of the original $\mathscr{M}$. The immediate next question is the other direction, that is, whether $D(M)$ simulates M_D, which would tell us that we obtain *exactly* $D(M)$ by our syntactical transformations from this section.

Given *exactly* the introduced syntactical transformations, the answer is negative. We'll see why this is the case and in how far we can improve the transformation in the following paragraphs.

The last paragraph of this section discusses further possible improvements of the transformation, which don't directly affect the precision but may lead to more compact

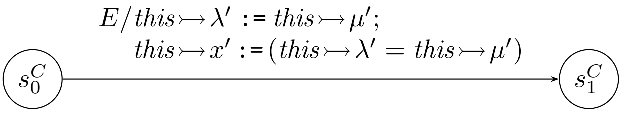

$E/\mathit{this}\rightarrowtail\lambda' := \mathit{this}\rightarrowtail\mu';$
$\mathit{this}\rightarrowtail x' := (\mathit{this}\rightarrowtail\lambda' = \mathit{this}\rightarrowtail\mu')$
s_0^C
s_1^C

Figure 9.9.: **One Transition of C.**

encodings in Section 9.4. Some of the countermeasures have already been identified in [178], yet at that time only under the aspect of reducing complexity in the employed array program encoding (cf. Section 9.4).

We can understand why, in general, we don't have $M_D \simeq D(M) = M^\sharp$, i.e. bisimulation, by considering what we would need to prove this relation. Namely given the simulation relation H from Lemma 9.3.7 and a pair of states $(s^\sharp, s_D) \in H$, we have to show that if there is a transition from s_D to s'_D in M_D, then there is a transition from $s^\sharp$ to a H-related state $s^{\sharp\prime}$ in $M^\sharp$.

This is in general not the case for two reasons

1. $M^\sharp$ reflects *aliasing* in action sequences, that is basically reachability of the same individual by different navigation expressions established by a single local transition.

2. $M^\sharp$ reflects properties of graphs, like transitivity, because topologies in $M^\sharp$ *are* graphs with regular links, while there is the artificial encoding with other-flags in $\mathscr{M}_D^\sharp$.

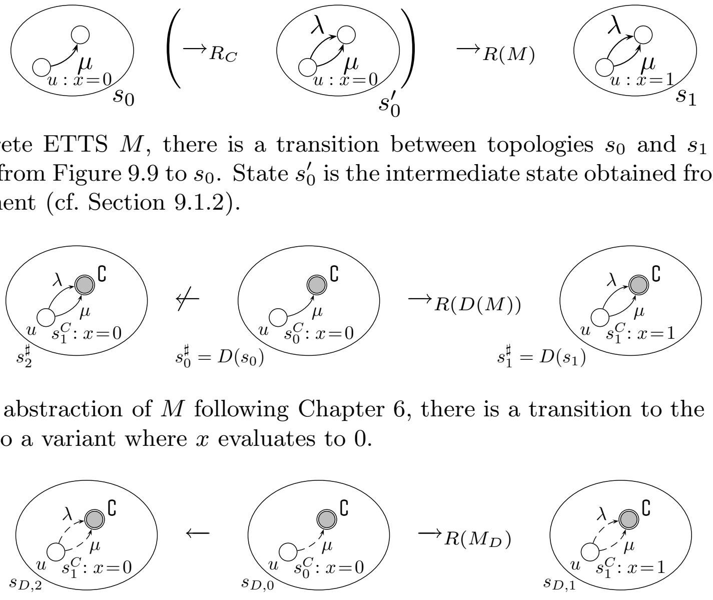
In the following, we'll have examples for both cases and discuss countermeasures in the transformation.

Aliasing Aliasing in action sequences is probably best understood by way of an example. Assume an HLL model $\mathscr{M}$ with a class C having two links λ and μ, both of object type τ_C, that is, pointing to instances of class C, and a local variable x of a boolean type. Further assume that there are at least two states s_0^C, $s_1^C \in S_C$ and we have at least the transition shown in Figure 9.9, that is, when receiving an event E, we change state to s_1 and assign to x whether λ points to the same individual as μ. As λ is updated first, and because the primed variants are compared, the outcome is actually fixed: x necessarily obtains value 1, independent from the remaining local state of the individual for which the transition is executed and independent from the global state in which it executes.

This is depicted in Figure 9.10(a). Let M be the ETTS of $\mathscr{M}$. Then s_0 is a (global) state (or topology) of M with individual u in a state where the local state is s_0^C (not shown in the picture), x is 0, μ points to a different individual, and λ is not set.

Assuming an event E ready in the ether (also not shown in the picture), there is a M-transition to state s_1 by the (local) transition from Figure 9.9 following Section 9.1.2.

The first of the local micro-steps is shown in parentheses as s'_0: the link λ is updated to point to the individual denoted by μ.



(a) In the concrete ETTS M, there is a transition between topologies s_0 and s_1 by applying the action sequence from Figure 9.9 to s_0. State s_0' is the intermediate state obtained from s_0 by applying the first assignment (cf. Section 9.1.2).

(b) In the DTR abstraction of M following Chapter 6, there is a transition to the abstraction of s_1 above, but not to a variant where x evaluates to 0.

(c) In the abstraction of M obtained via $\mathscr{M}_D^\sharp$, the transition to the variant of $s_{D,1}$ where x evaluates to 0 exists. The identity id_0 belongs to the class $C^{\mathtt{C}}$ introduced by the transformation process.

Figure 9.10.: **Aliasing** in action sequences.

Then the assignment of x yields 1 because the comparison between λ and μ is carried out in s_0' where they're equal. This outcome is, as said above, obviously independent from the individual and from its relation to others.

By Chapter 6, the state set of the DTR abstraction is defined as the DTR'd topologies of the original systems and transitions are introduced by an exists/exists pattern. That is, there is certainly a transition between the abstract states $D(s_0)$ and $D(s_1)$ in $D(M)$ because s_0 and s_1 are concretisations of these two states and in transition relation (cf. Figure 9.10(b)).

But there is *no* transition from $s_0^\sharp$ to the abstract state $s_2^\sharp$, which differs from $s_1^\sharp$ only in x having value 0 for u, because in M there is no transition between any concretisation of $s_0^\sharp$ and $s_2^\sharp$.

In the ETTS of $\mathscr{M}_D^\sharp$, the corresponding transition does exist (cf. Figure 9.10(c)) for the following reason. Consider the action sequence of the transition in Figure 9.9. In normal form, the second assignment becomes

$$\mathit{this} \rightarrowtail h_0 \rightarrowtail x' \mathrel{:=} (\mathit{this} \rightarrowtail h_1' = \mathit{this} \rightarrowtail h_2') \tag{9.163}$$

which, by (9.128) becomes

$$\begin{aligned} \mathit{this} \rightarrowtail h_0 \rightarrowtail x' \mathrel{:=} \Big(& \mathit{this} \rightarrowtail h_1^{\mathtt{C}'} \land \mathit{this} \rightarrowtail h_2^{\mathtt{C}'} \, ? \\ & i : \big(\neg \mathit{this} \rightarrowtail h_1^{\mathtt{C}'} \land \neg \mathit{this} \rightarrowtail h_2^{\mathtt{C}'} \, ? \, \mathit{this} \rightarrowtail h_1' = \mathit{this} \rightarrowtail h_2' : 0\big)\Big) \end{aligned} \tag{9.164}$$

where i is the boolean input introduced for this term.

When executing (9.164) on $s_{D,0}$, both h_1 and h_2 have the other-flag set, thus the outcome of the comparison is given by i, which may well *not* hold, thus assigning 0 to x and leading to the global state $s_{D,2}$.

The point is that the DTR of Chapter 6 operates on the ETTS where certain aspects of the HLL model are already lost, or have become atomic. For example, the kind of aliasing just discussed above. The syntactically obtained abstraction has access to each HLL action, it has a non-atomic view, which may give rise to additional spurious behaviour.

A similar, but slightly different example would be the sequence of actions

$$\begin{aligned} \mathit{this} \rightarrowtail x' &:= \mathit{this} \rightarrowtail h' \rightarrowtail x'; \\ \mathit{this} \rightarrowtail y' &:= \mathit{this} \rightarrowtail h' \rightarrowtail x' \end{aligned} \tag{9.165}$$

by which the local variables x and y obtain the same value in the concrete HLL.

After a DTR transformation, we have something like

$$\begin{aligned} \mathit{this} \rightarrowtail x' &:= (\mathit{this} \rightarrowtail h^{\mathtt{C}'} \;?\; i_1 : \mathit{this} \rightarrowtail h' \rightarrowtail x'); \\ \mathit{this} \rightarrowtail y' &:= (\mathit{this} \rightarrowtail h^{\mathtt{C}'} \;?\; i_2 : \mathit{this} \rightarrowtail h' \rightarrowtail x') \end{aligned} \tag{9.166}$$

where i_1 and i_2 are *different* inputs corresponding to the different occurrences of navigation expressions. Thus in M_D there will be spurious transitions leading to all combinations of different values in x and y.

Instances of the latter example can be solved by a more restrictive normal form. Namely, one has to ensure that for a common expression, always the same input is considered.

This can be achieved by extending the normal form such that auxiliary variables are used for common expressions, for example, (9.165) would turn into

$$\begin{aligned} \mathit{this} \rightarrowtail h_0' &:= \mathit{this} \rightarrowtail h' \rightarrowtail x'; \\ \mathit{this} \rightarrowtail x' &:= \mathit{this} \rightarrowtail h_0'; \\ \mathit{this} \rightarrowtail y' &:= \mathit{this} \rightarrowtail h_0' \end{aligned} \tag{9.167}$$

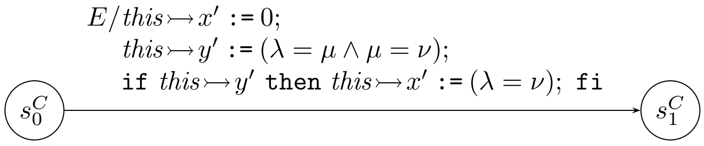
Then the unchanged transformation procedure yields

$$\begin{aligned} \mathit{this} \rightarrowtail h_0' &:= (\mathit{this} \rightarrowtail h^{\mathtt{C}'} \;?\; i_1 : \mathit{this} \rightarrowtail h' \rightarrowtail x'); \\ \mathit{this} \rightarrowtail x' &:= \mathit{this} \rightarrowtail h_0'; \\ \mathit{this} \rightarrowtail y' &:= \mathit{this} \rightarrowtail h_0' \end{aligned} \tag{9.168}$$

which preserves that property that x and y obtain the same value. Thus the ETTS of this $\mathscr{M}_D^\sharp$ won't have the spurious transition to states with different values of x and y.

The former example can be treated similarly but in addition requires a modification of the transformation procedure. Firstly, one would require to have the second action from Figure 9.9 in a normal form like

$$\mathit{this} \rightarrowtail x' := (\mathit{this} \rightarrowtail h' = \mathit{this} \rightarrowtail h'), \tag{9.169}$$



$E/\mathit{this}\rightarrowtail x' := 0;$
$\quad \mathit{this}\rightarrowtail y' := (\lambda = \mu \wedge \mu = \nu);$
$\quad$ `if` $\mathit{this}\rightarrowtail y'$ `then` $\mathit{this}\rightarrowtail x' := (\lambda = \nu);$ `fi`

Figure 9.11.: **One Transition of C.**

that is, identify that the same value is compared for equality.

Then a modified transformation could directly yield

$$\mathit{this}\rightarrowtail x' := 1. \tag{9.170}$$

This is basically a simplification of expressions based on a static analysis of the transition program, a general topic of compiler theory with differently expensive and precise solutions which can also be applied to HLL transition programs.

What the classes of reasons for spurious behaviour defined by the two examples intuitively have in common, is that M_D is the more precise, the more the transition programs define common expressions only once and avoid aliasing within one and the same transition program.

Consequently we conjecture, that in the extreme case, that is, for an HLL with only a single action per transition *and* without impacts of graph properties as discussed in the following, the abstract transition system M_D will bisimulate the DTR abstraction $D(M)$ obtained by Chapter 6.

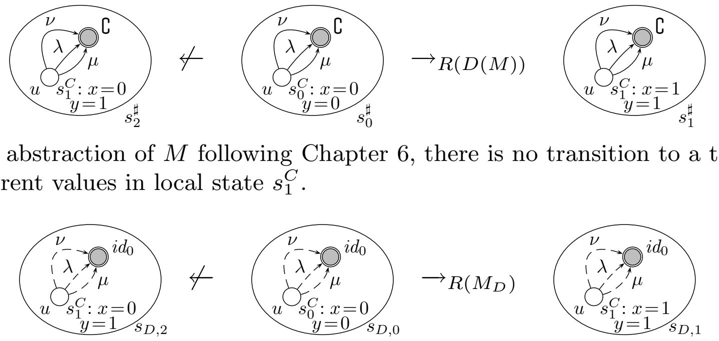
Yet HLL of this form cannot be recommended because splitting all actions into atoms typically first of all yields a significantly increased number of local states which multiplicate up when forming the global state space. Secondly, it is desirable to leave as much actions as possible together at a transition because these combinations are preserved (or reflected) by the syntactical DTR. This is discussed in more detail in Section 9.3.3 below.

Graph Properties A second difference between the abstract ETTS $M^\sharp$ obtained via Chapter 6 and M_D obtained via $\mathscr{M}_D^\sharp$ is that in M_D, when considering links to C (encoded by other-flags) the worst case is assumed in form of inputs. This doesn't preserve some properties of graphs, which do propagate to $M^\sharp$ similar to the first example for aliasing above.

The following example shows how this difference can lead to spurious transitions in M_D. Assume an HLL model $\mathscr{M}$ with a class C having three links λ, μ, and ν, all of object type τ_C, that is, pointing to instances of class C, and two local variables x and y of a boolean type.

Further assume that there are at least two states s_0^C, $s_1^C \in S_C$ and we have at least the transition shown in Figure 9.11, that is, when receiving an event E, we change state to s_1 and assign to x whether λ points to the same individual as ν.

In M, the ETTS of $\mathscr{M}$, the only possible valuations of x and y in local state s_1^C are $x = y = 0$ and $x = y = 1$ because equality in topologies is transitive, the truth of the expression assigned to x is implied by the expression assigned to y.



(a) In the DTR abstraction of M following Chapter 6, there is no transition to a topology where x and y have different values in local state s_1^C.

(b) In the abstraction of M obtained $\mathscr{M}_D^\sharp$, the transition to the topology where x and y evaluate differently exists.The identity id_0 belongs to the class $C^{\mathtt{C}}$ introduced by the transformation process.

Figure 9.12.: **Graph properties** in action sequences.

Similarly to the examples in the previous section (cf. Figure 9.10), there is no transition to a situation where u is in local state s_1^C and x and y have different values (cf. Figure 9.12(a)). In M_D, this transition is present (cf. Figure 9.12(b)). The reason looks similar to the second example of aliasing above, namely the action sequence from Figure 9.11 is turned into

$$
\begin{aligned}
&\mathit{this}\rightarrowtail h_0' \mathrel{:=} \mathit{this}\rightarrowtail \lambda; \mathit{this}\rightarrowtail h_1' \mathrel{:=} \mathit{this}\rightarrowtail \mu; \mathit{this}\rightarrowtail h_2' \mathrel{:=} \mathit{this}\rightarrowtail \nu;\\
&\mathit{this}\rightarrowtail y' \mathrel{:=} \Big(h_0^{\mathtt{C}'} \wedge h_1^{\mathtt{C}'} \mathbin{?} i_0 : \big(\neg h_0^{\mathtt{C}'} \wedge \neg h_1^{\mathtt{C}'} \mathbin{?} h_0' = h_1' : 0\big)\Big)\\
&\qquad\qquad \wedge \Big(h_1^{\mathtt{C}'} \wedge h_2^{\mathtt{C}'} \mathbin{?} i_1 : \big(\neg h_1^{\mathtt{C}'} \wedge \neg h_2^{\mathtt{C}'} \mathbin{?} h_1' = h_2' : 0\big)\Big); \qquad (9.171)\\
&\texttt{if } \mathit{this}\rightarrowtail y' \texttt{ then}\\
&\quad \mathit{this}\rightarrowtail x' \mathrel{:=} \Big(h_0^{\mathtt{C}'} \wedge h_2^{\mathtt{C}'} \mathbin{?} i_2 : \big(\neg h_0^{\mathtt{C}'} \wedge \neg h_2^{\mathtt{C}'} \mathbin{?} h_0' = h_2' : 0\big)\Big); \texttt{ fi}
\end{aligned}
$$

in $\mathscr{M}_D^\sharp$ by the syntactical transformation (9.128) where i_0, i_1, i_2 are inputs corresponding to the particular occurrences of expressions.

With $i_0 = i_1 = 1$ and $i_2 = 0$ we can justify the spurious transition to $s_{D,2}$ in Figure 9.12(b).

The reason only *looks* similar to the second example of aliasing above because it cannot be treated by ensuring that common expressions are accessed via the same auxiliary variable. This is already the case for (9.171), there are no duplicate expressions. The reason for the spuriousness is that the uninformed usage of inputs doesn't respect properties of graphs as, for instance, transitivity of the comparison for equality.

This issue of the example can be addressed by grouping all comparisons of links and considering (by external assumptions, as in Section 6.4 on approaches to refinement) only those adhering to the rules in graphs.

In the example, we'd have

$$\begin{aligned}
&\mathit{this}{\rightarrowtail}x' := 0; \mathit{this}{\rightarrowtail}h_0' := (\lambda = \mu);\\
&\mathit{this}{\rightarrowtail}h_1' := (\mu = \nu); \mathit{this}{\rightarrowtail}h_2' := (\lambda = \nu);\\
&\mathit{this}{\rightarrowtail}y' := (\mathit{this}{\rightarrowtail}h_0' \wedge \mathit{this}{\rightarrowtail}h_1');\\
&\mathtt{if}\ \mathit{this}{\rightarrowtail}y'\ \mathtt{then}\ \mathit{this}{\rightarrowtail}x' := \mathit{this}{\rightarrowtail}h_2';\ \mathtt{fi}
\end{aligned} \tag{9.172}$$

with boolean auxiliary variables h_0, h_1, h_2. The external assumption would comprise

$$h_0 \wedge h_1 \implies h_2. \tag{9.173}$$

We conjecture that the approach generalises to single link HLL but will benefit from a normal form addressing aliasing as discussed in the previous section. Together with this treatment, we further conjecture that we find M_D to simulate $D(M)$.

Another, far more drastic approach, would be to only consider HLL *without* comparison for equality – or, in other words, the fewer comparisons for equality in HLL the less spurious is M_D compared to $D(M)$.

Towards Bisimulation Given the absence of aliasing and the preservation of graph properties, we conjecture that the bisimulation proof as outlined above is possible and yields that $M_D \simeq D(M)$ basically *is* $M^\sharp$.

Then given states s_D, s_D' in the transition relation of M_D, we can consider the valuations of inputs and the branches of conditional expressions which justify this transition and can re-construct a topology of $M^\sharp$, because absence of aliasing and preservation of graph properties is supposed to ensure that such a topology exists. Subsequently, we show that applying the scheduled transition program actually leads to a topology, which is in simulation relation with s_D'.

Reflection Properties

From the construction it is obvious that $C^{\mathtt{C}}$ is far from complete chaos. Instead, it adheres to the sequence of actions given by transition programs.

For example, if an event E_1 is sent only on the same transition as an E_2 event, then there will never be an alone E_1 in the abstract system.

And if some events are not sent at all by instances of a particular class, then they'll also not sent in the abstract transition system. Thereby, the environment of the concrete individuals provided by the non-concrete instances automatically adhere to some ordering invariants of the system.

9.3.4. Full Concurrency

In Section 9.3.3 we've discussed the interleaving case with the outcome that it's principally possible to obtain $D(M)$ (or something arbitrarily close to it) by syntactical modification of the HLL model and the regular ETTS semantics of HLL. The intuition is, that if only one "thing" can happen at a time, then this thing is either happening

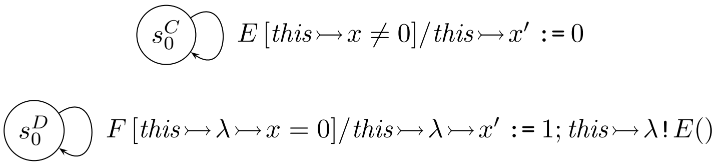
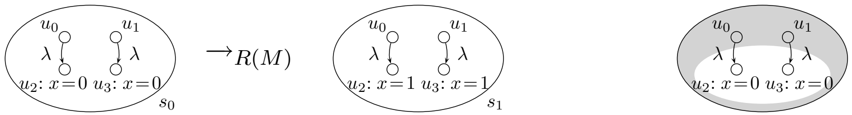

s_0^C
$E\,[\mathit{this}\rightarrowtail x \neq 0]/\mathit{this}\rightarrowtail x' := 0$
s_0^D
$F\,[\mathit{this}\rightarrowtail\lambda\rightarrowtail x = 0]/\mathit{this}\rightarrowtail\lambda\rightarrowtail x' := 1; \mathit{this}\rightarrowtail\lambda\,!\,E()$

Figure 9.13.: **Transitions of C and D.**



(a) Two instances u_0 and u_1 of class D and two instances u_2 and u_3 of class C.

(b) Putting a spotlight on only the instances of class C.

Figure 9.14.: **Full concurrency example.**

driven by a concrete individual or by another one. That is, the overall behaviour is an arbitrarily sequence of concrete and non-concrete transitions, and all non-concrete transitions with an effect to the concrete part can be mimicked by scheduling the instance of the additional class C^{C}. With full concurrency, the situation is significantly harder.

General HLL Example

As an example, assume an HLL model $\mathscr{M}$ with at least two classes C and D. Class C has a local boolean variable x and D a link λ pointing to C objects. Both classes have a single state with a single transition as shown in Figure 9.13.

Further assume a topology comprising s_0 shown in Figure 9.14(a) and concurrency, that is, multiple individuals may take local transitions simultaneously in each global state.

Now if there is one F-event ready to consume for each of the two shown D instances in s_0, and if u_0 and u_1 in Figure 9.14(a) consume the event simultaneously, then they simultaneously update the x variable of their C-object to 0 and send an E-event, leading to global state s_1 Figure 9.14(a) where the local variables of both C-objects have value 1.

Turning to a syntactically obtained abstraction, assume the DTR is $D = \{(\{u_3, u_4\}, Id_C)\}$, that is, considers only class C (cf. Figure 9.14(b)).

Then we obtain a topology like $s_{D,0}$ in Figure 9.15 if we naïvely apply the transformation of Section 9.3.3, note that there is only a single instance of the additional class

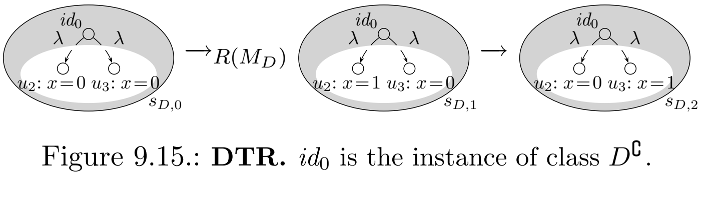


Figure 9.15.: **DTR.** id_0 is the instance of class D^{C}.

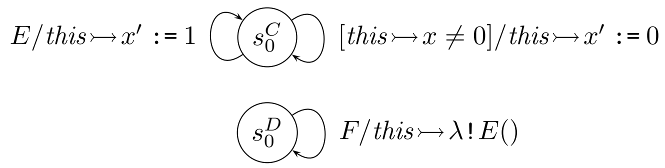

$E/\mathit{this}{\rightarrowtail}x' := 1$
s_0^C
$[\mathit{this}{\rightarrowtail}x \neq 0]/\mathit{this}{\rightarrowtail}x' := 0$
s_0^D
$F/\mathit{this}{\rightarrowtail}\lambda\,!\,E()$

Figure 9.16.: **Transitions of** C **and** D without direct manipulation.

D^{C}. Thus with concurrent execution, at most *one* of the C-instances is modified at a time, and at the same time receives an E-event, for example leading to $s_{D,1}$.

Hence before changing the second one, the first one will have changed back its local state by its transition (cf. Figure 9.13), for example leading to $s_{D,2}$ in Figure 9.15. That is, the abstract representation of s_0 from Figure 9.14(a) is not reachable at all in M_D in this case, thus the syntactical abstraction is in general not sound for concurrent schedulings. The abstract transition system $D(M)$ obtained by Chapter 6 in contrast is sound, it has by definition the transition between the DTR abstractions of the two topologies from Figure 9.14(a).[9]

HLL Without Direct Manipulation

A nasty aspect of the example in Section 9.3.4 is that the objects of class D manipulate the local state of C objects directly via links. On the one hand, this is not completely uncommon, as the C's may be subsidiaries of the D's and thus naturally be manipulated by their "owners". On the other hand, it's against the well-known information hiding paradigm, so one could instead require D-objects to only interact with C-objects via a certain interfaces, maybe even restricted to message communication.

Unfortunately, such a restriction is not strong enough. We still only observe at most one event sending in $s_{D,0}$ while two simultaneous events are possible in s, thus we still lose a transition in the abstract and are unsound. As an example consider the modified local behaviour of C and D as given by Figure 9.16. Now the C objects act completely local, but still a single D^{C} instance is not sufficient to reach the abstraction of s_1 from $s_{D,0}$.

Syntactic Transformation and Concurrency

From the understanding we gained in the previous chapters, we are able to analyse the underlying problem as sketched in this section's introduction. The point is that there is only one control flow of C.

This is fine with interleaving because there is always only one point in time where the environment can interact with a single individual, and we can schedule the C as often as we like to serve all individuals. With true concurrency we've got to preserve the possible *simultaneous* interactions.

[9] As pointed out by one of the referees, this problem seems to be connected to the ordering of choice and branching in the IO and OI hierarchies.

An easy criterion to ensure this is to require that the inhabitants of the shadows, i.e., the non-concrete objects, are passive, that is, only read and written by the concrete ones but don't take own transitions. It's clear that if there is no interaction *originating* from any non-concrete object, then nothing is lost in the syntactically obtained abstraction.

A more complex criterion would be to identify, or to analyse for, dependencies. In the examples of Sections 9.3.4 and 9.3.4, we can see that each C is influenced by exactly one D. Consequently, when reducing C to $\{u_1, u_2\}$ by DTR, there need to be two "instances" of D^{C} executing simultaneously. This is in general non-trivial because principally, a system can, by exchanging identities, arrange that any individual in the system has a link to the ones in the spotlight and interacts with them.

On the other hand, the relevant perspective is that of concrete individuals, and the number of their thread, i.e. which of them are executing concurrently. For example, if each concrete individual has an own thread and if sending messages were limited to a single sender per system step, then it would be sufficient to have as many non-concrete threads as concrete ones. This seems related to the active-object concept of [79], though they allow to receive events from any other individual in a step. Yet there is possibly a still useful subclass with sufficiently restricted communication model.

Another different case where the approach remains sound is if the interaction with multiple other instances cannot be distinguished. For example if there is a common bus or (boolean) signal which can be raised by one or more other instances but from the perspective *within* an instance the number of issuers can't be determined, but from the inside, there is only the distinction between "none" or "at least one".

Identifying whether a given HLL model (or a model in any other formalism) belongs to this system class is in general difficult. A possible approach would be to try to devise a particular sub-language in which all models have this property (cf. general discussion of symmetry detection in Section 9.2).

Concurrency in the Literature

The insights of Section 9.3.4, that a syntactically obtained DTR abstraction is safe for strict interleaving semantics but in general not for truly concurrent systems, is new to the best of our knowledge. So let us briefly discuss whether this issue has effects on the already published experiences with DTR abstraction.

Tomasulo and FLASH in the Cadence SMV The first application of DTR is by McMillan [124, 129, 127] who verifies an implementation of the Tomasulo algorithm [169] in Cadence SMV [125, 126]. The Cadence SMV modelling language has a fully concurrent semantics, it is able to model fully concurrent components, which is also reasonable for the case study: registers, reservation stations, and functional unit should in general do not run interleaved.

Does this mean a conflict with the findings of the previous sections?. Not necessarily. First of all, the named publications *don't tell* how the abstract transition system is obtained; maybe it's done exactly, then there's no problem because Chapter 6 ensures soundness. And even if it employed the syntactical way introduced for HLL above and elaborated for array programs below, it's on the safe side because the reservation stations and functional units communicate via a bus which serialises communication.

That is, at most one functional unit speaks to at most one reservation station. So in a sense, the architecture of the case-study already provides some of the countermeasures we proposed, yet it may not be sufficient to support all kinds of properties and choices of spotlights.

The same applies to the later work [130], a verification of the FLASH cache coherency protocol. It's on the safe side because of the system structure. Each cache line has a hosts and a number of clients where communication is between host and client (one communication at a time) or between two clients. The host is not abstracted and from the perspective of a client, there is only interaction with the host or one other client, thus it is not visible whether other clients execute concurrently.

German and FLASH Protocol in Murphi Chou and others [31] employ a syntactical transformation similar to ours in order to verify the German [145] and the FLASH cache coherency protocol [130] with the Murphi model-checker [56].

First of all, their syntactical transformation is carefully designed, not as general as our presentation for HLL, and they outline the discussion of soundness. On the other hand, the same argument given for the previous case applies, as the German protocol is also following a home/slave architecture.

Telecommunication Features in SPIN Calder and Miller [28, 29] apply a syntactical transformation to a Promela [83] description of a telecommunication system. They are on the safe side because Promela has a strict interleaving semantics in the targeted SPIN model-checker [83].

UML and DCS with VIS Similarly, our case studies on UML and DCS models from Chapter 10, which have partially been published in [178] (UML) and [9, 10] (DCS), are fine because we also use a strict interleaving semantics.

9.3.5. Conclusion

A possible bottom line of Sections 9.3.3 and 9.3.3 is the following. Strict interleaving is good as it allows to obtain (or at least closely approximate) the DTR of an HLL model's ETTS by a syntactical transformation.

It is common to employ interleaving semantics to analyse concurrent systems, yet it naturally doesn't preserve simultaneity, and there is in general no direct use of the "next" modality ("X"), which could be considered a good effect given the discussion of Chapter 4 where the "next" modality was one of the main hindrances in obtaining a definite fragment of EvoCTL*.

With full concurrency, syntactical transformation have to be treated with great care. As some of the case-studies from Section 9.3.4 don't discuss this issue, one could (drastically speaking) suspect that they only happen to be sound but aren't robustly so.

$$\begin{tikzcd}
\mathscr{M} \arrow[rr, mapsto, "\alpha_D"] \arrow[d, mapsto, "\llbracket\cdot\rrbracket"'] \arrow[rrd, bend left=15, "\llbracket\cdot\rrbracket^\sharp"] \arrow[rrdd, bend left=10, "\langle\!\langle\cdot\rangle\!\rangle^\sharp_D"'] & & \mathscr{M}^\sharp_D \arrow[d, mapsto, "\llbracket\cdot\rrbracket"] & \text{\} static analysis} \\
M \arrow[r, mapsto, "\alpha"] \arrow[d, mapsto, "\langle\!\langle\cdot\rangle\!\rangle", "\equiv"'] & M^\sharp \arrow[d, mapsto, "\langle\!\langle\cdot\rangle\!\rangle", "\equiv"'] \arrow[l, phantom, "\preceq"' near start] \arrow[r, phantom, "\preceq"] & M^\sharp_D \arrow[d, mapsto, "\langle\!\langle\cdot\rangle\!\rangle", "\equiv"'] & \text{\} ETTS model-ch. (if existent)} \\
A \arrow[r, phantom, "\preceq"] & A^\sharp \arrow[r, phantom, "\preceq"] & A^\sharp_D & \text{\} fin.-state model-ch. (VIS,SMV,SPIN)} \\
\underbrace{\qquad}_{\text{poss. infin.}} & \underbrace{\qquad\qquad\qquad}_{\text{finite}} & &
\end{tikzcd}$$

Figure 9.17.: **Commuting Diagram Revisited (2).** The parts of Figure 9.1 relevant for Section 9.4 (cf. Figure 9.6). The shaded region indicates the subject of Section 9.4.3, namely obtaining an array program encoding of a DTR abstraction by a modified encoding of the HLL model, bypassing the regular HLL semantics.

9.4. Encoding in Array Programs

The current section serves two purposes. Up to now we've discussed an HLL syntax with a parameterised semantics which translates to ETTS on a very elementary level. But in order to actually apply model-checking to HLL models, it is desirable to employ existing tools, in our case preferably finite-state model-checkers for their maturity and wide accessibility.

Model-checkers, like VIS [154] and SMV [128], typically support a higher-level input language providing a rich expression language and typing, either natively (SMV) or as an add-on with compilers (VIS). So first of all, this section provides an encoding of HLL in array programs, which, in a sense, closes the circle from Chapter 3 where we briefly compared ETTS to array programs.

Then in case of HLL models with *finite* ETTS, for instance with finite ethers and a maximum number on alive individuals encoded in the creation oracle, we can use the CTL* or LTL checking capabilities of common of-the-shelf model-checkers as the basis to treat EvoCTL* properties. Figure 9.17 shows this third layer in addition to the two already present in Figure 9.6.

Secondly, we'll address the mapping marked gray in Figure 9.6. In Section 9.3, the rationale was to implement the DTR abstraction while *staying within* the HLL syntax and semantics. Now if we're encoding HLL models in a lower level modelling language, namely that of common finite-state model-checkers, then the encoding function is giving the semantics and we are free to change it in a way such that the (modified) encoding yields the DTR abstraction of the original HLL model.

9.4.1. Array Programs: Abstract Syntax and Intuitive Semantics

Basic Types. We assume the following basic types. Booleans (`bool`), finite enumerations ($\{E_1, ..., E_n\}$, E_i identifiers), finite integer intervals ($[N, M]$, $N, M \in \mathbb{Z}$), and infinite integers (`int`).

We assume that type definitions of the form

$$\texttt{typedef}\ \mathit{type}_1 : \mathit{type}_0, \tag{9.174}$$

allow us to refer to type type_0 by the name of type_1. This is in a sense redundant, but will be useful as abbreviations and may be useful when desiring type-checking.

Record and Array Declarations. Record types shall be declared in the form

$$\texttt{record}\ : \{\mathit{name} : \mathit{type}\}, \tag{9.175}$$

array types with elements of type type_1 indexed by type_0, which is either an integer interval or the general integer type, by

$$\texttt{array}\ \mathit{type}_0\ \texttt{of}\ \mathit{type}_1 \tag{9.176}$$

Variable Declarations. We further assume

$$\begin{aligned} &\mathit{mode}\ x : \mathit{type} \\ &\mathit{mode}\ x : \mathit{type}\ \texttt{:=}\ \mathit{expr} \end{aligned} \tag{9.177}$$

to introduce a new variable x of a previously declared type *type*. The mode $\mathit{mode} \in \{\texttt{input}\ , \texttt{aux}\ , \texttt{local}\ , \texttt{const}\ \}$ denotes whether the variable is treated as an input, an auxiliary, a regular variable, or as a constant[10], where a distinction between the latter two cases may or may not be provided by the employed model-checker.

An initial value can be provided via the optional expression *expr*. If it is not given, the initial value is the left-most value of the type's domain or 0, which recursively defines the initial value of records and arrays.

Expressions. Considering the expression language, we firstly define a variable reference to be an word of the following grammar

$$\mathit{nav} ::= \langle \mathit{var} \rangle \mid \langle \mathit{nav} \rangle \texttt{[} \langle \mathit{expr} \rangle \texttt{]} \mid \langle \mathit{nav} \rangle . \langle \mathit{comp} \rangle \tag{9.178}$$

where *var* is a previously declared variable and *cond* a record component.

Common well-formedness rules apply, for example, the array access is well-formed if the type of *nav* is an array type and *expr* is of the array's index type. Similarly for record access.

Then both *nav* an *nav'* are expressions, the former denoting the value in the last stable state, the latter denoting the value in the current intermediate step (see semantics of programs below).

We assume that the features and semantics of the expression language, including function symbols, coincide with the signature and structure employed by the HLL model.

[10] somehow stretching the concept of "variable"

Actions. As actions *act*, we assume the following.

A *skip* statement `skip` doing nothing.

An *assignment*

$$\mathit{nav}' \; \texttt{:=} \; \mathit{expr} \tag{9.179}$$

updating a valuation of variables such that the expression *nav′* yields the value *expr* had in the original valuation and the rest remains unchanged.

Guarded conditions

$$\begin{array}{l} \texttt{if} \\ \quad \Box\, \mathit{cond}_1 : \mathit{act}_1; \\ \quad \vdots \\ \quad \Box\, \mathit{cond}_n : \mathit{act}_n; \\ \texttt{fi} \end{array} \tag{9.180}$$

executing non-deterministically one of the act_i for which cond_i holds, $1 \leq i \leq n$, if none of the condition holds, no action is executed.

For simplicity, we assume support for *sending events, querying the ether, and consuming events*: these facilities are, for example, provided by the SPIN model-checker.

In others environments, it can be mimicked with the means we've introduced, for instance by implementing a FIFO queue by arrays. Yet it becomes somewhat tedious if events have vastly different signatures, that is, sets of parameters. This is one reason why the DCS language discussed in Chapter 10 restricts events to only two signatures, namely ones comprising a single parameter of object type and ones without parameters.

We use $\mathit{indv}.\epsilon.E$ to denote the query operation, which yields true if and only if the ether of individual *indv* has a ready to consume E-event, we use

$$p_1, \ldots, p_n \; \texttt{:=} \; \mathit{consume}(\mathit{indv}), n \in \mathbb{N}_0 \tag{9.181}$$

to denote that the next event is consumed with a simultaneous assignment of the carried parameters to the local names $p_1, \ldots, p_n$, and we use

$$\mathit{indv}_1 \; \texttt{->} \; \mathit{send}(\mathit{indv}_0, E, (\mathit{expr}_1, \ldots, \mathit{expr}_n)) \tag{9.182}$$

to denote the sending operation, that is, individual indv_0 sends an E-event with parameter values given by expressions expr_i to individual indv_1.

We assume that the provided (or implemented) ether is initially empty

We don't assume native support for *creation and destruction*, because it's not present in VIS and SMV and not directly usable in the plain SPIN (cf. [2]).

Instead, we'll encode begin alive by a boolean flag in the record structure representing object's state. The (pseudo-code) operation

$$\texttt{delete}(\mathit{this} \rightarrowtail \lambda) \tag{9.183}$$

in Section 9.4.2 below will then basically correspond to

$$Ds\texttt{[}Cs\texttt{[}this\_C\texttt{]}.\lambda\texttt{]}.alive := 0 \tag{9.184}$$

if λ points to instances of class D, depending on whether duplicate deletion is considered to be a fatal error or not.

The (pseudo-code) create operation

$$this \rightarrowtail \lambda := \texttt{new}(C) \tag{9.185}$$

will correspond to choosing any unused identity, choosing an initial state, and setting up the local state, i.e.

$$\begin{aligned} &Cs\texttt{[}i_0\texttt{]}.alive := 1; Cs\texttt{[}i_0\texttt{]}.this := i_0; Cs\texttt{[}i_0\texttt{]}.st := i_1; \\ &Cs\texttt{[}i_0\texttt{]}.x_0 := 0; \dots\ Cs\texttt{[}i_0\texttt{]}.x_n := 0; \end{aligned} \tag{9.186}$$

where i_0 denotes an input providing a non-alive identity and i_1 denotes an input providing an initial state of class C. The last row exemplifies how a local boolean variable would be updated to the left-most value, alternatively it could be chosen arbitrarily.

Alternatively, the reset to initial values can be done together with the destruction instead of with the creation, this has been discussed and partly experimentally evaluated, for instance, in [150], [95], and [2].

Program. A program in this language shall have the following structure.

```
⟨decl⟩;
do
    ⟨act⟩;
od
```

That is, there is a sequence of type and variable declaration and then a sequence of actions enclosed in a loop, called *loop-body.*

The semantics is a Kripke structure where states are all possible valuations of the declared variables and where there is a transition between two states s and s' if and only if the variable valuation s' is a possible result of applying the loop-body to s stepwise, basically similar to HLL.

9.4.2. From HLL to Array Programs

The purpose of this section is to sketch an encoding in as much detail as necessary to enable straightforward actual implementations it in any model-checker input language reasonably resembling imperative programs, and thus to make plausible how we conducted our experiments in Chapter 10.

It is chosen minimal for this purpose and partly has to remain sketchy. It is not trying to define a good input language, and abstracts from practical requirements like some model-checkers having the restriction of only one assignment per variable in addition to a default assignment.

For instance, we consider only single link case (as in Section 9.3), and don't give a formal semantics, but only point out the principles. If in doubt, assume we have SMI in mind, or, less directly, SMV or SPIN.

The whole encoding is not novel but resembling the standard encoding of state machines in imperative languages as demonstrated, for example, for Statecharts [78] in [24], or numerous other works. For UML, for instance, the literature ranges from [111, 108, 65] over [118, 117] to [187, 184] ordered by sophistication. Ideas for the particular non-DTR encoding occur, for example, in [45, 46] and [159], the implementation of DTR has already been outlined in [48, 49], there assuming symbolic transition systems (STS) [120].

The novel point will be a general presentation of syntactical DTR for array programs in Section 9.4.3, of which this section is a prerequisite.

For more elaborate versions of this encoding, in particular including a discussion of issues with non-native but implemented communication facilities and object creation/destruction we refer to the works [150] and [2], which provide an encoding of the DCS language (cf. Chapter 10).

Basic Types, Events, Initial State Condition

Let $\mathscr{M} = (\mathscr{T}, \mathscr{E}, \mathscr{C}, \mathit{cond}_0)$ be an HLL model. As said above, we assume that the basic types in $\mathscr{T}$ coincide with the basic types supported by our array programming language.

Let $\mathscr{C} = \{C_1, \dots, C_n\}$ be set of classes. Then we use $\langle\!\langle \mathscr{C} \rangle\!\rangle$ to denote the type declarations

$$\texttt{typedef}\ C_1\mathit{id} : \texttt{int}\ ; \dots \texttt{typedef}\ C_n\mathit{id} : \texttt{int}\ ; \tag{9.187}$$

which introduce Cid as identity type of class C.

Let $\mathscr{E} = \{E_1, \dots, E_m\}$ be the set of events. Then we use $\langle\!\langle \mathscr{E} \rangle\!\rangle$ to denote the declaration of the enumeration

$$\texttt{typedef}\ \mathit{Evs\_enum} : \texttt{enum}\ \{E_1, \dots, E_m\} \tag{9.188}$$

and for each event $E \in \mathscr{E}$ with parameters $p_1, \dots, p_k$ the record

$$\texttt{typedef}\ \mathit{E\_rec} : \texttt{record}\ \{p_1 : \tau(p_1); \dots p_k : \tau(p_k); \} \tag{9.189}$$

of parameters where we use τ as in the definition of class state below.

Class State

Let C be a class with typed local variables $X_C = \{x_1, \dots, x_n\} \dot\cup \{x_{C,st}\}$, typed links $\Lambda_C = \{\lambda_1, \dots, \lambda_m\} \dot\cup \{\mathit{this}\}$, a non-empty set of states S_C, some of them initial $S_{0_C} \subseteq S_C$, and a (possibly empty) set of transitions R_C.

Then we use $\langle\!\langle C \rangle\!\rangle$ to denote the following sequence of one enumeration, three record type, and three array-typed variable declarations. Firstly,

$$\texttt{typedef}\ \mathit{C\_enum} : \texttt{enum}\ \{s_1^C, \dots, s_N^C\}; \tag{9.190}$$

provides the range for the "current state" variable.

It's followed by the three records

$$\begin{aligned}\texttt{typedef}\ \mathit{C\_rec}^{local} : \texttt{record}\ \{\ & \mathit{alive} : \texttt{bool}\ ; \\ & \mathit{this} : \mathit{Cid}; \\ & \mathit{st} : \mathit{C\_enum}; \\ & x_1^{local} : \tau(x_1^{local}); \dots x_j^{local} : \tau(x_j^{local}); \};\end{aligned} \tag{9.191}$$

if $X \setminus (X_I \cup X_A) = \{x_{st}, x_1^{local}, \dots, x_j^{local}\}$,

$$\texttt{typedef}\ \mathit{C\_rec}^{aux} : \texttt{record}\ \{x_1^{aux} : \tau(x_1^{aux}); \dots x_k^{aux} : \tau(x_k^{aux}); \}; \tag{9.192}$$

if $X_A = \{x_1^{aux}, \dots, x_k^{aux}\}$, and

$$\texttt{typedef}\ \mathit{C\_rec}^{in} : \texttt{record}\ \{x_1^{in} : \tau(x_1^{in}); \dots x_\ell^{in} : \tau(x_\ell^{in}); \}; \tag{9.193}$$

if $X_I = \{x_1^{in}, \dots, x_\ell^{in}\}$. Together they provide the type of single instances of class C, split into the three aspects of local, auxiliary, and input variables.

We use $\tau(x)$ denotes the array program basic type corresponding to the type τ of variable x and $\tau(\lambda)$ yields Cid if λ is of type τ_C.

Then the three array-typed variables

$$\begin{aligned}&\mathit{Cs}^{local} : \texttt{local array}\ \mathit{Cid}\ \texttt{of}\ \mathit{C\_rec}^{local}; \\ &\mathit{Cs}^{aux} : \texttt{aux array}\ \mathit{Cid}\ \texttt{of}\ \mathit{C\_rec}^{aux}; \\ &\mathit{Cs}^{in} : \texttt{input array}\ \mathit{Cid}\ \texttt{of}\ \mathit{C\_rec}^{in};\end{aligned} \tag{9.194}$$

keep the states of the instances of class C, split into the three modes.

For simplicity, we assume cond_0 to denote the empty topology as in Section 9.3.3. Then the initial values of variables directly provide an encoding of the initial state as we'll see below.

Transition Programs

The encoding of a transition $r = (s_C, \ell, s'_C) \in R_C$ is

$$\begin{aligned}\langle\!\langle r \rangle\!\rangle = \ & \mathit{Cs}\texttt{[}\mathit{this\_C}\texttt{]}.\mathit{st} = s_C \wedge \mathit{Cs}\texttt{[}\mathit{this\_C}\texttt{]}.\epsilon.E \wedge \mathit{cond} : \\ & p_1, \dots, p_n \mathrel{\texttt{:=}} \mathit{consume}(\mathit{this\_C}); \\ & \langle\!\langle \mathit{act} \rangle\!\rangle; \\ & \mathit{Cs}'\texttt{[}\mathit{this\_C}\texttt{]}.\mathit{st} \mathrel{\texttt{:=}} s'_C;\end{aligned} \tag{9.195}$$

if the labelling is $\ell = E(p_1, \dots, p_n)\,[\mathit{cond}]/\mathit{act}$.

If the event part is empty, then the middle condition in the guard and the assignment of parameters are omitted. Note that the navigation expression normal form of Section 9.3.1 permits let-expressions in guarding conditions, in order to set up the auxiliary variables and links used in the guarding condition. The encoding naturally extends to this case by moving this setting up of auxiliary variables right before the large condition ranging over all transitions (cf. Section 9.4.2).

The encoding of actions is inductively defined as follows.

1. The skip statement is encoded by the skip statements of array programs, i.e. $\langle\!\langle \texttt{skip} \rangle\!\rangle = \texttt{skip}$.

2. By Def. 9.3.1, we only have to consider assignments of the following forms

   - $\langle\!\langle \mathit{this} \rightarrowtail h' := \mathit{this} \rangle\!\rangle = \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle := \mathit{this\_C}$
   - $\langle\!\langle \mathit{this} \rightarrowtail h' := p \rangle\!\rangle = \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle := p$
   - $\langle\!\langle \mathit{this} \rightarrowtail h' := \mathit{this} \rightarrowtail h' \rightarrowtail \lambda \rangle\!\rangle = \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle := \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda \rangle\!\rangle$

   or

   - $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' := \mathit{nav} \rangle\!\rangle = \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \rangle\!\rangle := \langle\!\langle \mathit{nav} \rangle\!\rangle$
   - $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail x' := \mathit{expr} \rangle\!\rangle = \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail x' \rangle\!\rangle := \langle\!\langle \mathit{expr} \rangle\!\rangle$

   (See below for encoding of expressions.)

3. Creation and deletion are encoded by the creation and deletion primitives of array programs or corresponding implementation, i.e.

   - $\langle\!\langle \mathit{nav} \rightarrowtail \lambda' := \texttt{new}\ C \rangle\!\rangle = \langle\!\langle \mathit{nav} \rightarrowtail \lambda' \rangle\!\rangle := \texttt{new}(C)$
   - $\langle\!\langle \texttt{delete}\ \mathit{nav} \rangle\!\rangle = \texttt{delete}(\langle\!\langle \mathit{nav} \rangle\!\rangle)$

4. Event sending is encoded by the event sending primitive of array programs or a corresponding implementation, i.e.

   - $\langle\!\langle \mathit{nav}\,\texttt{!}\,E(\mathit{expr}_1, \ldots, \mathit{expr}_n) \rangle\!\rangle$
     $= \langle\!\langle \mathit{nav} \rangle\!\rangle\ \texttt{->}\ \mathit{send}(\mathit{this\_C}, E, (\langle\!\langle \mathit{expr}_1 \rangle\!\rangle, \ldots, \langle\!\langle \mathit{expr}_n \rangle\!\rangle))$

5. The conditional statement of transitions is naturally encoded as the conditional statement, i.e.

   - $\langle\!\langle \texttt{if}\ \mathit{cond}\ \texttt{then}\ \mathit{act}_1\texttt{;}\ \texttt{else}\ \mathit{act}_2\texttt{;}\ \texttt{fi} \rangle\!\rangle$
     $= \texttt{if}\ \square \langle\!\langle \mathit{cond} \rangle\!\rangle : \langle\!\langle \mathit{act}_1 \rangle\!\rangle; \square \neg \langle\!\langle \mathit{cond} \rangle\!\rangle : \langle\!\langle \mathit{act}_2 \rangle\!\rangle;\ \texttt{fi}$

6. Sequential composition similarly becomes sequential composition, i.e.

   - $\langle\!\langle \mathit{nav}_1; \mathit{nav}_2 \rangle\!\rangle = \mathit{nav}_1; \mathit{nav}_2$

Here, the encoding $\langle\!\langle \mathit{expr} \rangle\!\rangle$ of HLL expressions *expr* is assumed to be clear by the shared signature, except for navigation expressions (in normal form following Def. 9.3.1) whose encoding is defined as

- $\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle = \mathit{Cs}'\texttt{[}\mathit{this\_C}\texttt{]}.h$
- $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda \rangle\!\rangle = \mathit{Ds}\texttt{[}\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle\texttt{]}.\lambda$
- $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \rangle\!\rangle = \mathit{Ds}'\texttt{[}\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle\texttt{]}.\lambda$
- $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail x \rangle\!\rangle = \mathit{Ds}\texttt{[}\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle\texttt{]}.x$

- $\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail x' \rangle\!\rangle = \mathit{Ds}'\,\texttt{[}\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle\texttt{]}\,.x$

if h is of object type τ_D.

Note that, for readability, we don't distinguish between the three arrays Ds^{local}, Ds^{aux}, and Ds^{in} because the accessed record component uniquely determines which one is denoted by *Ds*.

Putting It All Together

Given interpretation ι, assuming it coincides with the semantics of expressions, a definition of ethers $\mathcal{E}$, assuming it coincides with the query, send and consume primitives of our array programs or is implemented accordingly, a strict interleaving scheduler $\mathcal{S}$, and an input and creation oracle $\mathcal{O}$, we obtain the following program.

```
$\langle\!\langle \mathscr{C} \rangle\!\rangle$;                        // identity types of classes
$\langle\!\langle \mathscr{E} \rangle\!\rangle$;                        // types for events
$\langle\!\langle C_1 \rangle\!\rangle; \dots ; \langle\!\langle C_n \rangle\!\rangle$;           // class arrays
do
   $\langle\!\langle \mathcal{S} \rangle\!\rangle$; // scheduler, select $sched\_C_i$ and $this\_C_i$
   if
      $\Box\ sched\_C_1 : \Box_{r \in R_{C_1}} \langle\!\langle r \rangle\!\rangle$
      $\vdots$
      $\Box\ sched\_C_n : \Box_{r \in R_{C_n}} \langle\!\langle r \rangle\!\rangle$
   fi
od
```

Alternatively, there may be a native scheduler like in SPIN.

Example 9.4.1 (Car Platooning Encoded)**.** *The following array program is the encoding of the example HLL model from Figure 9.2, omitting details like the scheduling, and the assumed additional class(es) which model an environment that creates cars and cares for their mutual recognition.*

```
typedef Cid : int ;

typedef Evs_enum : {ack, car_ahead, nack, new_flw, new_ldr, req};
typedef car_ahead_rec : record {p : Cid; };
typedef new_flw_rec, new_ldr_rec : car_ahead_rec;
typedef req_rec : car_ahead_rec;

typedef C_enum : enum {fa, ld, faf, ldf, fl, fls};
typedef $C\_rec^{local}$ : record {
   alive : bool ; this : Cid; st : C_enum; ldr : Cid; flw : Cid;
};
$Cs^{local}$ : local array Cid of $C\_rec^{local}$;
do
```

```
$\langle\langle\mathcal{S}\rangle\rangle$; // scheduler, select sched_$C_i$ and this_$C_i$
if $\square$ sched_C : if
   $\square$ Cs[this_C].st = fa $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      Cs' [this_C].flw := p; p -> send(this_C, ack, ());
      Cs' [this_C].st := ld;
   $\square$ Cs[this_C].st = fa $\wedge$ this_C.$\epsilon$.nack : consume(this_C);
      Cs' [this_C].st := faf;
   $\square$ Cs[this_C].st = faf $\wedge$ this_C.$\epsilon$.car_ahead : p := consume(this_C);
      p -> send(this_C, req, (this)); Cs' [this_C].st := fa;
   $\square$ Cs[this_C].st = faf $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      p -> send(this_C, nack, ()); Cs' [this_C].st := faf;
   $\square$ Cs[this_C].st = faf $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      p -> send(this_C, nack, ()); Cs' [this_C].st := faf;
   $\square$ Cs[this_C].st = faf $\wedge$ this_C.$\epsilon$.ack : p := consume(this_C);
      Cs' [this_C].ldr := p; Cs' [this_C].st := fl;
   $\square$ Cs[this_C].st = ld $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      flw -> send(this_C, new_flw, (p)); p -> send(this_C, ack, ());
      Cs' [this_C].st := ld;
   $\square$ Cs[this_C].st = ld $\wedge$ this_C.$\epsilon$.car_ahead : p := consume(this_C);
      p -> send(this_C, req, (this)); Cs' [this_C].st := ldf;
   $\square$ Cs[this_C].st = ldf $\wedge$ this_C.$\epsilon$.nack : consume(this_C);
      Cs' [this_C].st := ld;
   $\square$ Cs[this_C].st = ldf $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      p -> send(this_C, nack, ()); Cs' [this_C].st := ldf;
   $\square$ Cs[this_C].st = ldf $\wedge$ this_C.$\epsilon$.ack : p := consume(this_C);
      Cs' [this_C].ldr := p; Cs' [this_C].st := fls;
   $\square$ Cs[this_C].st = fl $\wedge$ this_C.$\epsilon$.new_ldr : p := consume(this_C);
      Cs' [this_C].ldr := p; Cs' [this_C].st := fl;
   $\square$ Cs[this_C].st = fl $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      p -> send(this_C, nack, ()); Cs' [this_C].st := fl;
   $\square$ Cs[this_C].st = fl $\wedge$ this_C.$\epsilon$.new_flw : p := consume(this_C);
      Cs' [this_C].flw := p; Cs' [this_C].st := fls;
   $\square$ Cs[this_C].st = fls $\wedge$ this_C.$\epsilon$.new_ldr : p := consume(this_C);
      Cs' [this_C].ldr := p; flw -> send(this_C, new_ldr, (p));
      Cs' [this_C].st := fls;
   $\square$ Cs[this_C].st = fls $\wedge$ this_C.$\epsilon$.new_flw : p := consume(this_C);
      flw -> send(this_C, new_flw, (p)); Cs' [this_C].st := fls;
   $\square$ Cs[this_C].st = fls $\wedge$ this_C.$\epsilon$.req : p := consume(this_C);
      p -> send(this_C, nack, ()); Cs' [this_C].st := fls;
   fi; fi;
od
```

$\diamondsuit$

9.4.3. Syntactical DTR for Array Program Encodings

The adaptation of the procedure from Section 9.3 to our array programs is rather straightforward because the action language of the array programs introduced above closely resembles the one of HLL, notably without sacrificing a close relation to common model-checker input languages.

The less strict typing in array programs, namely the lack of native support for classes, makes the construction even easier. We namely don't have to add a class similar to C^{C} and we don't *have to* encode links to non-concrete individuals by additional flags. If the targeted model-checker input language provides a native concept of process identities conflicting with the following transformation, an encoding similar to Section 9.3 is still an option.

We'll discuss the transformation in an order following Section 9.4.2 such that Section 9.4.2 applies without major changes.

Transformation: Classes, Events, Initial State Condition

Let $D = \{(d_{j_1}, Id_{j_1}), \dots, (d_{j_m}, Id_{j_m})\}$ be a DTR. For each class C with a set $d = \{1, \dots, N\}$, $N \in \mathbb{N}_0$, in D, the definition of $\langle\!\langle \mathscr{C} \rangle\!\rangle$ changes to

$$\begin{aligned} &\texttt{typedef}\ \mathit{Cid} : [1, N+1]; \\ &\texttt{const}\ \mathsf{C}_C := N+1; \end{aligned} \tag{9.196}$$

That is, the finite set given by d plus one additional number is used as identities of C objects instead of unbounded integers.

The additional number $N + 1$ corresponds to the identity C in Chapter 6, so for convenience we introduce this name as a constant . The declarations in $\langle\!\langle \mathscr{E} \rangle\!\rangle$ remain unchanged.

Transformation: Class State

The definition of the encoding of a class $\langle\!\langle C \rangle\!\rangle$ remains unaltered, we only add, for each occurrence of a navigation expression in the encoding of transition programs a fresh input of the corresponding type, exactly similar to the procedure in Section 9.3.3. The array declarations remain the same, only the index range changes implicitly by the altered definitions of the *Cid* types.

Transformation: Transition Programs

The encoding of transition programs changes analogously to Section 9.3.3. That is, for each navigation via C, an input is considered, and each assignment to the C_C-th field of *Cs* and message sending to C is replaced by a skip statement. The main difference to Section 9.3.3 is that we don't need an additional class C^{C}, but that we modify the encoding of transitions such that the array program code behaves regularly when *this_C* is from the range $[1, N]$ and that they behave over-approximating when *this_C* is $N+1 = \mathsf{C}_C$.

More technically, navigation expressions are firstly changed as follows.

Access of Auxiliary Variables remains unchanged because they're assigned before first use. When executing the loop for C_C, the *right-hand sides* will be determined by inputs

$$\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle^{\sharp}_{D} = \mathit{Cs}'\,\texttt{[}\mathit{this\_C}\texttt{]}\,.h \tag{9.197}$$

Navigation changes similar to Section 9.3.3. When navigation to a variable or link value via C_C is requested, then a free input value is considered instead, that is,

$$\langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda \rangle\!\rangle^{\sharp}_{D} = (\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle = \mathsf{C}_C \;?\; i : \mathit{Ds}\texttt{[}\langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle\texttt{]}\,.\lambda) \tag{9.198}$$

where we assume the conditional expression syntax from Section 9.3.2 and where i is the input dedicated to this occurrence of the navigation expression introduced in the previous section.

In this case, i will have the same type as λ. Thus in case of C and the DTR given above, it may yield any value from $[1, N{+}1]$, including the identity C_C of the non-concrete individuals.

The encodings of *variable access* and the *primed cases* change similarly.

The encoding of assignments remains unchanged, except for

$$\begin{aligned} \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \;\texttt{:=}\; \mathit{nav} \rangle\!\rangle^{\sharp}_{D} = {} & \texttt{if} \\ & \square \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle = \mathsf{C}_C : \texttt{skip}; \\ & \square \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle \neq \mathsf{C}_C : \\ & \qquad \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \rangle\!\rangle \;\texttt{:=}\; \langle\!\langle \mathit{nav} \rangle\!\rangle^{\sharp}_{D}; \\ & \texttt{fi} \end{aligned} \tag{9.199}$$

or, equivalent but shorter,

$$\begin{aligned} & \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \;\texttt{:=}\; \mathit{nav} \rangle\!\rangle^{\sharp}_{D} \\ & \quad = \texttt{if}\ \square \langle\!\langle \mathit{this} \rightarrowtail h' \rangle\!\rangle \neq \mathsf{C}_C : \langle\!\langle \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \rangle\!\rangle \;\texttt{:=}\; \langle\!\langle \mathit{nav} \rangle\!\rangle^{\sharp}_{D};\ \texttt{fi} \end{aligned} \tag{9.200}$$

and similarly for assignments to variables.

Creation and destruction are treated similarly to Section 9.3.3. There is a choice to directly consider C_C to be created, otherwise it is tried whether there is an unused concrete identity. If not, C_C is chosen. Here it is also important, that the outcome is completely deterministic in order to ensure that all identities can be used in all situations, such that the symmetry argument can be applied.

Sending is treated similarly to assignment, namely

$$\begin{aligned} & \langle\!\langle \mathit{nav}\,\texttt{!}\,E(\mathit{expr}_1, \dots, \mathit{expr}_n) \rangle\!\rangle^{\sharp}_{D} \\ & \quad = \texttt{if}\ \square\ \langle\!\langle \mathit{nav} \rangle\!\rangle^{\sharp}_{D} \neq \mathsf{C}_C : \langle\!\langle \mathit{nav}\,\texttt{!}\,E(\mathit{expr}_1, \dots, \mathit{expr}_n) \rangle\!\rangle;\ \texttt{fi} \end{aligned} \tag{9.201}$$

Control Structures. The encoding of *conditional statements* and *sequential composition* remains unchanged.

Transition. The modified encoding of a *transition* caters for the case where *this_C* denotes C_C. in that case, the current state is guessed, and the content of the queue such that the non-concrete individuals can take any transition whenever they're scheduled. That is, we obtain

$$\begin{aligned}
\langle\!\langle r \rangle\!\rangle^{\sharp}_{D} = {} & \mathit{this\_C} = \mathsf{C}_{hlcl} \vee (\mathit{Cs}\texttt{[}\mathit{this\_C}\texttt{]}.\mathit{st} = s_C \wedge \mathit{Cs}\texttt{[}\mathit{this\_C}\texttt{]}.\epsilon.E \wedge \mathit{cond}) : \\
& \texttt{if } \square\, \mathit{this\_C} = \mathsf{C}_C : p_1, \ldots, p_n \texttt{ := } i_1, \ldots, i_n; \\
& \quad \square\, \mathit{this\_C} \neq \mathsf{C}_C : p_1, \ldots, p_n \texttt{ := } \mathit{consume}(\mathit{this\_C}); \\
& \texttt{fi} \\
& \langle\!\langle \mathit{act} \rangle\!\rangle^{\sharp}_{D}; \\
& \texttt{if } \square\, \mathit{this\_C} \neq \mathsf{C}_C : \mathit{Cs}'\texttt{[}\mathit{this\_C}\texttt{]}.\mathit{st} \texttt{ := } s'_C;\ \texttt{fi}
\end{aligned} \tag{9.202}$$

Alternatively, all encodings of transitions could be changed to refer to an additional auxiliary variable h_{st} keeping the state of the currently scheduled object, like in

$$\langle\!\langle r \rangle\!\rangle^{\sharp}_{D} = (h_{st} = s_C) \wedge \mathit{Cs}\texttt{[}\mathit{this\_C}\texttt{]}.\epsilon.E \wedge \mathit{cond} : \ldots \tag{9.203}$$

Then the initialisation of h_{st} before the transition branches would regularly employ the modified transformation and read the current state from an input, leaving the array program closer to the original one. The query of the ether would have to be treated similarly, of course.

In case the ether is explicitly implemented, and not assumed to be native as we do here, correct treatment of the ether often follows by simply applying the regular rules for navigation and assignment given above.

Note that, in order to establish true liveness properties, including "U" or "F" operators as discussed in Chapter 10, one will need to add fairness constraints if the scheduling is not native. Otherwise, the C process can be scheduled forever and let the concrete individuals suffer from starvation.

Optimisations and Precision

The issues discussed in Section 9.3.3 apply directly to the array program case. For example, the transformation presented above is sound but looses precision as it may use different input for common expressions. Furthermore, the number of used inputs is far from optimal, there is large potential for sharing as in the HLL case.

An additional optimisation with array programs is based on recognising that the $(N+1)$-th entry of the array *Cs* is actually never accessed. Write access by assignment is re-directed to skip statements and read access in expressions is re-directed to the corresponding inputs. Thus, if the array program is sufficiently *weakly* typed, the declaration for class C can change to

$$\begin{aligned}&\texttt{typedef}\ \mathit{Cid} : [1, N+1];\\ &\mathit{Cs} : \texttt{local array}\ [1, N]\ \texttt{of}\ \mathit{C\_rec};\end{aligned} \tag{9.204}$$

On the other hand, it is a common built-in optimisation of model-checkers like VIS or SMV to identify effective constants automatically, so in some cases, the results of the change may not be visible in the overall checking time.

Finiteness

Given an HLL model with structure and oracle as assumed above, the question is whether $\langle\!\langle \mathscr{M} \rangle\!\rangle^{\sharp}_{D}$ is finite, because then it can directly be treated with the named finite-state model-checking tools as SMV, VIS, and SPIN.

As in the HLL case, the transition system defined by the encoding $\langle\!\langle \mathscr{M} \rangle\!\rangle$ is finite iff the ETTS of HLL is finite, which could be the case if the creation oracle considers only finitely many identities, which can alternatively be seen as an under-approximation of the full HLL, possibly useful for debugging purposes.

Furthermore, the communication medium may only keep a bounded number of messages at a time. This can be a model property or be imposed by choosing a finite ether $\mathcal{E}$ which, for instance, discards events once the finite limit is reached. Also imaginable is blocking the sender until the receiver is ready to enqueue a message, yet this approach doesn't directly blend well with the current design of HLL.

Consequently, as in the HLL case, the transition system defined by the abstract array program encoding $\langle\!\langle \mathscr{M} \rangle\!\rangle^{\sharp}_{D}$ is finite if the DTR D is finite and considers all classes in $\mathscr{C}$ and the communication medium is finite as named above.

Treating Unbounded Sets of Links

Recall that from Section 9.1.1 on, we assumed a single-link property for most of this chapter, that is, each individual has at most one link of a given link-name. This assumption makes in particular the syntactical transformation easier as we needn't consider the case where an individual has a link to multiple (different) non-concrete individuals.

One solution of Chapters 5 and 6 is to leave links non-abstracted, that is, possibly having an unbounded number of links to the non-concrete identity C, a case for which we don't have a straightforward treatment in the syntactical transformation. In the following, we'll briefly discuss a special case in which we are actually able to also soundly treat an unbounded number of links in the syntactical transformation.

A possible encoding of link sets in the array program is to employ an array with index type *Cid* and boolean range as the type of link name λ instead of a value of type *Cid*, that is, a characteristic function of the unary "this-has-a-λ-link-to" relation,

$$\lambda^{\mathit{local}} : \texttt{array}\ \mathit{Cid}\ \texttt{of}\ \mathbb{B}; \tag{9.205}$$

One possible concrete syntax to work with link sets is the one shown in Figure 9.5 of Section 9.2.5 comprising a "pick" operation. If we apply the syntactical transformation from this section naïvely, we obtain something which needn't be sound as the declaration effectively changes to (cf. Section 9.4.3)

$$\lambda^{local} : \texttt{array}\ [1, N+1]\ \texttt{of}\ \mathbb{B}; \tag{9.206}$$

If the pick operation keeps its semantics, then it iterates the cycle at most $N+1$ times, and the counter n doesn't grow larger than $N+1$, which are to few iterations because we can easily have $N+2$ links to non-concrete objects alone.

Furthermore, if we have a situation like the one above where a pick-operation takes a transition and thus a global system step, properties employing the "next" modality ("X") are easily lost (but cf. Section 9.3.4 for a discussion of the "next" modality in interleaving semantics).

What may go wrong in a naïve application of the syntactic transformation is that the definition of set is not respected. If the set was implemented as usual, that is, adding an element *id* to the set only if there is no *id′* with $id = id'$, then we could add the summary node many times to the set if comparison for equality is treated as usual, i.e. the outcome is determined by an input.

In the bit-representation, in contrast, we simply cut off the index, so we have to adjust iteration, too. In principle, the "pick" operation should be able to do any (finite) number of iterations, or even infinite if sets are that general. In some variants of the DCS language (cf. Chapter 10), we even know that sets comprises only finitely many links, because in DCS we can add at most one link per step so after finitely many steps, we have only finitely many links.

Note that the transformation is not *generally* unsound. For example, if we remove the counter n from Figure 9.5, then only event sending remains as action. This action is transformed to a skip statement by Section 9.4.3, thus all iterations where the pick-statement yields C_C actually don't have *any* effect, thus iteration over the $N+1$ entries has the desired effect: if there is a relation to a concrete individual, it will receive an event, and if a relation to C_C is indicated, a skip instead of event sending will be executed.

This fits into the overall picture because at the next execution of C_C it can, as all its transitions are always enabled, act as if it obtained the event from the loop. In some variants of DCS though, we can ensure that we're in the safe case because it employs broadcasting instead of a loop, and it doesn't count with the broadcast, that is, for these DCS the generic transformation procedure also yields an approach to treat unbounded links.

9.4.4. Encoding EvoCTL* in CTL or LTL

Finite-state model-checkers typically support CTL or LTL, but not EvoCTL*, which we recommend as adequate for ETTS and thus for HLL models. The main differences being that EvoCTL* is a first-order logic, that we propose primitives to query aliveness, and to employ a three-valued in order to treat pre-mature disappearance.

Hence obtaining an array program encoding which further translates to the input languages of such model checkers as presented in the previous section is only a part of support for ETTS verification. The second difficulty lies in the assumption of an abstract structure in Chapter 5 which provides a semantics for function symbols in *abstract* states, while in the syntactically transformed HLL or array program, there is a priori no such counterpart.

In Section 9.4.4, we begin with a brief discussion how to support at least fragments of EvoCTL* given CTL or LTL facilities in a finite, not abstracted model. Partly anticipating the discussion of case-studies in Chapter 10, note that the proposal of EvoCTL* is rather new and meant to support the results of Chapter 5, how such particular specification logics behave under abstractions like DTR.

The case-studies presented in Chapter 10 mainly focus on the aspect of obtaining DTR by syntactical encodings and the principal applicability of DTR to the class of ETTS defined by languages like HLL. For this reason, it is to be understood that the following discussion only gives preliminary answers in particular on how to reduce the three-valued semantics of EvoCTL* to classical temporal logics. In the case-studies, properties have been carefully chosen to remain in the definite fragment of EvoCTL* and to be biased towards the safe side, considerations which are only enabled by having such issues clearly prepared in form of EvoCTL*.

Section 9.4.4 provides a procedure to interpret terms in the abstract system, the idea is basically to effectively also apply the syntactical transformations of Sections 9.3.3 and 9.4.3 by turning terms in the formula to propositional observers in the model.

Finally, Section 9.4.4 points out an alternative which is particularly suitable if the property to verify is given in form of a Live Sequence Chart (cf. Chapter 10), namely to employ what is known under so various names as test automaton, observer, or monitor.

First-Order and Life-Cycle Queries in LTL or CTL

In Chapter 4, we've defined the LTL [11] fragment of EvoCTL*, which is basically terms over modalities for "next", "globally", "finally", and "until". That is, in a common model-checker like VIS or SMV we don't find support of quantification and typically only terms over the variables in the array program, but no explicit primitives for querying an object's life-cycle or refer to message send and consume actions, which we'll consider for DCS/METT in Chapter 10.

First of all, we have to assume an HLL with a finite ETTS semantics in order to apply the named model-checkers at all without abstractions. In that case, outermost quantification can be spelled out, just like practiced with parameterised systems.

Unless we're able to argue with Query Reduction (cf. Chapter 7), all possible valuations of logical variables have to be considered. With nested quantification, we inherit the problems discussed in Chapter 4 in the context of prenex normal forms.

Life-cycle queries, that is, operators $\odot$, $\circledcirc$, and $\otimes$ can directly or indirectly be expressed in terms of the alive flag introduced in the encoding Section 9.4.2. Aliveness is directly equivalent to the valuation of the boolean aliveness flag. Being new or doomed are path or state properties as they intuitively relate to rising or falling edges of the aliveness

[11] for brevity, we'll only discuss LTL in the following

flag, yet in corner cases, a deletion directly followed by a creation may leave the alive flag constantly set.

This can be overcome by adding additional variables to the array program with the purpose to *observe* creation and destruction, a common technique in practical formal verification. They're then updated whenever a create or destroy operation is executed, we only have to take care of the pathological situation where an individual is undergoing multiple creations and destructions in a single transition program. These cases can be syntactically excluded, for example by allowing at most one creation and destruction per transition program, or traced to indicate that the named observers may not be interpreted literally.

In case object creation is supported natively by the employed model-checker, one may assume that there is also support for querying the state in objects' life-cycle.

Three-Valuedness in LTL or CTL

In Chapter 4, we proposed a three-valued semantics for EvoCTL* in order to adequately treat pre-mature disappearance of individuals referred to in formulae. There is typically no native support for three-valued outcomes in the model-checkers named above, so we see basically three choices.

Firstly, with the precise EvoCTL* semantics of Chapter 4, we can in some cases mimic it with means of LTL and CTL employing life-cycle queries. Particularly interesting are formulae known to evaluate definite. Recall from Section 4.3.4 that we demonstrated how to express some (definite) evolution chain quantifications by identity quantification and additional life-cycle queries. This applies, for instance, to the large and relevant class of EvoCTL* properties equivalent to Live Sequence Charts where life-lines indicate how to add life-cycle queries (cf. Chapter 10).

A second option, as long as we're lacking automatic support, is to carefully craft EvoCTL* formulae such that they're biased in a safe way, that is, that all indefinite outcomes in the three-valued semantics map to a negative answer in the two-valued semantics of LTL and CTL model-checkers. Note that, in a sense EvoCTL* vs. plain LTL or CTL with life-cycle queries provides a separation of concerns between two cognitive levels of abstraction. With EvoCTL*, we can write specifications firstly without caring about life-cycle queries. With the EvoCTL* semantics, we can then identity whether the formula can safely be checked in a biased environment.

Thirdly, we can extend the procedure of Section 9.4.4 below, namely to move propositional terms from a formula into fresh observer variables in the array program, from the case of non-concrete individuals to the case of non-alive individuals.

Then (at least for some) indefinite formulae, the result is that both, the formula itself and its negation don't hold, a valid representation of an indefinite outcome.[12]

For example, a proposition observer

$$h' := f(\langle\!\langle \mu \rightarrowtail x \rangle\!\rangle) > 0 \tag{9.207}$$

[12] in the sense that $1/2$ can alternatively be read as the set $\{0, 1\}$

turned into

$$h' \mathrel{:=} (f((Cs\texttt{[}\mu\texttt{]}.\mathit{alive} \mathbin{?} i : Cs\texttt{[}\mu\texttt{]}.x) > 0 \tag{9.208}$$

(see Section 9.4.4 for the declaration of μ) which employs the fresh input i in case the identity μ doesn't denote an alive individual.

Note that this procedure works best if we know the binding of logical variables in advance, which is the case with QR or with the procedures of Section 9.4.4.

EvoCTL* and Syntactically Obtained DTR

Recall from Chapter 5 that we assumed an abstract structure together with the abstract topologies. The reason is the local state of the abstract individual with identity C.

Its local state is top element of the set of abstract local states, i.e. $\top \in \Sigma^\sharp$. Evaluation of function symbols in $\Sigma^\sharp$ is provided by the assumed abstract structure, for instance when considering formulae like

$$\forall\, x : C \,.\, \mathsf{G}\, f(\sigma(x).x) > 0 \tag{9.209}$$

When syntactically transforming an HLL model $\mathscr{M}$ into $\mathscr{M}_D^\sharp$ or directly into the array program $\langle\!\langle \mathscr{M} \rangle\!\rangle_D{}^\sharp$, there is a priori not such thing as an abstract structure.

Note that even when employing DTR in combination with QR, that is, when logical variables obtain a fixed binding to *concrete* individuals, the problem remains because navigation expressions may occur in EvoCTL* formulae and may navigate via links pointing to C, for instance

$$\varphi = \mathsf{G}\, f(\sigma(x \rightarrowtail \lambda).x) > 0. \tag{9.210}$$

If $x \rightarrowtail \lambda$ denotes C, then the valuation of $\sigma(x \rightarrowtail \lambda)$ shall also be $\top \in \Sigma^\sharp$.

One straightforward idea to treat this issue technically is to apply the same transformation discussed in Section 9.3.3 for HLL or in Section 9.4.3 for array programs. To this end, the transformed model $\mathscr{M}_D^\sharp$ or the array program $\langle\!\langle \mathscr{M} \rangle\!\rangle_D^\sharp$ has to provide additional inputs with the sole purpose to be referred to in the transformed formula. For example, (9.210) turned into

$$\varphi_D = \mathsf{G}\, f((x \rightarrowtail \lambda = \mathsf{C} \mathbin{?} i : \sigma(x \rightarrowtail \lambda \rightarrowtail x))) > 0 \tag{9.211}$$

where i is a fresh input and we need the constant C in the signature, which constantly yields the identity C, and where we assume conditional expressions similar to Section 9.3.2.

Recall that we may (non-symmetrically) refer to particular identities *after* a possibly applied QR. Optimisations as discussed in Section 9.3.3 apply correspondingly.

Considering (9.211) a bit closer shows that this simple example actually doesn't carry any information: the input value can be freely chosen from Σ thus, unless f is constant, the formula (9.211) doesn't hold, neither does it's negation. This indicates that, when choosing a DTR for a given property, one should also consider the amount of navigation taking place, in addition to the bare number of quantified variables (cf. Chapter 8); a natural aim is to ensure, by case-splits, that the navigated objects are also concrete whenever they're navigated and contribute to the truth value of the property.

Still, the transformation is necessary for soundness; even if one *aims* at navigating only concrete objects, the model is free to have errors which spoil such aims. Recall from Chapter 8 that case-splits can only control certain points in time, for example ensure concrete objects at the time of activation of parts of the formula but not necessarily for all points in time.

By soundness, the transformed formula (and its negation) will fail in case navigation via a non-concrete object which means errors are biased towards the safe side.

Technically, a transformation of formulae can be implemented by turning all terms, that is, all non-temporal sub-formulae into predicates or boolean propositions that are *driven* by the model.

For example, assume we're considering formula (9.211) with the fixed representative assignment $\theta = \{x \mapsto 1\}$ (cf. Chapters 8 and 7). Then we'd introduce a fresh constant link name

$$\texttt{const}\ \mu : \mathit{Cid} = 1; \tag{9.212}$$

in the declaration part of $\langle\!\langle \mathscr{M} \rangle\!\rangle$ and introduce a boolean variable h which are updated as

$$h' := f(\langle\!\langle \mu \rightarrowtail \lambda \rightarrowtail x \rangle\!\rangle) > 0 \tag{9.213}$$

at an appropriate point in time, then (9.211) turns into

$$\varphi = \mathsf{G}\, h' \tag{9.214}$$

Then applying Section 9.4.3 to the array program including (9.213) changes (9.213) to

$$\begin{aligned} h' &:= (f(\langle\!\langle \mu \rightarrowtail \lambda \rightarrowtail x \rangle\!\rangle^{\sharp}_{D}) > 0 \\ = h' &:= (f((\mathit{Cs}\texttt{[}\mu\texttt{]}.\lambda = \mathtt{C}_C \mathbin{?} i : \mathit{Ds}\texttt{[}\mathit{Cs}\texttt{[}\mu\texttt{]}.\lambda\texttt{]}.x) > 0 \end{aligned} \tag{9.215}$$

assuming λ denotes instances of class D.

Thereby we achieve that formula (9.214) needn't be touched but we concentrate all modifications in the model description, which less theoretical than technical advantage. Then there's only a single place to implement and support more intricate syntactical transformation, namely the array program, treatment of formulae consists only of the simple splitting into propositions.

Observers (or Test Automata (or Monitors))

A consequent continuation of the introduction of observer variables for propositions in the formula is to employ *observers* [182] (also known under the names of test automata or monitors) also for temporal parts.

In [98], we lay out how the procedure in particular applies to the visual formalism Live Sequence Charts (LSC), which we consider in Chapter 10. The basic idea is to add a transition system implementation of an automaton to the array program which observes the model by having transitions that depend on program variables and which thereby tracks the state of the model. Checking whether the HLL (or its array program encoding) satisfies an LSC then in some cases reduces to checking whether a designated error state of the observer is reachable.

More expressive subclasses of LSCs reduce to checking certain small and fixed formulae, thus in particular all treatment of propositions (or EvoCTL* terms) is part of the model transformation as discussed in Section 9.4.4. It applies particularly well after QR because we can then encode the fixed assignment of logical variables as demonstrated with the example in Section 9.4.4.

9.5. Discussion

As HLL is intended to generalise DCS [9] as well as the core fragment of UML [45, 46], it is certainly related to both. It borrows its syntax mostly from UML with a C-like action language. The difference to DCS and any approach to UML verification and semantics we're aware of, is that we explicate scheduling and non-determinism in creation and inputs.

Abstracting from the communication means has already been done in the earlier UML specifications. It basically only requires that the order of messages from the same sender is preserved but allows to generalise to priority queues. Our usage of an ether is just taking the step to a completely parameterised setting. We discuss the relation to DCS and UML in more detail in Chapter 10.

The rich literature on the syntactical declaration and detection of symmetry is discussed in Section 9.2, a good survey is provided by [134]. The main difference to our approach is that the works surveyed [134] typically remain within a given, limited semantical setting, for example, the most recent [59] operates in the context of the Promela language, which obtains its semantics from the SPIN model-checker, thus there is in particular a fixed scheduling and a fixed treatment of inputs. That the symmetry of the creation oracle has to be taken into account has lately been discussed in [2], who found that the process creation in SPIN is highly non-symmetric[13]

Given any abstraction procedure, it is a natural desire to obtain the abstract transition system by an easy syntactical transformation for reasons discussed in the introduction of this section, a good reason among them being the enabled re-use of existing model-checking tools. For the quotient structure-oriented branch of symmetry reduction (cf. Section 9.3), this has for example been proposed by [63].

The DTR implementation of Sections 9.3 and 9.4.3 is, as discussed in Sections 9.3, related to [28, 29] and [31] and of course to [127]. The main difference to the former is that they remain on a rather technical level, that is, compare to Section 9.4.3 rather than to Section 9.3, similar to our earlier works [48, 49]. Furthermore, [28, 29] and [31] propose syntactical procedures only for particular examples, which are less general than

[13] this doesn't pose a problem for [59], as they seem to consider only systems with fixed extension, that is, where all participating processes are created at once in the initial step and don't disappear at runtime

ours in [48, 49], and far from the wide generalisation as presented here. In particular the insights of Section 9.3.4 are new and highly relevant as they tell us that the procedures of [28, 29] and [31] don't generalise as easily as one would imagine, but obtain their soundness partially from the structure of the model.

Also new is our discussion of levels of precision in Section 9.3.3, that is, the exact relation between the abstraction obtained by Chapter 6 for the ETTS of an HLL model and the transformed HLL model obtained by Section 9.3; the discussion in [178] only argued in terms of efficiency (using less auxiliary variables, using fewer inputs), but didn't go as far as identifying the relation to precision. The difference to [127] and related publications is that it is unknown how the abstract transition system is actually obtained, that is, whether it employs a procedure similar to the syntactical transformations of [28, 29], [31], and ours, or a completely different procedure, possibly also treating concurrency.

The procedure of 9.3 also gives a hint why the procedure is named "Compositional Model-Checking" in the original work [127]. A classical understanding of compositional verification is that parts or components of a system are checked in isolation in an environment which over-approximates the behaviour of the remaining components. Then the overall reasoning for two components is than is that if component A satisfies its requirements under the assumption of B's requirements, and if B satisfies its requirements under the assumption of A's requirements, then the composition of A and B together satisfies certain requirements.

The DTR approach has in common with this reasoning that certain parts, or components of the system, namely those named in the DTR (in the sense of Def. 6.1.1), are put into a hostile environment which resembles chaos on first sight but preserves certain properties, in particular of the HLL description, thus is a very coarse abstraction of the real, concrete environment of these components.

10. Case Studies

With this chapter, and in particular Section 10.2, we can finally close the circle to the problem, which originally raised our interest in this particular abstraction for this particular class of systems: the formal verification of scenarios (given as LSCs) against UML models, of which [177] can be seen as a pre-study.

With Section 10.2, we confirm the findings of [177] on solid grounds. We have by Chapter 3 a rigid formal model of computation, while, for instance, dynamic creation and destruction and evolution played a far less clearer role in [177]. And we have by Chapter 4 a comprehensive temporal logic, which is aware of pre-mature disappearance of individuals and can confirm that the dynamic LSC extension proposed in [177] lies in the definite fragment of EvoCTL*.

For completeness, Section 10.1 discusses a more focused, less baroque setting than UML. The DCS language emerged in a project with the dedicated aim to analyse systems of systems and in a sense only comprises the essence of means that are necessary to formally model, for instance, the Car Platooning application [9]. One difference to UML is that individuals are more encapsulated and accept stimuli only in form of messages, but not via direct read or write access to local variables, which is possible in UML models.

Both sections are organised similarly. For instance, Sections 10.1.1 and 10.1.2 recall syntax and intuitive semantics of DCS and METT and discuss the relation to HLL from Chapter 9 and EvoCTL* from Chapter 4. Section 10.1.3 introduces a case-study, in this case the Car Platooning application, and reports results. As we've got the extensive discussion of encodings and implementations of the QR/DTR approach, we won't elaborate much on the details here. All results have been obtained based on an array program encoding in the SMI language and tools implementing the syntactical transformations of Section 9.4.3.

Note that this chapter is about case-studies in two senses. Firstly, the languages DCS/METT and UML/LSC are cases challenging HLL from Chapter 9 and EvoCTL* from Chapter 4, and secondly, the applications Car Platooning and Automated Rail Cars System indicate whether the QR/DTR approach is useful in these two settings.

10.1. DCS and METT

The Dynamic Communication System description language [9] (DCS) has been designed (under our participation) to provide a minimal language to capture the topological aspects of applications like Car Platooning.

This application class can be characterised as a system of systems. Each of the sub-systems has a finite-state control part and may interact with other sub-systems via asynchronous message communication. This choice was based on the observation that systems like Car Platooning or the European Train Control System (ETCS) Level 3, where trains negotiate moving authorities with roadside equipment, employ wireless communication that shouldn't be represented by synchronous communication.

It is inspired by Communicating Finite State Machines [21] (CFSM), and its relation to parameterised systems is similar to UML or our HLL: there is a single, finite-state description of behaviour which is executed by a number of processes, but this number of processes and their interconnection is completely unrestricted.

The DCS language has, for instance, fostered the development of the partner abstraction [7, 12] and has been subject of (partly joint) works on refinement for the DTR abstraction [10, 166].

10.1.1. DCS

Formally (quoting from [10]), a *DCS protocol* is a seven-tuple

$$\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\mathrm{msg}}, \mathit{succ}) \tag{10.1}$$

with a finite set Q of *states* a process may assume, a set of *initial states* $A \subseteq Q$ assumed by newly appeared processes and a set of *fragile states* $\Omega \subseteq Q$, in which processes may disappear, A finite set χ of *channels*, each providing potential links to other processes, A finite set Σ of *messages* and *environment messages* $\mathcal{E}_{\mathrm{msg}} \subseteq \Sigma$, that is, messages that may non-deterministically be sent by the environment, and a *transition relation* '*succ*', determining each processes' behaviour.

The transition relation *succ* comprises four different kinds of labelled transitions between two states from Q, namely send, receive, modify, and conditional transitions.

A transition $(q, c, m, c', q') \in \mathit{Snd}$ sends, if in state q, a m-message carrying one of the identities stored in channel c' or the own identity to c and changes state to q'.

A receive transition $(q, m, c, \mathit{op}, q') \in \mathit{Rec}$ consumes an m event and execute operation op on channel c in the current local state. Such operations are typically channel assignment or adding an identity to a channel.

Other transitions are the trigger-less $(q, c_1, \mathit{op}, c_2, q') \in \mathit{Mod}$ to set the content of channel c_1 to the result of applying op to channel c_2, and $(q, \mathit{em}, c, q') \in \mathit{Cnd}$ which tests channel c for emptiness.

Less minimal variants of DCS have been proposed in [9, 150, 95, 2], which differ in details but agree in fundamental concepts.

There are, for instance, variants with multiple such DCS protocols, which are considered to be compiled into a single one above. Others include explicit create and destroy operations, in addition to environment-driven creation.

DCS and HLL

The relation to HLL is rather straightforward. Assuming the setting of multiple DCS protocols, there is one HLL class per protocol. DCS protocol states become HLL states and transitions translate directly into transition programs assuming a corresponding signature with function symbols for link manipulation.

Specialties of DCS protocols are fragile states. Instances of DCS protocols are nondeterministically created by the environment and they may be destroyed by the environment whenever they're in a fragile state.

This can be modelled by an explicit HLL class playing the environment. As HLL allows to query the local states of other processes, the destruction transitions can have guards which check whether the destination is fragile. Candidates would be determined by link inputs. Similarly, events carrying identities can be sent by the explicit environment to individuals.

For readability, we'll right-away use the HLL action syntax to label DCS models in Section 10.1.3, instead of the tuple notation from Section 10.1.1.

One particular property of DCS is that its inherently symmetric in identities. This can easily be checked with the Ip/Dill criteria. The reason is that DCS doesn't have data, which could be mixed with identities. Furthermore, neither the logic nor the requirements specification logic support singularities.

10.1.2. METT

The DCS language is complemented by a first-order variant of LTL which, in addition to life-cycle queries, has primitives to refer to communication, namely sending and consumption of events, and whether an event is pending.

It is, like EvoCTL*, a direct descendent of VTL/ETL [191, 190] without transitive closure (cf. Chapter 4) but with the just named communication primitives. It shares with VTL/ETL the biased semantics.

The syntax of Mett is given by the following grammar.

$$\begin{aligned}\phi ::= {} & p_1 = p_2 \mid \mathit{instate}[q](p) \mid \mathit{conn}[c](p_1, p_2) \\ & \mid \mathit{pend}[m](p_1, p_2, p) \mid \mathit{send}[m](p_1, p_2, p) \mid \mathit{recv}[m](p_1, p_2, p) \\ & \mid \odot\ p \mid \otimes p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \forall\ p.\phi \mid \mathbf{X}\ \phi \mid \phi_1\ \mathbf{U}\ \phi_2\end{aligned} \tag{10.2}$$

Here, c denotes a channel and m a message, p is a logical variable.

The $\mathit{instate}[q](p)$ predicate holds if the individual denoted by p is in the given local state, $\mathit{conn}[c](p_1, p_2)$ holds if there is a link from p_1 to p_2.

The predicates $\mathit{send}[m](p_1, p_2, p)$, $\mathit{recv}[m](p_1, p_2, p)$ and $\mathit{pend}[m](p_1, p_2, p)$ indicate whether p_1 has send a message m to p_2, or whether such a message has just been consumed, or is currently pending.

Both, $\odot\, p$ and $\otimes p$ are known from ETL/VTL and EvoCTL*.

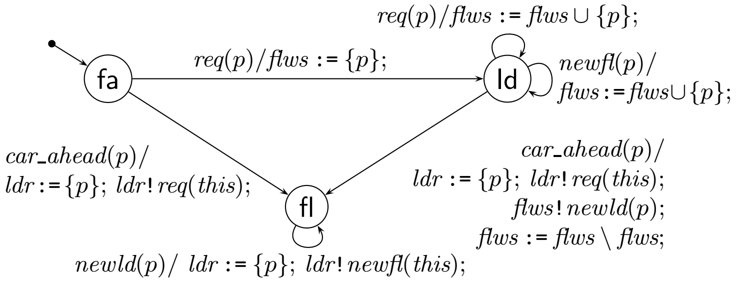

req(p)/flws := flws ∪ {p};
req(p)/flws := {p};
fa
ld
newfl(p)/
flws := flws ∪ {p};
car_ahead(p)/
ldr := {p}; ldr ! req(this);
car_ahead(p)/
ldr := {p}; ldr ! req(this);
flws ! newld(p);
flws := flws \ flws;
fl
newld(p)/ ldr := {p}; ldr ! newfl(this);

Figure 10.1.: **Platoon merge.** Simplest implementation. The action language syntax is borrowed from HLL.

METT and EvoCTL*

The relation between METT and EvoCTL* is quite obvious as both have common ancestors. The only new aspects are communication queries. To support these, we've got to provide access to the current state of the ether, which is part of an individual's local state.

10.1.3. Case-Study: Car Platooning

The QR/DTR combination has been applied to certain variants of our running example, the Car Platooning case-study. The focus was on the merge protocol, for instance in [9, 10], the complementing split procedure has been considered in [166].

In [9], a certain strategy to approach this class of systems is proposed. Firstly, a given requirement should be checked on a finite instantiation of the system, that is, with a finite upper bound. This may already reveal errors, and can benefit from fast underapproximating tools. Only in case no errors are found, the step to the abstraction should be taken.

The experiments we cite in the following have been carried out on different hosts, employing the automated translation from an XML representation of DCS protocols to the SMI model-checker input language from [150]. This result, stated as an infinite array program, has then been post-processed with our implementation of the procedure of Section 9.4.3. Similar to the UML case below, the precision issues have not been known back then.

Model

Figure 10.1 shows a minimal implementation of the merge procedure. Each car starts out as a free agent when entering the highway, thus the initial state is fa. Then there are two options. Either the car detects via some sensors another car in front, as modelled by reception of an environment event *car_ahead*, or it is approached by some other car from the back and receives a merge request *req*.

In the former case, the car in front is assumed as new leader, the identity of the front car is part of the *car_ahead* message, a request is sent with the own identity as parameter,

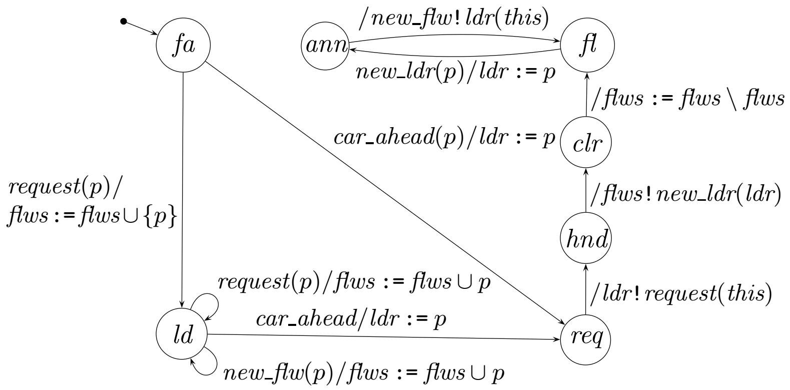


Figure 10.2.: **DCS Protocol for merge** [9].

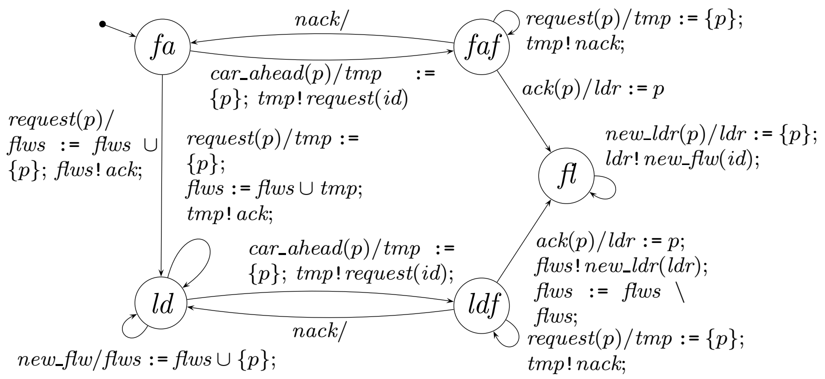


Figure 10.3.: **Fixed DCS Protocol for merge** [9].

and the car changes state to fl, thereby becoming a follower.

In the latter case, the request is accepted, the sender is assumed as (only) follower, and the car changes state to ld. In the leader state, further cars can join the platoon by sending requests.

A critical operation is the merge of two platoons. In that case, the leader approaching from behind receives a *car_ahead* notification, this time with the identity of the front leader. The back leader assumes the front leader as new leader, informs all of its followers by a *newld* message of the new leader, dismisses all its followers, and becomes itself a follower. The followers of the former back leader react on the *newld* message by assuming the received identity as new leader and announcing themselves by a *newfl* message. The front-leader consumes these messages and extends its follower links.

Note that Figure 10.1 employs sequences of actions per transition, which is not provided by the formal definition of Section 10.1.1, thus they're read as abbreviations for a variant with correspondingly many intermediate states. Figure 10.2 shows a completely unfolded variant. In [150], the variant with action sequences is considered.

Analysing this implementation with the under-approximation approach (see above) in a setting with at most three cars already reveals problems (see below). These problems

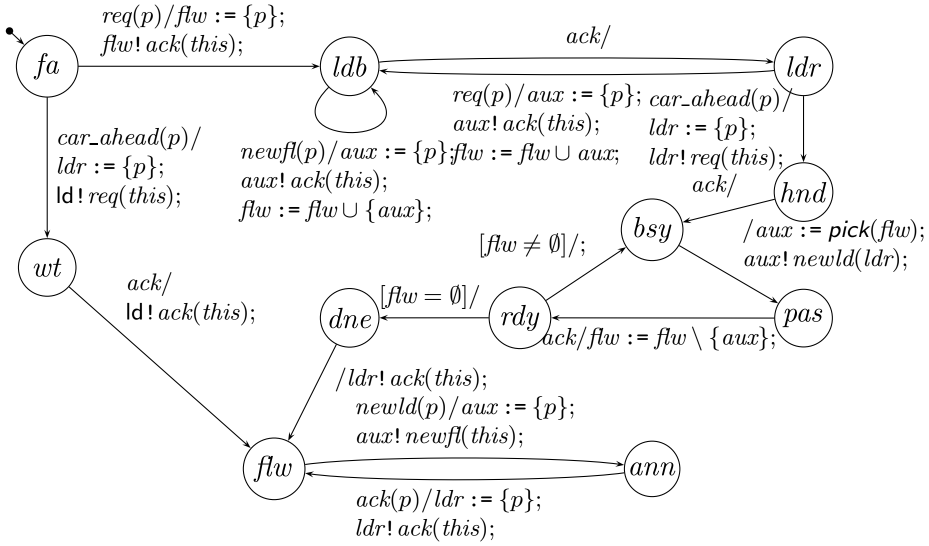


Figure 10.4.: **The strengthened merge protocol** [10].

are fixed by introducing the possibility to positively or negatively answer a request. This is shown in Figure 10.3, again in abbreviated form.

For the combination of Partner Abstraction [7, 12], as demonstrated in the joint work [10], a further strengthened variant had to be employed in order to have both techniques succeed, the static analysis and the QR/DTR model-checking.

This variant is shown in Figure 10.4. Note that it employs the pick operation as discussed in Section 9.2.5. In this case, we convinced ourselves manually of the symmetry of this loop, the automatic analysis is not yet this sophisticated. Further note that this variant is not a single link model. The number of followers per car is not limited at all. Thus with the results named below, we have successfully approached a setting with principally unbounded links, which is possible because the case-study meets the criteria discussed in Section 9.4.3.

10.1.4. Results

As a first demonstration of the feasibility to analyse DCS, the following property has been considered for the DCS protocol shown in Figure 10.2.

$$\begin{aligned}&\mathsf{G}\ \forall p_1, p_2 . \mathit{instate}[\mathit{fl}](p_1) \land \mathit{instate}[\mathit{fl}](p_2) \land \\ &p_1 \neq p_2 \rightarrow \neg(\mathit{conn}[\mathit{ldr}](p_1, p_2) \land \mathit{conn}[\mathit{ldr}](p_2, p_1))\end{aligned} \tag{10.3}$$

It requires the absence of a pathological topology, namely that two followers mutually consider each other to be the leader.

Following the proposed methodology, an analysis restricted to a concrete system of at most three cars already revealed that the lack of interlocking in the merge protocol allows overlapping merges which lead to the undesired situation.

This can be overcome by modifying the model to Figure 10.3. On this model, the under-approximation didn't reveal a violation of (10.3). Applying QR/DTR yields a counter-example for the initial spotlight. Its spuriousness has been identified by close examination. The source of spuriousness is that the abstracted cars may send *newld* messages at times where they're not supposed to, that is, in abstract topologies which don't concretise to a concrete topology where one of the other cars would send this message.

We effectively refine the abstraction by explicitly excluding the spurious behaviour using the following assumption.

$$\mathsf{G}\,\forall p_1, p_2, p_3.\mathit{send}[\mathit{new\_ldr}](p_1, p_2, p_3) \rightarrow \mathit{conn}[\mathit{flws}](p_3, p_1) \tag{10.4}$$

It basically says that cars only send *newld* messages if they're supposed to do so, namely if the recipient is currently a follower. With this assumption added, verification succeeds. The message queue has to be restricted to a finite length though, as discussed in Chapter 9.

Note that property (10.3) above is a rather simple topology invariant which can alternatively (and faster) be established by Partner Abstraction. In [10], the following more involved property is considered.

$$\forall\, c_b, c, c_f\,.\,\mathsf{G}\;(\mathit{send}[\mathit{newld}](c_b, c, c_f) \rightarrow (\mathit{conn}[\mathit{ldr}](c, c_b)\;\mathsf{U}\;\mathit{conn}[\mathit{ldr}](c, c_f))). \tag{10.5}$$

It states that, if a platoon led by car c_b merges with a platoon in front led by car c_f, then during the merge c_b hands over its followers to c_f.

Note that this is, by the employed U operator, a true liveness property, which cannot be established with Partner Abstraction.

Without refinement, this property fails for the model shown in Figure 10.4 with a spurious counter-example similar to the one of the previous property. In this case, refinement employs Topology Invariants obtained via Partner Abstraction based analysis. Restricting the model to adhere to the legal topologies allowed us to verify this property under the fairness assumption that each concrete car is always finally scheduled, and with a finite queue-length as discussed in Chapter 9.

10.2. UML and LSCs

As discussed in Chapter 3, the Unified Modelling Language [138, 141, 140] (UML) gains popularity for the model driven development of safety or mission critical system, thus there is a clear need for formal verification.

There are two primary obstacles. Firstly, UML still lacks a complete formal semantics. Secondly, an aim of the UML is to support all stages of software development, from the capturing of *use cases* and *scenarios* as requirements, to fully executable models of the system behaviour in terms of *class diagrams*, *state machine diagrams* and *activity diagrams*, to the deployment of tasks to a distributed infrastructure in *deployment diagrams*.[1]

[1] And these are only half of the diagram names provided by UML 2.0.

The underlying idea is to have orthogonal *views*. For example, a class diagram declares the (data) types of objects in the system while state machines give their temporal behaviour. Together they define traces of *object diagrams*. An object diagram is basically a graph whose nodes are class instances, called *objects*, interconnected via named links, thus resembling our topologies. Two object diagrams are in transition relation if the destination is just the effect of applying a state-machine to the source, resembling our HLL from Chapter 9. Given the set of traces of a UML model, a main application for formal verification is to ask whether it adheres to the required scenarios given by *sequence diagrams* (or the equivalent *collaboration diagrams*, or the related *timing diagrams*).

For the sheer number of different diagrams, with differently clear interrelation, we follow the typical approach in this area. Namely, to identify a relevant set of diagrams and of these a relevant core for which one can give a formal semantics. A first criterion for relevance is basically that the expressive power of the chosen cores is sufficient to describe *executable* models, that is, which are sufficiently restrictive to enable code generation. A second, and weaker criterion is that a reasonable subset of models is covered, that is, that interesting case studies with the aspects which make UML difficult are possible.

This section is structured as follows. In Section 10.2.1, we introduce the executable UML core we consider, which roughly follows [45, 46].

As property specification language, we don't consider UML's sequence diagrams, but Live Sequence Charts [42, 43]. Both visual languages have a common ancestor, the Message Sequence Charts of the ITU [91, 90, 92], but SDs inherited from MSCs serious issues concerning expressive power and formal semantics [80]. For example, they directly don't have means to distinguish necessary progress (liveness) from possible, non-necessary progress.

Section 10.2.2 briefly recalls syntax and semantics of LSCs, for the relation to EvoCTL* we'll refer to a publication we co-authored. We'll only discuss one issue in more detail which has already been sketched in [178], namely how to fit the observation of creation in the LSC sense into the QR/DTR framework. This issue would fit into Section 4.5, but is very particular to LSCs.

10.2.1. UML

Following [46], a UML model is a quadruple $U = (E, C, L, M)$ comprising a finite set E of events, a finite set C of classes, all active, and functions L and M providing classes with associations and state-machines. Events from E are typed, just like HLL events, in addition we distinguish whether an event may be sent by the environment or whether it is only used internally in the system.

Given a class $c \in C$, its set of associations $L(c) = \{l_1, \dots, l_n\}$, $n \in \mathbb{N}_0$, is finite and may be empty. A class has finitely many local variables $X = \{x_1, \dots, x_m\}$ of basic types in the sense of Chapter 9.

The state machine $M(c)$ of a class is a quadruple (S, S_0, R, A) comprising a finite set of states S, sets of initial states $S_0 \subseteq S$, a transition relation $R \subseteq S \times S$, and a transition labelling A assigning each transition $r \in R$ a trigger, a trigger/action pair, or only an action, where a trigger is an event from E, possibly also guarded by a condition similar to HLL.

An action manipulates associations and local variables and may refer to event parameters, and that plain actions at least comprise association manipulation and event sending.

As discussed in more detail in [46], this simplistic notion of UML models is not a severe restriction of generality of our proposal as it already captures many essential features by appropriate encodings.

For instance, hierarchical state machines unfold into the flat ones considered here following well-known procedures. Methods, unless recursive, can be encoded by "inlining" them into transition annotations. Finally, inheritance can be translated into one class per feature added in a specialisation and a new one-to-one association pointing to the superclass (cf. [176, 46]).

UML and HLL

The encoding of core UML models as introduced above in HLL is completely straightforward.

UML classes become HLL classes, UML associations become link names, and statemachines correspond directly to HLL transitions with transition programs.

The only specialty is the optional environment, which may non-deterministically send events marked accordingly (see above). This environment would be introduced by an additional class with always enabled transitions.

The semantics is a strict interleaving semantics, inputs and creation are not restricted and symmetric. The ether is instantiated by one FIFO queue per object.

In the UVE tool-chain [159, 158], symmetry in identities is not given for granted because the action language of the employed UML tool Rhapsody [87] is C++. This action language allows all forms of pointer arithmetics and conversion into data, so symmetry has to be established separately. For the case-studies, we employ an implementation of the Ip/Dill criteria check which operates on the SMI [24, 179] model-checker input language and is aware of the NULL singularity. The check typically succeeds because there are development guidelines for UVE which encourage symmetric designs.

10.2.2. LSC

Scenario-based approaches, and LSCs in particular, are highly relevant for the formal specification of requirements in model based development [175, 3, 102, 36, 26, 18, 17]. They're in particular of first choice because many model based development models propose to capture early requirements in form of use-cases and scenarios, which are later refined. In the joint work [22, 23], an elaborated refinement procedure is proposed.

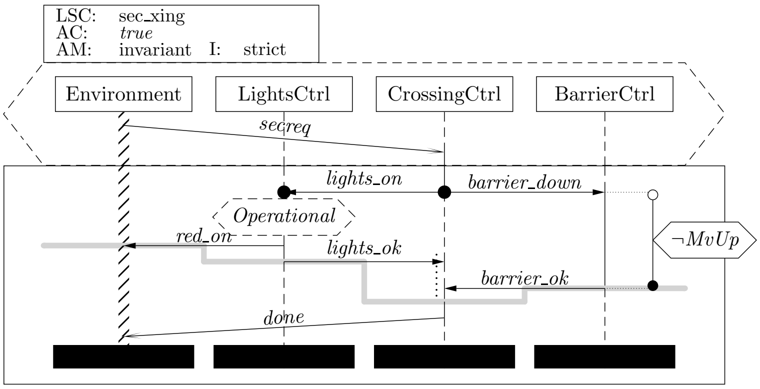


Figure 10.5.: **Live Sequence Chart** for a system of level-crossing controllers.

Static Binding LSCs

For our purposes, it's sufficient to introduce the visual formalism LSC by an example. For a more formal and complete treatment, we refer to [42, 43, 97].[2]

As an example, think about a requirements specification for a level crossing in form of an LSC. The level crossing shall be driven in a decentralised fashion by a group of controllers, a central *crossing controller* and separate controllers for the traffic *lights* as well as for the *barriers*.

Whenever the environment, typically a train, request that the level crossing shall be secured, we expect a certain sequence of communication and behaviour, unless the system is not operational.

In form of an LSC, such requirements can be captured as shown in Figure 10.5. Each controller and the environment are represented by vertical *life lines* (or *instance lines*). The implication between the request and the following behaviour is captured by showing the initial communication in dashed hexagon, the so-called *pre-chart*, and showing the following behaviour in the *main chart*. Its solid outline indicates that *whenever* the pre-chart is observed, the remainder shall follow.

The particular communication begins with the crossing controller simultaneously starting both sub-controllers. The small dashed hexagon is a *condition*. It queries the state of the lights controller when *lights_on* is received and exits the chart if the controller is not operational. Then another chart is in charge, this chart is *successfully exited*.

On the side of the barrier controller, the solid hexagon is a *local invariant* which requires the barrier to move downwards until completion is reported to the main controller. The reports from the sub-controllers may occur in any order as indicated by the dashed

[2] For completeness, note that two LSC dialects spawned from the original [42, 43]. The branch defined by [97], which we consider, has a more classical view on LSCs. In a sense they're simply viewed as a more readable variant of temporal logics in formal methods based development, which complement an implementation. The implementation can then be checked for satisfying the LSC requirements specifications.

The branch defined by [81], views LSCs as the programming language itself. A so-called *play engine* animates a set of LSCs, it plays out the scenarios according to environmental stimuli.

lines in parallel to the life line.

After both sub-controllers reported, the main controller *may* send back an (asynchronous) acknowledge to the requester. In contrast to this, the report of the barrier controller *must* occur finally. This difference is graphically indicated by the solid line segment between both messages on the barrier controller's life line and the dashed life line before the acknowledgments.

That is, LSCs are a conservative extension of MSCs. Each element obtains an additional *modality*, graphically indicated by dashed or solid lines. Intuitively, a solid outline denotes *mandatory* requirements: in order to satisfy the chart, a system *must* make the local invariant hold, the barrier controller report *must* finally be observed, and the whole chart *must* be completed whenever the pre-chart has been observed. A dashed outline denotes *possible* requirements: for example, in order to satisfy the chart, a system needs at least one run adhering to the chart. Mandatory elements are often called *hot*, possible ones *cold*.

The semantics of an LSCs is a Timed Symbolic Büchi Automaton [97] whose states are the *cuts* of the chart. The gray step-line in Figure 10.5 is a cut, which records how far the LSC has been traversed. The legal sequence of cuts is determined by

1. the order of elements along the instance line, elements closer to an instance line's *head* are required to occur strictly before strictly farer ones, and

2. the principle that asynchronous messages, as indicated by sloped lines with open arrows, are received strictly after sending.

Occurrences can in addition be grouped in *simultaneous regions*, as indicated by black circles on the life lines, and explicitly relaxed in *coregions*, as indicated by the dashed line in parallel to the life line.

Dynamic Binding LSCs

Note that we've implicitly assumed Section 10.2.2 that the relation between instance lines and controllers is fixed. This is the setting of [97]: they consider a model of a single level crossing and expect a given (fixed) mapping between the components in the model and the instance lines.

In [177, 99], we've proposed to take a more dynamic view on LSCs.[3] That is, we view instance lines as logical variables and the whole chart as existential or universal quantification over components in the model.

Viewing the LSC from 10.5 as dynamic binding, it would read out loud as

> "for all combinations of crossing, lights, and barrier controller, whenever there is a request..."

which would in practice need to be supported by an activation condition (cf. [97]) that restricts it to all combinations of *associated* controllers in the system.

[3] The play-in/play-out correspondence is [121].

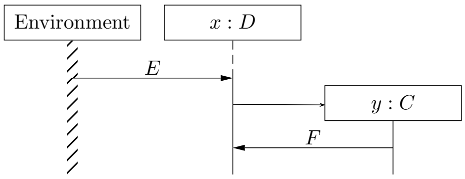


Figure 10.6.: **Creation** in a Live Sequence Chart.

In our proposal, we already have UML with (possibly premature) disappearance in mind. To this end, we read life lines as implicit local invariant requiring continuous aliveness throughout activation of the LSC, that is, $\neg \otimes x$ in terms of EvoCTL*.

Thereby, the EvoCTL* formulae corresponding to LSCs (see below) are definite. Premature disappearance is clearly a violation of the LSC, by intention, and not accidentally by being biased (cf. Chapter 4).

LSCs and EvoCTL*

In the joint work [168, 47], we've outlined how LSCs relate to temporal logic. Our findings are that each universal LSC (solid outline main chart) is equivalent to an LTL formula, and there is a syntactically characterisable strict fragment of LTL, which can be translated back into LSCs. Existential LSCs (dashed outline main chart) map into CTL.

In the joint work [98], we discuss sub-classes of LSCs which can be checked with differently powerful approaches. The classes range from the *bonded, time bounded* fragment, whose LSCs are equivalent to safety properties, to *non-bonded, general liveness* LSCs, which cannot be checked by the observer (or monitor (or test automaton)) approach of [182]. The discussion of Section 9.4.4 applies accordingly.

Treating Creation

With [177, 99], we also considered the notation for object creation from UML SDs in LSCs as shown in Figure 10.6 (cf. Chapter 10). It says that after object x (of class D) has received an event E from the environment, it (finally) creates a new object of class C, which shall from now on be known under the name y in this chart.

It is in particular *this* object of class C which is supposed to (finally) reply to x with an F event, not any other object of this class, and the semantics of the LSC from Figure 10.6 requires the F answer for all created C, shall there be more than one. This is something different from case-split, and different from treating links (cf. Chapter 4).

We can treat this case if we can assume the two predicates, which can be defined by auxiliary variables (or observers) as discussed in Chapter 9. Namely,

1. $new_C(\cdot)$, holding for an identity id iff id has just created a new object of class C, and

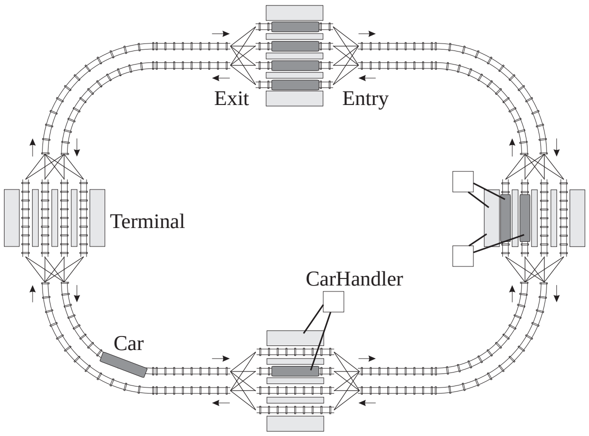


Figure 10.7.: **Automated Rail Cars System.** Closed variant with eight cars and four terminals (cf. Section 1.1.1).

2. $\mathit{thenew}_C(\cdot)$, yielding for an identity *id* the identity *id′* which denotes a new object of class C and has just been created by *id*.

Then the EvoCTL* formula[4]

$$\forall\, x : T_1, y : T_2 \,.\, \mathit{rcv}_E(x) \rightarrow \mathsf{X}\,\mathsf{F}(\mathit{new}_C(x) \wedge (\odot\, y \wedge y = \mathit{thenew}_C(x) \rightarrow \dots)) \qquad (10.6)$$

(with some omitted requirements on x staying alive) states that finally, x creates a new C, this then happens to either not be y, or to be y, and in this case it continues as required. Which coincides with the requirement stated by the LSC.

We conjecture that the auxiliary predicates *new* and *thenew* (or something equivalent) are necessary. It seems not to be sufficient to know the destination link, that is, the link name under which the D instance keeps the newly created C instance.

10.2.3. Automated Rail Cars System

Recall the Automated Rail Cars System [79] from the introduction in Chapter 1. In its original definition, it's a parameterised system with a fixed number $N \in \mathbb{N}^+$ of rail cars shuttling on opposite direction one-way tracks between a fixed number of M terminals, each with a fixed number of platforms. Cars autonomously plan their route and automatically keep a safety distance to the car in front.

Dynamic topologies come into play with the arrival and departure procedure. Cars announce themselves at the terminal and are assigned a newly created *car handler*, which reserves a platform and sets up the ingoing switches accordingly. Only afterwards, the car is notified that it may enter the terminal, if this notification doesn't arrive in time, the car waits. If the car wants to leave a terminal, its handler sets up the outgoing switches and clears the platform reservation afterwards, before it is destroyed.

[4] the relation between LSCs and temporal logic is briefly discussed in a paragraph above

model	inst.	model-gen.	model-checking	errorpath prep.
ARCSystem	DTR	0:02:18	4:45:50	(prop. holds)
	Witness	0:02:12	0:00:06	0:00:05

Table 10.1.: **Verification times.** Model-generation, -checking, and translation of the counter-example back to UML terms (hours:minutes:seconds).

So the link topology clearly changes over time, as does the extension because car handlers are created and destroyed. If the system is implemented without programming errors, then the number of cars induces a finite upper bound on the number of car handlers, thus the system is then finite-state.

But this premise, absence of programming errors cannot easily be made in the domain of critical system. This has to be properly established, which is admittedly possible in some cases.

More interestingly, the case study easily extends to an open system where rail cars dynamically enter and exit the scope of a terminal. Then there may still be a finite upper bound on the number of handlers, but it will depend in a non-trivial way on the safety distance between cars, possibly the strategy of platform management, etc..

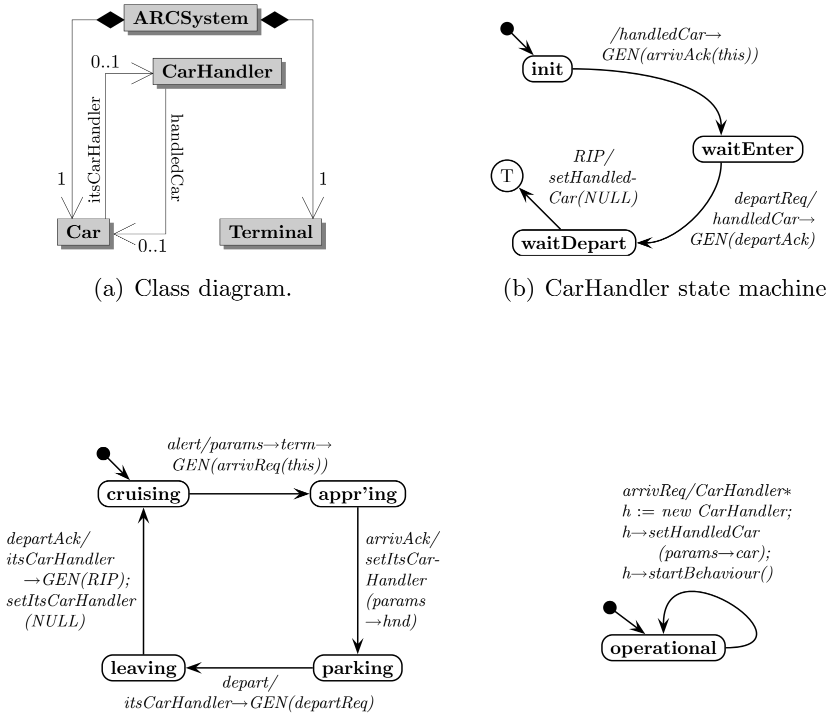
The benefit of the QR/DTR approach is that we *needn't* establish these premises but can analyse the system even if it is suspected to have an unbounded extension.

Model

In [178], we've presented the verification of a simplified version of the ARCS, which focuses on the arrival departure procedure at a single terminal. This is the interesting case as outlined above, because this is where topology dynamics are observed.

The relevant parts of the model, the class diagram and the state machines of car handlers, rail cars, and terminals, are shown in Figure 10.8. We've modelled it with the UML schematic entry tool Rhapsody and employed the UVE tool-chain [159, 158] with integrated QR/DTR transformations to obtain an array program encoding in the model-checker input language SMI [24, 179] (cf. [178]). It is basically implementing the less sophisticated encoding presented in Section 9.4.3, the precision problems (cf. Section 9.3.3) were not understood back then.

A notable aspect of the integration into UVE is that it was already prepared to exploit that the different representative cases yielded by QR are completely independent, thus can be verified completely independently on concurrent hosts. The implementation of the integration took care to interpret the result and to stop running tasks if the first task completed with a negative result.



(a) Class diagram.

(b) CarHandler state machine.

(c) Car state machine.

(d) Terminal state machine.

Figure 10.8.: **Automated Rail Cars System** class and state chart diagram.[178]

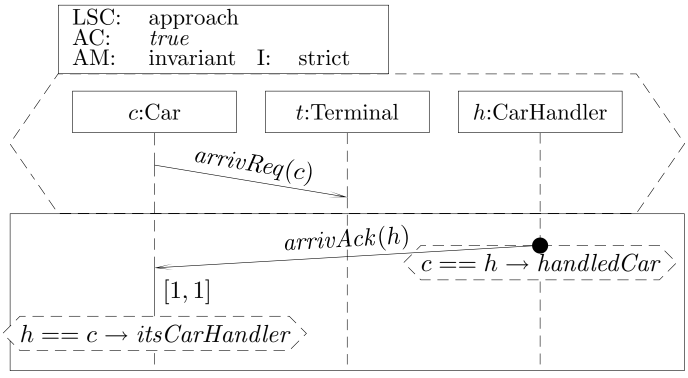


Figure 10.9.: **Live Sequence Chart.** If c contacts terminal t, then it is granted access by the h responsible for it and establishes the link *itsCarHandler*.

10.2.4. Requirements and Results

Table 10.1 shows verification times[5] for the LSC requirement given in Figure 10.9. The first line includes determining and executing the single representative verification task. The second line, for comparison, refers to the verification that there exists a run of the (not abstracted) ARCS which satisfies the LSC. The property already holds in the initial abstraction, refinement was not necessary.

The reason for the long verification times (nearly five hours) is not completely understood. We conjecture that the main source are the FIFO queues used for communication even when restricted to small lengths.

10.2.5. Discussion

Formal verification for UML already has a long history, as we've seen in Chapter 3 when discussion their computational model. The main difference to our approach is that, up to now, there is as far as we know no approach which supports true liveness properties (like the finally modality) in a truly dynamic topology setting, that is, with unbounded creation and destruction.

The earliest works, [111, 108, 65] basically considered single state machines in isolation, which is easy as formal verification for Harel's Statecharts [78] have been established then, e.g. [44, 25, 24].

Similarly, in [52, 51] the focus is on timed state-machines in UML, not in topologies in their Uppaal-based tool UML RT/Uppaal.

In [118, 117], multiple state-machines are considered in the tool vUML, which is based on the SPIN model-checker [83]. The topology is not dynamic in any sense, each state machine may communicate with all others. Object creation and destruction is also not considered in the ASM-based tool VeriUML [37, 38, 161, 162, 163].

In [157, 100, 101], verification of collaboration diagrams is considered which are largely equivalent to sequence diagrams. Thus they're principally closely related. The difference is that collaboration diagrams suffer from the same issues as sequence diagrams (see above), so they're strictly weaker than LSCs, and they exploit the capability of collaboration diagrams to also give a topology. Verification is then conducted in the fixed topology.

The most sophisticated approach next to UVE, the tool ObjectCheck [187, 185, 188, 186, 160, 184, 189] also employs Query Reduction, but didn't make the step to the DTR abstraction. Thus they rely on a completely concrete model with dynamic topology but with a fixed finite upper bound on the number of objects concurrently alive. The same holds for plain UVE without our extension [159, 158].

Graph-oriented approaches like UMLAUT/CADP [74, 73, 93], USE [153], or [8] fully support dynamic topologies, but address only safety properties.

[5] Sun Blade 2000, 900 MHz UltraSparc III+, 2 GByte.

Part V.

Conclusion

11. Conclusion and Further Work

11.1. Summary

In the pre-study [177], we set out to verify scenario properties given in form of Life Sequence Charts [42, 43] (LSC) against UML models in the sense of [79] by a particular abstraction following the spotlight principle. With this work, we can now confirm the expectations and have gained a good understanding of how and why the abstraction works, in particular in the context of Dynamic Topology Systems (DTS). This has been far from clear with the original proposals and provides valuable starting points for further investigations, in particular concerning refinement.

It took us a long way to achieve this. First of all, there is currently no agreed fully adequate formal computational model of DTS in the literature, so we propose evolving topology transition systems (ETTS) in Chapter 3. It serves, in comparison to existing proposals, the purpose to in particular represent nasty effects like dangling links and is prepared to cover also other approaches for formal comparison in a unified framework. The definition proves useful in the subsequent chapters.

Secondly, we widened the scope from scenarios to complete temporal logic. From the literature it is known that the scenario language of LSCs is equivalent to only a strict fragment of temporal logic, while the overall approach promises to apply to a wider range of properties. Employing any alternative representation of LSC properties than temporal logic, for instance Büchi automata, doesn't ease the proofs in the affected chapters.

To this end, we introduce EvoCTL* in Chapter 4 as a first-order variant of CTL* quantifying over identities or individual destinies. Again, the reason is that there is no agreed formalism as we lay out in Section 4.6. The proposals seem to converge in syntax, but have vastly different semantics, that are in our opinion in most cases not adequate to the DTS domain.

Our proposal is basically the union of the syntax of existing proposals, one exception is transitive closure which is present in some of the underlying proposals and which we omit because it would add an additional level of complexity. We intentionally refrained from adding new constructs, which are certainly imaginable, to keep the presentation focused. A first investigation of monotonicity and definiteness properties shows that our proposal is adequate.

Our discussion of prenex normal forms gains a better understanding of the fragment of EvoCTL* amenable to the overall approach. We understand that it depends on a quantification over identities (instead of destinies), that it clearly covers the fragment equivalent to LSCs, but interestingly also properties with normal form and the fragment of destiny quantification properties with equivalent identity form. We also see properties not amenable to the overall approach and can explain the reasons.

In Chapter 5, we establish a new simulation relation on ETTS and discuss how EvoCTL* properties on simulating systems are related. This is the basis for the (re-stated pre-existing) soundness proof of the Data-Type Reduction abstraction in Chapter 6.

The value of Chapter 6 lies in its new, closed definition of DTR, while existing approach are rather inductive, or implementation oriented. The intuitive statement of DTR in the graph-based setting of ETTS allows us to discuss reflection properties and give guidelines on possible refinements if the unique properties of DTR shall be preserved.

Chapters 7 and 8 basically re-state the methodology from [124, 127] in our context. A difference is that Chapter 7 introduces a notion of *symmetry in identities*, which is a special case of the existing approaches but of major concern with ETTS. Furthermore, it discusses how to treat singularities, which even furthers the range of applicability.

Having discussed the QR/DTR approach in full breadth theoretically in Part III, we address more practical issues in Part IV in full depth because a theoretical definition of an abstraction is close to useless if it is not clear how to obtain the abstract transition system in practice, or if it is prohibitively expensive. On the other hand, a theoretically non-understood implementation is far from satisfactory.

To this end, we introduce a higher-level description language (HLL) for DTS in Chapter 9. It is intended to cover the more concrete formalism DCS and UML as addressed in Chapter 10. With HLL, we discuss for the first time the impact of side-conditions for symmetry in identities. We find that it depends on the communication means, the treatment of inputs and creation, and the scheduling. In existing applications of QR/DTR, it is typically discussed only in the context of a particular model-checker, without reflecting in how far means provided by these tools are necessary.

Furthermore, we find that QR/DTR is best suited for interleaving semantics, full concurrency makes matters more complicated. Interestingly, the existing applications fall into this class but seldom reflect that they depend on it.

Finally, we demonstrate how to obtain an HLL model whose ETTS semantics simulates the theoretically described DTR'd ETTS and we can even discuss new issues of precision. With Section 9.4, we get back to array programs, the domain in which QR/DTR has originally been introduced. With a sufficiently detailed procedure to obtain, for a given general HLL model, a finite-state array program which simulates the DTR'd model we're ready to employ any readily available symbolic model-checker.

Chapter 10 closes the circle by reporting properties that have been established for the examples from the introduction with the QR/DTR approach employing an own implementation.

This breadth and width is in our opinion necessary because leaving out a discussion of any one of them would endanger the justifications of all other parts. As in particular the formal description languages are new proposals, their intuition has to be discussed in some detail.

This comes at the price that the discussion has to be somehow shallow and preliminary in some places. For example, we only have preliminary results on properties of EvoCTL* and the comparison of QR/DTR to other approaches is far from complete, for instance, given the amount of recently proposed approaches for the certainly related analysis of pointer programs.

On the other hand, while isolated and clearly focused investigations are typically to be preferred, sometimes somebody has to take a step back to see the whole picture, in particular in order to see whether all simplifying assumptions are still in scope. With this work, we have undertaken such a step.[1]

11.2. Discussion

The QR/DTR approach is suitable for proving scenario properties on Dynamic Topology Systems. The main benefits compared to other approaches are that it preserves liveness properties (under fairness assumptions), while the vast majority of other approaches only addresses reachability or safety properties.

Secondly, it doesn't suffer from identity blurring (cf. Section 6.6) which is inevitable when considering scenarios that refer to the behaviour of certain *particular* individuals over time.

And thirdly, obtaining the abstract transition system is computationally cheap, in contrast to, for instance, approaches based on predicate abstraction.

The analysis of Dynamic Topology Systems is a field which clearly gained momentum in last few years, for instance, think of the Partner Abstraction of [7, 12], approaches to model-checking for the π-calculus like [132], or Hoare-style verification of graph programs [76], not to speak of the above mentioned approaches to the analysis of pointer programs, and ongoing efforts to extend Craig interpolation to array programs.

For this reasons, we will see how QR/DTR proves over time, whether it remains. What we expect to remain is the characterisation of Dynamic Topology Systems (partly rooted in the joint work [9]))))) and the issues raised for a complementing property specification logic. Later approaches should be measured on the range of covered properties, in particular scenarios comprising liveness aspects.

11.3. Further Work

It is possibly a good sign, if one can conclude that we're left with more open than answered questions. This is clearly the case.

Concerning the modelling, the questions are whether ETTS and EvoCTL* are already completely adequate, and whether there is an interrelation as between CTL* and Kripke structures.

The EvoCTL* as such raises numerous questions in its own right, which we only could address briefly. Namely exact criteria for monotonicity and definiteness, or the exact relation between identity and destiny quantification, or whether we can tighten the findings

[1] And in our next undertakings, we will clearly aim to limit the scope substantially for a change.

on normal forms. Furthermore, the impact of different modeling means or restriction of links on the decidability of the model-checking problem should be investigated.

For DTR, the topic of refinement is most prominent, we were only able to scratch the surface and prepare the setting for further investigation. Interestingly, this work has already inspired independent work on this topic, for instance [166]. Such work in particular addresses the issue of automation. Currently, refinement takes a significant amount of expertise in both, the considered model and property, and the procedure as such.

For the syntactic transformations of Chapter 9, we have identified the impact of concurrency vs. interleaving which has to be further investigated, as well as adding precision by ensuring sharing of common expressions. Last but not least, more case-studies are in order to get a better feeling for the class of properties that can be established in practice, in addition to the ones reported in Chapter 10.

☙ *fin.* ❧

Bibliography

[1] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In Nancy A. Lynch and Bruce H. Krogh, editors, *HSCC*, volume 1790 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 2000. 3.7.7

[2] Christian Ammann. Verifikation von DCS beschreibungen mit dem modelchecker Spin, September 2007. Individuelles Projekt. 9.2.3, 9.2.3, 9.4.1, 9.4.1, 9.4.2, 9.5, 10.1.1

[3] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunications Systems Journal*, 24(1):61–94, September 2003. 10.2.2

[4] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, May 1986. 4.4.2

[5] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In Dill [55], pages 234–246. 4.4.2

[6] Ruth C. Barcan. A functional calculus of first order based on strict implication. *Journal of Symbolic Logic*, 11:1–16, 1946. 4.6

[7] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, September 2006. 3.2, 3.3, 3.6, 3.6, 3.7.4, 3.7.6, 4.5.3, 6.4.6, 6.4.6, 6.6.3, 10.1, 10.1.3, 11.2

[8] Jörg Bauer, Werner Damm, Tobe Toben, and Bernd Westphal. Verification and synthesis of OCL constraints via topology analysis: A case study. Reports of SFB/TR 14 AVACS 23, SFB/TR 14 AVACS, June 2007. ISSN: 1860-9821, http://www.avacs.org. 6.6.3, 10.2.5

[9] Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and verification of dynamic communication systems. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on.* IEEE Computer Society Press, 2006. 1.2, 1.4.1, 1.5.1, 4.6, 9, 9.2.5, 9.3.4, 9.5, 10, 10.1, 10.1.1, 10.1.3, 10.2, 10.3, 11.2

[10] Jörg Bauer, Tobe Toben, and Bernd Westphal. Mind the shapes: Refining datatype reduction via topology invariants, 2007. Submitted. 1.5.1, 6.4, 6.4.6, 6.6.3, 9.3.4, 10.1, 10.1.1, 10.1.3, 10.4, 10.1.3, 10.1.4

[11] Jörg Bauer, Tobe Toben, and Bernd Westphal. The temporal logic of appearance and disappearance. Reports of SFB/TR 14 AVACS 24, SFB/TR 14 AVACS, June 2007. ISSN: 1860-9821, http://www.avacs.org. 1.5, 1.5.1, 4.4, 4.6

[12] Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*. Springer, 2007. 3.2, 3.3, 3.6, 3.6, 3.7.4, 3.7.6, 4.5.3, 6.6.3, 10.1, 10.1.3, 11.2

[13] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, 22-24 October*, 1995. 7

[14] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Hu and Vardi [85], pages 369–386. 3

[15] Sergey Berezin, Edmund Clarke, Armin Biere, and Yunshan Zhu. Verification of out-of-order processor designs using model checking and a light-weight completion function. *Formal Methods in System Design*, 20(2):159–186, March 2002. 3

[16] F.S.d. Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors. *Formal Methods for Components and Objects First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, number 2852 in Lecture Notes in Computer Science. Springer-Verlag, 2003. 45, 48

[17] Jürgen Bohn, Werner Damm, Hartmut Wittke, Jochen Klose, and Adam Moik. Modelling and validating train system applications using Statemate and Live Sequence Charts. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT 2002)* [164]. 10.2.2

[18] Y. Bontemps, P. Heymans, and H. Kugler. Applying LSCs to the specification of an air traffic control system. In *International Workshop on Scenarios and State Machines (SCESM'03)*, 2003. 10.2.2

[19] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer*, 4(1):65–80, 2002. 9.2.2

[20] J. Bradfield, J. Küster Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In Kutsche and Weber [107], pages 203–217. 4.6

[21] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the Association for Computing Machinery*, 30(2):323–342, April 1983. 3.7.2, 10.1

[22] Matthias Brill, Ralf Buschermöhle, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Formal verification of LSCs in the development process. In Ehrig et al. [61], pages 494–516. 10.2.2

[23] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live sequence charts. In Ehrig et al. [61], pages 374–399. 10.2.2

[24] Udo Brockmeyer. *Verifikation von Statemate Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 1999. 9.2.2, 9.4.2, 10.2.1, 10.2.3, 10.2.5

[25] Udo Brockmeyer and Gunnar Wittich. Tamagotchis need not die. In Bernhard Steffen, editor, *TACAS*, number 1384 in Lecture Notes in Computer Science, pages 217–231. Springer-Verlag, 1998. 10.2.5

[26] Annette Bunker, Ganesh Gopalakrishnan, and Konrad Slind. Live Sequence Charts applied to hardware requirements specification and verification: A VCI bus interface model. *Software Tools for Technology Transfer*, 7(4):341–350, August 2004. 10.2.2

[27] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In Dill [55], pages 68–80. 3

[28] Muffy Calder and Alice Miller. Automatic verification of any number of concurrent, communicating processes. In Emmerich and Wile [64], pages 227–230. 1, 9, 9.2.1, 9.3.4, 9.5

[29] Muffy Calder and Alice Miller. Feature validation for any number of processes. Technical Report TR-2002-10, Glasgow University Department of Computer Science, May 2002. 9, 9.2.1, 9.3.4, 9.5

[30] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 390–409. Springer-Verlag, 2002. 4.6

[31] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterised verification of cache coherence protocols. In Alan J. Hu and Andrew K. Martin, editors, *FMCAD*, number 3312 in Lecture Notes in Computer Science, pages 382–398. Springer-Verlag, 2004. 1, 6.4.6, 9, 9.2.1, 9.3.4, 9.5

[32] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC*, pages 240–248. ACM, 1986. 3.7.1

[33] E. M. Clarke, O. Grumberg, and D. E. Long. *Model Checking*, volume 152 of *Nato ASI Series F*, pages 305–349. Springer-Verlag, 1996. 1, 3.7.1, 7.2

[34] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. 1.4.1, 2, 2.6, 4, 4.1, 5, 5.1, A.1, A.2, A.3

[35] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer-Verlag, 2006. 6.6.3

[36] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using Live Sequence Charts and the Play-Engine tool. In Doron Peled and Yih-Kuen Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. 10.2.2

[37] Kevin Compton, Yuri Gurevich, James K. Huggins, and Wuwei Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan EECS Department, 2000. 3.7.6, 10.2.5

[38] Kevin Compton, James Huggins, and Wuwei Shen. A semantic model for the state machine in the UML. In Gianna Reggio, Alexander Knapp, Bernhard Rumpe, Bran Selic, and Roel Wieringa, editors, *Dynamic Behaviour in UML Models: Semantic Questions, Workshop Proceedings, UML 2000 Workshop*, number Bericht 0006 in LMU Technical Reports, pages 25–31. Ludwig-Maximilians-Universität München, Institut für Informatik, oct 2000. 3.7.6, 10.2.5

[39] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1):34–56, October 2002. 4.6

[40] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. 2, 5.1

[41] Gy. Csertán, G. Huszerl, I. Majzik, Zs. Pap, A. Pataricza, and D. Varró. VIATRA - visual automated transformations for formal verification of UML models. In Emmerich and Wile [64]. 3.7.6

[42] Werner Damm and David Harel. Lsc's: Breathing life into message sequence charts. In Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), February 15-18, 1999, Florence, Italy*, volume 139 of *IFIP Conference Proceedings*. Kluwer, 1999. 1.2, 10.2, 10.2.2, 2, 11.1

[43] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001. 1.2, 4.6, 10.2, 10.2.2, 2, 11.1

[44] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of statemate designs. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, number 1536 in Lecture Notes in Computer Science. Springer-Verlag, 1997. 3.7.6, 10.2.5

[45] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Boer et al. [16], pages 71–98. 3.7.6, 9, 9.4.2, 9.5, 10.2

[46] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time uml semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 55(1–3):81–115, March 2005. 3.7.6, 9.4.2, 9.5, 10.2, 10.2.1

[47] Werner Damm, Tobe Toben, and Bernd Westphal. On the expressive power of live sequence charts. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, number 4444 in Lecture Notes in Computer Science, pages 225–246. Springer-Verlag, 2007. 4.6, 10.2.2

[48] Werner Damm and Bernd Westphal. Live and Let Die: LSC-based Verification of UML-Models. In Boer et al. [16], pages 99–135. 1.5.1, 4.6, 6.4.6, 6.6.3, 9.2.2, 9.4.2, 9.5

[49] Werner Damm and Bernd Westphal. Live and let die: LSC-based verification of UML-models. *Science of Computer Programming*, 55(1–3):117–159, March 2005. 1.5.1, 4.6, 6.4.6, 6.6.3, 9.2.2, 9.4.2, 9.5

[50] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002. 2.4

[51] Alexandre David and M. Oliver Möller. From HUppaal to Uppaal: A translation from hierarchical timed automata to flat timed automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001. 3.7.6, 10.2.5

[52] Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In Kutsche and Weber [107], pages 218–232. 3.7.6, 10.2.5

[53] Akash Deshpande, Aleks Göllü, and Luigi Semenzato. The SHIFT programming language and run-time system for dynamic networks of hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):584–587, 1998. 1.1.2, 3.7.7

[54] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, 2005. Electronic Edition 2005. 1

[55] David L. Dill, editor. *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, number 818 in Lecture Notes in Computer Science. Springer-Verlag, 1994. 5, 27

[56] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525. IEEE Computer Society, 1992. 9.3.4

[57] Dino Distefano. *On Model Checking the Dynamics of Object-based Software*. PhD thesis, University of Twente, 2003. 3.4, 4.4.4, 4.6

[58] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000. 4.6

[59] Alastair F. Donaldson. *Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking*. PhD thesis, University of Glasgow, June 2007. 9.2.2, 9.5, 13

[60] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In Palsberg [142], pages 115–134. 3.7.4, 3.7.4

[61] Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering*. Number 3147 in Lecture Notes in Computer Science. Springer-Verlag, 2004. 22, 23

[62] E. Allen Emerson and Prasad Sistla. Symmetry and model-checking. *Formal Methods in System Design*, 9(1–2):105–131, August 1996. 7, 2, 1, 9, 9.2

[63] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHARME99: 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer-Verlag, 1999. 9.5

[64] W. Emmerich and D. Wile, editors. *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*. IEEE Computer Society, 2002. 28, 41

[65] Rik Eshuis and Roel Wieringa. Requirements-level semantics for UML statecharts. In Scott F. Smith and Carolyn L. Talcott, editors, *FMOODS*, volume 177 of *IFIP Conference Proceedings*. Kluwer, 2000. 3.7.6, 9.4.2, 10.2.5

[66] Javier Esparza. An automata-theoretic approach to software verification. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, number 2710 in Lecture Notes in Computer Science, page 21. Springer-Verlag, 2003. 1

[67] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna - a static model checker. In *FMICS*, 2006. 3.7.4

[68] Melvin Fitting and Richard L. Mendelsohn. *First Order Modal Logic.* Kluwer, 1998. 4.6

[69] Stephan Flake and Wolfgang Müller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling*, 2(3):164–186, October 2003. 4.6

[70] Stephan Flake and Wolfgang Müller. Past- and future-oriented time-bounded temporal properties with OCL. In *SEFM*, pages 154–163. IEEE, IEEE Computer Society, 2004. 4.6

[71] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. 6.3, 6.6.3

[72] Orna Grumberg, editor. *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997. 110, 123

[73] Alain Le Guennec. Méthodes formelles avec UML. In *CFIP'2000 : Colloque Francophone sur l'Ingénierie des Protocoles.* Hermes, October 1999. 3.7.6, 10.2.5

[74] Alain Le Guennec. *Genie Logiciel et Methodes Formelles avec UML - Specification, Validation et Generation de tests.* PhD thesis, Université de Rennes 1, 2001. 3.7.6, 10.2.5

[75] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation methods*, pages 9–36. Oxford University Press, 1995. 6

[76] Annegret Habel and Karl-Heinz Pennemann. Satisfiability of high-level conditions. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 430–444. Springer, 2006. 11.2

[77] Chris Hankin, Flemming Nielson, and Hanne Riis Nielson. *Principles of Program Analysis.* Springer-Verlag, 1999. 2.4, 2.4.18, 2.4.19

[78] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 9.4.2, 10.2.5

[79] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997. 1.1.1, 1.5.1, 9.3.4, 10.2.3, 11.1

[80] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling (SoSyM)*, 2007. To

appear. (Early version in 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), 2006, pp. 13-20.). 10.2

[81] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003. 1.2, 2

[82] Rolf Hennicker, Heinrich Hussmann, and Michel Bidoit. On the precise meaning of OCL constraints. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, number 2263 in LNCS, pages 70–85. Springer-Verlag, 2002. 3.7.6

[83] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), may 1997. 3.7.6, 9.3.4, 10.2.5

[84] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. The Design of Platoon Maneuver Protocols for IVHS. Technical report, UCB-ITS-PRR-91-06, 1991. 1.1.2

[85] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, number 1427 in Lecture Notes in Computer Science. Springer-Verlag, 1998. 14, 124

[86] Heinrich Hussmann. Loose semantics for UML, OCL. In *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT 2002)* [164]. 3.7.6

[87] I-Logix Inc. *Rhapsody User Guide, Release 3.0.* Three Riverside Drive, Andover, Massachusetts 01810, 2001. Part No. 2215. 1.5.1, 10.2.1

[88] C. Norris Ip and David L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993. 1.4.1

[89] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, August 1996. 1.4.1, 6.5, 7, 7.1.1, 7.1.3, 7.4.1, 7.4.3, 2, 7.6, 9.2, 9.2.1, 9.2.1, 9.2.2, 9.2.3, 9.2.5, 9.2.5, 9.2.5

[90] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, Geneva, 1996. 10.2

[91] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, 1999. 10.2

[92] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC).* ITU-T, Geneva, 1999. 10.2

[93] Jean-Marc Jézéquel, Wai Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In R. Hall and E. Tyugu, editors, *ASE*. IEEE Computer Society, 1999. Also available as INRIA Technical Report RR3775. 3.7.6, 10.2.5

[94] Bernhard Josko. *Modular Specification and Verification of Reactive Systems*. Habilitationsschrift, Carl von Ossietzky Universität Oldenburg, 1993. 6.4.6, 6.4.6

[95] Henning Jost. Eliminating FIFO message queues from dynamic communication systems. Master's thesis, Carl von Ossietzky Universität Oldenburg, December 2006. 9.4.1, 10.1.1

[96] Yonit Kesten and Amir Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000. 6.4(a), 6.6.2, 6.6.2, 6.6.2, 3

[97] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003. 1.2, 2, 10.2.2, 2, 10.2.2

[98] Jochen Klose, Tobe Toben, Bernd Westphal, and Hartmut Wittke. Check it out: On the efficient formal verification of Live Sequence Charts. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, 2006. 6.4.6, 9.4.4, 10.2.2

[99] Jochen Klose and Bernd Westphal. Relating LSC specifications to UML models. In Hartmut Ehrig and Martin Grosse-Rhode, editors, *INT*, April 2002. 4.6, 10.2.2, 10.2.2

[100] Alexander Knapp and Stephan Merz. Model checking and code generation for UML state machines and collaborations. In Dominik Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, number 2002-11 in Technical Report, Institut für Informatik, Universität Augsburg, pages 59–64, 2002. 3.7.6, 10.2.5

[101] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, number 2469 in Lecture Notes in Computer Science. Springer-Verlag, 2002. 3.7.6, 10.2.5

[102] Christoph Knieke, Michaela Huhn, and Ursula Goltz. Modelling and simulation of an automotive system using LSCs. In Jan Jürjens Siv Hilde Houmb, editor, *Proceedings Workshop on Critical System Development Using Modeling Languages (CSDUML'2005)*, pages 73–87. TUM, September 2005. TUM-TR. 10.2.2

[103] Fabian Kratz, Oleg Sokolsky, George J. Pappas, and Insup Lee. R-charon, a modeling language for reconfigurable hybrid systems. In João P. Hespanha and

Ashish Tiwari, editors, *HSCC*, volume 3927 of *Lecture Notes in Computer Science*, pages 392–406. Springer-Verlag, 2006. 1.2, 3.7.7

[104] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963. 4.6

[105] O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, number 2154 in Lecture Notes in Computer Science, pages 519–535. Springer-Verlag, 2001. 3

[106] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1995. 3.7.6

[107] Ralf-Detlef Kutsche and Herbert Weber, editors. *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, number 2306 in Lecture Notes in Computer Science. Springer-Verlag, 2002. 20, 52, 185, 188

[108] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML*, number 1939 in Lecture Notes in Computer Science, pages 528–540. Springer-Verlag, 2000. 9.4.2, 10.2.5

[109] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, December 1997. 7

[110] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal: Status and developments. In Grumberg [72], pages 456–459. 7

[111] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999. 3.7.6, 9.4.2, 10.2.5

[112] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999.* Kluwer, 1999. 3.7.6

[113] Ranko Lazic and David Nowak. A unifying approach to data-independence. Programming Research Group Technical Report TR-4-00, Oxford University Computing Laboratory, June 2000. 9.2.1

[114] Marc Lettrari. *Efficient State Space Exploration of Reactive Object-Oriented Programs.* PhD thesis, Carl von Ossietzky Universität Oldenburg, November 2005. 8.2

[115] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analysis. In Palsberg [142], pages 280–301. 6.6.3

[116] David Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968. 3, 4.6

[117] Johan Lilius and Ivan Porres. Formalising UML state machines for model checking. In Robert B. France and Bernhard Rumpe, editors, *UML*, number 1723 in Lecture Notes in Computer Science, pages 430–445. Springer-Verlag, 1999. The TUCS Technical Report 273 (The Semantics of UML Statecharts) is an earlier version of this conference paper. 3.7.6, 9.4.2, 10.2.5

[118] Johan Lilius and Ivan Porres. vUML: a tool for verifying UML models. In *Proceedings of the Automatic Software Engineering Conference (ASE'99)*, pages 255–258. IEEE Computer Society, oct 1999. The TUCS Technical Report 272 is an extended version of this conference paper. 3.7.6, 9.4.2, 10.2.5

[119] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs i. *Acta Inf.*, 21:125–169, 1984. 6.3, 6.6.3

[120] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: specification.* Springer-Verlag, 1991. 9.4.2

[121] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA*, volume 37:11 of *SIGPLAN Notices*, pages 83–100. ACM, November 2002. 3

[122] Tiziana Margaria and Wang Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, number 2031 in Lecture Notes in Computer Science. Springer-Verlag, 2001. 145, 171

[123] K. L. McMillan. A compositional rule for hardware design refinement. In Grumberg [72], pages 24–35. 1, 1.4.1

[124] K. L. McMillan. Verification of an implementation of tomasulos algorithm by compositional model checking. In Hu and Vardi [85], pages 110–121. 1, 1.4, 1.4.1, 2, 1.4.1, 6.6.1, 6.6.2, 6.6.3, 7, 7.6, 8.2, 9.3.4, 11.1

[125] K. L. McMillan. *Getting Started with SMV.* Cadence Design Systems, March 2001. 6.4.6, 9.3.4

[126] K. L. McMillan. *The SMV Language.* Cadence Design Systems, March 2001. 9.3.4

[127] Ken L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37:279–309, 2000. (document), 1, 1.4, 1.4.1, 1.6, 1, 2, 1.4.1, 1.7(a), 1.4.1, 1.4.1, 1.4.1, 3.4, 3.7.3,

4.5.3, 4.5.3, 6, 6.4, 6.4.5, 6.4.5, 6.4.6, 6.4.6, 6.5, 6.6, 6.6.1, 6.6.2, 6.6.3, 7, 7.4.3, 7.6, 8.1, 8.1, 8.2, 9.2.2, 9.2.3, 9.3.4, 9.5, 11.1

[128] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 9, 9.4

[129] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1999. 1.4, 1.4.1, 2, 1.4.1, 1.4.1, 1.4.1, 1.4.2, 9.3.4

[130] Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2001. 9.3.4, 9.3.4

[131] Roland Meyer. Personal communication. 3.7.5

[132] Roland Meyer. Ein strukturtheoretischer Ansatz zur Verifikation des π-Kalküls (Abstract). In Martin Wenig, editor, *Dagstuhl "zehn plus eins". Zehn Informatik-Graduiertenkollegs und ein Informatik-Forschungskolleg stellen sich vor.* Verlagshaus Mainz GmbH Aachen, 2007. 11.2

[133] Roland Meyer. A theory of structural stationarity in the π-calculus, 2007. Accepted for Publication. 3.7.5

[134] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3):8, 2006. 7, 9.2.1, 9.2.2, 9.5

[135] Robin Milner. An algebraic definition of simulation between programs. In D. J. Cooper, editor, *IJCAI*, pages 481–489. William Kaufmann, September 1971. 5.1

[136] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. 3.7.5

[137] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. 3.7.5

[138] OMG. Unified modeling language specification. Technical Report 1.4-UML-01-09-67, OMG, September 2001. 1.1.1, 3.7.6, 4.6, 9, 10.2

[139] OMG. Object Constraint Language, version 2.0. Technical Report formal/06-05-01, OMG, 2006. 4.2.1, 4.6

[140] OMG. Unified Modeling Language: Infrastructure 2.1.1. Technical Report formal/07-02-06, OMG, February 2007. 1.1.1, 3.7.6, 4.6, 10.2

[141] OMG. Unified Modeling Language: Superstructure, version 2.1.1. Technical Report formal/07-02-05, OMG, February 2007. 1.1.1, 3.7.6, 4.6, 10.2

[142] Jens Palsberg, editor. *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, number 1824 in Lecture Notes in Computer Science. Springer-Verlag, 2000. 60, 115

[143] A. Pataricza, D. Varró, I. Majzik, Zs. Pap, and G. Huszerl. VIATRA - visual automated transformation based frameword for UML based dependability evaluation, 2002. Submitted (but not accepted?) to DSN 2002: Dependable Systems and Networks Washington, USA, 2002. 3.7.6

[144] Amir Pnueli. Compositional model checking is not what it appears to be, September 2000. Talk at IFIP 2.2 meeting in Oldenburg. 1.4.1, 6.6.2

[145] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In Margaria and Yi [122], pages 82–97. 9.3.4

[146] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with (0,1,infty)-counter abstraction. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 107–133. Springer-Verlag, 2003. 6.3, 6.6.3

[147] Andreas Podelski and Thomas Wies. Personal communication. 3.7.4

[148] Andreas Podelski and Thomas Wies. Boolean heaps. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2005. 3.7.4, 6.6.3

[149] Fong Pong and Michel Dubois. Formal verification of complex coherence protocols using symbolic state models. *J. ACM*, 45(4):557–587, 1998. 6.3, 6.6.3

[150] Jan Rakow. Verification of dynamic communication systems. Master's thesis, Carl von Ossietzky Universität Oldenburg, April 2006. 9.4.1, 9.4.2, 10.1.1, 10.1.3, 10.1.3

[151] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. A CASL formal definition of UML active classes and associated state machines. Technical Report DISI-TR-99-16, DISI - Università di Genova, Italy, 1999. Revised March 2000. 3.7.6

[152] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. Analyzing UML active classes and associated state machines - a lightweight formal approach. In T. S. E. Maibaum, editor, *FASE*, number 1783 in Lecture Notes in Computer Science. Springer-Verlag, 2000. 3.7.6

[153] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In *ProcUML2000*, pages 265–277, 2000. 3.7.6, 10.2.5

[154] R.K.Brayton, G.D. Machtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification

and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, number 1102 in Lecture Notes in Computer Science, pages 428–432. Springer-Verlag, 1996. 9, 9.4

[155] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. 3.7.4, 3.7.4, 5.1, 5.2, 5.7, 6.1.2, 6.6.3

[156] Davide Sangiorgi and David Walker. *The Pi Calculus*. Cambridge University Press, 2001. 3.7.5

[157] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001. 3.7.6, 10.2.5

[158] Ingo Schinz. *(UML Verification)*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2004. to appear. 4.5.3, 4.5.3, 7, 7.5, 7.6, 10.2.1, 10.2.3, 10.2.5

[159] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The rhapsody uml verification environment. In Jorge R. Cuellar and Zhiming Liu, editors, *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China*, pages 174–183. IEEE, September 2004. 1.5.1, 3.7.6, 7, 7.5, 7.6, 9.2.3, 9.4.2, 10.2.1, 10.2.3, 10.2.5

[160] Natasha Sharygina, James Browne, Fei Xie, Robert Kurshan, and Vladimir Levin. Lessons learned from model checking a NASA robot controller. *Formal Methods in System Design*, 25(2–3):241–270, September 2004. 3.7.6, 10.2.5

[161] Wuwei Shen, Kevin Compton, and James K. Huggins. A toolset for supporting UML static and dynamic model checking. In M. Feather and M. Goedicke, editors, *ASE*, pages 315–318. IEEE Computer Society, 2001. 3.7.6, 10.2.5

[162] Wuwei Shen, Kevin Compton, and James K. Huggins. A validation method for a UML model based on abstract state machines. In R. Moreno-Diaz and A. Quesada-Arencibia, editors, *Formal methods and Tools for Computer Science, Proceedings of EUROCAST 2001*, pages 220–223, 2001. 3.7.6, 10.2.5

[163] Wuwei Shen, Kevin Compton, and James K. Huggins. A toolset for supporting UML static and dynamic model checking. In *COMPSAC*, pages 147–152. IEEE Computer Society, 2002. 3.7.6, 10.2.5

[164] Society for Design and Process Science. *IDPT*, June 2002. 17, 86

[165] Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988. 4.4.2

[166] Tobe Toben. Non-interference properties for data-type reduction of communicating systems. In Jim Davies and Jeremy Gibbons, editors, *IFM*, volume 4591 of *Lecture*

Notes in Computer Science, pages 619–638. Springer, 2007. 6.4, 6.4.4, 6.4.6, 10.1, 10.1.3, 11.3

[167] Tobe Toben. *Communication Invariants (Tentative Title).* PhD thesis, Carl von Ossietzky Universität Oldenburg, 2008. to appear. 6, 6.4, 6.4.3, 6.4.4, 6.4.6, 6.4.6, 6.6.1, 8.1

[168] Tobe Toben and Bernd Westphal. On the expressive power of LSCs. In Jiří Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Július Štuller, editors, *SOFSEM 2006: Theory and Practice of Computer Science, 32nd Conference on Current Trends in Theory and Practice of Computer Science, Měřín, Czech Republic, January 2006*, volume 2, pages 33–43. Institute of Computer Science AS CR, Prague, January 2006. 4.6, 10.2.2

[169] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967. 1.4.1, 6.4.6, 9.3.4

[170] Moshe Y. Vardi. Sometimes and not-never re-revisited: On branching versus linear time. In *International Conference on Concurrency Theory*, pages 1–17, 1998. 4.6

[171] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Margaria and Yi [122], pages 1–22. 4.6

[172] Dániel Varró. A formal semantics of UML Statecharts by model transition systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, number 2505 in Lecture Notes in Computer Science. Springer-Verlag, 2002. Accepted paper. 3.7.6

[173] Björn Wachter. Checking universally quantified temporal properties with three-valued analysis. Master's thesis, Universität des Saarlandes, 2005. 6.6.3, 6.6.3, 6.6.3

[174] Björn Wachter and Bernd Westphal. The spotlight principle. On combining process-summarising state abstractions. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 182–198. Springer-Verlag, 2007. 1.5.1, 6.6.3, 6.6.3, 6.6.3

[175] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. Scenarios in system development: Current practice. *IEEE Software*, 15(2):34–45, March 1998. 10.2.2

[176] Bernd Westphal. Z-simulationstechnike als grundlage für csp-oz-simulation, June 1999. Minor Thesis ("Studienarbeit"). 10.2.1

[177] Bernd Westphal. Exploiting Object Symmetry in Verification of UML-Designs. Master's thesis, Carl von Ossietzky Universität Oldenburg, April 2001. 1, 1.5.1, 10, 10.2.2, 10.2.2, 11.1

[178] Bernd Westphal. LSC verification for uml models with unbounded creation and destruction. In Willem Visser Byron Cook, Scott Stoller, editor, *Proceedings of the Workshop on Software Model Checking (SoftMC 2005)*, volume 144:3 of *ENTCS*, pages 133–145. Elsevier B.V., July 2005. 1.5.1, 6.4.6, 9.2.1, 9.2.2, 9.2.3, 9.2.3, 9.3.3, 9.3.4, 9.5, 10.2, 10.2.3, 10.2.3, 10.8

[179] Bernd Westphal, Tom Bienmüller, Jürgen Bohn, and Rainer Lochmann. *The World's Ugliest Language a.k.a. SMI, Syntax and Semantics*, 2002. 9.2.2, 10.2.1, 10.2.3

[180] Thomas Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, September 2004. 6.6.3

[181] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. Verifying complex properties using symbolic shape analysis. distributed CD-ROM, 2007. 3.7.4, 6.6.3

[182] Hartmut Wittke. *An Environment for Compositional Specification Verification of Complex Embedded Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, November 2005. 6.4.6, 9.4.4, 10.2.2

[183] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983. 9.2.1

[184] Fei Xie. *Integration of Model Checking into Software Development Processes*. PhD thesis, The University of Texas at Austin, August 2004. UTCS Technique Report TR-04-29. 3.7.6, 9.4.2, 10.2.5

[185] Fei Xie and James C. Browne. Integrated state space reduction for model checking executable object-oriented software system designs. In Kutsche and Weber [107], pages 64–79. 1.4, 3.7.6, 7, 10.2.5

[186] Fei Xie, James C. Browne, and Robert P. Kurshan. Translation-based compositional reasoning for software systems. In *Proc. of 12th International Formal Method Europe (FME) Symposium*, 2003. 3.7.6, 10.2.5

[187] Fei Xie, Vladimir Levin, and James C. Browne. Model checking for an executable subset of UML. In M. Feather and M. Goedicke, editors, *Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering*. IEEE CS Press, nov 2001. 3.7.6, 9.4.2, 10.2.5

[188] Fei Xie, Vladimir Levin, and James C. Browne. Objectcheck: A model checking tool for executable object-oriented software system designs. In Kutsche and Weber [107], pages 331–335. 3.7.6, 10.2.5

[189] Fei Xie, Vladimir Levin, Robert P. Kurshan, and James C. Browne. Translating software designs for model checking. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, number 2984 in Lecture Notes in Computer Science, pages 324–338. Springer-Verlag, 2004. 3.7.6, 10.2.5

[190] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. *Logic Journal of the IGPL*, 14(5):755–783, 2006. 3.1.3, 3.2, 3.4, 3.6, 3.7.4, 3.7.4, 4.1.3, 4.3.1, 4.4, 3, 4.4.2, 4.6, 5.7, 6.3, 6.6.3, 10.1.2

[191] E. Yahav, T. Reps, S. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In Pierpaolo Degano, editor, *ESOP*, number 2618 in Lecture Notes in Computer Science, pages 204–222. Springer-Verlag, 2003. 3.1.3, 3.2, 3.4, 3.6, 3.7.4, 3.7.4, 4.1.3, 4.3.1, 3, 4.4.2, 4.6, 5.7, 6.3, 6.6.3, 10.1.2

[192] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001. 6.6.3, 6.6.3

[193] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 351–357. Springer-Verlag, 2003. 4.6

A. Proofs

A.1. Proofs of Chapter 4: Property Specification Logic

Proof of Lemma 4.3.6, page 91. Let $\mathcal{S} = (\mathcal{T}, \mathcal{V}, \mathcal{F}, \Lambda)$ be a signature, $G_1 \sqsubseteq G_2$ topologies over *Id* compatible with $\mathcal{S}$, and $\mathcal{M}$ a canonical structure wrt. G_1 and G_2.

1. In the statement of the lemma, we use the information order "$\sqsubseteq$" on $\mathbb{B}_3$ as introduced in Chapter 2. In the following, we prove the more general case not only for $\mathbb{B}_3$ but for any semantical domain $\mathcal{D}$ assuming the natural completion of a set with a top element $\top$ and the information order

$$\sqsubseteq = \{(d_1, d_2) \in \mathcal{D} \times \mathcal{D} \mid d_1 = d_2 \vee d_2 = \top\}. \tag{A.1}$$

We then use $\top$ of $\mathcal{D}(\tau)$ if a functional term of result type τ is undefined.

Let a be a boolean functional term and $\theta_1 \sqsubseteq \theta_2$ assignments of the free variables of a.

Induction base:

- a is of the form x:
  The semantics of these terms is independent from the employed topology, thus

$$\iota[\![x]\!](G_2, \theta_2) = \theta(v) = \iota[\![x]\!](G_1, \theta_1). \tag{A.2}$$

- a is of the form $\boldsymbol{x}$:
  If $\theta_2(\boldsymbol{x}) = \varepsilon$, then the evaluation is independent from the topology. If $\theta_2(\boldsymbol{x})$ is not the empty sequence in G_2 then by definition of information order on topologies and assignments, $\theta_1(\boldsymbol{x})$ is also alive in G_1, and the evaluation is identical in both topologies.

Induction step:

- a is of the form $\circledcirc\, a_1$:
  This case is explicitly excluded by premises.
- a is of the form $\sigma(a_1)$:
  By induction hypothesis, $\iota[\![a_1]\!](G_1, \theta_1) \sqsubseteq \iota[\![a_1]\!](G_2, \theta_2)$, that is, either both yield the same identity *id* or the valuation of a_1 in G_2 yields $\top$, i.e. is not defined, and in G_1 any identity or also $\top$.
  In the former case, the identity *id* is either an individual in both topologies, than the local state is identical because $G_1 \sqsubseteq G_2$. If *id* is not an

individual in G_2, or in the latter case where the whole term yields $\top$, we have

$$\iota[\![\sigma(a_1)]\!](G_2, \theta_2) = 1/2 \sqsupseteq \iota[\![\sigma(a_1)]\!](G_1, \theta_1) \tag{A.3}$$

independent from the situation in G_1 by definition of the local state access operator.

- a is of the form $\lambda(a_1)$:
  Similar to the case above using that information order of topologies requires that links in G_2 are also present in G_1 and navigation functions are identical.
- a is of the form $f(a_1, \ldots, a_k)$:
  If one of the parameters evaluates to $\top$ in G_2, then the valuation of the whole term is immediately undefined or $1/2$, depending on the return type, thus $\top$ in the extended definition.
  If none of the parameter terms evaluates to $\top$ in G_2, we obtain the same valuations in G_1 by induction hypothesis. If the interpretation of f is not defined at that point, the valuation in both topologies is $\top$, otherwise it is the same value, thus

$$\iota[\![f(a_1, \ldots, a_k)]\!](G, \theta_1) \sqsubseteq \iota[\![f(a_1, \ldots, a_k)]\!](G, \theta_2). \tag{A.4}$$

- a is of the form $a_1 = a_2$:
  Similar to the previous case because the valuation of comparison for equality also turns indefinite as soon as one side doesn't have a valuation.

2. Let t be a logical term over $\mathcal{S}$ and let θ_1 and θ_2 be two assignments of (a subset of) the free variables of t.

   Induction base:

   - t is of the form 0, 1, or a:
     In the former two cases, monotonicity is obvious as the interpretation is independent from the valuation. In the latter case, monotonicity follows from the first part of this lemma.

   Induction step:

   - t is of the form $\neg(t_1)$:
     If t_1 evaluates to $1/2$ under θ_2, then the whole term by definition evaluates to $1/2$, thus

$$\iota[\![\neg(a_1)]\!](G, \theta_2) = 1/2 \sqsupseteq \iota[\![\neg(a_1)]\!](G, \theta_1). \tag{A.5}$$

     independent from θ_1. If t_1 evaluates to a definite value under θ_2, then the same value is obtained for θ_1 by induction hypothesis.
   - t is of the form $(t_1 \vee t_2)$:
     It is sufficient to consider three cases. If t_1 or t_2 evaluate to 1 under θ_2, then they also evaluates to 1 under θ_1 correspondingly. The whole term

becomes 1 under both assignments. If t_1 evaluates to $1/2$ and t_2 evaluates to 0 or $1/2$ under θ_2, then

$$\iota[\![\neg(a_1)]\!](G, \theta_2) = 1/2 \sqsupseteq \iota[\![\neg(a_1)]\!](G, \theta_1) \tag{A.6}$$

independent from the outcome under θ_1. If both sub-terms evaluate to 0 under θ_2, then they also do so under θ_1 and the overall outcome for t is identical.

- t is of the form $(t_1 \wedge t_2)$:
  If one of the sub-terms evaluates to $1/2$ under θ_2, then the whole term evaluates to $1/2$. If both evaluate definite, then they obtain the same value under θ_1 and the outcome is identical.
- t is of the form $\forall \boldsymbol{x} : T \,.\, t_1$ or $\exists \boldsymbol{x} : T \,.\, t_1$:
  This case is explicitly excluded by premises.
- t is of the form $\forall x : T \,.\, t_1$ or $\exists x : T \,.\, t_1$:
  By induction hypothesis, we have

$$\iota[\![a_1]\!](G, \theta_1) \sqsubseteq \iota[\![a_1]\!](G, \theta_2) \tag{A.7}$$

  for any assignments $\theta_1 \sqsubseteq \theta_2$, thus in particular for each $\theta_1[x \mapsto id] \sqsubseteq \theta_2[x \mapsto id]$, $id \in Id$, because modifying variables of an identity type doesn't affect the information order relation between the two assignments. That is, each evaluation changes from 0 or 1 on the left-hand side to $1/2$ on the right hand side or remains unchanged. The resulting maximum (or minimum) then changes to $1/2$ or remains unchanged.

3. Follows directly from the first two parts of the Lemma.

□

Proof of Lemma 4.3.8, page 93. Let $\mathcal{S}$ be a signature, G a compatible topology over identities *Id*, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. G.

Note that in the following we show definiteness in an extended sense for a functional or logical term t. For logical terms, definiteness means values different from $1/2$ and for functional terms it means being defined at all, and, when evaluating to an identity type T or to L, that all employed identities denote individuals, that is, are alive. Furthermore, we use an assignment θ which assigns each free logical variable of type $\boldsymbol{T}$ in a term a non-empty evolution chain of identities denoting alive individuals. The claim then follows directly.

Induction base:

- t is of the form 0 or 1:
  Trivially definite because the evaluation is independent from topology and structure.

- t is of the form x:

  This case is explicitly excluded by premises.
- t is of the form $\boldsymbol{x}$:

  By premises, $\boldsymbol{x}$ appears bound, that is, in a sub-term of the form $\forall \boldsymbol{x} : T \,.\, t_1$ (or $\exists \boldsymbol{x} : T \,.\, t_1$). Thus if this term is evaluated as such, it is ensured that it is evaluated under an assignment θ which doesn't assign the empty sequence, that is, $\theta(\boldsymbol{x}) \neq \varepsilon$. Thus $\boldsymbol{x}$ evaluates definite to an alive individual.

Induction step:

- t is of the form $\circledcirc\, t_1$ or $\sigma(t_1)$:

  By induction hypothesis, t_1 evaluates definite, i.e. has a valuation. The whole term then obtains a definite value by definition.
- t is of the form $\lambda(t_1)$:

  This case is explicitly excluded.
- t is of the form $f(t_1, \ldots, t_k)$ or $p(t_1, \ldots, t_k)$:

  By induction hypothesis, t_i, $1 \leq i \leq n$, evaluate definite. By premise, function and predicate symbols appearing in t are total, hence t has a valuation under $\mathcal{M}$, i.e. evaluates definite in the extended sense.
- t is of the form $t_1 = t_2$:

  Similar to the previous case.
- t is of the form $\neg t_1$, $t_1 \vee t_2$, $t_1 \wedge t_2$:

  By induction hypotheses, t_1 and t_2 evaluate definite, thus by definition the valuation of t is also definite.
- t is of the form $\forall x : T \,.\, t_1$ or $\exists x : T \,.\, t_1$:

  These cases are implicitly excluded by the premises, which require that all variables occurring in term t are of type $\boldsymbol{T}$.
- t is of the form $\forall \boldsymbol{x} : T \,.\, t_1$ or $\exists \boldsymbol{x} : T \,.\, t_1$:

  By induction hypothesis, we have

  $$\iota[\![t_1]\!](G, \theta) \neq 1/2 \tag{A.8}$$

  for all assignments θ assigning non-empty evolution chains to free variables of type $\boldsymbol{T}$ in t. To evaluate the overall term, the definition modifies θ at $\boldsymbol{x}$ to non-empty evolution chains if there are any. Thus all values over which the minimum or maximum is computed, are definite and hence also the result.

□

Proof of Lemma 4.4.17, page 107. (by induction over the structure of φ)

Let $\mathcal{S}$ be a signature, $M_1 \sqsubseteq M_2$ two compatible evolving topology transition systems, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M_1 and M_2. By Cor. 4.4.16, we can without loss of generality assume that φ is a normal EvoCTL* formula.

Let $s \in S(M_1) = S(M_2)$ be a state of M_1 and M_2 and π a path in M_1 and M_2, and let $\theta_1 \sqsubseteq \theta_2$ be assignments.

Induction base:

- φ is of the form t:
  Lemma 4.3.6.
- φ is of the form $\odot\, a$ or $\otimes\, a$:
  These cases are explicitly excluded by premises.

Induction step:

- φ is of the form $\neg\phi_1$ or $\phi_1 \wedge \phi_2$, or $\neg\psi_1$ or $\psi_1 \wedge \psi_2$:
  By induction hypothesis, similar to Lemma 4.3.6.
- φ is of the form $\forall\, x : T \,.\, \phi_1$
  By induction hypothesis, we have

  $$\mathcal{M}_{M_1}[\![\phi_1]\!](s, \theta_1) \sqsubseteq \mathcal{M}_{M_2}[\![\phi_1]\!](s, \theta_2) \tag{A.9}$$

  for any assignments $\theta_1 \sqsubseteq \theta_2$, thus in particular for each $\theta_1[x \mapsto id] \sqsubseteq \theta_2[x \mapsto id]$, $id \in Id$, because modifying variables of an identity type T doesn't affect the relation between the two assignments. That is, each evaluation changes from 0 or 1 on the left-hand side to $1/2$ on the right hand side or remains unchanged. The resulting maximum then changes to $1/2$ or remains unchanged.
- φ is of the form $\forall\, x : T \,.\, \psi_1$
  Similar to the previous case.
- φ is of the form $\forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \varphi_1$
  This case is explicitly excluded by premises since variables of type $\boldsymbol{T}$ are assumed to appear free in φ.
- φ is of the form $\mathsf{A}\,\psi$:
  By induction hypothesis, the evaluation is monotone for each path and each assignment, thus in particular for the paths starting at s and the modifications of θ_1 and θ_2 considering all evolution chains. The relation between the two assignments is preserved because the resulting evolution chains depend on the evolution relation e, which is smaller for M_2, thus yields shorter evolution chains. The claim follows similar to quantification over identities.
- φ is of the form ϕ:
  In order to apply the induction hypothesis, we aim for a sub-formula of ϕ which is syntactically shorter, i.e. has fewer nodes in the parse tree. Following [34], we can view the right hand side as $\mathtt{path}(\phi)$, that is, as comprising the otherwise invisible operator '$\mathtt{path}$', which casts path formulae into state formulae. Then the claim follows directly by induction hypothesis.

- φ is of the form $\mathsf{X}\,\psi_1$:

  Distinguish whether evolution chains shorter than 2 are employed by the assignments for $\boldsymbol{T}$-variables in *Free*ψ_1. As $\theta_1 \sqsubseteq \theta_2$, there are only three cases. The evolution chains employed by θ_1 for these variables aren't strictly shorter than the ones in θ_2.

  If both assignments employ at least one shorter evolution chain, then φ evaluates to $1/2$ in M_1 and M_2. If only θ_2 employs one, than φ evaluates to $1/2$ in M_2, thus

  $$\mathcal{M}_{M_1}[\![\varphi]\!](\pi, \theta_2) = 1/2 \sqsupseteq \mathcal{M}_{M_2}[\![\varphi]\!](\pi, \theta_1). \tag{A.10}$$

  If none employs too short evolution chains, consider that by induction hypothesis, we have

  $$\mathcal{M}_{M_1}[\![\psi_1]\!](\pi, \theta_1) \sqsubseteq \mathcal{M}_{M_2}[\![\psi_1]\!](\pi, \theta_2). \tag{A.11}$$

  These values directly provide the evaluation of $\mathsf{X}\,\psi_1$ by definition.

- φ is of the form $\psi_1 \,\mathsf{U}\, \psi_2$:

  Distinguish the possible evaluations. If φ evaluates to 0 or 1 in M_1, then there is a (shortest) positive or negative witness along π. By induction hypothesis, ψ_1 and ψ_2 either have the same values or $1/2$ on this section of π. In the latter case, φ evaluates to $1/2$ in M_2 and otherwise to the same value as in M_1.

  If φ evaluates to $1/2$ in M_1, then by definition either ψ_1 or ψ_2 evaluate to $1/2$ prematurely. By induction hypothesis, they (at least) then also evaluate to $1/2$ in M_2, thus φ also evaluates to $1/2$ in M_2.

□

Proof of Note 4.4.18, page 107.

1. Consider the ETTS M_1 and M_2 shown in Figure 4.6 on page 108. We have $M_1 \sqsubseteq M_2$ because both are identical up to evolution relation. Assuming θ assigns *id* to v, as sequence of length one in case v is of an evolution chain type $\boldsymbol{T}$ and plainly otherwise, we obtain

   $$\mathcal{M}_{M_1}[\![\odot\, v]\!](s_2, \theta) = 1 \not\sqsubseteq 0 = \mathcal{M}_{M_2}[\![\odot\, v]\!](s_2, \theta). \tag{A.12}$$

2. In the same ETTS as in the previous case, we obtain

   $$\mathcal{M}_{M_1}[\![\otimes\, v]\!](s_1, \theta) = 1 \not\sqsubseteq 0 = \mathcal{M}_{M_2}[\![\otimes\, v]\!](s_1, \theta). \tag{A.13}$$

3. Note 4.3.7.

4. Note 4.3.7.

5. To discharge this case, it is sufficient to consider the formula $\mathsf{F} \circledcirc \boldsymbol{x}$, i.e. we set $\psi_1 := 1$ and $\psi_2 := \circledcirc \boldsymbol{x}$. With $\theta_1 := \{\boldsymbol{x} \mapsto \varepsilon\}$ and $\theta_2 := \{\boldsymbol{x} \mapsto id\}$ we have $\theta_1 \sqsubseteq \theta_2$, but
$$\mathcal{M}_{M_1}[\![\mathsf{F} \circledcirc \boldsymbol{x}]\!](s, \theta_1) = 0 \not\sqsubseteq 1 = \mathcal{M}_{M_2}[\![\mathsf{F} \circledcirc \boldsymbol{x}]\!](s, \theta_2). \tag{A.14}$$
because in the former case, there is no positive witness for $\boldsymbol{x}$ being alive. In the second case there is.

□

Proof of Lemma 4.4.19, page 108. Let $\mathcal{S}$ be a signature, M a compatible ETTS, and $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M.

Let φ be an EvoCTL* state formula over $\mathcal{S}$ satisfying prerequisites (1)–(4) and let $s \in S(M)$ be a state of M and π a path in M. By Cor. 4.4.16, we can without loss of generality assume that φ is a normal EvoCTL* formula.

Similar to the proof of Lemma 4.3.8, we assume (without loss of generality) that the assignment θ maps free variables of an evolution chain type $\boldsymbol{T}$ to non-empty evolution chains of alive individuals.

Induction base:

- φ is of the form t:
  Lemma 4.3.8.
- φ is of the form $\odot\, a$ or $\otimes\, a$:
  By Lemma 4.3.8, the functional term a is defined, that is, evaluates to an alive (cf. Lemma 4.3.8) identity. Then the formula yields a definite valuation by definition.

Induction step:

- φ is of the form $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\neg\psi_1$, or $\psi_1 \wedge \psi_2$:
  By induction hypothesis.
- φ is of the form $\forall\, x : T \,.\, \varphi_1$:
  This case is explicitly excluded by the premises.
- φ is of the form $\forall\, \boldsymbol{x} : T \,.\, \varphi_1$:
  By definition, the considered assignment θ is modified at $\boldsymbol{x}$ to an evolution chain of alive individuals (if φ is a state formula, then the chain has length 1). Thus each of the considered cases evaluate definite, and so the whole formula.
- φ is of the form $\mathsf{A}\, \psi$:
  By definition, the considered assignment θ is possibly modified at variables of an evolution chain type $\boldsymbol{T}$, branching the existing evolution chains. As these modifications preserve the property of θ to only employ chains of alive individuals, the claim follows by induction hypothesis.
- φ is of the form ϕ:
  Similar to the proof of Lemma 4.4.17.

- φ is of the form $\mathsf{X}\,\psi$:

  This case is explicitly excluded by the premises.

- φ is of the form $\psi_1 \,\mathsf{U}\, \psi_2$:

  By definition, the minimum length of evolution chains assigned by θ to the variables from $\mathit{Free}\psi_2$, which are by premise all of an evolution chain type $\boldsymbol{T}$, determines the prefix of π where ψ_1 and ψ_2 are evaluated.

  By induction hypothesis, both sub-formulae ψ_1 and ψ_2 evaluate definite, where the subset-requirement from the premises ensures applicability of the induction hypothesis. The whole formula then evaluates definite by definition.

□

Proof of Note 4.4.20, page 109.

1. By assigning the empty evolution chain to $\boldsymbol{x}$ the evaluation of φ immediately turns indefinite in the former cases. In the latter case, an evolution chain of length 1 has the same effect, independent from the sub-formula.

2. For example, choose $\otimes\, x$ for ψ_1 and ψ_2. Then choose the ETTS M such that the identity $id \in Id$ doesn't denote an alive individual in the initial state $s \in S_0(M)$ and all of its successors (for example by having s as the only successor of s).

   Then setting $\theta(x) := id$ has the desired effect independent from the canonical structure.

3. For example, choose $\odot\, x$ for ψ_2 and the ETTS from the previous case. Then the whole formula evaluates indefinite independent from the form of ψ_1 because the ψ_2 turns indefinite before a positive or negative witness could've been obtained.

□

Proof of Lemma 4.5.1, page 112. Let $\mathcal{S}$ be a signature, M a compatible evolving topology transition system, $\mathcal{M} = (\iota, \mathcal{D})$ a canonical structure of $\mathcal{S}$ wrt. M, and θ an assignment.

1. Cases 1 and 2 follow by computation rules on minima and maxima and Def. 4.4.6.

3. By case 2.

4. Consider the three possible evaluations, firstly

$$\mathcal{M}[\![\mathsf{X}\,\forall\, x : T \,.\, \varphi_1]\!](\pi, \theta) = 1. \tag{A.15}$$

This is the case iff

$$\forall\, id \in Id : \mathcal{M}[\![\varphi_1]\!](\pi/1, \theta/1[x/id]) = 1 \tag{A.16}$$

which is equivalent to

$$\forall\, id \in Id : \mathcal{M}[\![\varphi_1]\!](\pi/1, \theta[x/id]/1) = 1 \tag{A.17}$$

because the quantification is over an identity, not an evolution chain. The latter holds iff $\forall\, x : T \,.\, \mathsf{X}\, \varphi_1$ evaluates to 1. The case 0 is similar, the remaining cases for both sides are $1/2$.

5. Similar to the previous case. The difference is, that not the single next step is considered but *all* steps $k \in \mathbb{N}_0$, that is,

   $$\forall\, k \in \mathbb{N}_0\ \forall\, id \in Id : \mathcal{M}[\![\varphi_1]\!](\pi/k, \theta/k[x/id]) = 1 \tag{A.18}$$

   and the universal quantification in (A.18) is commutative.

6. Also by distinction of cases. If the left hand side holds, then there is a smallest step $k \in \mathbb{N}_0$ where φ_2 finally holds. In addition, φ_1 holds for all $id \in Id$ for steps $j < k$. This implies that, for $id \in Id$ considered in isolation, φ_1 holds until φ_2, which is the right hand side.

   The other direction follows because the right hand side states that there is one $k \in \mathbb{N}_0$ for each individual. These k have a minimum in $\mathbb{N}_0$, for which the left hand side holds.

7. Similar to the previous case.

8. If the left hand side holds, then given a path π, there is a step k such that all identities satisfy φ_1 globally from k on. This k serves as a witness for the right hand side, although there may be cases there φ_1 is already satisfied at steps smaller than k.

   In the other direction, there is one such k for each of the (by premise) finitely many identities in *Id*. The maximum of these k satisfies the right hand side.

   The negative and indefinite cases are similar.

□

Proof of Lemma 4.5.5, page 116. We only consider case 1. as the remaining cases follow by definition of (restricted) equivalence. Then we have to show that both sides evaluate to the same value for a given state and assignment under the assumption that a is valid in s under θ.

So assume φ_1 is a state formula and let $s \in S$ be a state and θ an assignment. By premises, a is valid in s under θ, thus it has a certain valuation in this state,

$$\iota[\![a]\!](s, \theta) =: id_a \in Id. \tag{A.19}$$

Then

$$\begin{aligned}
&\mathcal{M}[\![\forall x : T \,.\, a = x \rightarrow \varphi_1]\!](s,\theta)\\
&= \min\{\mathcal{M}[\![a = x \rightarrow \varphi_1]\!](s,\theta[x \mapsto \mathit{id}]) \mid \mathit{id} \in \mathit{Id}\}\\
&= \min(\mathcal{M}[\![a = x \rightarrow \varphi_1]\!](s,\theta[x \mapsto \mathit{id}_a]),\\
&\qquad \underbrace{\min\{\mathcal{M}[\![a = x \rightarrow \varphi_1]\!](s,\theta[x \mapsto \mathit{id}]) \mid \mathit{id} \in \mathit{Id} \setminus \{\mathit{id}_a\}}_{=1})\\
&= \mathcal{M}[\![a = x \rightarrow \varphi_1]\!](s,\theta[x \mapsto \mathit{id}_a]),\\
&= \max(\underbrace{1 - \mathcal{M}[\![a = x]\!](s,\theta[x \mapsto \mathit{id}_a])}_{=0}, \mathcal{M}[\![\varphi_1]\!](s,\theta[x \mapsto \mathit{id}_a]))\\
&= \mathcal{M}[\![\varphi_1]\!](s,\theta[x \mapsto \mathit{id}_a]) = \mathcal{M}[\![\varphi_1]\!](s,\theta)
\end{aligned} \tag{A.20}$$

The last equation holds because $x \notin \mathit{Free}(\varphi_1)$ by premises. □

A.2. Proofs of Chapter 5: Abstraction

Note that the inductive invariant of the following proof is apparently complicated because it has to cater for two things:

1. from the side of the assignment, we may encounter id_0, which has certain properties (being unlabelled)
2. from the side of the formula, we may hit $\top_{\mathcal{D}^\sharp(T)}$, which possibly propagates through

Proof of Lemma 5.3.5, page 135. By induction over the structure of general terms, not only logical terms. That is, we're actually going to prove an inductive property which is stronger than (5.32) by also considering functional terms. Namely, we'll show that for each term we have

$$\alpha(\iota[\![t]\!](G,\theta)) \sqsubseteq \iota^\sharp[\![t]\!](G^\sharp,\theta^\sharp), \tag{A.21}$$

where we consider an undefined valuation of the right hand side to be in $\sqsubseteq$ relation to any right hand side if it is of type S or an additional type, or, in case t is of an identity type T, the additional possibility

$$\begin{aligned}
&\iota[\![t]\!](G,\theta) \notin U(G)\\
&\quad \wedge \iota^\sharp[\![t]\!](G^\sharp,\theta^\sharp) \in U(G) \setminus U^{\circledcirc}(G^\sharp)\\
&\quad \wedge \iota^\sharp[\![\sigma(t)]\!](G^\sharp,\theta^\sharp) = \top \wedge \iota^\sharp[\![\lambda(t)]\!](G^\sharp,\theta^\sharp) = \top.
\end{aligned} \tag{A.22}$$

The original claim is implied by (A.21). Extending the meaning of "$\sqsubseteq$" in (A.21) doesn't break the reasoning because, by definition, functions propagate undefined values up to the level of boolean terms and boolean terms turn indefinite if any subterm is undefined.

Induction base:

- t is of the form x or $\boldsymbol{x}$: Def. 5.3.3.

- t is of the form 1: Trivial.

Induction step:

- t is of the form $\circledcirc\, a_1$:

  By induction hypothesis, either (A.21) or (A.22). In the former case, either $f(\iota[\![a_1]\!](G,\theta)) = \iota^\sharp[\![a_1]\!](G^\sharp,\theta^\sharp)$, then the claim follows because the embedding of individuals respects aliveness according to Def. 5.2.1, or $\top = \iota^\sharp[\![a_1]\!](G^\sharp,\theta^\sharp)$, then the claim follows by Def. 4.4.6.

  In the latter case, (A.22) entails that both $\iota[\![a_1]\!](G,\theta)$ and $\iota^\sharp[\![a_1]\!](G^\sharp,\theta^\sharp)$ exist but both are not in $U^{\circledcirc}(G)$, thus t evaluates to 0 in both structures.

- t is of the form $\sigma(a_1)$:

  By induction hypothesis, either (A.21) or (A.22). In the former case, either

  $$f(\underbrace{\iota[\![a_1]\!](G,\theta)}_{=:id_1}) \sqsubseteq \iota^\sharp[\![a_1]\!](G^\sharp,\theta^\sharp) =: id_1^\sharp, \tag{A.23}$$

  then the embedding of individuals assures that the local state of $id_1^\sharp$ is a safe over-approximation of the local state of id_1 by Def. 5.2.1, or $id_1^\sharp = \top$, then (A.21) follows because $\top$ is not in $Id^\sharp$ and thus $\sigma^\sharp$ and $\rightarrowtail_\lambda^\sharp$ are not defined for $\top$.

  In the latter case we have $\iota[\![a_1]\!](G,\theta) \notin U^{\circledcirc}(G)$, and (A.22), which entails the claim.

- t is of the form $\lambda(a_1)$:

  Similar to the previous case set $id_1 := \iota[\![a_1]\!](G,\theta)$ and $id_1^\sharp =: \iota^\sharp[\![a_1]\!](G^\sharp,\theta^\sharp)$. The induction hypothesis provides either (A.21) or (A.22).

  In the former case, either

  $$f(id_1) \sqsubseteq id_1^\sharp, \tag{A.24}$$

  then the embedding of links assures that navigating link λ from $id_1^\sharp$ in $G^\sharp$ is a safe over-approximation of the result of navigating it from id_1 in G, that is we obtain (A.21). Otherwise we have $id_1^\sharp = \top$, then navigation of links is not defined, that is, we also obtain (A.21) with the extended definition of "$\sqsubseteq$".

  In the latter case, (A.21) directly follows.

- t is of the form $a_1 = a_2$:

  In case both functional terms evaluate to identities in $U(G)$ and $U(G^\sharp)$, the claim is a consequence of preservation of equality by the embedding, cf. Def. 5.2.1.2b.

  Otherwise if they evaluate to identities outside of $U(G)$ in G, then they do so in $G^\sharp$ or become $\top \in \mathcal{D}(T)$ by the definition of corresponding assignments, cf. Def. 5.3.3. Then the whole term turns indefinite in $\mathcal{M}^\sharp$, which entails the claim.

- t is of the form $f(a_1, \ldots, a_k)$:
  In case τ is T or $\boldsymbol{T}$ we have to show
  $$\iota^\sharp[\![f(a_1, \ldots, a_k)]\!](G, \theta^\sharp) = f(\iota[\![f(a_1, \ldots, a_k)]\!](G, \theta)) \tag{A.25}$$
  or (A.22) and otherwise
  $$\iota^\sharp[\![f(a_1, \ldots, a_k)]\!](G, f(\theta)) \sqsupseteq \alpha(\iota[\![f(a_1, \ldots, a_k)]\!](G, \theta)). \tag{A.26}$$
  Let
  $$d_i := \iota[\![a_i]\!](G, \theta) \tag{A.27}$$
  $$d_i^\sharp := \iota^\sharp[\![a_i]\!](G, \theta^\sharp) \tag{A.28}$$
  By induction hypothesis,
  $$d_i^\sharp = f(d_i) \text{ or } d_i^\sharp \sqsupseteq \alpha(d_i) \tag{A.29}$$
  or (A.22) depending on the type of the i-th argument. With these premises, Def. 5.3.2 of canonical structure wrt. G, $G^\sharp$, and (f, g) yields the claim.
- t is of the form $\neg(t_1)$, $t_1 \vee t_2$, $t_1 \wedge t_2$:
  By induction hypothesis and the three-valued definition of term semantics, similar to the proof of Lemma 4.3.6.

□

Proof of Lemma 5.5.5, page 142. Let $M_1 \preceq_H M_2$ be two evolving topology transition systems compatible with signature $\mathcal{S}$.

1. Direct consequence of Def. 5.5.4.
2. The evolution of identity variables is constant for all $k \in \mathbb{N}_0$, thus preserves correspondence. The evolution chains assigned to destiny variables already correspond pointwise by Def. 5.3.3, thus any pair of k-suffixes of evolution chains for a given destiny variable also corresponds.

   The second claim follows similarly from Def. 5.3.3. If an evolution chain in θ ends with step $k-1$, then it is directly supposed to end in $\theta^\sharp$, too. In the other direction, if evolution ended only in $\theta^\sharp$ and not in θ, then the assignments didn't correspond.

□

Proof of Lemma 5.5.6, page 142. Let $s \sim_H s^\sharp$ be two corresponding states and $\pi \in \Pi_s(M)$ a path starting at s. For Def. 5.5.4, it is sufficient to show by induction that for each $n \in \mathbb{N}_0$ there is a finite sequence $s_0^\sharp s_1^\sharp \ldots s_n^\sharp$ of states from $S(M^\sharp)$ such that consecutive states are in transition relation, states with the same index correspond, i.e. $\pi^j \sim_H s_j^\sharp$, and individuals evolve and disappear correspondingly along the transitions $(\pi^j, \pi^{j+1}) \in R(M)$ and $(s_j^\sharp, s_{j+1}^\sharp) \in R(M^\sharp)$, i.e. $(\pi^j, \pi^{j+1}) \sim_e (s_j^\sharp, s_{j+1}^\sharp)$, for $0 \leq j \leq n$.

Induction base: Set $s_0^\sharp := s^\sharp$, then $s \sim_H s^\sharp$ by premises, consecutive states are trivially in transition relation, and individuals trivially evolve and disappear correspondingly.

Induction step: Assume a state sequence $s_0^\sharp s_1^\sharp \dots s_n^\sharp$ such that $\pi^i \sim_H s_i^\sharp$ for each $0 \leq i \leq n$. Then in particular $\pi^n \sim_H s_n^\sharp$.

By premises, $(\pi^n, \pi^{n+1}) \in R(M)$. Thus by Def. 5.4.2.2, there is a state $s_n^\sharp{}' \in S(M^\sharp)$ such that $\pi^{n+1} \sim_H s_n^\sharp{}'$, $(s_n^\sharp, s_n^\sharp{}') \in R(M^\sharp)$, and $(\pi^n, \pi^{n+1}) \sim_e (s_n^\sharp, s_n^\sharp{}')$.

□

Proof of Lemma 5.5.9, page 143. Assume M and $M^\sharp$, and $\theta^\sharp$ as in the premises.

1. Let $x \in V \cap \mathcal{V}^T$ and $(s, s^\sharp) \in H$ such that

$$\forall\, id^\sharp \in Id^\sharp\ \forall\, (\theta, s) \sim_H (\theta^\sharp[x \mapsto id^\sharp], s^\sharp) : \mathcal{M}[\![\phi]\!](s, \theta) = 1. \tag{A.30}$$

We prove (5.53) pointwise. To this end, let $\theta \in Assign_{\mathcal{M}^\sharp}(V)$ be an assignment of V such that $(\theta, s) \sim_H (\theta^\sharp, s^\sharp)$ and let $id \in Id$ be an identity of M. Then we set

$$\theta^{\sharp\prime} := \theta^\sharp[x \mapsto f(id)] \tag{A.31}$$

if $id \in U(s)$ and

$$\theta^{\sharp\prime} := \theta^\sharp[x \mapsto id_0^\sharp] \tag{A.32}$$

otherwise, where $id_0^\sharp \in Id^\sharp$ is the designated identity in $Id^\sharp$ which exists by Def. 5.4.2 and which either lies outside of $U(s^\sharp)$ or is labelled with $\top \in \mathcal{D}^\sharp(T)$ and $\top \in \mathcal{D}^\sharp(L)$, cf. Def. 5.3.3. Then we have

$$(\theta[x \mapsto id], s) \sim_H (\theta^{\sharp\prime}, s^\sharp) \tag{A.33}$$

by Def. 5.3.3. Together with (5.52) we obtain

$$\mathcal{M}[\![\phi]\!](s, \theta[x \mapsto id]) = 1 \tag{A.34}$$

as required.

2. Assume (5.54). Let $\theta \in Assign(s)$ be an assignment corresponding to $\theta^\sharp$ wrt. $(s, s^\sharp)$ and let π be a path in M starting at s. Assume, $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ are the destiny variables in the domain of θ (and $\theta^\sharp$, as they correspond). Let $\theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1] \dots [\boldsymbol{x}_n \mapsto \delta_n]$ be a modification of θ such that each destiny variable $\boldsymbol{x}_i$ is assigned an evolution chain δ_i along path, more precisely, $\delta_i = \Delta(\theta(\boldsymbol{x}_i)(0), \pi)$ $1 \leq i \leq n$.

By the Corresponding Path Lemma, Lemma 5.5.6, there is a path $\pi^\sharp$ in $M^\sharp$ which starts at $s^\sharp$ and corresponds to π, i.e. $\pi \sim_H \pi^\sharp$. By the Corrresponding Evolution Chain Lemma, Lemma 5.5.8, there is an evolution chain $\delta_i^\sharp$ along $\pi^\sharp$ which corresponds to δ_i for each $1 \leq i \leq n$. Then we have

$$(\theta', \pi) \sim_H (\theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1^\sharp] \dots [\boldsymbol{x}_n \mapsto \delta_n^\sharp], \pi^\sharp) \tag{A.35}$$

by Def. 5.3.3. Thus by (5.54), we have

$$\mathcal{M}[\![\phi]\!](\pi, \theta') = 1 \tag{A.36}$$

as required.

□

Proof of Lemma 5.5.10, page 144. Let φ be an AEvoCTL* formula over signature $\mathcal{S}^\sharp$.

Let $s \in S(M)$ and $s^\sharp \in S(M^\sharp)$ be corresponding states, i.e. $s \sim_H s^\sharp$, and let $\pi \in \Pi(M)$ and $\pi^\sharp \in \Pi(M^\sharp)$ be corresponding paths, i.e. $\pi \sim_H \pi^\sharp$. By $\theta^\sharp$ we denote an assignment of the free variables of φ in $s^\sharp$ or in $\pi^\sharp$.

We show by induction over the structure of φ that, in case φ is a state formula, we have

$$\mathcal{M}^\sharp[\![\varphi]\!](s^\sharp, \theta^\sharp) = 1 \implies \mathcal{M}[\![\varphi]\!](s, \theta) = 1 \tag{A.37}$$

for each pair of corresponding assignments $(\theta, s) \sim_H (\theta^\sharp, s^\sharp)$ and, in case φ is a path formula, we have

$$\mathcal{M}^\sharp[\![\varphi]\!](\pi^\sharp, \theta^\sharp) = 1 \implies \mathcal{M}[\![\varphi]\!](\pi, \theta) = 1 \tag{A.38}$$

for each pair of corresponding assignments $(\theta, \pi) \sim_H (\theta^\sharp, \pi^\sharp)$. The claim follows directly by definition of the satisfaction relation (cf. Section 4.4.2).

The proof doesn't assume that φ is $\odot$-free. The necessity of this premise in case we only have $M \preceq M^\sharp$ but not $M \preceq^\odot M^\sharp$ becomes obvious when considering the $\odot$ operator in the induction. This is the only place where this premise is needed.

Induction base: φ is of the form t, i.e., a boolean term:

Then

$$\begin{aligned}
&\mathcal{M}^\sharp[\![t]\!](s^\sharp, \theta^\sharp) = 1\\
&\iff\\
&\iota^\sharp[\![t]\!](\mathscr{L}^\sharp(s^\sharp), \theta^\sharp) = 1\\
&\implies \quad (\textit{Cor. 5.3.6})\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \iota[\![t]\!](\mathscr{L}(s), \theta) = 1\\
&\iff\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \mathcal{M}[\![t]\!](s, \theta) = 1
\end{aligned}$$

Induction step:

- φ is a state formula of the form $\odot\,\phi$:
  If $\odot$ appears in φ, we assume $M \preceq^\odot M^\sharp$. Then the claim follows by Def. 5.4.3.
- φ is a state formula of the form $\neg\phi$:
  As we consider AEvoCTL*, φ is in negative normal form, thus ϕ is in fact a boolean term $t \in \mathit{Term}_{\mathbb{B}}(\mathcal{S})$.

$\mathcal{M}^\sharp[\![\neg t]\!](s^\sharp, \theta^\sharp) = 1$
$\iff$ (*Def. 4.4.6*)
$1 - \iota^\sharp[\![t]\!](\mathscr{L}^\sharp(s^\sharp), \theta^\sharp) = 1$
$\iff$
$\iota^\sharp[\![t]\!](\mathscr{L}^\sharp(s^\sharp), \theta^\sharp) = 0$
$\implies$ (*Cor. 5.3.6*)
$\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \iota[\![t]\!](\mathscr{L}(s), \theta) = 0$
$\iff$ (*Def. 4.4.6*)
$\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \mathcal{M}[\![t]\!](s, \theta) = 0$

- φ is a state formula of the form $\phi_1 \vee \phi_2$:

$\mathcal{M}^\sharp[\![\phi_1 \vee \phi_2]\!](s^\sharp, \theta^\sharp) = 1$
$\iff$ (*Def. 4.4.6*)
$\max(\mathcal{M}^\sharp[\![\phi_1]\!](s^\sharp, \theta^\sharp), \mathcal{M}^\sharp[\![\phi_2]\!](s^\sharp, \theta^\sharp)) = 1$
$\iff$
$\mathcal{M}^\sharp[\![\phi_1]\!](s^\sharp, \theta^\sharp) = 1 \vee \mathcal{M}^\sharp[\![\phi_2]\!](s^\sharp, \theta^\sharp) = 1$
$\implies$ (*premise $s \sim_H s^\sharp$, induction hypothesis*)
$(\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \mathcal{M}[\![\phi_1]\!](s, \theta) = 1)$
$\vee(\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \mathcal{M}[\![\phi_2]\!](s, \theta) = 1)$
$\iff$
$\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) :$
$\mathcal{M}[\![\phi_1]\!](s, \theta) = 1 \vee \mathcal{M}[\![\phi_2]\!](s, \theta) = 1$
$\iff$
$\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) :$
$\max(\mathcal{M}[\![\phi_1]\!](s, \theta), \mathcal{M}[\![\phi_2]\!](s, \theta)) = 1$
$\iff$ (*Def. 4.4.6*)
$\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : \mathcal{M}[\![\phi_1 \vee \phi_2]\!](s, \theta) = 1$

- φ is a state formula of the form $\phi_1 \wedge \phi_2$:
  Similar to the previous case.
- φ is a state formula of the form $\forall\, x : T \,.\, \phi$:

$M^\sharp[\![\forall\, x : T \,.\, \phi]\!](s^\sharp, \theta^\sharp) = 1$
$\iff$ (*Def. 4.4.6*)
$\min\{M^\sharp[\![\phi]\!](s^\sharp, \theta^\sharp[x \mapsto \mathit{id}^\sharp]) \mid \mathit{id}^\sharp \in \mathit{Id}^\sharp \cap \mathcal{D}^\sharp(T)\} = 1$
$\iff$
$\forall\, \mathit{id}^\sharp \in \mathit{Id}^\sharp \cap \mathcal{D}^\sharp(T) : M^\sharp[\![\phi]\!](s^\sharp, \theta^\sharp[x \mapsto \mathit{id}^\sharp]) = 1$
$\implies$ (*premise $s \sim_H s^\sharp$, induction hypothesis*)
$\forall\, \mathit{id}^\sharp \in \mathit{Id}^\sharp \cap \mathcal{D}^\sharp(T)\ \forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : M[\![\phi]\!](s, \theta) = 1$

$$\begin{aligned}
&\Longrightarrow \quad (\mathit{Lemma\ 5.5.9.1})\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp)\ \forall\, \mathit{id} \in \mathit{Id} \cap \mathcal{D}(T) :\\
&\quad M\llbracket \phi \rrbracket(s, \theta[x \mapsto \mathit{id}]) = 1\\
&\Longleftrightarrow\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) :\\
&\quad \min\{M\llbracket \phi \rrbracket(s, \theta[x \mapsto \mathit{id}]) \mid \mathit{id} \in \mathit{Id} \cap \mathcal{D}(T)\} = 1\\
&\Longleftrightarrow\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : M\llbracket \forall\, x : T \,.\, \phi \rrbracket(s, \theta) = 1
\end{aligned}$$

- φ is a state formula of the form $\forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \phi$:

$$\begin{aligned}
&M^\sharp\llbracket \forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \phi \rrbracket(s^\sharp, \theta^\sharp) = 1\\
&\Longleftrightarrow \quad (\mathit{Def.\ 4.4.6})\\
&\min\{M^\sharp\llbracket \phi \rrbracket(s^\sharp, \theta^\sharp[\boldsymbol{x} \mapsto \mathit{id}^\sharp]) \mid \mathit{id}^\sharp \in U^{\circledcirc}(s^\sharp) \cap \mathcal{D}^\sharp(T)\} = 1\\
&\Longleftrightarrow\\
&\forall\, \mathit{id}^\sharp \in U^{\circledcirc}(\mathit{Id}^\sharp) \cap \mathcal{D}^\sharp(T) : M^\sharp\llbracket \phi \rrbracket(s^\sharp, \theta^\sharp[\boldsymbol{x} \mapsto \mathit{id}^\sharp]) = 1\\
&\Longrightarrow \quad (\mathit{premise}\ s \sim_H s^\sharp,\ \mathit{induction\ hypothesis})\\
&\forall\, \mathit{id}^\sharp \in U^{\circledcirc}(\mathit{Id}^\sharp) \cap \mathcal{D}^\sharp(T)\ \forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : M\llbracket \phi \rrbracket(s, \theta) = 1\\
&\Longrightarrow \quad (*)\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp)\ \forall\, \mathit{id} \in U^{\circledcirc}(s) \cap \mathcal{D}(T) :\\
&\quad M\llbracket \phi \rrbracket(s, \theta[\boldsymbol{x} \mapsto \mathit{id}]) = 1\\
&\Longleftrightarrow\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) :\\
&\quad \min\{M\llbracket \phi \rrbracket(s, \theta[\boldsymbol{x} \mapsto \mathit{id}]) \mid \mathit{id} \in U^{\circledcirc}(s) \cap \mathcal{D}(T)\} = 1\\
&\Longleftrightarrow\\
&\forall\, \theta \in \mathit{Assign}(s), (\theta, s) \sim_H (\theta^\sharp, s^\sharp) : M\llbracket \forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \phi \rrbracket(s, \theta) = 1
\end{aligned}$$

The justification for $(*)$ is similar to the reasoning in the proof of Lemma 5.5.9.1. Consider the embedding provided by the simulation relation. By Def. 5.2.1, it has the property that each identity alive in state s is mapped to an alive identity in $s^\sharp$, that is, $f(\mathit{id}) \in U^{\circledcirc}(s^\sharp)$ if $\mathit{id} \in U^{\circledcirc}(s)$. Thus the premise of $(*)$ covers in particular the cases in the conclusion.

- φ is a state formula of the form $\exists\, v \,.\, \phi$:

Explicitly excluded by the premise that φ is in AEvoCTL*.

- φ is a state formula of the form $\mathsf{A}\, \psi$:

$$\begin{aligned}
&\mathcal{M}^\sharp\llbracket \mathsf{A}\, \psi \rrbracket(\pi^\sharp, \theta^\sharp)\\
&\Longleftrightarrow \quad (\mathit{Def.\ 4.4.6})\\
&\min\{\mathcal{M}^\sharp\llbracket \psi \rrbracket(\pi^\sharp, \theta^{\sharp\prime}) \mid \pi^\sharp \in \Pi_{s^\sharp}(M^\sharp),\\
&\qquad\qquad \theta^{\sharp\prime} = \theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta^\sharp(\boldsymbol{x}_i)(0), \pi^\sharp), 1 \leq i \leq n\}
\end{aligned}$$

$\iff$

$\forall\,\pi^\sharp \in \Pi_{s^\sharp}(M^\sharp)\ \forall\,\theta^{\sharp\prime} = \theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1]\ldots[\boldsymbol{x}_n \mapsto \delta_n]:$

$\quad \mathcal{M}^\sharp\llbracket\psi\rrbracket(\pi^\sharp,\theta^{\sharp\prime}) = 1$

$\implies$ (*induction hypothesis*)

$\forall\,\pi^\sharp \in \Pi_{s^\sharp}(M^\sharp)\ \forall\,\theta^{\sharp\prime} = \theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1]\ldots[\boldsymbol{x}_n \mapsto \delta_n]$

$\quad \forall\,\pi \in \Pi_s(M), \pi \sim_H \pi^\sharp\ \forall\,\theta' \in \mathit{Assign}(\pi), (\theta',\pi) \sim_H (\theta^{\sharp\prime},\pi^\sharp):$

$\quad\quad \mathcal{M}\llbracket\psi\rrbracket(\pi,\theta') = 1$

$\implies$ (*Lemma 5.5.9.2*)

$\forall\,\theta \in \mathit{Assign}(s), (\theta,s) \sim_H (\theta^\sharp,s^\sharp)$

$\quad \forall\,\pi \in \Pi_s(M)\ \forall\,\theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1]\ldots[\boldsymbol{x}_n \mapsto \delta_n] : \mathcal{M}\llbracket\psi\rrbracket(\pi,\theta') = 1$

$\implies$

$\forall\,\theta \in \mathit{Assign}(s), (\theta,s) \sim_H (\theta^\sharp,s^\sharp):$

$\quad \min\{\mathcal{M}^\sharp\llbracket\psi\rrbracket(\pi^\sharp,\theta^{\sharp\prime}) \mid \pi^\sharp \in \Pi_{s^\sharp}(M^\sharp),$

$\quad\quad\quad \theta^{\sharp\prime} = \theta^\sharp[\boldsymbol{x}_1 \mapsto \delta_1]\ldots[\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta^\sharp(\boldsymbol{x}_i)(0),\pi^\sharp), 1 \le i \le n\}$

$\iff$ (*Def. 4.4.6*)

$\forall\,\theta \in \mathit{Assign}(s), (\theta,s) \sim_H (\theta^\sharp,s^\sharp) : \mathcal{M}\llbracket \mathsf{A}\,\psi\rrbracket(\pi,\theta)$

- φ is a state formula of the form $\mathsf{E}\,\psi$:
  Explicitly excluded by the premise that φ is in AEvoCTL*.
- $\varphi = \psi$, a path formula where ϕ is a state formula.
  In order to apply the induction hypothesis, we aim for a syntactically shorter sub-formula of φ, i.e. which has fewer nodes in the parse tree. Following [34], we can view the right hand side as $\mathsf{path}(\phi)$, that is, comprising the otherwise invisible operator 'path', which casts path formulae into state formulae. Then

  $M^\sharp\llbracket\mathsf{path}(\phi)\rrbracket(\pi^\sharp,\theta^\sharp) = 1$

  $\iff$ (*Definition*)

  $M^\sharp\llbracket\phi\rrbracket({\pi^\sharp}^0,\theta^\sharp) = 1$

  $\implies$ (*premise $\pi \sim_H \pi^\sharp$, induction hypothesis*)

  $\forall\,\theta \sim_H \theta^\sharp : M\llbracket\phi\rrbracket(\pi^0,\theta) = 1$

  $\iff$ (*Definition*)

  $\forall\,\theta \sim_H \theta^\sharp : M\llbracket\mathsf{path}(\phi)\rrbracket(\pi,\theta) = 1$
- φ is a path formula of the form $\otimes\, a$:

  $\mathcal{M}^\sharp\llbracket\otimes\, a\rrbracket(\pi^\sharp,\theta^\sharp) = 1$

  $\iff$

  $\mathit{id}^\sharp \in U^{\circledcirc}(s^\sharp)$, and $\mathit{id}^\sharp \notin \mathrm{dom}(e\langle(s^\sharp,s^{\sharp\prime})\rangle)$ or $e\langle(s^\sharp,s^{\sharp\prime})\rangle(\mathit{id}^\sharp) \in U^{\not\circledcirc}(s^\sharp) \setminus U^{\circledcirc}(s^\sharp)$, where $s^\sharp := {\pi^\sharp}^0$, $s^{\sharp\prime} := {\pi^\sharp}^1$, and $\mathit{id}^\sharp := \iota^\sharp\llbracket a\rrbracket(\mathscr{L}(s^\sharp),\theta^\sharp)$

  $\implies$ ($\pi \sim_H \pi^\sharp$, *Def. 5.4.1*, $(*)$)

$$\begin{aligned}
&\forall\,\theta \in \mathit{Assign}(\pi), (\theta,\pi) \sim_H (\theta^\sharp,\pi^\sharp),\\
&\mathit{id} \in U^{\circledcirc}(s), \text{ and } \mathit{id} \notin \operatorname{dom}(e\langle(s,s')\rangle) \text{ or } e\langle(s,s')\rangle(\mathit{id}) \in\\
&U^{\not\circledcirc}(s) \setminus U^{\circledcirc}(s), \text{ where } s := \pi^0,\ s' := \pi^1, \text{ and } \mathit{id} := \iota[\![a]\!](\mathscr{L}(s),\theta)\\
&\iff\\
&\forall\,\theta \in \mathit{Assign}(\pi), (\theta,\pi) \sim_H (\theta^\sharp,\pi^\sharp) : \mathcal{M}[\![\otimes\, a]\!](\pi,\theta) = 1
\end{aligned}$$

where $(*)$ refers to the fact that $f(\mathit{id}) = \mathit{id}^\sharp$ by Lemma 5.3.5, the Embedding Lemma.

- φ is a path formula of the form $\neg\psi$:

  By premises, φ is in negative normal form, that is ψ is of the form $\otimes\, a$. The reasoning is then similar to the previous case.

- φ is a path formula of the form $\psi_1 \vee \psi_2$ or $\psi_1 \wedge \psi_2$:

  Similar to the logical connectives for state formulae above.

- φ is a path formula of the form $\forall\, x : T \,.\, \psi$:

  Similar to state formulae above.

- φ is a path formula of the form $\forall\, \boldsymbol{x} : \boldsymbol{T} \,.\, \psi$:

  Similar to state formulae above using the Corresponding Evolution Chain Lemma, Lemma 5.5.8, to obtain corresponding evolution chains along π for the ones considered in $\pi^\sharp$.

- φ is a path formula of the form $\exists\, x : T \,.\, \psi$ or $\exists\, \boldsymbol{x} : \boldsymbol{T} \,.\, \psi$:

  Explicitly excluded by the premise that φ is in AEvoCTL*.

- φ is a path formula of the form $\mathsf{X}\,\psi$:

$$\begin{aligned}
&\mathcal{M}^\sharp[\![\mathsf{X}\,\psi]\!](\pi^\sharp,\theta^\sharp) = 1\\
&\iff \quad (\textit{Def. 4.4.6})\\
&\varepsilon \notin (\theta^\sharp/1)(\mathit{Free}(\psi)) \wedge \mathcal{M}^\sharp[\![\psi]\!](\pi^\sharp/1,\theta^\sharp/1) = 1\\
&\implies \quad (\textit{premise } \pi \sim_H \pi^\sharp\textit{, induction hypothesis, Lemma 5.5.5})\\
&\forall\,\theta \in \mathit{Assign}(\pi), (\theta,\pi) \sim_H (\theta^\sharp,\pi^\sharp) : \varepsilon \notin (\theta/1)(\mathit{Free}(\psi)) \text{ and}\\
&\forall\,\theta' \in \mathit{Assign}(\pi/1), (\theta',\pi/1) \sim_H (\theta^\sharp/1,\pi^\sharp/1) : \mathcal{M}[\![\psi]\!](\pi/1,\theta') = 1\\
&\implies \quad (\textit{similar to proof of Lemma 5.5.9})\\
&\forall\,\theta \in \mathit{Assign}(\pi), (\theta,\pi) \sim_H (\theta^\sharp,\pi^\sharp) :\\
&\quad \varepsilon \notin (\theta/1)(\mathit{Free}(\psi)) \wedge \mathcal{M}[\![\psi]\!](\pi/1,\theta/1) = 1\\
&\iff \quad (\textit{Def. 4.4.6})\\
&\forall\,\theta \in \mathit{Assign}(\pi), (\theta,\pi) \sim_H (\theta^\sharp,\pi^\sharp) : \mathcal{M}[\![\mathsf{X}\,\psi]\!](\pi,\theta) = 1
\end{aligned}$$

- φ is a path formula of the form $\mathsf{F}\,\psi$, $\mathsf{G}\,\psi$, $\psi_1\,\mathsf{U}\,\psi_2$, or $\psi_1\,\mathsf{R}\,\psi_2$:

  Similar to the previous case using Lemma 5.5.5.

□

A.3. Proofs of Chapter 7: Query Reduction

Proof of Lemma 7.1.12, page 181. Let $\mathcal{S}$ be a signature, G be a $\mathcal{S}$-compatible topology over *Id* and $\mathcal{M}$ a structure which is canonical wrt. G and symmetric in identities.

Let $t \in \mathit{Term}(\mathcal{S})$, p a permutation of *Id*, and $\theta \in \mathit{Assign}_{\mathcal{M}}(\mathit{Free}(t))$. We only have to consider occurrences of functional terms because $\tilde{p}$ is the identity on $\mathbb{B}$, that is, boolean terms have the same value on the left and right hand side of (7.7) thus completely boolean terms satisfy the requirement immediately.

Induction base:

- t is of the form x or $\boldsymbol{x}$:

$$\iota[\![x]\!](p(G), p(\theta)) = p(\theta(x)) = \tilde{p}(\theta(x)) = \tilde{p}(x[\![G, \theta]\!]()). \tag{A.39}$$

  Similarly for destiny variables by definition of permutation of evolution chains.
- t is of the form 1: Trivial.

Induction step:

- t is of the form $\circledcirc\, a_1$:

$$\iota[\![\circledcirc\, a_1]\!](p(G), p(\theta)) = 1$$
$$\iff \quad (\mathit{Def.\ 4.3.3})$$
$$\iota[\![a_1]\!](p(G), p(\theta)) \in U^{\circledcirc}(p(G)) \setminus U^{\not\circledcirc}(p(G))$$
$$\iff \quad (\mathit{induction\ hypothesis})$$
$$\tilde{p}(\iota[\![a_1]\!](G, \theta)) \in U^{\circledcirc}(p(G)) \setminus U^{\not\circledcirc}(p(G))$$
$$\iff \quad (\mathit{Def.\ 7.1.6.2\ and\ 3,\ i.e.\ permutation\ respects\ aliveness})$$
$$\tilde{p}(\iota[\![a_1]\!](G, \theta)) \in p(U^{\circledcirc}(G)) \setminus p(U^{\not\circledcirc}(G))$$
$$\iff \quad (\mathit{Def.\ 7.1.9,\ i.e.\ definition\ of}\ \tilde{p})$$
$$\iota[\![a_1]\!](G, \theta) \in U^{\circledcirc}(G) \setminus U^{\not\circledcirc}(G)$$
$$\iff \quad (\mathit{Def.\ 4.3.3})$$
$$\iota[\![\circledcirc\, a_1]\!](G, \theta) = 1$$
$$\iff \quad (\mathit{Def.\ 7.1.9,\ i.e.\ definition\ of}\ \tilde{p})$$
$$\tilde{p}(\iota[\![\circledcirc\, a_1]\!](G, \theta)) = 1$$

  Similarly for the negative case, the remaining cases yield $1/2$.
- t is of the form $\sigma(a_1)$:

$$\iota[\![\sigma(a_1)]\!](p(G), p(\theta))$$
$$= \quad (\mathit{Def.\ 4.3.3})$$
$$\sigma(p(G))(\iota[\![a_1]\!](p(G), p(\theta)))$$
$$= \quad (\mathit{induction\ hypothesis})$$
$$\sigma(p(G))(\tilde{p}(\iota[\![a_1]\!](G, \theta)))$$

$=$ $\quad$ (*Def. 7.1.9, i.e. definition of* $\tilde{p}$)
$\sigma(p(G))(p(\iota\llbracket a_1\rrbracket(G,\theta)))$
$=$ $\quad$ (*Def. 7.1.6.4*)
$\sigma(G)(p^{-1}(p(\iota\llbracket a_1\rrbracket(G,\theta))))$
$=$
$\sigma(G)(\iota\llbracket a_1\rrbracket(G,\theta))$,

where Note 7.1.7 assures that $\sigma(G)$ and $\sigma(t(G))$ are either both applied to values of their domain, or none of them.

- t is of the form $\lambda(a_1)$:
  Induction hypothesis and Def. 7.1.6.8.
- t is of the form $a_1 = a_2$:
  Induction hypothesis and Def. 7.1.6.1.
- t is of the form $f(a_1, \dots, a_k)$:
  Induction hypothesis and premise that $\mathcal{M}$ is a symmetric structure.

□

Proof of Lemma 7.2.4, page 183. Let $\mathcal{S}$ be a signature, M a $\mathcal{S}$-compatible ETTS over *Id*, and $\mathcal{M}$ a structure canonical wrt. M such that M and $\mathcal{M}$ are symmetric in identities. Let state $s \in S(M)$ be a state of M and π a path in M.

Let φ be an EvoCTL* formula over $\mathcal{S}$, $\theta \in \mathit{Assign}_{\mathcal{M}}(\mathit{Free}(\varphi))$ an assignment of the free variables of φ, and p a permutation of *Id*.

The proof is by induction over the structure of φ.

Induction base: φ is of the form t, i.e., a boolean term:

Then

$\mathcal{M}\llbracket t\rrbracket(s,\theta) = \iota\llbracket t\rrbracket(\mathscr{L}(s),\theta)$
$=$ $\quad$ (*Def. 7.2.1, Cor. 7.1.13*)
$\iota\llbracket t\rrbracket(p(\mathscr{L}(s)), p(\theta)) = \mathcal{M}\llbracket t\rrbracket(p(s), p(\theta))$

Induction step:

- φ is a state formula of the form $\odot\,\phi$:
  For the positive case, we have
  $\mathcal{M}\llbracket\odot\,\phi\rrbracket(s,\theta) = 1$
  $\iff$ $\quad$ (*Def. 4.4.6*)
  $\mathit{id} := \iota\llbracket a\rrbracket(\mathscr{L}(s),\theta)$ exists, is in $U^{\circledcirc}(s)$, and $\forall\, r = ('s, s) \in R(M)$:
  $\text{☼} \overset{r}{\rightsquigarrow}_e \mathit{id} \wedge \forall\, '\mathit{id} \in \mathit{Id} :' \mathit{id} \overset{r}{\rightsquigarrow}_e \mathit{id} \implies '\mathit{id} \notin U^{\circledcirc}('s)$
  $\iff$ $\quad$ (*Def. 7.2.1.2b, Def. 7.1.6.2, i.e. the permutation respects evolution and aliveness*)

$id := \iota[\![a]\!](p(\mathscr{L}(s)), p(\theta))$ exists, is in $U^{\circledcirc}(p(s))$, and $\forall\, r = ('s, p(s)) \in R(M):$

$\text{҉} \overset{r}{\rightsquigarrow}_e id \wedge \forall\, 'id \in Id :' id \overset{r}{\rightsquigarrow}_e id \implies 'id \notin U^{\circledcirc}('s)$

$\iff \quad (\mathit{Def.\ 4.4.6})$

$\mathcal{M}[\![\odot\, \phi]\!](p(s), p(\theta)) = 1.$

The negative case follows similarly, the remaining cases are indefinite for both sides.

- φ is a state formula of the form $\neg\phi$, $\phi_1 \vee \phi_2$, or $\phi_1 \wedge \phi_2$:

  Induction hypothesis.

- φ is a state formula of the form $\forall\, x : T\,.\,\phi$ or $\exists\, x : T\,.\,\phi$:

  By the following property of the sets over which the minimum and maximum is formed according to Def. 4.4.6.

  $$\{M[\![\phi]\!](s, \theta[x \mapsto id]) \mid id \in Id \cap \mathcal{D}(T)\}$$
  $$= \quad (\mathit{induction\ hypothesis})$$
  $$\{M[\![\phi]\!](p(s), p(\theta[x \mapsto id])) \mid id \in Id \cap \mathcal{D}(T)\}$$
  $$=$$
  $$\{M[\![\phi]\!](p(s), (p(\theta))[x \mapsto p(id)]) \mid id \in Id \cap \mathcal{D}(T)\}$$
  $$= \quad (\mathit{Def.\ 7.1.5,\ i.e.\ p\ bijective\ and\ compatible\ with\ partitions})$$
  $$\{M[\![\phi]\!](p(s), (p(\theta))[x \mapsto id]) \mid id \in Id \cap \mathcal{D}(T)\}$$

- φ is a state formula of the form $\forall\, \boldsymbol{x} : \boldsymbol{T}\,.\,\phi$ or $\exists\, \boldsymbol{x} : \boldsymbol{T}\,.\,\phi$:

  Similar to the previous case, in addition using property Def. 7.1.6.2, i.e. that permutation respects aliveness.

- φ is a state formula of the form $\mathsf{A}\,\psi$ or $\mathsf{E}\,\psi$:

  Similar to previous cases with the following property of the set over which the minimum and maximum is formed according to Def. 4.4.6.

  $$\{\mathcal{M}[\![\psi]\!](\pi, \theta') \mid \pi \in \Pi_s(M),$$
  $$\theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), 1 \leq i \leq n\}$$
  $$= \quad (\mathit{induction\ hypothesis})$$
  $$\{\mathcal{M}[\![\psi]\!](p(\pi), p(\theta')) \mid \pi \in \Pi_s(M),$$
  $$\theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), 1 \leq i \leq n\}$$
  $$=$$
  $$\{\mathcal{M}[\![\psi]\!](p(\pi), \theta') \mid \pi \in \Pi_s(M),$$
  $$\theta' = p(\theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n]), \delta_i \in \Delta(\theta(\boldsymbol{x}_i)(0), \pi), 1 \leq i \leq n\}$$
  $$= \quad (\mathit{Lemma\ 7.2.2})$$
  $$\{\mathcal{M}[\![\psi]\!](\pi, \theta') \mid \pi \in \Pi_{p(s)}(M),$$
  $$\theta' = \theta[\boldsymbol{x}_1 \mapsto \delta_1] \ldots [\boldsymbol{x}_n \mapsto \delta_n], \delta_i \in \Delta(p(\theta)(\boldsymbol{x}_i)(0), \pi), 1 \leq i \leq n\}$$

- φ is a path formula of the form ϕ, i.e. a state formula:

  In order to apply the induction hypothesis, we aim for a syntactically shorter sub-formula of φ, i.e. which has fewer nodes in the parse tree. Following [34], we can view the right hand side as $\mathsf{path}(\phi)$, that is, comprising the otherwise invisible operator 'path', which casts path formulae into state formulae. Then

  $$\begin{aligned}&\mathcal{M}[\![\mathsf{path}(\phi)]\!](\pi,\theta)\\ =&\quad (\textit{Definition})\\ &\mathcal{M}[\![\phi]\!](\pi,\theta)\\ =&\quad (\textit{induction hypothesis})\\ &\mathcal{M}[\![\phi]\!](p(\pi),p(\theta))\\ =&\quad (\textit{Definition})\\ &\mathcal{M}[\![\mathsf{path}(\phi)]\!](p(\pi),p(\theta))\end{aligned}$$

- φ is a path formula of the form $\otimes\, a$:

  Similar to the $\odot\, a$ case above, by preservation of aliveness and non-aliveness, consistent evolution in symmetric ETTS, and the induction hypothesis.

- φ is a path formula of the form $\neg\psi$:

  By premises, φ is in negative normal form, that is ψ is of the form $\otimes\, a$. The reasoning is then similar to the previous case.

- φ is a path formula of the form $\psi_1 \vee \psi_2$ or $\psi_1 \wedge \psi_2$:

  Similar to the logical connectives for state formulae above.

- φ is a path formula of the form $\forall\, x : T\, .\, \psi$ or $\exists\, x : T\, .\, \psi$:

  Similar to state formulae above.

- φ is a path formula of the form $\forall\, \boldsymbol{x} : \boldsymbol{T}\, .\, \psi$ or $\exists\, \boldsymbol{x} : \boldsymbol{T}\, .\, \psi$:

  Similar to state formulae above.

- φ is a path formula of the form $\mathsf{X}\,\psi$:

  $$\begin{aligned}&\mathcal{M}[\![\mathsf{X}\,\psi]\!](\pi,\theta) = 1\\ \iff&\quad (\textit{Def. 4.4.6})\\ &\varepsilon \notin (\theta/1)(\textit{Free}(\psi)) \wedge \mathcal{M}[\![\psi]\!](\pi/1,\theta/1) = 1\\ \iff&\quad (\textit{Note 7.2.3, induction hypothesis})\\ &\varepsilon \notin (p(\theta)/1)(\textit{Free}(\psi)) \wedge \mathcal{M}[\![\psi]\!](p(\pi)/1,p(\theta)/1) = 1\\ \iff&\quad (\textit{Def. 4.4.6})\\ &\mathcal{M}[\![\mathsf{X}\,\psi]\!](p(\pi),p(\theta)) = 1\end{aligned}$$

  The negative case follows similarly, the remaining cases are indefinite for both sides.

- φ is a path formula of the form $\mathsf{F}\,\psi$, $\mathsf{G}\,\psi$, $\psi_1\,\mathsf{U}\,\psi_2$, or $\psi_1\,\mathsf{R}\,\psi_2$:

  Similar to the previous case using Note 7.2.3 for general $k \in \mathbb{N}_0$.

□

Proof of Lemma 7.3.2, page 184. Let $V = \{x_1, \dots, x_n\} \subseteq \mathcal{V}^T$, $n \in \mathbb{N}_0$, be a finite set of identity variables of the same type T and let Id_0 denote the partition used as $\mathcal{D}(T)$.

Then $\Theta := \Theta_n$, recursively defined by

$$\Theta_0 := \{\emptyset\} \tag{A.40}$$

$$\Theta_{k+1} := \{\theta \cup \{x_{k+1} \mapsto id_{0_1}\}, \dots, \theta \cup \{x_{k+1} \mapsto id_{0_{k+1}}\} \mid \theta \in \Theta_k\}, \tag{A.41}$$

where the $id_{0_1}, \dots, id_{0_k}$ are pairwise different as long as $|Id_0| \geq k$ and $id_{0_{k+1}} = id_{0_k}$ if $k \geq |Id_0|$, is a finite assignment basis of V as demonstrated in the following. It comprises $N! \cdot N^{N-n}$ different assignments where $N = \min(n, |Id_0|)$, that is, $n!$ if $|Id_0| \geq n$.

Let $\theta \in Assign_{\mathcal{M}}(V,)$ be an assignment of V in $\mathcal{M}$. Then it is of the form

$$\theta = \{x_1 \mapsto \theta(x_1), \dots, x_n \mapsto \theta(x_n)\}. \tag{A.42}$$

Let $m \in \{1, \dots, n\}$ denote the number of *different* identities $\{id_1, \dots, id_m\}$ employed by θ. Note that, as θ ranges over Id_0, it cannot use more identities than present in Id_0, i.e. we have $m \leq |Id_0|$.

Define $[id_i] := \{x_j \mid \theta(x_j) = id_i, 1 \leq j \leq n\}$ to denote the (non-empty) set of variables mapped to identity id_i, $1 \leq i \leq m$.

Without loss of generality we may assume that the variables sets are ordered by size, i.e. that

$$|[id_1]| \geq |[id_2]| \geq \cdots \geq |[id_m]|. \tag{A.43}$$

Set

$$p := \{id_{0_1} \mapsto id_1, \dots, id_{0_m} \mapsto id_m\} \tag{A.44}$$

and choose

$$\theta_0 := \{x_j \mapsto id_{0_i} \mid x_j \in [id_i], 1 \leq i \leq m, 1 \leq j \leq n\} \tag{A.45}$$

from Θ. Then

$$p(\theta_0)(x_j) = p(\underbrace{\theta_0(x_j)}_{=id_{0_i}}) = id_i = \theta(x_j), \tag{A.46}$$

where $i \in \{1, \dots, m\}$ such that $x_j \in [id_i]$, thus $p(\theta_0) = \theta$. □

Proof of Lemma 7.3.4, page 185. Let $V \subseteq \mathcal{V}^T$ be a finite set of $n \in \mathbb{N}_0$ identity variables of the same type and Θ constructed as defined above.

If Θ were not minimal, there would be an assignment $\theta \in \Theta$ such that $\Theta \setminus \{\theta\}$ is still an assignment basis. In other words, there were a permutation p of Id such that there is an $\theta \neq \theta_0 \in \Theta$ such that $\theta = p(\theta_0)$

Assume, θ were such a redundant assignment. Then it were of the form

$$\theta = \{x_1 \mapsto id_1, \dots, x_n \mapsto id_n\} \tag{A.47}$$

with $id_i \in \{id_{0_1}, \dots, id_{0_n}\}$. Consider the minimal permutation p, that is, the permutation which coincides with the identity function on as many points as possible, and the assignment θ_0 with $p(\theta_0) = \theta$. If there were an $1 \leq i \leq n$ such that $p(id_i) = p(id_{0_j}) = id_{0_k}$

with $k < j$, then $p(\theta_0)$ would map two logical variables to the same value which obtain different values in θ, thus they would not be equivalent. If there were an $1 \leq i \leq n$ such that $p(id_i) = p(id_{0_j}) = id_{0_k}$ with $j < k$, then id_{0_j} would be missing in the range as p is assumed minimal, thus $p(\theta_0)$ would not be in Θ, in particular not equal to θ. Thus p is the identity function and $\theta = p(\theta_0) = \theta_0$ in contradiction to the assumption. □

Proof of Cor. 7.3.5, page 185. Partition V into non-empty sets $V_1 \,\dot{\cup}\, \ldots \,\dot{\cup}\, V_n$ such that V_i comprises all identity variables from V of type $T_i \in \mathcal{T}(\mathcal{S})$.

Let Θ_i be a minimal assignment basis of V_i, which exists by Lemma 7.3.4, $1 \leq i \leq n$. Then $\Theta = \Theta_1 \times \cdots \times \Theta_n$ is minimal finite assignment basis of V, where the cross-product of assigment bases is defined as

$$\Theta_1 \times \cdots \times \Theta_n := \{\theta_1 \,\dot{\cup}\, \ldots \,\dot{\cup}\, \theta_n \mid \theta_1 \in \Theta_1, \ldots, \theta_n \in \Theta_n\}, \tag{A.48}$$

that is, all possible unions of one assignment from each base.

It is a basis because all assignments of V are reachable by considering the occurring types independently and minimal because removal of one assignment from Θ can't be type-consistently compensated by one of the other bases. □

Proof of Lemma 7.4.1, page 189. Let $V = \{x_1, \ldots, x_n\} \subseteq \mathcal{V}^T$, $n \in \mathbb{N}_0$, be a finite set of identity variables of the same type T and let $Id_{0,1}, \ldots, Id_{0,m}$ be the sub-partitioning of the partition $\mathcal{D}(T)$.

Then $\Theta := \Theta_n$, recursively defined by

$$\Theta_0 := \{\emptyset\} \tag{A.49}$$

$$\begin{aligned}\Theta_{k+1} := \{&\theta \cup \{x_{k+1} \mapsto id_{1,1}\}, \ldots, \theta \cup \{x_{k+1} \mapsto id_{1,k+1}\},\\ &\ldots,\\ &\theta \cup \{x_{k+1} \mapsto id_{m,1}\}, \ldots, \theta \cup \{x_{k+1} \mapsto id_{m,k+1}\} \mid \theta \in \Theta_k\},\end{aligned} \tag{A.50}$$

where the $id_{i,1}, \ldots, id_{i,k} \in Id_{0,i}$ are pairwise different as long as $|Id_{0,i}| \geq k$ and $id_{0_{k+1}} = id_{0_k}$ if $k \geq |Id_{0,i}|$, is a finite assignment basis of V. The proof is similar to the one of Lemma 7.4.1. □

A.4. Proofs of Chapter 9: DTR/QR for Higher-Level Languages

Proof of Lemma 9.2.1, page 225. According to Def. 7.2.1, we have to show that labels discriminate states, and that each partitioning consistent permutation p of Id has a well-defined automorphism function, which is an automorphism of M and consistent with evolution.

The first requirements is trivially satisfied as the set of states is identified with the labelling domain.

So let $p : Id \to Id$ be a partitioning consistent permutation of Id, i.e. respect the partitioning by classes. Its permutation function is $a_p = \{s \mapsto p(s)\}$ where $p(s)$ denotes the p-permutation of topology s, thus well-defined and bijective.

Following Note 7.1.4, it is sufficient to show only one direction for the automorphism property.

- Let $s \in S_0(M)$ be an initial state of M. That is, s satisfies $cond_0$, or $\iota[\![cond_0]\!](s, id, \emptyset) = 1$. By premises, the interpretation ι is symmetric in the sense of Section 9.2.3, that is, $\iota[\![cond_0]\!](p(s), p(id), p(\emptyset)) = 1$. Thus $p(s) \in S_0(M)$.
- Let $(s, s') \in R(M)$ be a transition in M. By Section 9.1.2, there is necessarily a scheduled sequence
$$u_1/r_{C_1,1}, \ldots, u_n/r_{C_n,n} \in \mathcal{S}(s) \tag{A.51}$$
with
  - $id_i \in U^{\circledcirc}(s)$,
  - if r_i is of the form
$$(s_{C_i}, ev_i\,[cond_i]/act_i, s'_{C_i}) \tag{A.52}$$
then
    - $s(u_i).x_{st} = s_{C_i}$
    - either $ev_i \neq \varepsilon$ or $ev_i = E(p_1, \ldots, p_m)$ and
$$s(u_i).\epsilon = \epsilon_0.E(d_1, \ldots, d_m) \tag{A.53}$$
, and
    - $\iota[\![cond_i]\!](s, u_i, \theta) = 1$ with $\theta = \{p_1 \mapsto d_1, \ldots, p_m \mapsto d_m\}$

  and s' is the result of
  - subsequently applying the transition programs to s,
  - consuming the ready-to-consume events, and
  - setting the input valuations in s to values from the input oracle $\mathcal{O}$.

  By premises, the scheduler is symmetric, thus according to Section 9.2.3,
$$p(u_1)/r_{C_1,1}, \ldots, p(u_n)/r_{C_n,n} \in \mathcal{S}(p(s)). \tag{A.54}$$
  We have
  - $p(id_i) \in U^{\circledcirc}(p(s))$. because permutation of topologies preserves aliveness,
  - $(p(s))(u_i).x_{st} = s(u_i).x_{st} = s_{C_i}$ because permutation of topologies preserves the local state,
  - either $ev_i \neq \varepsilon$ or $ev_i = E(p_1, \ldots, p_m)$ and
$$(p(s))(u_i).\epsilon = \epsilon_0.E(p(d_1), \ldots, p(d_m)) \tag{A.55}$$
, by symmetry of the ether, and

- $\iota[\![cond_i]\!](p(s), p(u_i), p(\theta)) = 1$ by symmetry of the ether and the interpretation.

With (9.70) to (9.72) we can construct $p(s)'$. As, by premises, interpretation and creation oracle are also symmetric, $p(s)'$ exists and is equal to $p(s')$ according to Section 9.2.3. Thus $(p(s), p(s')) \in R(M)$.

The permutation p is consistent with evolution because

$$u \overset{r}{\rightsquigarrow}_e u' \iff u' = e\langle r\rangle(u) \iff u' = id_{U^{\circledcirc}(s)\cap U^{\circledcirc}(s')}(u) \tag{A.56}$$

which is equivalent to the application of the chosen local transitions to s leaving u alive. By Section 9.2.3, this is the case if and only if the application of the chosen local transitions to $p(s)$ leaves the individual $p(u)$ alive, thus to

$$p(u) \overset{r}{\rightsquigarrow}_e p(u'). \tag{A.57}$$

□

Proof of Lemma 9.3.7, page 249. Let $\mathscr{M}$, ι, $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{O}$ as required above and let

$$D = \{(d_{j_1}, Id_{j_1}), \ldots, (d_{j_m}, Id_{j_m})\} \tag{A.58}$$

be a DTR. Let $M = \iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})$ be the ETTS of $\mathscr{M}$ and $M^\sharp = D(M)$ the DTR abstraction according to Chapter 6. Furthermore, let $M_D = \iota[\![\mathscr{M}^\sharp_D]\!](\mathcal{E}, \mathcal{S}, \mathcal{O}^\sharp))$ be the ETTS of the syntactically transformed HLL model $\mathscr{M}^\sharp_D$.

The claim is $M^\sharp \preceq M_D$, that is, that M_D simulates $M^\sharp$ in the sense of Def. 5.4.2. To verify this claim, we've got to provide a relation H between the state sets $S(M^\sharp)$ and $S(M_D)$ such that:

1. a pair from H determines an embedding between the two topologies,
2. initial states in $S(M^\sharp)$ have a correspondence in $S(M_D)$, and
3. given a pair of corresponding states, if $S(M^\sharp)$ can take a transition to a destination state, then $S(M_D)$ can make a transition to a corresponding destination.

We define the embedding relation based on the embedding of individuals

$$\begin{aligned} f : Id(M^\sharp) &\to Id(M_D) \\ id &\mapsto \begin{cases} \mathtt{C}_C \in \mathcal{D}(\tau_{C^{\mathtt{C}}}) & \text{, if } id = \mathtt{C}_C \in \mathcal{D}(\tau_C) \\ id & \text{, otherwise} \end{cases} \end{aligned} \tag{A.59}$$

This is principally the identity function on identities, but with the particularity that it maps $\mathtt{C}_C$, which is added to the domain $\mathcal{D}(\tau_C)$ of class C in $M^\sharp$ to the identity $\mathtt{C}_C$ from $\mathcal{D}(\tau_{C^{\mathtt{C}}})$, that is, to the identity of a newly introduced class which is present in every state of M_D.

Then the simulation relation is then

$$\begin{aligned}H = \{(s^\sharp, s_D) \in S(M^\sharp) \times S(M_D) \mid \\ & U(s_D) = f(U(s^\sharp)), \\ & \forall\, id \in D, id \neq \mathsf{C}_C : \sigma(M^\sharp)(id) = \sigma(M_D)(f(id)), \\ & \exists\, g : L(M^\sharp) \rightarrowtail L(M_D) : \\ & \quad \forall\, \ell \in L(M^\sharp) : \psi(\ell) = f(\psi(g(\ell))), \lambda(\ell) = \lambda(g(\ell))\}\end{aligned} \tag{A.60}$$

that is, it relates states comprising the same identities with identical local state (excluding C_C) and the same links, in particular the self-links of C_C.

The simulation relation is not empty, but rather a one-to-one mapping, because for each state $s^\sharp \in S(M^\sharp)$, it is possibly to construct a state s_D in $S(M_D)$ considering the topologies obtained for $\mathscr{M}_D^\sharp$ by the ETTS semantics of HLL models (cf. Figure 9.8). Item 1 requires a topology embedding for a pair $(s^\sharp, s_D)$. This is f from (A.59) restricted to $s^\sharp$ and the link embedding from (A.60).

Item 2 is trivial because we only consider the empty topology as initial state by Section 9.3.3. In general, given an initial state $s^\sharp$, one would get hold of a concrete s which is, by the ETTS semantics of HLL known to satisfy $cond_0$ with a particular binding of free variables and would transfer this binding to a binding for the simulating s_D, choosing inputs accordingly.

For item 3, consider $(s^\sharp, s_D) \in H$ and let $(s^\sharp, s^{\sharp\prime}) \in R(M^\sharp)$. Then by definition of $M^\sharp$ as DTR of M, there are states s, s' in M that are concretisations of the two abstract states, i.e. $D(s) = s^\sharp$ and $D(s') = s^{\sharp\prime}$, and that are in transition relation, i.e. $(s, s') \in R(M)$.

By the definition of the ETTS semantics of HLL and because $\mathcal{S}$ is an interleaving scheduler, there is a single scheduled individual/transition program pair

$$\mathcal{S}(s) = u/r \tag{A.61}$$

which is ready in s and s' is the result of applying the transition program of r to s, i.e. s' is obtained from

$$s_0' = \iota[\![act]\!](s, u, \theta, \mathcal{O})(s) \tag{A.62}$$

by the procedure of Section 9.1.2. To complete the proof, we'll argue that

$$f(u)/r_D, \tag{A.63}$$

where r_D is the syntactical transformation from Sections 9.3.3 and 9.3.3, is scheduled in M_D and show by induction over actions that applying the actions of r_D to s_D with a suitable choice of inputs yields a state s_D' which is in simulation relation with $s^{\sharp\prime}$. In addition, corresponding evolution is established because, for instance, creation and destruction are controlled by the transition program of transition r.

If u/r causes the transition, with u denoting an object of class C, is scheduled in s but not ready, then either there is no event to consume or the guarding condition doesn't hold. Distinguish two cases:

- If $u \in D$ or of a sort not considered by D, then u has the same local state in $M^\sharp$ and in M_D thus there is no event both, so u/r is also not ready in M_D. Similarly for the guarding condition.
- Otherwise $u_D = f(u) \in Id_{C^{\mathtt{C}}}$. The individual u still has an event to be consumed and a local state (including input valuation) which causes the guarding condition to hold.

  As all local states turn into inputs in M_D and as $\mathcal{O}^\sharp$ allows all possible values in any state for all newly introduced inputs, we can choose $o \in \mathcal{O}^\sharp$ such that each $x \in X_{C^{\mathtt{C}}}$, which is an input in $\mathscr{M}^\sharp_D$, obtains exactly the value $s(u).x$, i.e. such that

  $$o \subseteq \{(u_D, x) \mapsto s(u).x \mid x \in X_C\} \cup \{(u_D, \lambda) \mapsto s(u).\lambda \mid \lambda \in \Lambda_C\} \tag{A.64}$$

  Then r_D is ready in s_D because it has ε as event part and the guarding condition holds by construction of o and the transformation of the guarding condition into a let-expression in Section 9.3.3. The proof is by induction over the structure of the guarding condition in normal form, that is, as a let-expression, and the DTR transformation similar to the case of actions below.

Knowing that f_u/r_D is ready in s_D if u/r is in s, we now show by induction over the structure of actions and expressions, that

$$\begin{aligned} &\forall\, s \in S(M), s_D \in S(M_D) : (D(s), s_D) \in H \\ &\quad \implies \iota[\![act]\!](s, u, \theta, \mathcal{O})(s) = \iota[\![act_D]\!](s_D, f(u), \theta, \mathcal{O}^\sharp)(s_D) \end{aligned} \tag{A.65}$$

for all actions *act*.

So let *act* be the action part of the label of the ready transition r.

Induction base:

- *act* is of the form `skip`: Trivial.
- *act* is of the form $(\mathit{this} \rightarrowtail h' := \mathit{this})$:
  Then $act_D = \mathit{this} \rightarrowtail h' := \mathit{this}$. This action preserves (A.65).
- *act* is of the form $(\mathit{this} \rightarrowtail h' := p)$:
  Then $act_D = \mathit{this} \rightarrowtail h' := p$ in class C and i instead of p in class $C^{\mathtt{C}}$. In the former case, the event carries the same parameters in s and s_D, in the latter case, we can choose i accordingly in the oracle value o, thus (A.65).
- *act* is of the form $(\mathit{this} \rightarrowtail h' := \mathit{this} \rightarrowtail h' \rightarrowtail \lambda)$:
  Then

  $$act_D = \mathit{this} \rightarrowtail h' := \left(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \;?\; i_{h,\lambda} : \mathit{this} \rightarrowtail h' \rightarrowtail \lambda\right). \tag{A.66}$$

  Similar to the previous case, o can be chosen from the oracle to establish (A.65).

- act is of the form $(\mathit{this} \rightarrowtail h^{\mathtt{C}'} \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}})$:
  Similar to previous case.
- act is of the form $\mathit{this} \rightarrowtail h' \rightarrowtail x' \mathrel{\mathtt{:=}} \mathit{expr}$:
  Then

  $$\mathit{act}_D = \mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then}\ \mathtt{skip};\ \mathtt{else}\ \mathit{this} \rightarrowtail h' \rightarrowtail x' \mathrel{\mathtt{:=}} \mathit{expr}^{\sim};\ \mathtt{fi}. \tag{A.67}$$

  If the individual u denoted by $\mathit{this} \rightarrowtail h$ is in D or of a sort not considered by D, then the else-branch is taken in act_D and $f(u) = u$ is updated preserving (A.65) given the transformation of expressions.

  Otherwise, $f(u) \in \mathit{Id}_{C^{\mathtt{C}}}$ is not updated, which preserves (A.65), too.
- act is of the form $\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0$, which is the only form we need to consider by Def. 9.3.1:
  Then

  $$\begin{aligned} \mathit{act}_D = {} & \mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then}\ \mathtt{skip}; \mathtt{else} \\ & \qquad \mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0'; \\ & \qquad \mathit{this} \rightarrowtail h' \rightarrowtail \lambda^{\mathtt{C}'} \mathrel{\mathtt{:=}} \mathit{this} \rightarrowtail h_0^{\mathtt{C}'}; \\ & \mathtt{fi} \end{aligned} \tag{A.68}$$

  It preserves (A.65) analogously to the previous case.
- act is of the form $\mathit{this} \rightarrowtail h' \rightarrowtail \lambda' \mathrel{\mathtt{:=}} \mathtt{new}\ C$:

  Let u be the newly created individual in M. If u is in D or of a sort not considered by D, then u can also be created in M_D by choice of the oracle. Otherwise, i in (9.136) can be chosen to take the first branch and leave the concrete individuals in M_D untouched.

  Update of the left-hand side is analogous to the two previous cases, thus (A.65) is preserved.
- act is of the form $\mathit{this} \rightarrowtail h' \,\mathtt{!}\, E(\mathit{expr}_1, \ldots, \mathit{expr}_n)$:

  If the destination individual u as denoted by $\mathit{this} \rightarrowtail h$ is in D or of a sort not considered by D, then (A.65) is preserved by (9.137) and the treatment of expressions. Otherwise, (A.65) is also preserved because local states remain unchanged.
- act is of the form $\mathtt{delete}\ \mathit{this} \rightarrowtail h'$:
  Then

  $$\mathit{act}_D = \mathtt{if}\ \mathit{this} \rightarrowtail h^{\mathtt{C}'}\ \mathtt{then}\ \mathtt{skip};\ \mathtt{else}\ \mathtt{delete}\ \mathit{this} \rightarrowtail h';\ \mathtt{fi} \tag{A.69}$$

  If the destination individual u as denoted by $\mathit{this} \rightarrowtail h$ is in D or of a sort not considered by D, then (9.140) deletes u. Otherwise, (A.65) is also preserved because local states remain unchanged.

Induction step:

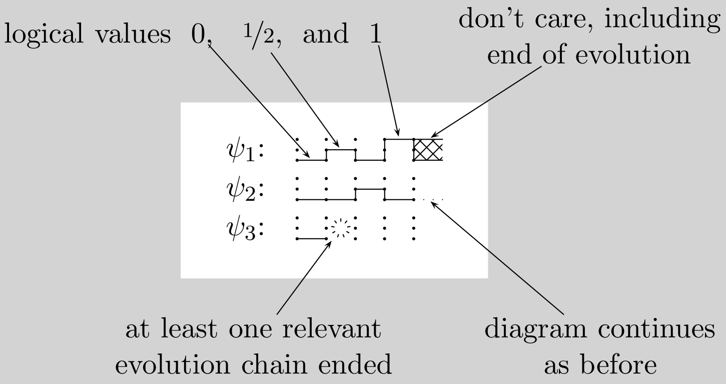
- act is of the form `if` $cond$ `then` act_1 `else` act_2 `fi`:
  Then
  $$nav_D = \texttt{if } cond^{\sim} \texttt{ then } act_1^{\sim} \texttt{ else } act_2^{\sim} \texttt{ fi}. \qquad \text{(A.70)}$$
  By treatment of expressions and induction hypothesis, (A.65) is preserved.
- act is of the form $(nav_1; nav_2)$:
  Directly by induction hypothesis.

□

B. Additional Figures

B.1. Additional Figures for Chapter 4: Property Specification Logic

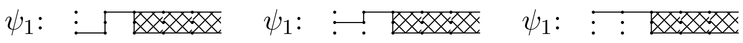
A variant of timing diagrams is used to visualise the evaluation of temporal operators in terms of the evaluations of the sub-formulae. The timing diagram given below shows the evaluations of three path formulae $\psi_1, \ldots, \psi_3$ over five steps of a computation path, without explicitly denoting at *which* step the evaluation starts. In the first step, ψ_1 evaluates to 0, in the second step to $1/2$, in the third step to 0, etc. The dots in the fifth step in the row of ψ_2 indicate that the diagram continues as before. In the particular case values of 0 or $1/2$.
Note that the hatch-filled box indicates *don't care* values, that is, the evaluation of ψ_1 at the fifth step is not relevant for the case illustrated by this diagram. It may even include the end of (formally) relevant evolution chains. An evolution chain is relevant if it is bound to a free $\boldsymbol{T}$-typed variable of the formula. For example, the third row in the timing diagram below indicates that an evolution chain bound to a variable from $\mathit{Free}(\psi_3) \cap \mathcal{V}^{\boldsymbol{T}}$ ended after the first step.


logical values 0, 1/2, and 1
don't care, including end of evolution
ψ1:
ψ2:
ψ3:
at least one relevant evolution chain ended
diagram continues as before

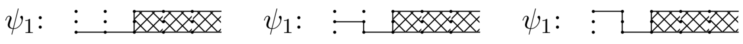
B.1.1. Semantics of "X ψ_1".

The evaluation indicates to which value ψ_1 evaluates in the next step. In contrast to the other operators, indefinite evaluation of ψ_1 *now* does not force the whole formula to indefinite evaluation; only the *next* step is considered.

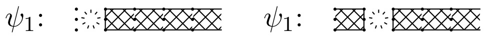
- **Positive:** ψ_1 holds in the next step and no relevant evolution chain ended beforehand.



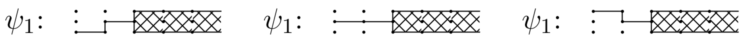
- **Negative:** ψ_1 evaluates to 0 in the next step and no relevant evolution chain ended beforehand.



- **Indefinite (by disappearance):** A relevant evolution chain ends before or in the next step.



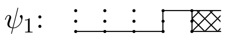
- **Indefinite (by indefiniteness):** ψ_1 evaluates to 1/2 in the next step and no relevant evolution chain ended beforehand.



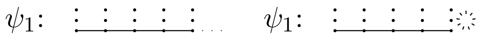
B.1.2. Semantics of "F ψ_1".

The evaluation indicates whether there is a witness for the classical positive or negative interpretation. That is, as long as ψ_1 didn't hold once, it evaluates to 0. In the negative case, variables of kind $\boldsymbol{T}$ may shorten the considered length of the path. Otherwise, the outcome is indefinite.

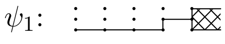
- **Positive:** ψ_1 finally holds and evaluates to 0 beforehand.



- **Negative:** ψ_1 evaluates to 0 forever or for the length of the shortest relevant evolution chain.



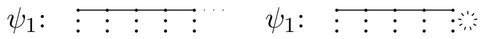
- **Indefinite:** ψ_1 evaluates indefinite before it held once.



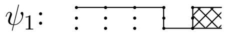
B.1.3. Semantics of "G ψ_1".

The evaluation indicates whether ψ_1 holds globally. Disappearance of relevant individuals shortens the considered path to finite length. If sub-formula ψ_1 evaluates to 0 and held beforehand, the outcome is negative. Otherwise, the outcome is indefinite.

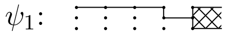
- **Positive:** ψ_1 globally holds as long as no relevant evolution chain ends.



- **Negative:** ψ_1 evaluates to 0 before any relevant evolution chain ends.

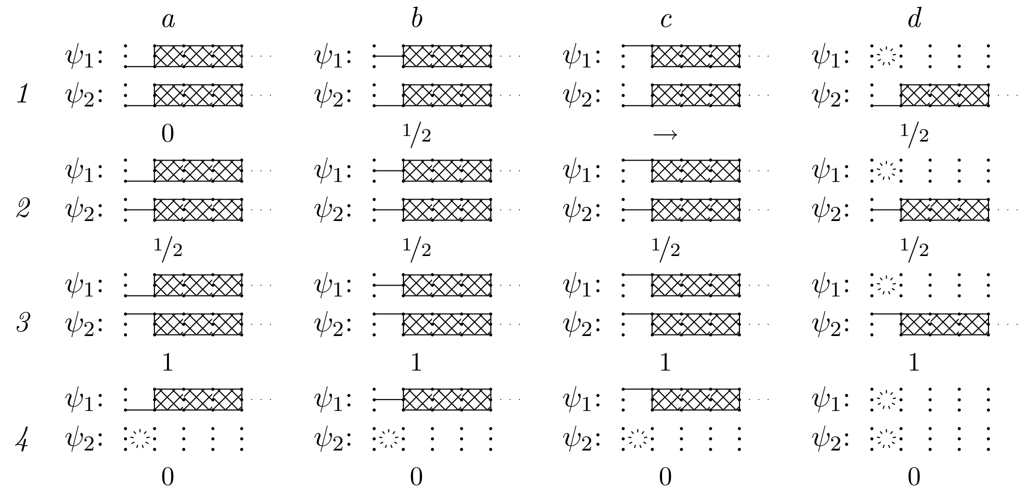


- **Indefinite:** ψ_1 evaluates to 1/2 before any relevant evolution chain ends.



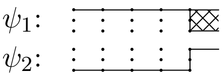
B.1.4. Semantics of "ψ_1 U ψ_2".

The following table gives the semantics of "ψ_1 U ψ_2" by consideration of the first state. Subsequently we give the semantics grouped by outcome as exercised previously for the other temporal operators.



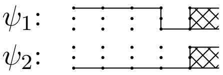
The whole formula holds immediately if ψ_2 evaluates to 1, independent from the first one (cf. row *3*). It evaluates indefinite if ψ_1 or ψ_2 become indefinite pre-maturely (cf. *1.b* and *1.d* as well as row *2*). If ψ_1 still holds, and ψ_2 not yet, we're supposed to observe further (cf. *1.c*). Otherwise, it evaluates to 0.

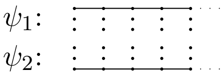
- **Positive:** ψ_2 finally holds, ψ_1 constantly holds

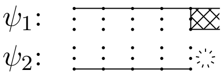


until right before that point in time, and nobody is lost on-route.

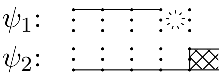
- **Negative:** Either ψ_2 never holds before a relevant individual disappears while ψ_1 constantly holds, or there is a state where both don't hold and ψ_2 didn't hold earlier, and none of the relevant evolution chains ended beforehand.



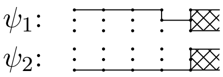


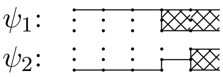


- **Indefinite (by disappearance):** Individuals relevant for the first sub-formula disappear before a definite evaluation could've been obtained.

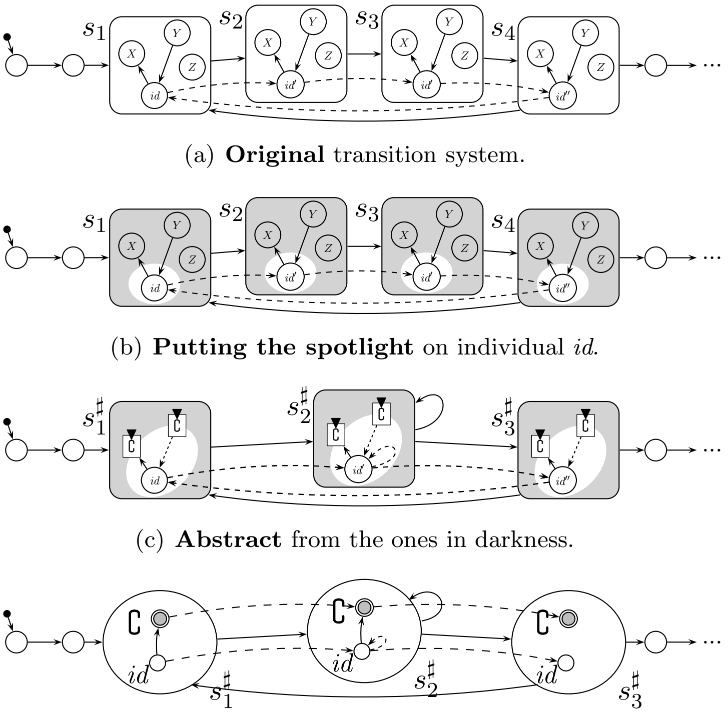


- **Indefinite (by indefiniteness):** Otherwise.





B.2. Additional Figures for Chapter 6: Data-Type Reduction



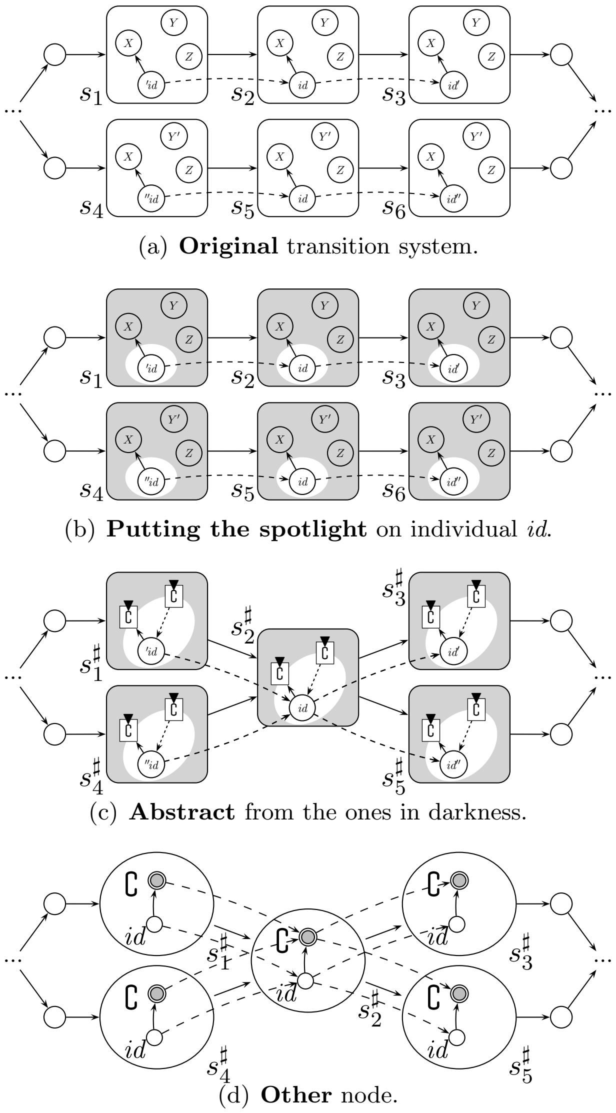
(a) **Original** transition system.

(b) **Putting the spotlight** on individual *id*.

(c) **Abstract** from the ones in darkness.

(d) **Other** node.

Figure B.1.: **An intuition** of Data-type reduction within a transition system where predicates do not distinguish states. (Or: illustration of the effect of different states with the same label. Over-approximation is preserved, but information lost. Fortunately, generated systems are good, states are distinguishable.)



(a) **Original** transition system.

(b) **Putting the spotlight** on individual *id*.

(c) **Abstract** from the ones in darkness.

(d) **Other** node.

Figure B.2.: **An intuition** of Data-type reduction within a transition system where the evolution of individuals not only depends on the local state. (Or: illustration of another not-so-nice effect, namely that the focused individual depends on another individual which it doesn't know by links. Here's the question, whether this is actually a problem – maybe it's hard to capture by case-split, but in the end Y would be abstracted anyway, even if we had a link.)

List of Symbols

$\perp$	bottom element, page 30
$\dot{\cup}$	disjoint union, page 23
$\sqsupseteq$	partial order, page 27
$\sqcap$	greatest lower bound, page 28
$\sqsupset$	strict partial order, page 27
$\sqsubseteq$	partial order, page 27
$\sqsubset$	strict partial order, page 27
$\sqcup$	least upper bound, page 28
$\top$	top element, page 30
$\{\!\mid, \mid\!\}$	multi-set comprehension, page 25
$\emptyset$	function symbol, empty multi-set, page 81
$\odot\, p$	Mett, newly created, page 283
$\odot\, a$	state formula newly created, page 96
$\circledcirc\, a_1$	functional term aliveness, page 87
$\otimes p$	Mett, disappearing, page 283
$\otimes\, a$	path formula disappearing, page 96
❄	crystallisation function, page 191
$*$	function symbol navigation, page 82
$\lll$	binding priority, page 89
$=$	function symbol comparison for equality, page 81
$id \rightarrowtail \lambda$	application of link navigation, page 46
$\rightarrowtail^{\sharp}_{\lambda}$	abstract link navigation, page 129
$\rightarrowtail_{\lambda}$	link navigation, page 46

$\rightarrowtail^{G}_{\lambda}$	link navigation of topology G, page 46
$^1\!/_2$	uncertain, indefinite, page 27
a	automorphism of transition system, page 177
A	DCS initial states, page 282
a	functional term, page 87
A	UML state machine action labelling, page 288
A^*	finite sequences over A including ε, page 25
A^+	finite sequences over A, page 25
act	HLL action, page 206
act^{NF}	HLL action normal form, page 234
(α, γ)	Galois connection (short), page 31
A^ω	infinite sequences over A, page 25
$AP^\sharp$	abstract atomic propositions, page 127
AP	atomic propositions, page 33
$\mathsf{A}\,\psi$	universal path quantification, page 96
$Assign_{\mathcal{M}}(V)$	assignments of variables V in structure $\mathcal{M}$, page 84
$Assign_{\mathcal{M}}(V, \pi)$	assignments of V in path π, page 85
$Assign_{\mathcal{M}}(V, s)$	assignments of V in state s, page 85
$\forall\, x : T \,.\, \psi_1$	path formula universal identity quantification, page 96
$\forall\, x : T \,.\, \phi_1$	state formula universal identity quantification, page 96
$\forall\, x : T \,.\, t_1$	logical term universal identity quantification, page 87
$\forall\, \boldsymbol{x} : T \,.\, \psi_1$	path formula universal destiny quantification, page 96
$\forall\, \boldsymbol{x} : T \,.\, \phi_1$	state formula universal destiny quantification, page 96
$\forall\, \boldsymbol{x} : T \,.\, t_1$	logical term universal destiny quantification, page 87
$\mathbb{B}$	basic type booleans, page 80
$\mathbb{B}$	boolean truth values, page 27
$\mathbb{B}_3$	Kleene truth values, page 27

$\texttt{bool}$	array program booleans, page 261
$\langle\!\langle C \rangle\!\rangle$	encoding of HLL class C in array programs, page 265
c	Mett channel, page 283
c	UML class, page 288
C	UML classes, page 288
$\mathscr{C}$	HLL classes, page 203
χ	DCS channels, page 282
cond	HLL condition, page 204
cond_0	HLL model initial system state, page 203
$\mathit{conn}[c](p_1, p_2)$	Mett, link query, page 283
D	DTR, page 148
$\mathcal{D}$	domains, page 83
$D(G)$	DTR D applied to G, page 149
$D(M)$	DTR D applied to ETTS M, page 151
$\mathcal{D}(\tau)$	domain of type τ, page 83
$D_1 \sqsubseteq D_2$	precision order on DTRs, page 155
$\mathscr{D}$	state labelling domain, page 33
$\texttt{delete}(\mathit{this} \rightarrowtail \lambda)$	array program destruction, page 263
$\texttt{delete}\ \langle \mathit{nav} \rangle$	HLL destroy action, page 206
$\delta^\sharp$	evolution chain in abstract system, page 142
$\Delta(\mathit{id}, \pi)$	set of maximal finite and infinite evolution chains of id along π, page 52
$(\delta, \pi) \sim_H (\delta^\sharp, \pi^\sharp)$	corresponding evolution chain, page 142
$\mathrm{dom}(\rightarrowtail_\lambda)$	domain of link navigation, page 46
e	edge, page 26
E	edges of a graph, page 26
E	HLL event, page 203

E	UML events, page 288
$E(G)$	edges of graph G, page 26
$\epsilon \oplus (u_1, E, u_2)$	ether insert, page 211
$(\mathcal{E}, \oplus, \ominus)$	HLL ether, page 211
$\epsilon \ominus (u_1, E, u_2)$	ether remove, page 211
$\epsilon_0.E$	abbreviation for ether query, page 211
$\epsilon_0.u_1 \xrightarrow{E} u_2$	consumption of E from ether ϵ possible, page 211
$e\langle r\rangle$	evolution relation of transition r, page 51
e	evolution annotation, page 52
$\mathscr{E}$	HLL events, page 203
$\mathcal{E}_{\mathrm{msg}}$	DCS environment messages, page 282
$\mathsf{E}\,\psi$	existential path quantification, page 96
ε	empty sequence, page 25
eq_{Id}	equality function, page 45
E_v	outgoing edges at v, page 26
$\exists\, x : T \,.\, \psi_1$	path formula existential identity quantification, page 96
$\exists\, x : T \,.\, \phi_1$	state formula existential identity quantification, page 96
$\exists\, x : T \,.\, t_1$	logical term existential identity quantification, page 87
$expr$	HLL functional term, page 205
$\exists\, \boldsymbol{x} : T \,.\, \psi_1$	path formula existential destiny quantification, page 96
$\exists\, \boldsymbol{x} : T \,.\, \phi_1$	state formula existential destiny quantification, page 96
$\exists\, \boldsymbol{x} : T \,.\, t_1$	logical term existential destiny quantification, page 87
F	fairness constraints, page 32
f	function symbol, page 80
$\mathcal{F}$	function symbols, page 79
f	identity embedding, page 129
f	vertex labelling function, page 26

$f(G)$	vertex labelling of graph G, page 26
$F(M)$	fairness constraints of LTS M, page 33
$\mathcal{F}(\mathcal{S})$	function symbols of signature $\mathcal{S}$, page 80
(f, g)	embedding, page 129
$f^{-1}(b)$	set of pre-images, page 24
$f_1 \sqsubseteq f_2$	information order on functions, page 29
$f : A \rightarrow B$	total function, page 24
$(f_{G,G^\sharp}, g_{G,G^\sharp})$	embedding, topologies explicated, page 130
$\mathsf{F}\,\psi_1$	path formula finally, page 96
$\mathit{Free}(\varphi)$	free variables in formula φ, page 97
$\mathit{Free}(t)$	free variables in t, page 88
$G^\sharp$	abstract topology, page 129
g	edge labelling function, page 26
g	link embedding, page 130
$g(G)$	edge labelling of graph G, page 26
$G_1 \sqsubseteq G_2$	information order on topologies, page 86
$G \sqsubseteq^{(f,g)} G^\sharp$	(f, g) embeds topology G into $G^\sharp$, page 130
G	graph, page 26
G	topology, page 45
$\mathsf{G}\,\psi_1$	path formula globally, page 96
H	simulation relation, page 138
$\iota[\![\mathscr{M}]\!](\mathcal{E}, \mathcal{S}, \mathcal{O})$	ETTS semantics of HLL model $\mathscr{M}$, page 208
$\mathit{id}^\sharp$	abstract identity, page 129
$\mathit{Id}^\sharp$	abstract identities, page 129
$\mathit{Id}_i \in D$	partition Id_i is considered by D, page 148
$\mathit{id}, \mathit{id}_1, \mathit{id}_2$	identity, page 46
id_A	identity function, page 25

Id_i	sort, typing of identities, page 49
$\inf(\pi)$	states occurring infinitely often in path π, page 34
$instate[q](p)$	METT, in given local state, page 283
`int`	array program infinite integers, page 261
$\iota^\sharp$	abstract interpretation, page 135
ι	HLL interpretation, page 208
ι	interpretation, page 83
$\iota[\![cond]\!](s_0, s, u, \theta)$	HLL evaluation of condition, page 212
$\iota[\![P]\!](s_0, u, \theta, \mathcal{O})$	HLL transition program application, page 214
$\iota[\![t]\!](G, \theta)$	valuation of term t, page 89
k	arity, page 80
ℓ	HLL transition label, page 204
ℓ	link, page 46
L	links, page 46
L	basic type link sets, page 80
L	UML associations, page 288
$L(c)$	UML associations of a class, page 288
$L(G)$	links of topology G, page 46
$(L, \sqsubseteq, \bot)$	meet semi-lattice, page 30
$(L, \sqsubseteq, \top)$	join semi-lattice, page 30
$(L, \sqsubseteq, \top, \bot)$	complete lattice, page 30
$(L, \sqsubseteq_L)$	partially ordered set, page 27
(L, α, γ, M)	Galois connection, page 31
λ	HLL link name, page 203
Λ	link names, page 45
Λ	link names, page 79
$\lambda(a_1)$	functional term link navigation, page 87

$\lambda(G)$	link name function of topology G, page 46
$\lambda(\ell)$	link name of link ℓ, page 46
$\Lambda(\mathcal{S})$	link names of signature $\mathcal{S}$, page 80
$\Lambda_{A,C}$	auxiliary links of HLL class C, page 203
Λ_C	links of HLL class C, page 203
$\Lambda_{I,C}$	input links of HLL class C, page 203
$\mathrm{last}(\mathit{term}^L)$	last auxiliary link in HLL expression, page 232
$\mathscr{L}^\sharp$	state labelling of abstract system, page 130
$\mathscr{L}$	state labelling function, page 32
$\mathscr{L}(M)$	state labelling of LTS M, page 33
$M^\sharp$	abstract transition system, page 136
m	METT message, page 283
$\mathcal{M}$	structure, page 83
M	UML state-machines, page 288
$\mathfrak{M}(A)$	multi-sets over A, page 25
$M(c)$	UML state machine of class c, page 288
$\mathcal{M}[\![\varphi]\!](\pi,\theta)$	valuation of formula φ on path π under θ, page 99
$\mathcal{M}[\![\varphi]\!](s,\theta)$	valuation of formula φ in state s under θ, page 99
$\mathcal{M}[\![\varphi]\!](T)$	evaluation of formula φ in tree T, page 158
$M_1 \sqsubseteq M_2$	information order on ETTS, page 86
$M \preceq^{\odot} M^\sharp$	$M^\sharp$ simulates M with corresponding appearance, page 138
$M \preceq^{F} M^\sharp$	fair simulation, page 139
$M \preceq_{H} M^\sharp$	$M^\sharp$ simulates M by simulation relation H, page 138
$M \preceq M^\sharp$	$M^\sharp$ simulates M, page 138
M	labelled transition system, page 32
$\max(a,b)$	maximum in $\mathbb{B}_3$, page 32
$\max A$	maximum of set A, page 32

$\mathcal{M}[\![\varphi]\!]_F(s,\theta)$	fair valuation, page 103
$\min(a,b)$	minimum in $\mathbb{B}_3$, page 32
$\min A$	minimum of set A, page 32
$\langle\!\langle \mathscr{M} \rangle\!\rangle^{\sharp}_{D}$	encoding of DTR D on HLL model $\mathscr{M}$ in array programs, page 273
$\mathscr{M}$	HLL model, page 204
$\mathit{mode}\ x : \mathit{type}$	array program variable declaration, page 262
$\mathbb{N}^+$	postive natural numbers, page 23
$[N, M]$	array program integer interval, page 261
$\mathbb{N}_0$	natural numbers including 0, page 23
$\mathit{nav}' := \mathit{expr}$	array program assignment, page 263
nav	HLL navigation expression, page 205
nav	array program expression, page 262
$\texttt{new}\ C$	HLL creation expression, page 205
$.n_i$	function symbol attribute access, page 82
$\mathcal{O}$	HLL creation oracle (overloaded), page 214
$\mathcal{O}$	HLL input oracle (overloaded), page 214
$o(s) \xrightarrow{\{u_i/r_i\}} s'$	HLL transition via scheduling sequence, page 217
Ω	DCS fragile states, page 282
$\mathcal{P}$	DCS protocol, page 282
p	Mett logical variable, page 283
p	permutation, page 178
$\mathcal{P}$	predicate symbols, page 80
$\mathfrak{P}(A)$	power-set of A, page 25
$p(\delta)$	p-permutation of evolution chain δ, page 180
$p(G)$	p-permutation of topology, page 179
$p(\mathcal{O}(s))$	permutation of HLL oracle, page 223
$\mathcal{P}(\mathcal{S})$	predicate symbols of signature $\mathcal{S}$, page 80

$p(\theta)$	p-permuted assignment, page 180
$\mathit{pend}[m](p_1, p_2, p)$	Mett, pending query, page 283
φ	EvoCTL* formula, page 96
ϕ	state formula, page 96
$\varphi_1 \equiv \varphi_2$	semantical equivalence, page 103
$\varphi_1 \equiv_{\text{prop}} \varphi_2$	semantical equivalence under property, page 104
$\psi_1 \mathsf{R} \psi_2$	path formula release, page 96
$\psi_1 \mathsf{U} \psi_2$	path formula until, page 96
$\pi^\sharp$	path in abstract system, page 139
π	computation path, page 33
π^k	k-th state of path π, page 33
$\pi(k)$	k-th element of sequence π, page 25
$\Pi(M)$	set of paths in LTS M from an initial state, page 34
π/k	suffix of path π starting at k, page 33
π/k	suffix of sequence π from k, page 25
$\Pi^F(M)$	set of fair paths in LTS M from an initial state, page 34
$\Pi_s(M)$	set of paths from state s in LTS M, page 34
$\Pi_s^F(M)$	set of fair paths from state s in LTS M, page 34
$\Pi_{s,s'}(M)$	finite paths in LTS M from state s to state s', page 34
$\pi \sim_H \pi^\sharp$	corresponding paths, page 141
$\mathcal{P}^k$	predicate symbols of arity k, page 80
ψ	path formula, page 96
ψ	incidence function, page 26
$\psi(G)$	incidence function of graph G, page 26
$\psi(G)$	incidence function of topology G, page 46
Q	DCS states, page 282
$r^\sharp$	transition of abstract system, page 136

$R^\sharp$	transition relation of abstract system, page 138
$\mathbb{R}$	real numbers, page 23
R	transition relation, page 32
R	UML state machine transition relation, page 288
$R(\iota\llbracket\mathscr{M}\rrbracket(\mathcal{E},\mathcal{S},\mathcal{O}))$	transition relation of HLL model, page 217
$R(M)$	transition relation of LTS M, page 33
$\mathbb{R}_0^+$	positive real numbers including 0, page 23
(R, e)	transition relation with evolution annotation, page 57
r^{-1}	inversion of r, page 24
R_C	transitions of HLL class, page 203
$\mathit{recv}[m](p_1, p_2, p)$	Mett, receive query, page 283
$r \sim_e r^\sharp$	corresponding evolution and disappearance, page 137
$s^\sharp$	state of abstract transition system, page 130
$\mathcal{M}^\sharp$	abstract structure, page 133
$\mathcal{S}$	HLL scheduler, page 216
$\mathcal{S}$	signature, page 79
S	states, page 32
S	basic type local states, page 80
S	UML state machine states, page 288
$S(\iota\llbracket\mathscr{M}\rrbracket(\mathcal{E},\mathcal{S},\mathcal{O}))$	topologies of HLL model, page 210
$S(M)$	states of LTS M, page 33
$\mathcal{S}(s)$	schedulings for state s, page 210
$s(u).\epsilon$	ether-component of u in state s, page 210
$s(u).\lambda$	navigation of u via λ in state s, page 210
$s(u).x$	x-component of u's local state in state s, page 210
S_0	initial states, page 32
S_0	UML state machine initial states, page 288

$S_0(M)$	initial states of LTS M, page 33
S_{0_C}	initial states of HLL class, page 203
$\mathcal{S}_1 \subseteq \mathcal{S}_2$	subset-relation for signatures, page 80
S_C	states of HLL class, page 203
$\mathit{send}[m](p_1, p_2, p)$	Mett, send query, page 283
$\Sigma^\sharp$	abstract local states, page 129
Σ	DCS messages, page 282
Σ	local states, page 45
$\sigma(a_1)$	functional term local state, page 87
$\sigma(G)$	local state function of topology G, page 46
$\sigma(\mathit{id})$	local state of identity id, page 46
Σ_C	HLL local state domain, page 209
Σ_E	edge labels, page 26
Σ_i	typed local states, page 49
Σ_V	vertex labels, page 26
succ	DCS transition relation, page 282
$s \sim_H s^\sharp$	corresponding states, page 141
t	logical term, page 87
T	computation tree, page 158
$\mathcal{T}$	types, page 79
$\mathcal{T}(\mathcal{S})$	types of signature $\mathcal{S}$, page 80
$t.\lambda$	abbreviation for $\lambda(t)$, page 88
$t.\lambda.\cdots.\lambda.\sigma$	abbreviation for $\sigma(*(\lambda(\ldots*(\lambda(t))\ldots)))$, page 88
$t.\sigma$	abbreviation for $\sigma(t)$, page 88
$(t_1 \leftrightarrow t_2)$	abbreviation for equivalence, page 88
$t_1 \equiv t_2$	semantical equivalence, page 90
$(t_1 \rightarrow t_2)$	abbreviation for implication, page 88

$(t_1 \dot{\vee} t_2)$	abbreviation for exclusive-or, page 88
$\langle\!\langle r \rangle\!\rangle$	encoding of HLL transition r in array program, page 266
τ	HLL basic type, page 203
$\tau_{\mathscr{E}}$	HLL domain of event types, page 209
$\tau_{\mathscr{E}}$	HLL events type, page 203
term	HLL term, page 204
$\mathit{Term}_{\mathbb{B}}(\mathcal{S})$	boolean terms, page 88
term^{L}	HLL let-expression, page 231
term^{NF}	HLL guard normal form, page 231
$\mathit{Term}_{\tau}(\mathcal{S})$	terms of type τ over signature $\mathcal{S}$, page 88
$\theta^{\sharp}$	assignment in abstract structure, page 133
θ	assignment, page 84
Θ	assignment basis, page 184
$(\theta, G) \sim_{(f,g)} (\theta^{\sharp}, G^{\sharp})$	corresponding assignment wrt. topologies, page 133
$(\theta, \pi) \sim_{H} (\theta^{\sharp}, \pi^{\sharp})$	corresponding assignment, page 140
θ/k	k-step evolution of θ, page 84
$\theta_1 \sqsubseteq \theta_2$	information order on assignments, page 84
this	HLL self link, page 203
$\mathit{this} \rightarrowtail \lambda \mathrel{\texttt{:=}} \texttt{new}(C)$	array program creation, page 264
T_i	basic type identities, page 80
$\boldsymbol{T}_i$	basic type evolution chains, page 80
$u \in D$	individual u is in one of D's subsets, page 148
U	UML model, page 288
$U(G)$	identities of topology G, page 46
$U^{\circledcirc}$	individuals (or alive), page 45
$U^{\circledcirc}(G)$	alives of topology G, page 46
$U^{\circledcirc}(s)$	abbreviation for $U^{\circledcirc}(\mathscr{L}(s))$, page 50

$(u_i, r_{C_i,i})$	element of HLL scheduling sequence, page 216
$u \overset{\pi}{\rightsquigarrow}_e u'$	individual u evolves into u' along path π, page 52
$u \overset{r}{\rightsquigarrow}_e u'$	individual u evolves into u' along transition r, page 51
$u \overset{r}{\rightsquigarrow}_e$ ◌	individual u disappears along transition r, page 51
$\mathrm{u}\xrightarrow{(s,s')}_e$ u’	individual u evolves into u' along transition (s, s'), page 51
$u \xrightarrow{(s,s')}_e$ ◌	individual u disappears along transition (s, s'), page 51
$U^{\emptyset}$	non-alive, page 46
$U^{\emptyset}(G)$	non-alives of topology G, page 46
◌ $\overset{r}{\rightsquigarrow}_e u$	individual u appears along transition r, page 51
◌ $\xrightarrow{(s,s')}_e u$	individual u appears along transition (s, s'), page 51
$\mathcal{V}$	logical variables, page 79
V	set of logical variables, page 84
v, v_1, v_2	vertex, page 26
V	vertices of a graph, page 26
$V(G)$	vertices of graph G, page 26
$\mathcal{V}(\mathcal{S})$	logical variables of signature $\mathcal{S}$, page 80
v, w	logical variable, page 80
$\mathcal{V}^T(\mathcal{S})$	identity variables of signature $\mathcal{S}$, page 80
$\mathcal{V}^{\boldsymbol{T}}(\mathcal{S})$	destiny variables of signature $\mathcal{S}$, page 80
X	UML attributes, page 288
x, y	identity variable, page 80
$X_{A,C}$	auxiliary variables of HLL class C, page 203
X_C	local variables of HLL class C, page 203
$X_{I,C}$	input variables of HLL class C, page 203
$\mathsf{X}\,\psi_1$	path formula next, page 96
$\boldsymbol{x}, \boldsymbol{y}$	destiny variable, page 80

$\mathbb{Z}$	integers, page 23
$\mathbb{Z}^+$	positive integers, page 23
$\mathbb{Z}_0^-$	negative integers including 0, page 23
$\mathbb{Z}^-$	negative integers, page 23
$\mathbb{Z}_0^+$	positive integers including 0, page 23

Index

Lebenslauf

Bernd Westphal, geb. 2.10.1973 in Hamburg.

Beruflicher Werdegang

- seit 1.5.2008 — wiss. Mitarbeiter, Uni Freiburg, Abteilung Softwaretechnik, Prof. Dr. A. Podelski
- 1.5.2001 – 30.4.2008 — wiss. Mitarbeiter, Uni Oldenburg, Abteilung Sicherheitskritische Eingebettete Systeme, Prof. Dr. W. Damm
- Juli 1998 – Dezember 2000 — stud. Hilfskraft, Uni Oldenburg, Prof. Dr. Sonnenschein

Ausbildung

- April 2001 — Abschluss Dipl.-Inform., Gesamtnote 'Ausgezeichnet'

  Vertiefungsgebiet: Theoretische Informatik
  Nebenfach: Mathematik
  Thema der Diplomarbeit:
  "Exploiting Object Symmetry in Verification of UML-Designs"
- 1994–2001 — Studium der Informatik und Mathematik an der Carl von Ossietzky Universität Oldenburg
- 1993–1994 — Zivildienst
- 1987–1993 — Gymnasium Liebfrauenschule Oldenburg
- 1980–1986 — Grundschule und Orientierungsstufe in Oldenburg