CARL VON OSSIETZKY universität OLDENBURG

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

Analysis of Dynamic Evolution Systems by Spotlight Abstraction Refinement

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inform. Tobe Toben

Gutachter:
Prof. Dr. Werner Damm
Prof. Dr. Ernst-Rüdiger Olderog

Tag der Disputation: 10. Februar 2009

✉ toben@informatik.uni-oldenburg.de
http://purl.oclc.org/net/phd/toben09

Für meinen Vater.

Abstract

Dynamic Evolution Systems describe an emerging class of systems that vary dynamically in size and topology. Typical examples include the adhoc networking principle where a routing infrastructure over a changing set of participants is created and maintained. A corresponding routing protocol has to be able to handle an arbitrary number of nodes and must in particular be prepared for the integration of new and the proper dismissal of disappeared devices. Similar structures occur in dynamic traffic management systems like the car platooning scenario where physically adjacent cars establish interlinked groups.

This thesis presents a formal method to check whether a system in the sense above adheres to requirements given in form of temporal scenarios. We observe that the inherent unboundedness of the systems renders this task in general undecidable and we present a sound but necessarily incomplete solution that is based on the abstract-check-refine paradigm. Our approach employs the spotlight abstraction principle to obtain a finite description of both the system and the requirement. The idea of this abstraction technique is to exactly preserve the behaviour of a finite set of spotlight processes and to provide one dedicated abstract process to over-approximate the behaviour of the non-spotlight processes. As the validity of a requirement often cannot be definitely decided in the initial abstraction, we propose an automatic and iterative refinement of the abstraction that is guided by the analysis of abstract counterexamples. The refinement method is based on the observation that the precision of the abstraction can be tuned by two complementary principles, namely by enlarging the size of the spotlight and by refining the behaviour of the non-spotlight part of the abstraction. We demonstrate that the approach can be further improved by integrating auxiliary system invariants, and we devise a static analysis method to obtain such invariants for an existing modelling language.

Our approach is realised in form of a tool implementation. On this basis, we perform a practical evaluation of our approach on a number of case studies that cover a broad range of the addressed class of systems. We are in particular able to automatically establish system properties for which manual intervention was required before.

Zusammenfassung

Dynamic Evolution Systems beschreiben eine neue Klasse von Systemen welche sich dynamisch in ihrer Größe und Verbindungsstruktur verändern. Ein typisches Beispiel hierfür sind ad-hoc Netzwerke in welchen sich eine Routingstruktur über eine wechselnde Anzahl von Teilnehmern dynamisch bildet und anpasst. Ein zugehöriges Routingprotokoll ist dafür verantwortlich eine beliebige Anzahl von Knoten zu integrieren und insbesondere auch verschwundene Teilnehmer sicher aus dem Netz zu entfernen. Ähnliche Strukturen entstehen in Verkehrsmanagementsystemen wie dem Car Platooning in welchem sich benachbarte Autos zu einer Gruppe zusammenschließen.

Diese Arbeit präsentiert eine Methode mit welcher überprüft werden kann ob ein solches System formale Anforderungen in Form von Szenarien erfüllt. Wir zeigen, dass die Unbeschränktheit der Systeme diese Aufgabe im Allgemeinen unentscheidbar macht und präsentieren eine sichere aber notwendigerweise unvollständige Lösung die auf dem Prinzip *abstract-check-refine* basiert. Unser Ansatz benutzt die Technik der "Spotlight Abstraction" um eine endliche Darstellung des Systems und der Anforderung zu erhalten. Die Idee hinter dieser Technik ist eine endliche Anzahl von Spotlichtprozessen präzise darzustellen und alle anderen Prozesse zu einem abstrakten Prozess kollabieren zu lassen. Da eine Eigenschaft oftmals nicht in der initialen Abstraktion bewiesen oder widerlegt werden kann, schlagen wir eine Verfeinerungsmethode basierend auf abstrakten Gegenbeispielen vor. Wir zeigen, dass die Präzision der Abstraktion auf zwei grundsätzliche Arten verbessert werden kann, nämlich zum einen durch Vergrößerung des Spotlights und zum anderen durch eine Verfeinerung des Verhaltens des abstrakten Prozesses. Zudem präsentieren wir eine Methode aus dem Bereich der statischen Modellanalyse mittels welcher das Verfahren weiter verbessert werden kann.

Wir haben eine Implementierung vorgenommen um unseren Ansatz evaluieren zu können. Hierzu betrachten wir Fallstudien und Eigenschaften die einen großen Bereich der gesamten Systemklasse abdecken. Wir sind insbesondere in der Lage Eigenschaften automatisch zu beweisen, bei welchen bisher eine manuelle Unterstützung nötig war.

Acknowledgements

I want to thank my supervisor Prof. Dr. Werner Damm for his constant effort in creating a perfect research environment which offers a broad scope of cooperation and inspiration. In particular the collaborative work within the AVACS (`www.avacs.org`) project provided me with new and interesting aspects regarding my research topic. I'm very glad that Prof. Dr. Ernst-Rüdiger Olderog became the second reviewer of this thesis. My work in his group during my studies fostered my interest in formal methods. In this context I also very much appreciated the inspiring thoughts and lectures of Dr. habil. Henning Dierks.

The topic of this thesis has been developed in close collaboration with my former colleague Bernd Westphal. Bernd is always a source of inspiration in various areas like specification and abstraction techniques, innovative (yet sometimes weird) research ideas, travelling plans and on which text editor to use. From the AVACS family I like to express my gratitude to Jörg Kreiker and Ina Schaefer for openly sharing their ideas and scientific skills. I also thank my colleagues Jan-Hendrik Rakow for fruitful discussions concerning languages, tools and swingsets, Christian Ammann for his reliable implementation work, Ingo Schinz and Christian Mrugalla for thought-provoking conversations in the early era, Jürgen Niehaus for managing every possible and impossible request, and rest of the Mensakarawane, Eckard, Thomas, Pate Ralf, thias, Lars and Eike, for ensuring an enjoyable lunch break.

From my social circle I like to mention the members of DownStairs and the Pilshotline who provide sonorous and convivial times of recreation and who are remaining close friends for many years now. Finally I want to thank my whole family for their endless trust and support. This in particular addresses Manuela and Tammo who keep me focussed on the real important aspects of life.

Table of Contents

List of Figures

Chapter 1

Introduction

"Let there be Light."
AC⚡DC (australian rock band)

Ubiquity is prevalent. Nowadays, we are confronted with computing devices not only at our desktops but rather in almost every situation in our daily life. The ongoing decrease in cost, size and energy consumption allows computing devices to become *mobile*. For example, modern cars are equipped with navigation services, cellular phones have abilities to connect to internet services, and so-called wearables are even directly integrated in our clothes and provide health monitoring or entertainment functionalities. This kind of *ubiquitous computing* [Wei91] is often seen as the "third wave in computing": The beginning of the technology era was dominated by mainframes, where each of them was typically shared by a number of people. The second wave represents the era of personal computing, where each person works on his own machine, typically at one fixed location. And next comes ubiquitous computing, where several mobile devices are associated to one person and where devices dynamically establish and update connections to other devices depending on their internal status and actual physical position.

There is a clear demand for formal analysis techniques for this new class of systems. This demand is on the one hand based on financial aspects, as a proper functionality of a product should be established before the product is actually shipped to the customers. Any errors which are detected after the selling of the product typically generates high costs for recalls. On the other hand, mobile devices may also take over safety-critical functions like manoeuvre controlling in automotive applications [Var93] or the management of movement authorisations in train systems like in the emerging European Train Control System (ETCS) [UNI02]. In these cases, a rigorous proof of the proper functionality of the device is indispensable.

For finite systems working in a fixed infrastructure, formal verification techniques like model-checking [CGP99] or theorem-proving [Sch00] allow for a stringent analysis of the correctness of systems with respect to given requirements. Sophisticated tools have been developed and are already used in industrial development processes, as for example described in [Hol94, Low96, SAH+00, HLP01]. However, these techniques are not immediately applicable to dynamic and mobile systems in the sense described above, first of all as these systems vary dynamically in size and topology. By this, the size of the system is in general not bounded by any finite number. This fact rules out any technique that is purely based on the exhaustive analysis of the induced state space of the system.

1.1 Abstraction and Refinement

Our approach to create a formal verification method for dynamic systems is based on the thesis that the correctness of the overall system can often be established on a small fraction of the system. Likewise, any incorrect behaviour is visible already when considering only a certain part of the system. If this is the case we can *abstract* the large (in general even infinitely large) system to a much smaller (and in particular finite) system. Then, well-established techniques to analyse the resulting abstract system can be applied and the results transfer to the original system. According to [CGJ+00] a good abstraction technique should have two properties, namely *feasibility*, that is, the resulting system can actually be computed and handled by existing analysis techniques, and *preservation*, that is, the behaviour that is relevant for the analysis is still contained in the abstract system. Actually, abstraction techniques are quite heavily used in formal methods, mainly in order to defeat the inherent complexity of the verification procedures themselves [KP00].

> *"Abstraction is the elimination of the irrelevant and the amplification of the essential."* [Mar95]

In this sense our goal is to identify the essential part of the system that is sufficient to either establish or disprove the correctness of the system. There is actually a hope to reach this goal for our class of systems as the overall behaviour stems from *replicated components*, this means, we are dealing with the interaction of several devices that however all follow the same, or at least one of finitely many, behavioural descriptions. Intuitively, it might then be sufficient to consider only a finite number of devices. While any violation of the correctness specification within this finite set of devices immediately yields

a violation of the overall system, the fact that a specification holds for this part the system does not ensure the overall correctness of the system. There might actually be a violation involving more devices than considered. To address this problem of obtaining false-positives, one typically applies an *over-approximation* of the system by generating more behaviour than the original system, such that a satisfaction in the abstract system directly transfers to a satisfaction in the original system. The drawback of this method is that one now may obtain false-negatives as a violation in the abstract system not necessarily corresponds to a violation of the original system. In the model-checking community, these false-negatives are also called spurious counterexamples.

In general, the identification of the essential fragment of the system, which has to be emphasised by any over-approximating abstraction method, is hard. It is hard already for a human user, but even harder if this decision should be made automatically by some machine. Our approach to deal with this problem is the usage of *refinement*. The basic idea is that one may start which a knowingly coarse abstraction and then gradually refine the abstraction until the specification can be established or shown to be wrong. Refinement in this sense corresponds to the elimination of behaviour that is not contained in the original system. The decision which behaviour to eliminate can often be made on the basis of prior verification results, in particular by the analysis of the obtained counterexamples which demonstrate the (abstract) violation of the specification. Any behaviour within the counterexample that can be identified to be spurious can safely be removed from the abstract system. Then, the next verification attempt of the refined abstract system can be initiated. If however a counterexample comprises valid behaviour that it is completely reproducible in the original system, one has identified a real counterexample violating the specification also in the original system. This verification framework is known under the term *counterexample-guided abstraction refinement* (CEGAR), which has already been applied to many abstraction techniques for quite different classes of systems (cf. Sect. 6.2, page 101).

The particular abstraction technique that we employ for the analysis of the considered class of dynamic systems follows the *spotlight principle* [WW07]. This principle generates heterogeneous abstraction techniques in the sense that different parts of the systems are abstracted with different degrees of precision. The basics of this technique were initially developed under the name "Data-Type Reduction" in [McM99], which focussed on the verification of parameterized systems (cf. Sect. 3.5). It advanced to a generic abstraction technique that applies in particular to systems with runtime creation and destruction of objects in [DW05, Wes08]. In the particular instance discussed in this thesis,

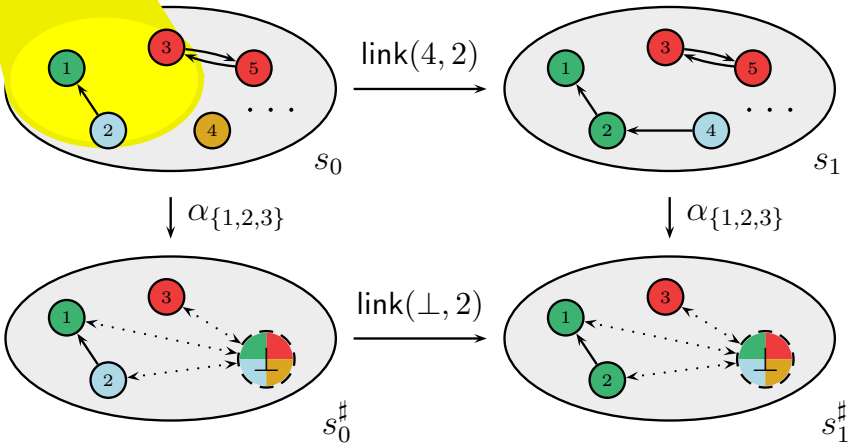


Figure 1.1: **Spotlight Abstraction.** The local states and connection topology among the spotlight processes $1, 2, 3$ is preserved while any information about non-spotlight processes is collapsed into the dedicated abstract process $\perp$ for which any configuration is assumed. Concrete evolutions of the system can be mapped to abstract evolutions which reflect the behaviour of the spotlight processes.

one finite part of the system, the so-called spotlight, is preserved without any abstraction while the rest, the so-called shadows, is coarsely abstracted to one summary processes (see Figure 1.1). The behaviour of this summary process is constructed such that it yields an over-approximation of the behaviour of an arbitrary number of concrete processes. In the sense of [Mar95] the summary processes realises the "elimination of the irrelevant" while the precisely represented spotlight part serves as the "amplification of the essential".

Contributions of this thesis

We observe that the refinement of spotlight abstractions first of all corresponds to the elimination of spurious behaviour of the summary process. In this context, spurious behaviour corresponds to behaviour that is not reproducible by any number of concrete processes. This description already indicates that the identification of spurious behaviour reveals a difficult problem as the analysis of an unbounded number of concrete processes corresponds to a general verification problem. Notably, there is another parameter by which the abstraction can be tuned, namely by the content of the spotlight. A larger spotlight intuitively increases the amount of concrete behaviour and allows us to identify concrete counterexamples directly in the abstract system if they occur within the spotlight.

We exploit this characteristic of the spotlight abstraction technique to devise a validation method for abstract counterexamples. Roughly speaking, this method gradually increases the spotlight until the abstract counterexample can either be validated, which means the spotlight has become large enough to replay the counterexample with concrete processes, or the abstract counterexample has been shown to be spurious. The spuriousness of an abstract counterexample is then used to obtain a refinement of the behaviour of the summary process by integrating temporal assumptions. We have published this new variant of the counterexample guided abstraction refinement framework for spotlight abstraction in [Tob08]. In this thesis, we extend the initial idea by establishing general termination and progress properties of the refinement algorithm and we provide a detailed practical evaluation of the approach via a prototypical tool implementation. Also, we extend the abstraction and refinement loop to preserve local liveness properties by considering strong fairness constraints.

Besides this general strategy of spotlight abstraction refinement, we also describe a complementary refinement approach that is tailored to a specific modelling language for the addressed class of systems, the so-called Dynamic Communication Systems (DCS). We have co-authored the initial publication of this language in [BSTW06], which has spawned a number of research activities [Rak06, Jos06, Bau06, Amm07, Tob07, BTW07a, Wes08, Bac08]. In [Tob07] we show how to exploit the inherent communication dependencies of a DCS variant that employs a synchronous communication paradigm. The method computes refinement information for eliminating *spurious message interferences*, which a major source of spurious behaviour stemming from the spotlight abstraction principle. In this thesis, we extend the method to also handle asynchronous buffered communication and we integrate the approach in the overall refinement loop. This resulting method is practically evaluated and we observe interesting synergy effects with the counterexample-guided refinement approach.

The modelling languages are complemented by a temporal logic to specify requirements of the addressed class of systems. The design of this logic is driven by our research on first-order temporal logic and on graphical specification languages that have been published in [BTW07b, DTW06, WT06, KTWW06]. The logic is tailored to the employed abstraction principle by allowing the user to specify desired or forbidden *scenarios*, that is, the interaction of a finite set of processes over the time.

In summary, we present a concise description of the spotlight abstraction principle applied to a general framework for the modelling and specification of dynamic evolution systems. We in particular provide a formal characterisation of the specific properties of the abstraction principle in terms of a three-valued

specification satisfaction relation. The proposed refinement strategies then turn the rather coarse abstraction procedure into a proper verification and falsification technique. We present formal models and requirement specifications of a number of case studies from the addressed class of systems, on which we demonstrate the suitability of our approach.

1.2 Running Example

One prominent example of the class of dynamic systems described above is characterised by the term *adhoc networking* [FJL00], which describes a general class of self-configuring networks. Participants of a so-called *mobile adhoc network* (MANET) are typically equipped with some sort of radio antenna by which short-ranged wireless connections to other nodes in the neighbourhood can be established. A routing protocol then establishes, maintains and distributes information on how to reach other nodes in the network which are not within the local scanning range. The difficulties for providing reliable routes stems from the fact that mobile device may freely enter and leave the overall network infrastructure. Thus the connection topology may change almost arbitrarily, typically leading to temporarily inconsistent routing information that have to be identified and restored.

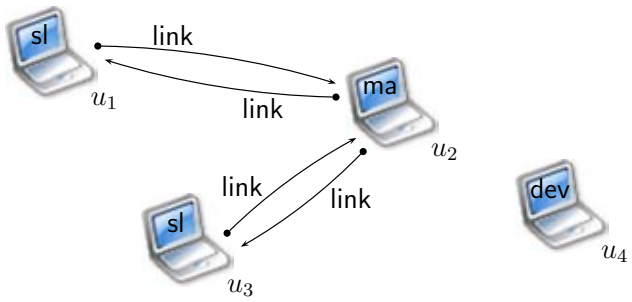


Figure 1.2: An MANET topology involving four participants.

The Bluetooth^{TM} protocol [Haa98] groups a certain number of physically adjacent nodes into so-called piconets where one node becomes the master of this subnet and all other nodes are slaves. Network routes are then established by interlinking the master nodes of different piconets. Figure 1.2 shows a snapshot of a MANET topology following the bluetooth principle. The three nodes u_1, u_2, u_3 form a piconet with u_2 being the master ('ma'), and u_1 and u_3 being slaves ('sl'). To exchange data, the master has established bidirectional links to its slaves. We have a forth device u_4 ('dev') which is yet unconnected. This

snapshot may now evolve into different topologies. For example, the master and the free device may connect such that u_4 becomes another slave in the piconet. Alternatively, another free device may appear and subsequently build a new piconet with u_4. Clearly, connections may also be removed, for example if some slave device leaves the networking area.

We will use this topology scenario as the running example in order to motivate and explain our approach for the specification and verification of dynamic evolution systems. This example is one the one hand small enough in order to concisely illustrate the approach and one the other hand it provides all the challenging features of the addressed class of systems, in particular a dynamically changing and a priori unbounded number of participants in combination with a varying communication topology among them.

1.3 Structure

The structure of this thesis reflects the main problems that have to be addressed in order to create the desired analysis method. We start in Chapter 2 by providing a number of basic notations and formalisms.

As a formal analysis technique can only be established on both a formal description of the model and its requirements, Chapter 3 introduces the formal language by with we will describe the systems to be analysed and Chapter 4 describes the logic to capture correctness specifications. We define the semantical domain where the requirements are evaluated on in Section 3.2 as well as two higher-level modelling languages in Sections 3.3 and 3.4 that translate into this domain.

The remainder of this thesis then addresses the question how to analyse whether a given system model satisfies its requirements. To this end, we provide in Chapter 5 an abstraction mechanism that is suitable for the addressed class of systems. This mechanism comprises the two techniques *spotlight abstraction* and *query reduction* which in combination reduces the analysis of the infinite state system into a finite set of finite state verification tasks. This mechanism is turned into a verification procedure in Chapter 6 by devising an iterative approach for *abstraction refinement*. This approach is separated into a general technique that instantiates the counterexample guided abstraction refinement framework in Section 6.2, which is improved by a complementary procedure that exploits the specifics of the DCS language in Section 6.3.

We discuss related work at the end of each chapter. For the sake of readability most of the formal proofs have been shifted into the appendix.

To evaluate our approach we have conducted a tool implementation which we will demonstrate on a number of case studies in Chapter 7. Chapter 8 concludes our thesis and gives perspectives on how this work might be continued.

Chapter 2

Preliminaries

This chapter introduces basic notations (Sect. 2.1) and formalisms (Sect. 2.2). In particular, we introduce labelled transition systems as a standard representation of system behaviour. Moreover, we give an overview of model-checking and abstraction techniques in Section 2.3, and we introduce a classical three-valued extension of propositional logic in Section 2.4, which allows us to formally capture partially imprecise information.

2.1 Notations

Symbols and Operations By $\mathbb{N}_0 := \{0, 1, 2, \ldots\}$ we denote the natural numbers including 0, and we set $\mathbb{N} := \mathbb{N}_0 \setminus \{0\}$. By $\mathbb{Z} := \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ we denote the set of integers. For two sets X and Y, we write $X \dot{\cup} Y$ to denote the union of X and Y if X and Y are disjoint, that is, if $X \cap Y = \emptyset$. By $2^X := \{X' \mid X' \subseteq X\}$ we denote the set of all subsets of X.

Relations and Functions Given two sets X and Y, a *relation* is a subset of $X \times Y$. The *domain* of a relation $R \subseteq X \times Y$ are those elements of X which are related to some element of Y, written as $\mathsf{dom}(R) := \{x \in X \mid \exists\, y \in Y : (x, y) \in R\}$. Analogously, the *range* of R are those elements of Y to which some element of X is related to, written as $\mathsf{ran}(R) := \{y \in Y \mid \exists\, x \in X : (x, y) \in R\}$.

A relation f is called a *(partial) function*, written as $f : X \rightharpoonup Y$, if each element $x \in X$ has at most one element from $y \in Y$ related to it. For $x \in \mathsf{dom}(f)$, we may then write $f(x)$ to denote this unique element $y \in Y$. We write $f(x) = \mathsf{undef}$ to indicate that $x \notin \mathsf{dom}(f)$.

A function f is called *total*, written as $f : X \to Y$, if $\mathsf{dom}(f) = X$, and *surjective* if $\mathsf{ran}(f) = Y$. The *inverse* of a function $f : X \rightharpoonup Y$ is the relation $f^{-1} := \{(f(x), x) \in Y \times X \mid x \in \mathsf{dom}(f)\}$. A function f is called *injective* if its inverse is a function, and *bijective* if f is both injective and surjective. A function f is called *monotone* if for all $x_1, x_2 \in \mathsf{dom}(X)$ we have that $x_1 < x_2$ implies $f(x_1) \leq f(x_2)$, and *strong monotone* if $x_1 < x_2$ implies $f(x_1) < f(x_2)$.

To declare a function explicitly, we write $f := [x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, yielding a function f with $f(x_1) = y_1, \ldots,$ and $f(x_n) = y_n$.

Restriction, Modification and Union Given a function $f : X \rightharpoonup Y$ and a set $X' \subseteq X$, we use $f|_{X'}$ to denote the *restriction* of f with respect to X', defined as $f|_{X'} := \{(x, f(x)) \in X \times Y \mid x \in \mathsf{dom}(f) \cap X'\}$.

For elements $x' \in X$ and $y' \in Y$, we use $f[x' \mapsto y']$ to denote the *modification* of x' by y' in f, defined as

$$f[x' \mapsto y'](x) := \begin{cases} y' & \text{if } x = x' \\ f(x) & \text{else} \end{cases}$$

where x ranges over X.

For two functions $f_1, f_2 : X \rightharpoonup Y$ with disjoint domains, i.e. $\mathsf{dom}(f_1) \cap \mathsf{dom}(f_2) = \emptyset$, we denote the *union* of f_1 and f_2 by $f_1 \cdot f_2$, defined as

$$(f_1 \cdot f_2)(x) = \begin{cases} f_1(x) & \text{if } x \in \mathsf{dom}(f_1) \\ f_2(x) & \text{if } x \in \mathsf{dom}(f_2) \\ \mathsf{undef} & \text{else} \end{cases}$$

where x ranges over X.

Predicates and Atoms A *predicate* is a symbol p associated with an arity k_p from $\mathbb{N}_0$. We use the notation p/k to denote that p has arity k. Given a set of predicates P and a set of logical variables X, an *atom* over P and X is of the form

$$p(x_1, \ldots, x_{k_p})$$

where $p \in P$ and $x_1, \ldots, x_{k_p} \in X$. The set $\{x_1, \ldots, x_{k_p}\}$ is called the arguments of the atom, and the set of all atoms over X and P is denoted by $Atoms_X(P)$.

Sequences For a set A, we use A^* to denote finite sequences over A, that is, the set of all sequences of the form

$$(a_i)_{0\leq i\leq n} = \ a_0, a_1, \ldots, a_n$$

with $n \in \mathbb{N}_0$ and $a_i \in A$ for all $0 \leq i \leq n$. The case for $n = 0$ yields the empty sequence, denoted by ϵ, and we call $\mathsf{len}((a_i)_{0\leq i\leq n}) := n$ the length of the sequence.

By A^ω we denote the set of all infinite sequences over A of the form

$$(a_i)_{i\in\mathbb{N}_0} = \ a_0, a_1, \ldots$$

with $a_i \in A$ for all $i \in \mathbb{N}_0$, and we set $\mathsf{len}((a_i)_{i\in\mathbb{N}_0}) := \infty$.

By $\mathit{Prefixes}(\sigma) := \{v \in A^* \mid \exists\, w \in A^\omega \,.\, vw = \sigma\}$ we denote the set of all finite prefixes of an infinite sequence $\sigma \in A^\omega$.

2.2 Basic Formalisms

Transition systems are a standard model to formally represent the behaviour of reactive systems [MP92]. A transition system denotes the set of possible *states* together with the set of *transitions* among these states. Additionally, a *label* is associated to each transition.

Definition 2.1 (Labelled Transition System) *A tuple* $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *where*

- $\mathbf{S}$ *is a set of states,*
- $\mathbf{S}_0 \subseteq \mathbf{S}$ *is a set of initial states,*
- $\mathbf{L}$ *is a set of labels, and*
- $\rightarrow \subseteq \mathbf{S} \times \mathbf{L} \times \mathbf{S}$ *is the transition relation,*

is called a labelled transition system *(LTS).* ♢

We call an LTS $(\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *finite* iff both $\mathbf{S}$ and $\rightarrow$ are finite. Note that we usually write a labelled transition $(S, L, S') \in \rightarrow$ in infix form as $S \xrightarrow{L} S'$.

Definition 2.2 (Runs of an LTS) *Let* $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *be an LTS. A* run *of* T *is an infinite sequence*

$$\pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in (\mathbf{L} \times \mathbf{S})^\omega$$

of labels $L_i \in \mathbf{L}$ *and states* $S_i \in \mathbf{S}$*, where*

1. *the sequence starts at an initial state, i.e.* $S_0 \in \mathbf{S}_0$, *and*
2. *consecutive pairs are in transition relation, i.e.* $S_i \xrightarrow{L_{i+1}} S_{i+1}$ *for all* $i \in \mathbb{N}_0$.

The set of all runs of T *is denoted by* $\mathit{Runs}(\mathsf{T})$. ♢

The unusual notation of using pairs of "labels and states" rather than of "states and labels" is motivated by our treatment of prefixes of runs in terms of counterexamples in Section 6.2. As the initial label of a run is unrestricted, it will not be shown in the representation of example runs in the following.

Based on the notion of a run, we define the reachability among two states with and without visiting a certain transition label, and we introduce the classical notion of reachability from an initial state of an LTS.

Definition 2.3 (Reachability) *Let* $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *be an LTS. The state* $S' \in \mathbf{S}$ *is reachable in* T *via label* $L' \in \mathbf{L}$ *from state* $S \in \mathbf{S}$, *written*

$$S \xdashrightarrow{L'}_{\mathsf{T}} S'$$

if there exists a run $\pi = ((L_i, S_i))_{i \in \mathbb{N}_0} \in \mathit{Runs}(\mathsf{T})$ *with*

1. $S_i = S$ *for some* $i \in \mathbb{N}_0$, *and*
2. $(L_j, S_j) = (L', S')$ *for some* $j \in \mathbb{N}_0$ *with* $i \leq j$.

We say that S' *is reachable from* S *in* T, *written* $S \dashrightarrow_{\mathsf{T}} S'$, *if* $S \xdashrightarrow{L'}_{\mathsf{T}} S'$ *for some label* $L' \in \mathbf{L}$. *We say that* S *is reachable in* T, *written* $\dashrightarrow_{\mathsf{T}} S$, *if* $S_0 \dashrightarrow_{\mathsf{T}} S$ *for some* $S_0 \in \mathbf{S}_0$. ♢

The subscript T of the reachability symbol may be omitted if the underlying LTS is clear from the context.

2.3 Model-Checking

The term *Model-Checking* designates a collection of techniques to automatically check whether a system model adheres to a given requirement specification. We will introduce the basic notations and results of model-checking in this section. For the overall thesis however, we regard model-checking merely as a black-box technique that allows us to exhaustively analyse a given finite state transition system with respect to some temporal requirement specification. For an in-depth discussion of related formalisms and algorithms we refer to the monographs [CGP99] and [HR00].

The classical formalism the specify the behaviour of the system to be analysed is the *Kripke Structure* [Kri63, CGP99], which is a finite state transition system where states are annotated by a set of properties that hold in the corresponding system state. Given a finite number of atomic propositions AP (i.e. predicates with arity zero), a Kripke Structure is a tuple $K = (S, s_0, R, L)$ over AP where

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $R \subseteq S \times S$ is a transition relation with $\mathsf{dom}(R) = S$, and
- $L : S \to 2^{AP}$ labels each state with a set of atomic propositions.

The Kripke Structure can be seen as a compact representation of the systems behaviour over infinite execution time. The semantics is given by unwinding the Kripke Structure to an infinite computation tree with root node s_0, which is build by adding child nodes while recursively traversing the transition relation.

Requirements are typically given in some variant of *Temporal Logic* [Pnu77] which allows a declarative specification of the system behaviour over qualitative time. Typical requirements may be divided into two categories, namely *safety* and *liveness* requirements. While requirements of the first categories states that the system will never exhibit any behaviour that is considered to be bad, a liveness specification requires to eventually observe some behaviour that is considered to be good.

The main variation point of the different temporal logics is their underlying model of time, namely whether they consider branching time or linear time. Prominent examples of these different paradigms are the Computational Tree Logic (CTL) [CES86] and Linear Time Logic (LTL) [MP92], respectively. While a time point in linear time always has one unique successor, there may be multiple futures in the branching time paradigm. Consequently, linear time logics are interpreted over individual paths of the computation tree, while branching time reasons about the computation tree itself. Its is known that CTL and LTL are expressively incomparable [CD89], and the question which timing paradigm is better suited for specification and verification of reactive systems is subject of a long and lively debate in the literature (see e.g. [Lam80, EH86, EL87, Var01]). While LTL is considered to be more intuitive, CTL specifications are computationally easier to handle. The time complexity of CTL model-checking is linear in both the size of the model and the size of specification, while LTL model-checking is exponential in the size of the formula, which is basically determined by the number of temporal operators. This difference in the worst-case complexities is however regarded to be of minor relevance in practice [Var02], as typical specifications belong to the common subset of LTL and CTL [Mai00].

In this thesis (cf. Chapter 4) we focus on LTL, defined as

$$\phi ::= \mathit{true} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\,\phi \mid \phi_1 \,\mathsf{U}\, \phi_2$$

where $p \in AP$ is some atomic proposition. A Kripke Structure K is said to satisfy an LTL specification ϕ, written

$$K \models \phi,$$

if ϕ holds in all computation paths of K that starts in the initial state. The intuition of the temporal operators is the following. Given a computation path $\pi = s_0, s_1, s_2, \ldots$, the formula '$\mathsf{X}\,\phi$' holds in some state s_i if ϕ holds in the next state s_{i+1}. The path π satisfies '$\phi_1 \,\mathsf{U}\, \phi_2$' if ϕ_1 holds until ϕ_2 eventually holds, that is, ϕ_1 holds in all states s_0 up to s_{j-1} for some $j \in \mathbb{N}_0$ and ϕ_2 holds in s_j. In particular, one may specify that some property *finally* holds by writing '$\mathsf{F}\,\phi$' as an abbreviation for '$\mathit{true}\,\mathsf{U}\,\phi$, and that some property *globally* holds by writing '$\mathsf{G}\,\phi$' as an abbreviation for '$\neg(\mathsf{F}\,\neg\phi)$'.

Given a Kripke Structure K and an LTL specification ϕ, the model-checking problem is to compute the satisfaction relation $K \models \phi$. An important result is that the model-checking problem is decidable for (finite state) Kripke Structure, that is, there exists an effective procedure to check whether $K \models \phi$ holds or not. A major benefit of model-checker implementations is their ability to present a counterexample if a specification does not hold. This counterexample represents a certain part of the computation tree (or a single computation run in the case of linear time) that violates the property. Using this information, the designer of the system may be able to identify errors in the system (or in the specification).

In practice, model-checking often suffers from the so-called state explosion problem as the Kripke Structure representations of real-world systems typically become rather large in terms of states. To attack this problem, various techniques including compact state representations [BCM+90], search space reductions [God90], and abstraction strategies [CGL94, KP00], have been developed.

As *abstraction* is one of the main topics of this thesis, we briefly recall the basics of abstraction based on the notion of *simulation* in the context of model-checking. For two Kripke Structures $K = (S, s_0, R, L)$ and $K' = (S', s'_0, R', L')$, the simulation preorder [Mil71] on $S \times S'$ requires that whenever a pair of states (s, s') are in simulation relation, then s and s' are labelled by the same set of atomic propositions, and for every successor s_2 of s there is a successor s'_2 of s' such that s_2 and s'_2 are in simulation relation. Intuitively, every transition in K can be matched by some transition in K'. If there exists a simulation preorder that in particular relates the initial states s_0 and s'_0, then K' is said to simulate K. If this is the case, then $K' \models \phi$ implies $K \models \phi$ for any LTL specification

ϕ by the property preservation theorem. Intuitively, K' over-approximates the behaviour of K as each run of K is reflected by a corresponding run in K'. However, as K' may comprise additional runs the converse of the above implication does not hold, this means a specification may be violated in K' although it is actually satisfied in K. Counterexamples for such kind of violations are called *spurious*. Note that we will introduce a notion of simulation for a particular kind of labelled transition systems in Section 3.2.2, and established a corresponding preservation theorem for our specification language in Section 4.2.

In order to reduce the occurrences of spurious counterexamples one aims at *refining* the abstraction. One particular approach to automatically obtain a gradual refinement is the framework of *counterexample-guided abstraction refinement* [Kur94, CGJ+00]. There, the analysis of a spurious counterexample provides information on how to refine the abstraction. We will elaborate on this technique in Section 6.2.

2.4 Three-Valued Logic

We will use the symbols '1' and '0' to represent the boolean values *true* and *false*, respectively. Three-valued logic now comprises the additionally symbol '$1/2$' that denotes the *unknown* value, that is, neither true nor false. Thus, we set the domain of three-valued logic as $\mathbb{B}_3 := \{0, 1, 1/2\}$, where the values 1 and 0 represent definite information while $1/2$ stands for indefinite information. This is reflected by the *information order* $\sqsubset \subseteq \mathbb{B}_3 \times \mathbb{B}_3$ defined as

$$\sqsubset := \{(0, 1/2), (1, 1/2)\}$$

which reads as "$1/2$ represents coarser information than both 0 and 1". Additionally, we assume the standard arithmetic order $< \subseteq \mathbb{B}_3 \times \mathbb{B}_3$ defined as $<:= \{(0, 1/2), (0, 1), (1/2, 1)\}$, and by $\min B$ ($\max B$) we denote the least (greatest) element of $B \subseteq \mathbb{B}_3$ with respect to the arithmetic order $<$. We write

$$\begin{aligned} b_1 \leq b_2 \quad &\text{if} \quad b_1 < b_2 \text{ or } b_1 = b_2 \\ b_1 \sqsubseteq b_2 \quad &\text{if} \quad b_1 \sqsubset b_2 \text{ or } b_1 = b_2 \end{aligned}$$

for boolean values $b_1, b_2 \in \mathbb{B}_3$.

To transfer the boolean connectives to the domain of three-valued logic, we follow the approach of Kleene [Kle52] which naturally preserves the indefinite character of expressions whenever the unknown value is involved. Regarding the negation operator, we have $\neg 1 = 0$ and $\neg 0 = 1$ just as in standard two-valued boolean logic, and we set $\neg 1/2 = 1/2$, that is, the negation of indefinite

b	$\neg b$
1	0
$1/2$	$1/2$
0	1

$\vee$	1	$1/2$	0
1	1	1	1
$1/2$	1	$1/2$	$1/2$
0	1	$1/2$	0

$\wedge$	1	$1/2$	0
1	1	$1/2$	0
$1/2$	$1/2$	$1/2$	0
0	0	0	0

Table 2.1: Interpretation of the boolean connectives in $\mathbb{B}_3$.

values remains indefinite. For disjunction, we have $1 \vee 1/2 = 1$ as, regardless of what definite value $1/2$ may actually stand for, the expression yields 1 also in standard boolean logic. For conjunction however, we have $1 \wedge 1/2 = 1/2$ as the result of conjunction of 1 with some value b in standard boolean logic differs if $b = 1$ or $b = 0$. As $1/2$ represents indefinite information, the conjunction remains indefinite, too. Note that $b_1 \wedge b_2 = \min\{b_1, b_2\}$ and $b_1 \vee b_2 = \max\{b_1, b_2\}$.

The semantics of negation, disjunction and conjunction in three-valued logic is given in the Table 2.1, and the following remarks states a number of standard relations between the information order and the boolean connectives.

Remark 2.4 (Three-Valued Logic) *Let $a, b, c, d \in \mathbb{B}_3$ be three-valued boolean values. Then the following conditions hold:*

$$\begin{aligned} b \in \{0,1\} \wedge (a \sqsubseteq b) &\implies a = b \\ (a \sqsubseteq b) &\implies \neg a \sqsubseteq \neg b \\ (a \sqsubseteq b) \wedge (c \sqsubseteq d) &\implies (a \wedge c) \sqsubseteq (b \wedge d) \\ (a \sqsubseteq b) \wedge (c \sqsubseteq d) &\implies (a \vee c) \sqsubseteq (b \vee d) \end{aligned}$$ ♢

Proof. By the definition of the information order $\sqsubseteq$ and the truth-tables for negation, disjunction and conjunction (cf. Table 2.1). ■

In the course of this thesis we will make use of three-valued logic in several formalisations. First of all, it allows us to formalise the loss of information that occurs during the abstraction of concrete, potentially unbounded structures to finite abstract structures (cf. Def. 5.5 on page 74). Any information that is not precisely represented in the abstract structure will become indefinite. Moreover, we will define a three-valued evaluation of a temporal specification in an abstracted system (cf. Def. 4.2 on page 57), with the intuition in mind that a definite result directly transfers to the evaluation result of the specification in the original system. Consequently, an indefinite evaluation result will trigger the refinement of the abstraction.

Chapter 3

Modelling Dynamic Evolution Systems

In this chapter we provide means to formally describe the systems to be analysed. We base our semantical domain (cf. Section 3.2) on the standard notion of a *logical structure*, which basically interprets a number of predicates for a given universe. To represent the appearance and disappearance of processes over time, a subset of the universe is dedicated to represent the set of currently *alive* processes. Both the content of this subset and the interpretation of the predicates may vary when one logical structure evolves into another logical structure.

We present two modelling languages for the developed semantical domain. Firstly, a Dynamic Evolution System (DES, cf. Section 3.3) provides a symbolic characterisation of the possible evolutions of logical structures in terms of so-called *evolution rules*. This language serves as a concise low-level modelling formalism throughout this thesis. Secondly, the language of Dynamic Communication Systems (DCS, cf. Section 3.4), which has been developed in a joint

paper [BSTW06], allows to specify the communication aspects of the system from the viewpoint of a single process. In particular, this language alleviates the gap between real-world systems and various low-level specification formalisms like DES and others (see Section 3.5 on related modelling formalisms). The semantics of a Dynamic Communication System will be defined by a translation from DCS into the DES language. By this, the analysis techniques developed for Dynamic Evolution Systems (cf. Chapters 5 and 6) are immediately lifted to the DCS language.

3.1 Intuition

A process in our systems is uniquely identified by its process identity. A central aspect of the considered class of systems is that processes may be created and destroyed at runtime, that is, the set of alive processes may vary over the time. Note that, however, during the lifetime of a process its identity will not change. We assume a set of process identities as follows, where we already introduce a dedicated process identity that will serve as a representation of the *abstract* summary process (cf. Chapter 5).

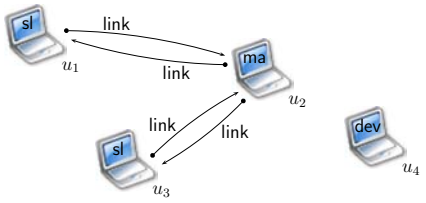
Definition 3.1 (Process Identities) *The countably infinite set Id is called the set of* process identities*. The symbol $\perp \notin Id$ denotes the* abstract process identity*, and for a set of identities $I \subseteq Id$ we define $I^{\perp} := I \cup \{\perp\}$.* $\diamondsuit$

To formally represent a single *snapshot* of a system, for example the adhoc networking topology from Figure 1.2, we actually need to preserve three kinds of information, namely

1. the set of alive processes,
2. the local status of each process, and
3. the connection topology among the processes.

To stress the importance of a process being alive or dead, we maintain a dedicated subset of process identities that are currently alive. The local status of each process (not necessarily only the alive ones) is given by an interpretation of unary predicates, and the connection topology by an interpretation of binary predicates. This encoding separates the different aspects of a system snapshot.

In Figure 3.1 we present the encoding of the snapshot of the adhoc networking system given in Figure 1.2. We have $\{u_1, u_2, u_3, u_4\} \subset Id$ as the set of alive identities. The interpretation of the corresponding predicates for these identities is given by the tables in Figure 3.1(b). The interpretation of the unary predicate



(a) Snapshot (cf. Fig. 1.2)

	sl	ma	dev
u_1	1	0	0
u_2	0	1	0
u_3	1	0	0
u_4	0	0	1

link	u_1	u_2	u_3	u_4
u_1	0	1	0	0
u_2	1	0	1	0
u_3	0	1	0	0
u_4	0	0	0	0

(b) Predicate Interpretation

Figure 3.1: A system snapshot and its representation as logical structure.

sl yields true for identities u_1 and u_3, the predicate ma yields true for identity u_2, and dev yields true for identity u_4. All other combinations evaluate to false. Analogously, the connection topology is given by the interpretation of a binary predicate link which yields *true* for the arguments (u_1, u_2), (u_2, u_1), (u_3, u_2), and (u_2, u_3), and *false* for all other pairs of process identities.

By a *configuration* of a process, we will denote the information that is local to this process, that is, firstly whether it is currently alive, secondly the local status of this process, and thirdly the set of connections with other processes. Formally, we will obtain the configuration of a set of processes by the focus operator given in Def. 3.7 below.

The behaviour of the system over time will be captured by *evolutions*, which are transitions from one logical structure into another logical structure. These evolutions comprise the creation and destruction of processes, as well as modifications of the interpretation functions. Evolutions will be given a *name*, indicating what kind of transition the involved processes perform in this evolution. Note that each evolution will only affect a certain subset of processes and we assume an interleaving semantics. Formally, we use a set of evolution predicates that serve, together with the set of affected process identities, as transition labels. In fact, labelled transition systems having logical structures as states and ground atoms over evolution predicates as transition labels form our basic semantic domain, namely structured labelled transition systems (cf. Def. 3.15).

Our semantical domain allows us in particular to observe two artifacts that are inherent to systems with runtime creation and destruction of processes, namely *dangling links* and the *re-use* of identities. The two evolutions shown in Figure 3.2 demonstrate these aspects. Starting from a snapshot with two alive processes being connected, we evolve into a snapshot where process u_2 has disappeared from the system, indicated by its dotted border. However, process u_1 still recognised the link to u_2 such that the connection is actually "dangling". The next evolution now comprises the creation of a new process which

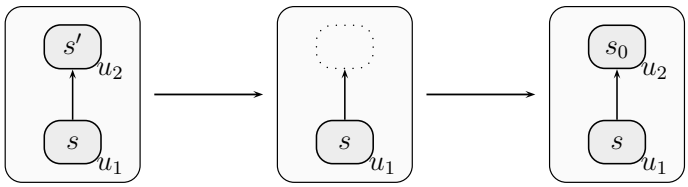


Figure 3.2: Dangling links and re-use of identities.

obtains the free identity u_2, that is, this identity is actually "re-used" by a new, potentially different process. Clearly, these situations may lead to system inconsistencies if process u_1 does not account for these evolutions of his connection partner. Note that our requirement specification language (cf. Chapter 4) and the corresponding analysis methods (cf. Chapter 5 and 6) are able to formalise and detect these artifacts.

3.2 Semantical Domain

In this section we will formally develop the semantical domain that we have sketched in the previous section. In particular, we define logical structures in Subsection 3.2.1 as a means to formally capture snapshots, and we introduce structured labelled transition systems in Subsection 3.2.2 as our basic formalism to represent the dynamic evolution of the considered class of systems.

3.2.1 Logical Structures

As sketched above, logical structures are suitable to represent system snapshots via an interpretation of predicates. Actually, we already motivated the need for different kinds of predicates, namely predicates to describe the local status, their interconnection, and the evolution of processes. Together with a set of logical variables, these sets of predicates form a *signature* as follows.

Definition 3.2 (Signature) *A tuple $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$, where*

- *$\mathcal{X}$ is a totally ordered set of variables,*
- *$\mathcal{P}_S$ is a finite set of unary predicates, called the set of* state predicates*,*
- *$\mathcal{P}_L$ is a finite set of binary predicates, called the set of* link predicates*, and*
- *$\mathcal{P}_E$ is a finite set of predicates, called the set of* evolution predicates*,*

is called a signature*. Without loss of generality, we require $\mathcal{P}_S$, $\mathcal{P}_L$ and $\mathcal{P}_E$ to be pairwise disjoint. We set $\mathcal{P}_{SL} := \mathcal{P}_S \cup \mathcal{P}_L$ and $\mathcal{P} := \mathcal{P}_S \cup \mathcal{P}_L \cup \mathcal{P}_E$.* ♢

In Section 2.1, we have introduced the notion of an *atom*, which is a predicate symbol $p \in \mathcal{P}$ taking a number of logical variables as argument, written as $p(x_1, \ldots, x_{k_p})$ where k_p denotes the arity of p. In our setting, logical variables will be used to *denote* process identities via a *valuation* function, that is, a mapping from logical variables to process identities. Clearly, such a function need not be injective such that two different variables may actually denote the same identity.

Definition 3.3 (Valuation) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $X \subseteq \mathcal{X}$ a set of variables, and $I \subseteq Id^{\perp}$ a set of identities.*

A valuation *of X to I is a total function $\mathcal{V} : X \to I$. The set of all valuations of X to I is denoted by $Vals_I(X)$.* ♢

Given an atom and a valuation of the arguments of the atom, we can instantiate the atom by replacing all variables by the identities they denote, yielding a so-called *ground atom*. Ground atoms will be used as a means to represent predicate interpretations (cf. page 22) and will play a central role when defining the evolution of logical structures (cf. Def. 3.15) and the concretisation of counterexamples (cf. Def. 6.8).

Definition 3.4 (Ground Atom) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $P \subseteq \mathcal{P}$ a set of predicates and $I \subseteq Id^{\perp}$ a set of identities.*

A ground atom over P and I *is of the form $p(u_1, \ldots, u_{k_p})$ where $p \in P$ and $u_1, \ldots, u_{k_p} \in I$. The set of all ground atoms over P and I is denoted by $GroundAtoms_I(P)$. The set of arguments of a ground atom $g = p(u_1, \ldots, u_{k_p})$ is defined as $A(g) := \{u_1, \ldots, u_{k_p}\}$.*

For an atom $a = p(x_1, \ldots, x_{k_p}) \in Atoms_X(P)$ and a valuation $\mathcal{V} \in Vals_I(X)$, we set

$$a[\mathcal{V}] := p(\mathcal{V}(x_1), \ldots, \mathcal{V}(x_{k_p})) \in GroundAtoms_I(P)$$

to be the ground atom of a under $\mathcal{V}$, where each variable in the arguments of the atom a is replaced by the identity it denotes by the valuation $\mathcal{V}$. ♢

The central ingredient of a logical structure is the *interpretation* of the state and link predicates. Formally, an interpretation is a set of functions from product-sets of process identities to three-valued boolean values as follows.

Definition 3.5 (Interpretation) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $P \subseteq \mathcal{P}$ a set of predicates and $I \subseteq Id^{\perp}$ a set of identities.*

An interpretation ι *of P for I defines for each $p \in P$ a total function*

$$\iota(p) : (I)^{k_p} \to \mathbb{B}_3.$$

The set of all interpretations of P for I is denoted by $Inter_I(P)$. An interpretation $\iota \in Inter_I(P)$ is called two-valued *if* $\mathsf{ran}(\iota(p)) \subseteq \mathbb{B}$ *for each $p \in P$, and* three-valued *else.*

By $\iota_e \in Inter_I(P)$ we denote the initial *interpretation which yields* 0 *for all predicates $p \in P$ and arguments I^{k_p}.* ♢

We observe that any interpretation $\iota \in Inter_I(P)$ can be precisely represented by two (possibly infinite) sets of ground atoms, namely as $(\iota_{\mathbf{1}}, \iota_{1/\mathbf{2}})$ where

$$\iota_{\mathbf{1}} := \{p(u_1, \ldots, u_{k_p}) \in GroundAtoms_I(P) \mid \iota(p)(u_1, \ldots, u_{k_p}) = 1\}$$
$$\iota_{1/\mathbf{2}} := \{p(u_1, \ldots, u_{k_p}) \in GroundAtoms_I(P) \mid \iota(p)(u_1, \ldots, u_{k_p}) = 1/2\}.$$

This encoding simplifies the textual representation of an interpretation, in particular in the case when I and P are finite.

We are now prepared to define *logical structures* over process identities, comprising a set of alive identities and a predicate interpretation as follows.

Definition 3.6 (Logical Structure) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature and $I \subseteq Id^\perp$ a set of identities. A tuple*

$$S = (U, \iota)$$

where $U \subseteq I$ is the alive universe of S and $\iota \in Inter_I(\mathcal{P}_{SL})$ is an interpretation of $\mathcal{P}_{SL}$ for I, is called a logical structure over $\mathcal{S}$ and I. *The set I is called the* domain *of S, written $\mathsf{dom}(S)$, and the set of all structures over $\mathcal{S}$ and I is denoted by $Strucs_{\mathcal{S}}(I)$.*

A structure (U, ι) is called two-valued *if ι is two-valued, and* three-valued *else. The set of two-valued structures over $\mathcal{S}$ and I is denoted by $Strucs^2_{\mathcal{S}}(I) \subseteq Strucs_{\mathcal{S}}(I)$, and we set $Strucs^3_{\mathcal{S}}(I) := Strucs_{\mathcal{S}}(I) \setminus Strucs^2_{\mathcal{S}}(I)$.*

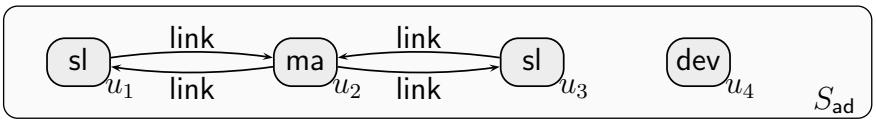
By $S_e = (\emptyset, \iota_e) \in Strucs_{\mathcal{S}}(I)$ we denote the empty *structure with empty universe and initial interpretation.* ♢

Note that a logical structure involves three sets of identities, namely firstly $Id^\perp$, the set of all possible identities, secondly a subset $I \subseteq Id^\perp$ as the domain over which the interpretation ranges, and thirdly $U \subseteq I$ denoting the set of currently alive processes in S. As U may be a proper subset of I, we in particular may represent dangling links, that is, a connection of some alive process to some dead process. For example, we allow (U, ι) to have $\iota(p_l)(u_1, u_2) = 1$ although $u_2 \notin U$ for some binary predicate $p_l \in \mathcal{P}_L$.

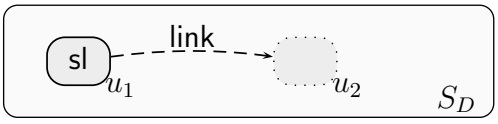
Let us reconsider the adhoc networking example. To faithfully represent local configurations of this case study, we define its signature as

$$\begin{aligned} \mathcal{S}_{\mathsf{ad}} &= (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E) \text{ with} \\ \text{state predicates } \mathcal{P}_S &:= \{\mathsf{dev}, \mathsf{ma}, \mathsf{sl}\}, \\ \text{link predicates } \mathcal{P}_L &:= \{\mathsf{link}\}, \text{ and} \\ \text{evolution predicates } \mathcal{P}_E &:= \{\mathsf{new}, \mathsf{con}, \mathsf{dis}, \mathsf{free}, \mathsf{del}\}. \end{aligned}$$

With this, the snapshot depicted in Figure 1.2 on page 6 corresponds to the two-valued logical structure $S_{\mathsf{ad}} = (U, (\iota_{\mathbf{1}}, \iota_{1/2})) \in \mathit{Strucs}_{\mathcal{S}_{\mathsf{ad}}}(\mathit{Id})$ with $\iota_{\mathbf{1}} = \{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{sl}(u_3), \mathsf{dev}(u_4), \mathsf{link}(u_1, u_2), \mathsf{link}(u_2, u_1), \mathsf{link}(u_3, u_2), \mathsf{link}(u_2, u_3)\}$, $\iota_{1/2} = \emptyset$ and $U = \{u_1, u_2, u_3, u_4\}$. Note that the evolution predicates are not used for the characterisation of a single snapshot, but only for the evolution between two snapshots. Often a logical structure becomes more readable if it is shown in a graphical notation. For example, the structure S_{ad} may be illustrated as



where each element $u \in U$ is represented as a node having its name u as index. The set of state predicates $\iota_{\mathbf{1}}$ is written inside of the node where we omit brackets if exactly one predicate is present, and each link predicate $p_l \in \mathcal{P}_L$ with $p_l(u_1, u_2) \in \iota_{\mathbf{1}}$ is illustrated as a solid directed arc from node u_1 to node u_2 that is labelled by p_l. For representing the indefinite value $1/2$ for ground atoms in $\iota_{1/2}$, we use *dashed* nodes and arcs. Moreover, if we need to depict a dead identity $u' \notin U$ in order to illustrate for example a dangling link, we draw the corresponding node for u' with a *dotted* border. For example, we represent the logical structure $S_D := (\{u_1\}, (\{\mathsf{sl}(u_1)\}, \{\mathsf{link}(u_1, u_2)\}))$ as



Note that this graphical illustration is not suitable for arbitrary logical structures. For instance, structures with infinite domain or dead identities with an indefinite predicate evaluation may not be faithfully represented. In these cases, we will switch to the formal representation in terms of the sets $(\iota_{\mathbf{1}}, \iota_{1/2})$.

Focus

To obtain the configuration of a certain subset of processes, we introduce the *focussing* of a logical structure. This operation can be seen as an restricted view on a subset of identities neglecting the information about the rest, however by keeping those predicate interpretations where both focused and unfocused processes are involved. Hence, a focus provides a concise view on the configuration of the focused processes *and* their relationship to unfocused ones while dismissing information which only concerns unfocused processes.

Definition 3.7 (Focus of a Structure) *Let $\mathcal{S}$ be a signature, $I \subseteq Id^{\perp}$ a set of identities, and $S = (U, \iota) \in Strucs_{\mathcal{S}}(I)$ a logical structure.*

For a set of identities $I' \subseteq I$, we define the I'-focus on S as

$$S\!\!\downarrow_{I'} := (U, \iota\!\!\downarrow_{I'}) \in Strucs_{\mathcal{S}}(I)$$

where

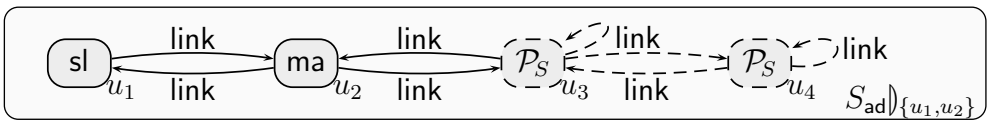
$$\iota\!\!\downarrow_{I'}(p)(u_1, \ldots, u_{k_p}) = \begin{cases} \iota(p)(u_1, \ldots, u_{k_p}) & \text{if } \{u_1, \ldots, u_{k_p}\} \cap I' \neq \emptyset \\ 1/2 & \text{else} \end{cases}$$

for $p \in \mathcal{P}_{SL}$ and $u_1, \ldots, u_{k_p} \in I$. ♢

For example, the $\{u_1, u_2\}$-focus on S_{ad} yields the structure

$$(U, (\{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1, u_2), \mathsf{link}(u_2, u_1), \mathsf{link}(u_3, u_2), \mathsf{link}(u_2, u_3)\}, \\ \{p_s(u), \mathsf{link}(u, u') \mid p_s \in \mathcal{P}_S, u, u' \in \{u_3, u_4\}\}))$$

with alive identities $U = \{u_1, u_2, u_3, u_4\}$. This focused snapshot is graphically represented as



and shows a proper master-slave connection among u_1 and u_2 and the fact that there is another bidirectional connection to some other process u_3. However, the information from S_{ad} that u_3 is actually an alive slave and that there exists another unconnected process u_4 being a free device is dismissed.

Embedding

Party anticipating the abstraction of dynamic evolution systems to be discussed in Chapter 5, we investigate the relation of two logical structures in terms of an *information ordering*. To this end, we formalise the intuition of one logical structure representing only a certain subset of the information which another logical structure represents. Such a relation will be called *embedding*, and the basic ingredient is a mapping between the domains of the two logical structures via an embedding function as follows.

Definition 3.8 (Embedding Function) *Let $I_1, I_2 \subseteq Id$ be two sets of identities with $I_2 \subseteq I_1$. The function $e^{I_1}_{I_2} : I_1 \to I_2^\perp$ with*

$$e^{I_1}_{I_2}(u) = \begin{cases} u & \text{if } u \in I_2 \\ \perp & \text{else} \end{cases}$$

is called the embedding function from I_1 to I_2.

For a valuation $\mathcal{V} \in Vals_{I_1}(\mathcal{X})$, we define the embedded valuation $e^{I_1}_{I_2}(\mathcal{V}) \in Vals_{I_2^\perp}(\mathcal{X})$ as $e^{I_1}_{I_2}(\mathcal{V})(x) := e^{I_1}_{I_2}(\mathcal{V}(x))$ where x ranges over $\mathcal{X}$. ♢

Each process identity in $I_1 \setminus I_2$ is mapped by the embedding function to the abstract process identity $\perp$. Restricted to I_2, the embedding function is actually the identity function.

Based on this function, an embedding of ground atoms is simply the pointwise application of the embedding function to the arguments of the ground atom as follows.

Definition 3.9 (Ground Atom Embedding) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $I_1, I_2 \subseteq Id^\perp$ with $I_2 \subseteq I_1$ two sets of identities, and*

$$g_1 = p(u^1_1, \ldots, u^1_{k_p}) \in GroundAtoms_{I_1}(\mathcal{P})$$
$$g_2 = p(u^2_1, \ldots, u^2_{k_p}) \in GroundAtoms_{I_2^\perp}(\mathcal{P})$$

be two ground atoms. We say that g_2 is embedded *in g_1, written $g_1 \Subset g_2$, if*

$$u^2_i = e^{I_1}_{I_2}(u^1_i)$$

for all $1 \leq i \leq k_p$. ♢

The embedding of logical structures is a bit more involved, as it requires to not only relate the domains and the universes, but also the interpretation functions. Here, the information order of the three-valued logic (cf. page 15) comes into play as it allows us to characterises a potential loss of definite information to indefinite information. We define the embedding of logical structures as follows.

Definition 3.10 (Structure Embedding) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $I_1, I_2 \subseteq Id$ with $I_2 \subseteq I_1$ two sets of identities, and $S_1 = (U_1, \iota_1) \in Strucs_{\mathcal{S}}(I_1)$ and $S_2 = (U_2, \iota_2) \in Strucs_{\mathcal{S}}(I_2^{\perp})$ two logical structures.*

We say that S_2 is embedded *in S_1, written $S_1 \Subset S_2$, if*

1. *S_2 preserves the aliveness of processes, that is,*

$$u \in U_1 \iff e_{I_2}^{I_1}(u) \in U_2$$

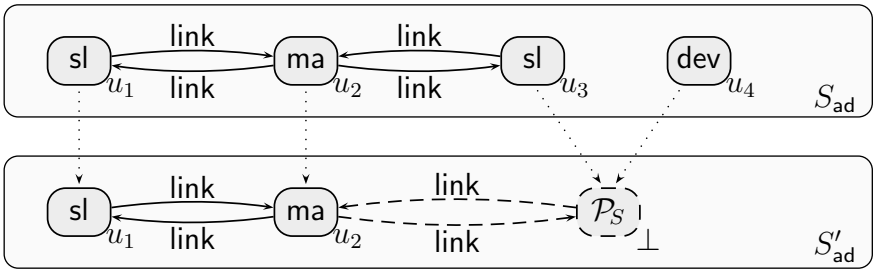
for all identities $u \in I_2$, and

2. *S_2 over-approximates the interpretation of predicates, that is,*

$$\iota_1(p)(u_1, \ldots, u_{k_p}) \sqsubseteq \iota_2(p)(e_{I_2}^{I_1}(u_1), \ldots, e_{I_2}^{I_1}(u_{k_p}))$$

for all predicates $p \in \mathcal{P}$ and identities $u_1, \ldots, u_{k_p} \in I_1$. ♢

Intuitively, if $S_1 \Subset S_2$ we have that S_2 represents less information and is more abstract than S_1. We consider the following example embedding of S_{ad}:



For the structures S_{ad} and $S'_{\mathsf{ad}} = (\{u_1, u_2, \perp\}, \iota')$, where ι' is given in the above box, we provide the embedding function from $\{u_1, u_2, u_3, u_4\}$ to $\{u_1, u_2, \perp\}$ by the dotted arcs. We have $S_{\mathsf{ad}} \Subset S'_{\mathsf{ad}}$ as S'_{ad} preserves the aliveness of processes u_1 and u_2 and the predicate interpretation ι is over-approximated by ι', as for example

$$\begin{aligned}
\iota(\mathsf{sl})(u_1) = 1 &\sqsubseteq 1 = \iota'(\mathsf{sl})(u_1) \\
\iota(\mathsf{ma})(u_1) = 0 &\sqsubseteq 0 = \iota'(\mathsf{ma})(u_1) \\
\iota(\mathsf{sl})(u_3) = 1 &\sqsubseteq 1/2 = \iota'(\mathsf{sl})(\perp) \\
\iota(\mathsf{sl})(u_4) = 0 &\sqsubseteq 1/2 = \iota'(\mathsf{sl})(\perp) \\
\iota(\mathsf{link})(u_1, u_4) = 0 &\sqsubseteq 0 = \iota'(\mathsf{link})(u_1, \perp) \\
\iota(\mathsf{link})(u_2, u_3) = 1 &\sqsubseteq 1/2 = \iota'(\mathsf{link})(u_2, \perp) \\
\iota(\mathsf{link})(u_2, u_4) = 0 &\sqsubseteq 1/2 = \iota'(\mathsf{link})(u_2, \perp)
\end{aligned}$$

In Chapter 5 we will introduce the spotlight abstraction of a logical structure as a systematic method to obtain an embedded structure. This provides us with a proper abstraction technique as an important observation is that the embedding of a logical structure weakly preserves its properties. For example, if some property holds in a logical structure, then this property can not be violated in a corresponding embedding structure. However, the satisfaction of the property may become unknown as the embedding structure may represent less precise information. To formalise this intuition, we firstly discuss how to specify properties of logical structures, and then show the preservation of properties by embedded structures in Lemma 3.14 below.

Formulas

Properties of a logical structure will be formally characterised by *formulas*. To this end, a formula may first of all reason about the *state* and *link* predicates, that is, it may comprise atoms over $\mathcal{P}_{SL}$. To denote that a process x has a connection to *some* (anonymous) process, we take the liberty of writing $p_l(x)$, disregarding the fact that $p_l \in \mathcal{P}_L$ is a actually a *binary* predicate. We also syntactically allow atoms over *evolution* predicates. However, a semantics for these terms can only be given for transitions systems over logical structures. We will do so in Def. 4.2 in Chapter 4. To express that a process is currently *alive* we use the notation $\circledcirc x$. Finally, the usual constructs to reason about equality and the standard boolean connectives are also contained in our property language.

Definition 3.11 (Formula) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature and $P \subseteq \mathcal{P}$ a set of predicates.*

A formula over P *is generated by the grammar*

$$\psi ::= \mathtt{tt} \mid a \mid p_l(x_1) \mid \circledcirc x_1 \mid x_1 = x_2 \mid \neg\psi \mid \psi_1 \wedge \psi_2$$

where $a \in Atoms_{\mathcal{X}}(P)$, $p_l \in \mathcal{P}_L \cap P$ and $x_1, x_2 \in \mathcal{X}$.

The set of all formulas over P is denoted by $Forms(P)$. ♢

The variables of a formula $\psi \in Forms(P)$ are denoted by $vars(\psi)$ and are canonically defined inductively as

$$\begin{aligned} vars(\mathtt{tt}) &:= \emptyset & vars(x_1 = x_2) &:= \{x_1, x_2\} \\ vars(p(x_1, \ldots, x_{k_p})) &:= \{x_1, \ldots, x_{k_p}\} & vars(\neg\psi) &:= vars(\psi) \\ vars(p_l(x_1)) = vars(\circledcirc x_1) &:= \{x_1\} & vars(\psi_1 \wedge \psi_2) &:= vars(\psi_1) \cup vars(\psi_2) \end{aligned}$$

where $p \in P$. By $evovars(\psi) \subseteq vars(\psi)$ we denote the set of variables which only occur as arguments for evolution predicates $\mathcal{P}_E$ in ψ.

We assume a number of abbreviations for a formula, namely

$$\begin{aligned} \mathbf{ff} &:= \neg\mathbf{tt} & x_1 \vee x_2 &:= \neg(\neg x_1 \wedge \neg x_2) \\ \oplus x &:= \neg\circledcirc x & x_1 \rightarrow x_2 &:= \neg x_1 \vee x_2 \\ x_1 \neq x_2 &:= \neg(x_1 = x_2) & x_1 \leftrightarrow x_2 &:= x_1 \rightarrow x_2 \wedge x_2 \rightarrow x_1, \end{aligned}$$

where the (only non-standard) notation $\oplus x$ allows us to express that the identity denoted by x is dead.

We assume the usual binding priorities, that is, the unary operators $\circledcirc, \oplus$ and $\neg$ bind most tightly, followed by equality ($=$), then conjunction ($\wedge$), then disjunction ($\vee$), then both kinds of implication ($\rightarrow, \leftrightarrow$).

Note that the formula language does not comprise any quantification mechanism, all variables occur free. To define the evaluation of a formula in a logical structure, we hence need a valuation that assigns to each variable of the formula some process identity from the domain of the logical structure.

Definition 3.12 (Formula Evaluation) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, and $I \subseteq Id^\perp$ a set of identities.*

The evaluation of a formula $\psi \in Forms(\mathcal{P}_{SL})$ *in a structure $S = (U, \iota) \in Strucs_{\mathcal{S}}(I)$ under a valuation $\mathcal{V} \in Vals_I(vars(\psi))$ is denoted by $S[\psi](\mathcal{V}) \in \mathbb{B}_3$ and is defined inductively as*

$$\begin{aligned} S[\mathbf{tt}](\mathcal{V}) &:= 1 \\ S[\circledcirc x_1](\mathcal{V}) &:= \begin{cases} 1/2 & \text{if } \mathcal{V}(x_1) = \perp \\ \mathcal{V}(x_1) \in U & \text{else} \end{cases} \\ S[p_s(x_1)](\mathcal{V}) &:= \iota(p_s)(\mathcal{V}(x_1)) \\ S[p_l(x_1, x_2)](\mathcal{V}) &:= \iota(p_l)(\mathcal{V}(x_1), \mathcal{V}(x_2)) \\ S[p_l(x_1)](\mathcal{V}) &:= \max\{\iota(p_l)(\mathcal{V}(x_1), u) \mid u \in U\} \\ S[x_1 = x_2](\mathcal{V}) &:= \begin{cases} \mathcal{V}(x_1) = \mathcal{V}(x_2) & \text{if } \{\mathcal{V}(x_1), \mathcal{V}(x_2)\} \subseteq I \setminus \{\perp\} \\ 1/2 & \text{if } \mathcal{V}(x_1) = \perp = \mathcal{V}(x_2) \\ 0 & \text{else} \end{cases} \\ S[\neg\psi](\mathcal{V}) &:= \neg S[\psi](\mathcal{V}) \\ S[\psi_1 \wedge \psi_2](\mathcal{V}) &:= S[\psi_1](\mathcal{V}) \wedge S[\psi_2](\mathcal{V}) \end{aligned}$$

where p_s ranges over $\mathcal{P}_S$, and p_l ranges over $\mathcal{P}_L$. ♢

The max operator denotes the greatest element with respect to the arithmetic order in the three-valued domain as introduced on page 15 such that $p_l(x_1)$ yields e.g. 1 if there is at least one definite link p_l from x_1 to some other process. In order to avoid the need for reasoning over an potentially infinite set, the evaluation of $p_l(x_1)$ only considers connections to *alive* processes, whose number will be unbounded but always finite. Further note that Def. 3.12 is tailored for the specific character of the abstract process $\perp$. This in particular affects the semantics for the comparison of variables. As the abstract process will be used to represent any number of concrete processes, comparing $\perp$ with itself yields $1/2$. In contrast, comparing some identity from Id with $\perp$ yields 0, as $\perp$ will represent only processes that are not concretely represented (cf. Def. 5.5).

Clearly, the evaluation of predicates is not limited to valuations that denote alive identities. In particular, one may query the existence of a dangling link, for example

$$\mathsf{sl}(x) \wedge \mathsf{link}(x, y) \wedge \oplus y$$

Actually, the reasoning over processes that may be dead yields some further intricacies. For example, the formula

$$\psi_x := \neg\mathsf{sl}(x)$$

holds for the adhoc networking structure S_{ad} (cf. page 23) both under the valuation $[x \mapsto u_4]$ and under $[x \mapsto u_5]$. However, it holds for u_4 because u_4 is rather a device than a slave while it holds for u_5 because no unary predicates holds for the dead process u_5. Note that we do not require that all state predicates yields 0 for some dead process, however typical system models will actually have this property. Then, although ψ_x is satisfied under these two valuations for the same reason, namely because the interpretation of the sl predicate yields 0 in both cases, the intuitive consequences of this satisfaction is different. While one expects for the alive process u_4 that either the interpretation of some other state predicate (like dev or ma in the adhoc example) yields 1, no interpretation of any state predicate for a dead process like u_5 is expected to yield 1. However, one does not obtain any information about the aliveness of the corresponding process unless the $\circledcirc$ operator is explicitly used in the formula. These specifics of reasoning over appearing and disappearing processes have to be carefully considered when using formulas to characterise logical structures, and in particular when specifying *temporal* system requirements. We have thoroughly discussed these aspects in a technical report [BTW07b], and we will reconsider the main issues in Section 4.3.

Let us consider some further example formulas. The expression

$$\psi_1 := \mathsf{ma}(x_1) \wedge \mathsf{link}(x_1, x_2) \wedge \mathsf{link}(x_1, x_3) \wedge x_2 \neq x_3$$

states that some master process has links to two different processes. It evaluates to $S_{\mathsf{ad}}[\psi_1](\mathcal{V}) = 1$ for $\mathcal{V} := [x_1 \mapsto u_2, x_2 \mapsto u_1, x_3 \mapsto u_3]$. The formula

$$\psi_2 := \neg\mathsf{link}(x)$$

says that there are no outgoing links from the process denoted by x. Here, we have $S_{\mathsf{ad}}[\psi_2](\mathcal{V}_1) = 0$ for e.g. $\mathcal{V}_1 := [x \mapsto u_1]$, and $S_{\mathsf{ad}}[\psi_2](\mathcal{V}_2) = 1$ for $\mathcal{V}_2 := [x \mapsto u_4]$.

Formulas may also evaluate indefinite, e.g. for $\mathcal{V}_3 := [x_1 \mapsto u_1, x_2 \mapsto u_2]$

$$\psi_3 := \mathsf{link}(x_1, x_2) \wedge \oplus x_2$$

evaluates to $S_D[\psi_3](\mathcal{V}_3) = 1/2$, indicating that the process denoted by x_1 possibly has a dangling link in the logical structure S_D (introduced on page 23).

We observe that besides the comparison and aliveness of variables denoting $\perp$, the only other reason for a formula evaluating indefinite is that some predicate interpretation yields the indefinite value. This entails the remark stating that two-valued logical structures (cf. Def. 3.6) over *Id* always evaluate definite.

Remark 3.13 (Definite Formula Evaluation) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $\psi \in Forms(\mathcal{P}_{SL})$ a formula, $I \subseteq Id$ a set of identities, $S \in Strucs_{\mathcal{S}}(I)$ a logical structure, and $\mathcal{V} \in Vals_I(vars(\psi))$ a valuation.*

If S is two-valued, then ψ evaluates definite*, that is,*

$$S \in Strucs^2_{\mathcal{S}}(I) \implies S[\psi](\mathcal{V}) \in \mathbb{B}. \qquad \diamondsuit$$

We now relate the evaluation of formulas and the embedding of logical structures (cf. Def. 3.10). As already mentioned, the embedding ensures a weak preservation of the evaluation of formulas, in the sense that the evaluation may become indefinite but not change from 1 to 0 or vice verse. We again use the information order of three-valued logic to formalise this property.

Lemma 3.14 (Property Preservation) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $I_1, I_2 \subseteq Id$ with $I_2 \subseteq I_1$ two sets of identities, and $S_1 = (U_1, \iota_1) \in Strucs_{\mathcal{S}}(I_1)$ and $S_2 = (U_2, \iota_2) \in Strucs_{\mathcal{S}}(I_2^{\perp})$ two logical structures with $S_1 \Subset S_2$.*

For any formula $\psi \in Forms(\mathcal{P})$ and valuation $\mathcal{V} \in Vals_{I_1}(vars(\psi))$, we have

$$S_1[\psi](\mathcal{V}) \sqsubseteq S_2[\psi](e^{I_1}_{I_2}(\mathcal{V})). \qquad \diamondsuit$$

Proof. By induction over the structure of the formula. The proof is given in the appendix (page 187). ■

This concludes our investigation of logical structures as a means to formally represent system snapshots. In particular, we have introduced the specification of *properties* in terms of a formula language (thereby partly anticipating Chapter 4 on specifying temporal requirements), and the *embedding* of logical structures, which will serve as a basic notion for the abstraction of dynamic evolution systems in Chapter 5.

3.2.2 Structured Labelled Transition Systems

To be able to define the *behaviour over time* of the considered systems we combine the notion of a labelled transition system (cf. Def. 2.1) with logical structures and ground atoms as introduced above, yielding so-called *structured labelled transition systems* (SLTS). The states of an SLTS correspond to logical structures, and the transitions among logical structures are labelled by ground atoms over evolution predicates, thereby indicating both what kind of transition has been made (by the predicate of the ground atom) and which set of identities are involved in this transition (by the arguments of the ground atom).

Definition 3.15 (Structured Labelled Transition System) *Let $\mathcal{S}$ be a signature and $I \subseteq Id^{\perp}$ a set of identities. An LTS $(\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ where*

1. *states are logical structures over $\mathcal{S}$ and I, i.e. $\mathbf{S} \subseteq Strucs_{\mathcal{S}}(I)$, and*
2. *labels are ground atoms over $\mathcal{P}_E$ and I, i.e. $\mathbf{L} \subseteq GroundAtoms_I(\mathcal{P}_E)$,*

is called a structured labelled transition system *(SLTS) over $\mathcal{S}$ and I. The set of all structured labelled transition systems over $\mathcal{S}$ and I is denoted by $\mathcal{T}_{\mathcal{S}}(I)$.*

A structured labelled transition system $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow) \in \mathcal{T}_{\mathcal{S}}(I)$ is called two-valued if all states are two-valued structures, i.e. if $\mathbf{S} \subseteq Strucs^2_{\mathcal{S}}(I)$. The set of all two-valued SLTS over $\mathcal{S}$ is denoted by $\mathcal{T}^2_{\mathcal{S}}(I)$. ♢

The notion of a run (Def. 2.1) transfers directly from labelled transition systems to structured labelled transition systems. A run of an SLTS $(\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ is an infinite sequence $((L_i, S_i))_{i \in \mathbb{N}_0}$ of pairs of ground atoms $L_i \in \mathbf{L}$ and logical structures $S_i \in \mathbf{S}$ such that $S_0 \in \mathbf{S}_0$ and $(S_i, L_{i+1}, S_{i+1}) \in \rightarrow$ for all $i \in \mathbb{N}_0$.

In the previous section, we have devised the embedding of logical structures in order to state their relation in terms of information order. We now lift this concept to Structured Labelled Transition Systems, yielding a notion of *simulation* for SLTSs. This generalises the simulation preorder ([Mil71], cf. Section 2.3) by relating those states and transition labels of two structured labelled transition systems which form an embedding relation.

Definition 3.16 (SLTS Simulation) *Let $\mathcal{S}$ be a signature, $I_1, I_2 \subseteq Id$ two sets of identities with $I_2 \subseteq I_1$ and*

$$\mathsf{T}_1 = (\mathbf{S}^1, \mathbf{S}_0^1, \mathbf{L}^1, \rightarrow) \in \mathcal{T}_{\mathcal{S}}(I_1)$$
$$\mathsf{T}_2 = (\mathbf{S}^2, \mathbf{S}_0^2, \mathbf{L}^2, \rightsquigarrow) \in \mathcal{T}_{\mathcal{S}}(I_2^{\perp})$$

two structured labelled transition systems over $\mathcal{S}$.

A relation $\sim \subseteq (\mathbf{S}^1 \times \mathbf{S}^2) \cup (\mathbf{L}^1 \times \mathbf{L}^2)$ is called a simulation relation if

1. *simulating elements form an embedding relation, that is,*

$$R_1 \sim R_2 \implies R_1 \Subset R_2$$

for $R_1 \in \mathbf{S}^1 \cup \mathbf{L}^1$ and $R_2 \in \mathbf{S}^2 \cup \mathbf{L}^2$, and

2. *each transition in T_1 is reflected by a transition in T_2, that is,*

$$S_1 \sim S_2 \implies \forall\, S_1' \in \mathbf{S}^1, L_1 \in \mathbf{L}^1 \text{ with } S_1 \xrightarrow{L_1} S_1'$$
$$\exists\, S_2' \in \mathbf{S}^2, L_2 \in \mathbf{L}^2 : S_2 \overset{L_2}{\rightsquigarrow} S_2' \wedge S_1' \sim S_2' \wedge L_1 \sim L_2$$

for any states $S_1 \in \mathbf{S}^1$ and $S_2 \in \mathbf{S}^2$.

We say that T_2 simulates T_1, written $\mathsf{T}_1 \preceq \mathsf{T}_2$, iff there exists a simulation relation $\sim \subseteq (\mathbf{S}^1 \times \mathbf{S}^2) \cup (\mathbf{L}^1 \times \mathbf{L}^2)$ that relates initial states, that is,

$$\forall\, S_0 \in \mathbf{S}_0^1\ \exists\, S_0' \in \mathbf{S}_0^2 : S_0 \sim S_0'.$$

T_1 and T_2 are similar, written $\mathsf{T}_1 \approx \mathsf{T}_2$, iff both $\mathsf{T}_1 \preceq \mathsf{T}_2$ and $\mathsf{T}_2 \preceq \mathsf{T}_1$. ♢

As in the classical variant of simulating transition systems we obtain a *corresponding path lemma*, stating that for each run of an SLTS there is corresponding run in a simulating SLTS that embeds each element of the original run.

Lemma 3.17 (Corresponding Runs) *Let $\mathcal{S}$ be a signature, $I_1, I_2 \subseteq Id$ two sets of identities, and $\mathsf{T}_1 \in \mathcal{T}_{\mathcal{S}}(I_1)$ and $\mathsf{T}_2 \in \mathcal{T}_{\mathcal{S}}(I_2^{\perp})$ two SLTSs with $\mathsf{T}_1 \preceq \mathsf{T}_2$.*

Then for every run $\pi_1 = ((L_i^1, S_i^1))_{i \in \mathbb{N}_0} \in Runs(\mathsf{T}_1)$ there exists a run $\pi_2 = ((L_i^2, S_i^2))_{i \in \mathbb{N}_0} \in Runs(\mathsf{T}_2)$ with

$$L_i^1 \Subset L_i^2 \text{ and } S_i^1 \Subset S_i^2$$

for all $i \in \mathbb{N}_0$. The run π_2 is called a corresponding run *of π_1.* ♢

Proof. By induction over the length of the run. The proof is given in the appendix (page 188). ■

We have presented the semantical domain for the considered class of systems, which in particular exhibits a notion of simulation which will be used to prove the soundness of the considered abstraction in Chapter 5. We proceed by developing suitable modelling languages for the developed semantical domain.

3.3 Dynamic Evolution Systems

In the following, we introduce a symbolic description language for structured labelled transition systems called Dynamic Evolution Systems (DES for short). The syntax of this language is rule-based and will be given in Subsection 3.3.1. The formal semantics of DES in terms of an SLTS follows in Subsection 3.3.2.

3.3.1 Syntax

Dynamic Evolution Systems will be modelled by a set of so-called *evolution rules*, each of them comprising a *name*, a *guard* and a sequence of *statements*. The effect of an evolution rule is that each logical structure satisfying the guard may evolve into the structure that results by executing the sequence of statements. The name of the rule will be used to label the corresponding transition by an evolution ground atom, that is, the name is actually an atom over evolution predicates. The guard of the rule is a formula over state and link predicates as introduced in Definition 3.11, and the language of statements is defined as follows.

Definition 3.18 (Statement) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature. A* statement over $\mathcal{S}$ *is generated by the following grammar*

$$stm ::= stm_1; stm_2 \mid a \mid \neg a \mid \neg p_l(x) \mid \circledast x \mid \otimes x \mid \mathsf{skip}$$

where $a \in Atoms_{\mathcal{X}}(\mathcal{P}_{SL})$, $p_l \in \mathcal{P}_L$ and $x \in \mathcal{X}$. The set of all statements over $\mathcal{S}$ is denoted by $Stms_{\mathcal{S}}$. ♢

The intuition of the seven ingredients of the statement grammar is as follows. The notion $stm_1; stm_2$ denotes sequential composition of the statements stm_1 and stm_2, which allows us to actually build *sequences of statements*. An atom $a = p(x_1, \ldots, x_{k_p}) \in Atoms_{\mathcal{X}}(\mathcal{P}_{SL})$ will be used to set the interpretation for p and the identities denoted by $x_1, \ldots, x_{k_p}$ to the boolean value 1. Analogously, the negated version of the atom '$\neg a$' sets the corresponding interpretation to 0. The negated variant of a binary predicate p_l comprising only one variable '$\neg p_l(x)$' will set all interpretations for p_l with the identity denoted by x as the first argument to 0. Speaking in terms of our application domain, $\neg p_l(x)$ will remove all outgoing p_l-connections of the process denoted by x. The notion $\circledast x$ represents the create statement that adds the identity denoted by x to the set of alive processes. Analogously, the notion $\otimes x$ represents the destroy statement that removes the identity denoted by x from the set of alive processes. The skip statement has no effect on the structure.

The variables of a statement $stm \in Stms_{\mathcal{S}}$ are denoted by $vars(stm)$ and are canonically defined inductively as

$$\begin{aligned} vars(stm_1; stm_2) &:= vars(stm_1) \cup vars(stm_2) \\ vars(\neg p_l(x)) = vars(\circledast x) = vars(\otimes x) &:= \{x\} \\ vars(\mathsf{skip}) &:= \emptyset \end{aligned}$$

and $vars(a)$ and $vars(\neg a)$ for atoms $a \in Atoms_{\mathcal{X}}(\mathcal{P}_{SL})$ are as defined on page 27.

For convenience, the positive variants of an atom statement may be followed by the *overwrite* modifier '!' that ensures that

- for state atoms $p_s(x) \in Atoms_{\mathcal{X}}(\mathcal{P}_S)$ the interpretations of all state predicates except for p_s becomes 0 for the identity denoted by x, and
- for link atoms $p_l(x_1, x_2) \in Atoms_x(\mathcal{P}_L)$ that all interpretations of the link predicate p_l becomes 0 for the identity denoted by x_1 as first argument, except of the two identities denoted by x_1 and x_2 as first and second argument, respectively. Speaking in terms of our application domain, the overwrite modifier for binary predicates will delete all outgoing p_l-connections of the identity denoted by x_1 first, and then establish a single p_l-connection to the identity denoted by x_2.

Given a signature $(\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ with $\mathcal{P}_S = \{p_s^1, \ldots, p_s^n\}$, we define the overwriting statement $a!$ for atoms $a \in Atoms_{\mathcal{X}}(\mathcal{P}_{SL})$ as the following two abbreviations

$$\begin{aligned} p_s!(x) &:= \neg p_s^1(x); \ldots; \neg p_s^n(x); p_s(x) \\ p_l!(x_1, x_2) &:= \neg p_l(x_1); p_l(x_1, x_2) \end{aligned}$$

for $p_s \in \mathcal{P}_S$ and $p_l \in \mathcal{P}_L$.

We are now prepared to define the syntax of Dynamic Evolution Systems as a set of evolution rules as motivated above.

Definition 3.19 (Dynamic Evolution System) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature. An* evolution rule *over $\mathcal{S}$ is a triple*

$$(name, guard, stm) \in Atoms_{\mathcal{X}}(\mathcal{P}_E) \times Forms(\mathcal{P}_{SL}) \times Stms_{\mathcal{S}}$$

where $vars(name) = vars(guard) \cup vars(stm)$. The set of all evolution rules over $\mathcal{S}$ is denoted by $EvoRule_{\mathcal{S}}$.

A Dynamic Evolution System (DES) *over $\mathcal{S}$ is a finite set of evolution rules $\mathsf{D} \subset EvoRule_{\mathcal{S}}$, and by $\mathcal{D}_{\mathcal{S}}$ we denote the set of all Dynamic Evolution Systems over $\mathcal{S}$.* ♢

Evolution rules will be written in the form '*name* • *guard* ▶ *stm*'. As an example system, we model the adhoc networking case study as the Dynamic Evolution System Ad over the signature $\mathcal{S}_{\mathsf{ad}}$ as given on page 23. It comprises the following five evolution rules

$$\begin{aligned}
\mathsf{new}(x) &\bullet \oplus x \blacktriangleright \circledast x; \mathsf{dev}(x)\\
\mathsf{con}(x,y) &\bullet \mathsf{dev}(x) \wedge (\mathsf{dev}(y) \vee \mathsf{ma}(y)) \wedge x \neq y \blacktriangleright \mathsf{sl!}(x); \mathsf{ma!}(y); \mathsf{link}(x,y); \mathsf{link}(y,x)\\
\mathsf{dis}(x,y) &\bullet \mathsf{ma}(x) \wedge \mathsf{link}(x,y) \blacktriangleright \mathsf{dev!}(y); \neg\mathsf{link}(x,y); \neg\mathsf{link}(y,x)\\
\mathsf{free}(x) &\bullet \mathsf{ma}(x) \wedge \neg\mathsf{link}(x) \blacktriangleright \mathsf{dev!}(x)\\
\mathsf{del}(x) &\bullet \mathsf{dev}(x) \blacktriangleright \otimes x
\end{aligned}$$

where the intuition of these rules are that

1. the first rule allows a dead identity to *appear* as a free device, and
2. the second rule allow two different identities, where one is a free device and the other is either a device or a master to *connect*, such that the first becomes a slave, the second becomes a master and links in both directions are established, and
3. the third rule allows a master having a link to some identity to *disconnect* this identity such that it becomes a free devices and the links to and from it are removed, and
4. the fourth rule allows a master device that has no link to any identity to become a *free* device again, and finally
5. the last rule allows a free device to *disappear* from the network.

3.3.2 Semantics

To be able to define the formal semantics of an DES, we need to formalise the intuition of the effects of the statements as discussed on page 33.

Definition 3.20 (Statement Execution) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, and $I \subseteq Id^{\perp}$ a set of identities.*

The execution of a statement *is a function*

$$\cdot\langle\cdot\rangle(\cdot) : Strucs_{\mathcal{S}}(I) \times Stms_{\mathcal{S}} \times Vals_I(\mathcal{X}) \to Strucs_{\mathcal{S}}(I)$$

which is defined inductively as

$$\begin{aligned}
S\langle stm_1; stm_2\rangle(\mathcal{V}) &:= S\langle stm_1\rangle(\mathcal{V})\langle stm_2\rangle(\mathcal{V})\\
S\langle p(x_1,\ldots,x_{k_p})\rangle(\mathcal{V}) &:= (U, \iota[p \mapsto \iota(p)[(\mathcal{V}(x_1),\ldots,\mathcal{V}(x_{k_p})) \mapsto 1]])
\end{aligned}$$

$$
\begin{aligned}
S\langle\neg p(x_1,\ldots,x_{k_p})\rangle(\mathcal{V}) &:= (U, \iota[p \mapsto \iota(p)[(\mathcal{V}(x_1),\ldots,\mathcal{V}(x_{k_p})) \mapsto 0]]) \\
S\langle\neg p_l(x)\rangle(\mathcal{V}) &:= (U, \iota[p_l \mapsto \iota(p_l)\lfloor_{\mathcal{V}(x)}]) \\
S\langle\circledast x\rangle(\mathcal{V}) &:= (U \cup \{\mathcal{V}(x)\}, \iota) \\
S\langle\otimes x\rangle(\mathcal{V}) &:= (U \setminus \{\mathcal{V}(x)\}, \iota) \\
S\langle\mathsf{skip}\rangle(\mathcal{V}) &:= S
\end{aligned}
$$

where $S = (U, \iota)$ *and* $p \in \mathcal{P}_{SL}$*. The notation* $\iota(p_l)\lfloor_u$ *sets all interpretations for predicate* p_l *and identity* u *as the first argument to* 0*, that is,*

$$
\iota(p_l)\lfloor_u(u_1,u_2) := \begin{cases} 0 & \text{if } u_1 = u \\ \iota(p_l)(u_1,u_2) & \text{else} \end{cases}
$$

for $p_l \in \mathcal{P}_L$ *and* $u, u_1, u_2 \in I$*.* ♢

We will now specify when two logical structures are related by an evolution rule, that is, when a (source) structure may evolve into a (target) structure via some evolution ground atom. This notion of *evolution* is the central concept for defining the formal semantics of a Dynamic Evolution System in terms of a SLTS below.

Definition 3.21 (Evolution) *Let* $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *be a signature,* $r = (name, guard, stm) \in EvoRule_{\mathcal{S}}$ *an evolution rule over* $\mathcal{S}$*, and* $I \subseteq Id$ *a set of identities.*

Then r *enables the* evolution of the logical structure $S \in Strucs_{\mathcal{S}}(I)$ into the logical structure $S' \in Strucs_{\mathcal{S}}(I)$ via the ground atom $g \in GroundAtoms_I(\mathcal{P}_E)$ under I*, denoted*

$$S \xrightarrow[r,I]{g} S'$$

if there is a valuation $\mathcal{V} \in Vals_I(vars(name))$ *such that the following three conditions are satisfied:*

1. *The source structure* S *satisfies the guard under* $\mathcal{V}$*, that is,*

$$S[\![guard]\!](\mathcal{V}) = 1.$$

2. *The rule name under* $\mathcal{V}$ *yields the evolution ground atom* g*, that is,*

$$name[\mathcal{V}] = g.$$

3. *Executing the statements* stm *on* S *yields the target structure* S'*, that is,*

$$S' = S\langle stm\rangle(\mathcal{V}).$$

We say that a Dynamic Evolution System $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *enables the evolution of a structure* $S \in \mathit{Strucs}_{\mathcal{S}}(I)$ *into* $S' \in \mathit{Strucs}_{\mathcal{S}}(I)$ *via the evolution ground atom* $g \in \mathit{GroundAtoms}_I(\mathcal{P}_E)$ *under* I*, denoted*

$$S \xrightarrow[\mathsf{D}, I]{g} S'$$

if there is a rule $r \in \mathsf{D}$ *such that* $S \xrightarrow[r, I]{g} S'$. ♢

To illustrate the concept of evolution, we consider some examples from the adhoc networking case study Ad as defined on page 35. We fix a set of identities $I := \{u_1, u_2\} \subset \mathit{Id}$, and have

$$S_1 := (\{u_1\}, (\{\mathsf{dev}(u_1)\}, \emptyset)) \xrightarrow[\mathsf{Ad}, I]{\mathsf{new}(u_2)} (\{u_1, u_2\}, (\{\mathsf{dev}(u_1), \mathsf{dev}(u_2)\}, \emptyset)) =: S_2$$

by the evolution rule

$$\mathsf{new}(x) \bullet \oplus x \blacktriangleright \circledast x; \mathsf{dev}(x)$$

as for the valuation $\mathcal{V} = [x \mapsto u_2]$ the guard is satisfied in S_1, that is,

$$S_1[\oplus x](\mathcal{V}) = 1,$$

and the rule name under $\mathcal{V}$ yields the evolution ground atom, that is,

$$\mathsf{new}(x)[\mathcal{V}] = \mathsf{new}(u_2),$$

and the target structure S_2 is the result of executing the statement in S_1, i.e.

$$S_2 = S_1\langle \circledast x; \mathsf{dev}(x)\rangle(\mathcal{V}).$$

Similarly, we have

$$S_2 \xrightarrow[\mathsf{Ad}, I]{\mathsf{con}(u_1, u_2)} (\{u_1, u_2\}, (\{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1, u_2), \mathsf{link}(u_2, u_1)\}, \emptyset))$$

by the evolution rule

$$\mathsf{con}(x, y) \bullet \mathsf{dev}(x) \wedge (\mathsf{dev}(y) \vee \mathsf{ma}(y)) \wedge x \neq y \blacktriangleright \mathsf{sl!}(x); \mathsf{ma!}(y); \mathsf{link}(x, y); \mathsf{link}(y, x)$$

under the valuation $[x \mapsto u_1, y \mapsto u_2]$. Thus, the two devices u_1 and u_2 may connect such that u_1 becomes a master and u_2 a slave. Note that the symmetric evolution, namely where u_2 becomes the master and u_1 the slave, is also possible under the valuation $[x \mapsto u_2, y \mapsto u_1]$. We will discuss symmetry properties of Dynamic Evolution Systems in more detail in Section 5.2.2.

We observe that an evolution only affects processes which are in the range of the valuation function according to Def. 3.21. As the name of a rule comprises all variables of both the guard and the statements, we obtain the following remark.

Remark 3.22 (Locality of Evolution) *Let $r \in EvoRule_{\mathcal{S}}$ be an evolution rule over signature $\mathcal{S}$, $I \subseteq Id$ a set of identities, and $S, S' \in Strucs_{\mathcal{S}}(I)$ two logical structures.*

If S evolves into S' via g, then the focus of S onto the arguments of g evolves into the same focus of S' and vice versa, that is,

$$S \xrightarrow[r,I]{g} S' \quad \Longleftrightarrow \quad S|\!|_{A(g)} \xrightarrow[r,I]{g} S'|\!|_{A(g)} \qquad \diamondsuit$$

Proof. The proof is given in the appendix (page 188). ■

The concrete semantics of a Dynamic Evolution System will be defined by collecting all evolving structures in a structured labelled transition system. To facilitate a flexible mechanism to characterise the *initial* snapshot of the considered system, we need to introduce a notion of *identity permutation* before. In fact, we allow to specify a designated logical structure characterising the initial snapshot of the system. However, in order to not break the symmetry properties of the induced transition system (cf. Section 5.2.2), we need to take all permutations of this given structure into account when constructing the set of initial states of the induced transition system.

Definition 3.23 (Permutation) *Let S be a set. A permutation of S is a bijective function $\sigma : S \to S$.*

The set of all permutations of S is denoted by Σ_S. $\diamondsuit$

For a set of identities $I \subseteq Id^{\perp}$, we canonically extend a permutation function $\sigma \in \Sigma_I$ to work on a set $I' \subseteq I$ of identities by setting $\sigma(I') := \{\sigma(u) \mid u \in I'\}$, and we define the application of a permutation to valuations, ground atoms and logical structures as follows.

Definition 3.24 (Permutations) *Let $I \subseteq Id$ be a set of identities, and $\sigma \in \Sigma_I$ a permutation of I.*

- *The permutation of a valuation $\mathcal{V} \in Vals_I(\mathcal{X})$ under σ is defined as*

  $$\sigma(\mathcal{V})(x) := \sigma(\mathcal{V}(x))$$

  for each variable $x \in \mathsf{dom}(\mathcal{V})$.
- *The permutation of a ground atom $g = p(u_1, \ldots, u_{k_p}) \in GroundAtoms_I(P)$ under σ is defined as*

  $$\sigma(g) := p(\sigma(u_1), \ldots, \sigma(u_{k_p})).$$

  and is extended to a set of ground atoms $V = \{g_1, \ldots, g_n\}$ as

  $$\sigma(V) := \{\sigma(g_1), \ldots, \sigma(g_n)\}$$

- *The permutation of a logical structure* $S = (U, \iota) \in \mathit{Strucs}_{\mathcal{S}}(I)$ *under* σ *is defined as*

$$\sigma(S) := (\sigma(U), (\sigma(\iota_{\mathbf{1}}), \sigma(\iota_{\mathbf{1/2}})). \qquad \diamondsuit$$

With this, we are able to define the concrete semantics of Dynamic Evolution System, parameterised by a given logical structure describing the initial snapshot of the system. Note that this initial structure may in particular be the empty structure S_e as defined on page 22.

Definition 3.25 (Semantics of DES) *Let* $\mathcal{S}$ *be a signature,* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over* $\mathcal{S}$*,* $I \subseteq Id$ *a set of identities, and* $S \in \mathit{Strucs}_{\mathcal{S}}(I)$ *a logical structure.*

The semantics of D and S under I*, denoted* $[\![\mathsf{D}, S]\!]_I$*, is the structured labelled transition system* $(\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow) \in \mathcal{T}^2_{\mathcal{S}}(I)$ *where*

- *the states are two-valued logical structures over* $\mathcal{S}$ *and* I*, that is,*

$$\mathbf{S} := \mathit{Strucs}^2_{\mathcal{S}}(I),$$

- *the given structure* S *determines the set of initial states, that is,*

$$\mathbf{S}_0 := \{\sigma(S) \in \mathit{Strucs}^2_{\mathcal{S}}(I) \mid \sigma \in \Sigma_I\}$$

- *the labels are evolution ground atoms over* $\mathcal{P}_E$ *and* I*, that is,*

$$\mathbf{L} := \mathit{GroundAtoms}_I(\mathcal{P}_E),$$

- *and the evolution determines the transition relation, that is,*

$$\rightarrow := \{(S, g, S') \in \mathbf{S} \times \mathbf{L} \times \mathbf{S} \mid S \xrightarrow[\mathsf{D}, I]{g} S'\}.$$

If no restriction on the set of available identities is given, that is, if $I = Id$*, we call*

$$[\![\mathsf{D}, S]\!] := [\![\mathsf{D}, S]\!]_{Id}$$

the concrete semantics *of* D *and* S*.*

If no logical structure S *is given, we take the empty structure as the initial state, that is, we set*

$$[\![\mathsf{D}]\!]_I := [\![\mathsf{D}, S_e]\!]_I$$

for any $I \subseteq Id$*.* $\diamondsuit$

For a subset of identities $I \subset Id$, we call $[\![\mathsf{D}, S]\!]_I$ the *I-underapproximated semantics* of D. This notion is well-defined by the following lemma, stating that each run in any underapproximated semantics of D corresponds to a run in the concrete semantics of D.

Lemma 3.26 (Under-Approximation) *Let $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ be a Dynamic Evolution System over $\mathcal{S}$, $I \subseteq Id$ a set of identities and $S \in Strucs_\mathcal{S}(I)$ a logical structure. Then*

$$\forall\, \pi' \in Runs([\![\mathsf{D}, S]\!]_I)\ \exists\, \pi \in Runs([\![\mathsf{D}, S]\!]_{Id}) : \pi' = \pi|_I$$

where $\pi|_I := ((L_i, (U_i, \iota_i|_I))_{i\in\mathbb{N}_0}$ denotes the element-wise restriction of the interpretation function to identities from I. ♢

Proof. According to Def. 3.21, evolution of logical structures requires the existence of a rule $(name, guard, stm) \in \mathsf{D}$ and a valuation $\mathcal{V} \in Vals_I(vars(name))$ of variables to identities I. As each valuation to a subset $I \subseteq Id$ is in particular a valuation for Id, we have that any two structures that are enabled under I are also enabled under Id. Then the claim follows by Remark 3.22 (Locality of Evolution) and by the semantics of DES (cf. Def. 3.25). ■

A *run* of the concrete semantics of a Dynamic Evolution System D (i.e. of the underlying structured labelled transition system) stands for one particular behaviour of the system. The set of all a runs of $[\![\mathsf{D}, S]\!]$ represents the complete behaviour of D and S. For the adhoc networking case study, we present a prefix of a run $\pi \in Runs([\![\mathsf{Ad}]\!])$ where two processes u_1 and u_2 appear, connect themselves, and disconnect again. Then the device u_1 disappears, and u_2 becomes a device and disappears.

$$\begin{aligned}
&\pi = (\emptyset, \iota_e),\\
&(\ \mathsf{new}(u_1),\ (\{u_1\}, (\{\mathsf{dev}(u_1)\}, \emptyset)\ ),\\
&(\ \mathsf{new}(u_2),\ (\{u_1, u_2\}, (\{\mathsf{dev}(u_1), \mathsf{dev}(u_2)\}, \emptyset)\ ),\\
&(\ \mathsf{con}(u_1, u_2),\ (\{u_1, u_2\}, (\{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1, u_2), \mathsf{link}(u_2, u_1)\}, \emptyset)\ ),\\
&(\ \mathsf{dis}(u_2, u_1),\ (\{u_1, u_2\}, (\{\mathsf{dev}(u_1), \mathsf{ma}(u_2)\}, \emptyset)\ ),\\
&(\ \mathsf{del}(u_1),\ (\{u_2\}, (\{\mathsf{ma}(u_2)\}, \emptyset)\ ),\\
&(\ \mathsf{free}(u_2),\ (\{u_2\}, (\{\mathsf{dev}(u_2)\}, \emptyset)\ ),\\
&(\ \mathsf{del}(u_2),\ (\emptyset, \iota_e)\ ),\ \ldots
\end{aligned}$$

Using the graphical notation for logical structures as introduced on page 23, we show an illustration of these evolution steps in Figure 3.3.

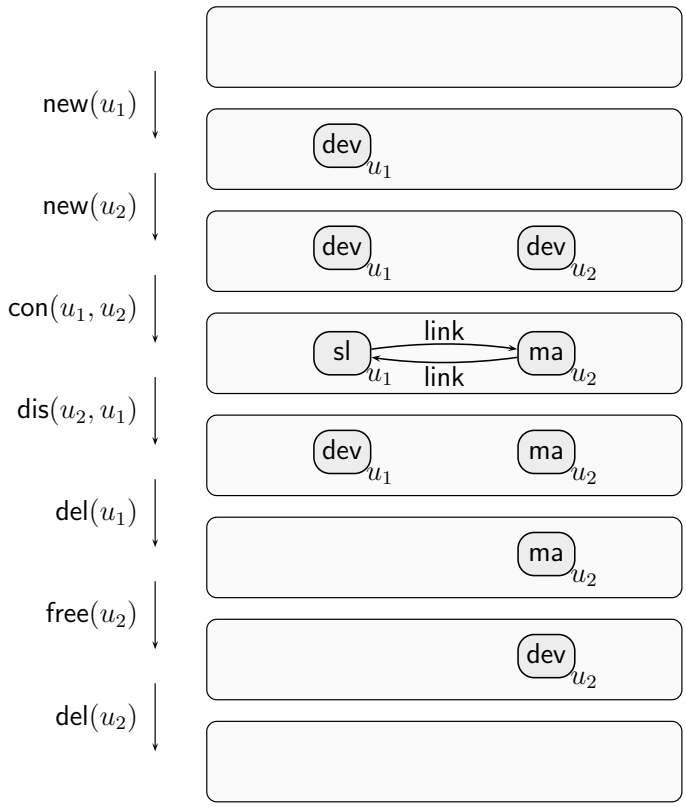


Figure 3.3: A prefix of a run of $[\![\mathsf{Ad}]\!]$.

3.3.3 Discussion

The language of Dynamic Evolution System equips us with a concise formalism to specify the behaviour of systems where both the number of alive processes and their configurations varies over the time. Clearly, DES is a rather low-level language as the evolution of processes is based on a "partly global view" on the system. For example, the possible evolutions of some process may depend on the local status of some other process. However, when implementing the behaviour of a process in a distributed system, each process has only a *local* view on the system, namely its own configuration. Information about other process is typically gathered by *communication* with other processes, that is, each process builds its own view of a certain part of the overall system by querying other processes and storing their answers as part of its local configuration.

In a joint paper [BSTW06] we have addressed this issue by developing a modelling language called Dynamic Communication Systems (DCS for short). There, the behaviour of a single process is given in terms of a finite transition

system, where the transitions are labelled by certain actions like sending and receiving of messages over named communication channels. The semantics of the overall system is then given by the parallel composition of arbitrarily many processes, each one following the same behavioural description.

We will introduce this language in the following section. However, the reader who is more interested in our approach to actually *analyse* the considered class of systems may safely skip the next section upon the first reading and directly proceed to Chapter 4. We will reconsider the DCS language in Section 6.3 where we exploit the specifics of this language for a new kind of abstraction refinement. However, the overall approach for requirement specification, abstraction and analysis given in Chapters 4 to 6 is purely defined in terms of the DES language and hence may be understood without knowing the DCS language.

3.4 Dynamic Communication Systems

As the name *Dynamic Communication Systems* already indicates, this language opens up a new issue in our modelling framework, namely *communication*. As motivated before, the behaviour of a single process operating in a distributed system is mainly driven by communication with other processes. This is because the local configuration of processes is not directly visible from the outside but has to be acquired via message communication. The DCS language accounts for this fact by modelling the behaviour from the *local view of a single process*, thereby specifying a behavioural template that each process will follow. This is clearly related to the notion of a class in object-oriented programming [Str00], where each instance of this class adheres to the given description during runtime.

A DCS process may only sent messages to processes it knows. This kind of acquaintanceship is realised by establishing named channels among processes. There are basically two possibility for a process to establish a new channel, either by creating a new process or by receiving a message that comprises a process identity as parameter. A process may also delete one of its channels.

A Dynamic Communication System may comprise a set of so-called environment messages, which are non-deterministically sent from the environment to alive processes. By these messages one can abstract from physical aspects of the real-world system, for example to model the detecting of processes via hardware sensors. Environment messages are in particular suitable to bootstrap the communication protocols by providing initial channel contents.

The previous paragraph has already introduced the four basic actions, namely

- *creation* of processes,
- *sending* and *receiving* of messages, and
- *reset* of channels.

Additionally, there are three kinds of environment actions, namely *appearance* and *disappearance* of processes and *sending* of environment messages. Note that sending a message always attaches the process identity of the sender to the message. In particular we do not consider the feature of the DCS variant in [BSTW06] that may use the content of a channel as a message parameter. Moreover, we assume that each channel holds at most one process identity at a time. These restrictions allow for a precise static analysis of communication dependencies (cf. Sect. 6.3) without limiting the expressiveness of the overall language (cf. page 72). We will give the formal syntax of Dynamic Communication Systems in the following Subsection 3.4.1, and its semantics in Subsection 3.4.2.

3.4.1 Syntax

A Dynamic Communication System is specified by a behavioural template, the so-called *DCS protocol*. Actually, there may be more than one DCS protocol in a Dynamic Communication System, that is, the processes are *typed*, and each type has its own protocol description. In particular, the creation action is parameterised in the type of process which is to be created. Formally, we define the set of DCS actions as follows.

Definition 3.27 (DCS Actions) *Let $\mathcal{T}$ be a set of types, C a set of channel names, and M a set of message names. We define*

- *the set of* create actions *over C and $\mathcal{T}$ as*

$$New_{C,\mathcal{T}} := \{*_t^c \mid c \in C, t \in \mathcal{T}\},$$

- *the set of* receive actions *over C and M as*

$$Rcv_{C,M} := \{?m(c) \mid c \in C, m \in M\},$$

- *the set of* send actions *over C and M as*

$$Snd_{C,M} := \{c!m \mid c \in C, m \in M\}, \text{ and}$$

- *the set of* reset actions *over C as*

$$Clr_C := \{\bar{c} \mid c \in C\}.$$ ♢

Note that the channel of a receive action will not specify the channel over which the message is to be received but rather the channel in which the identity of the sender of the message will be stored. Besides creation of new processes, this storing mechanism allows a process to acquire new process identities. As each channel holds at most one identity, each process has at most $|C|$ communication partners at a certain point in time, however these partners may of course vary over the time.

A DCS protocol is given as a transition system where the transitions are annotated by some DCS action as defined above. A DCS protocol declares a subset of its states to be *initial* and *fragile*. The set of initial state determine in which state a new process may appear. If some process is in a fragile state, it may disappear from the system. Additionally, we declare a set of channel names under which a process can establish connections to other processes.

Definition 3.28 (DCS Protocol) *Let $\mathcal{T}$ be a set of types, M a set of message names, and M_X a subset of M.*

A DCS Protocol *over $\mathcal{T}$, M and M_X is given by a tuple*

$$\mathsf{P} = (Q, A, F, C, succ)$$

where

- *Q is a finite set of states,*
- *$A \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of fragile states,*
- *C is a finite set of channel names,*
- *$succ \subseteq Q \times (New_{C,\mathcal{T}} \cup Snd_{C,M \setminus M_X} \cup Rcv_{C,M} \cup Clr_C) \times Q$ is the transition relation.*

The set of all DCS Protocols over $\mathcal{T}$, M and M_X is denoted by $\mathcal{P}(\mathcal{T}, M, M_X)$.

We use the notations Q_P, A_P, F_P, C_P and $succ_\mathsf{P}$ to access the respective components of a DCS protocol $\mathsf{P} = (Q, A, F, C, succ)$. ♢

A Dynamic Communication System is basically a set of DCS protocols. It additionally declares a common set of message names by which the different instances of the protocols may communicate. A subset of these message is tagged as *environment messages*, indicating that these message will only by sent from the environment.

Definition 3.29 (Dynamic Communication System) *A tuple*

$$\mathsf{C} = (M, M_X, P)$$

where

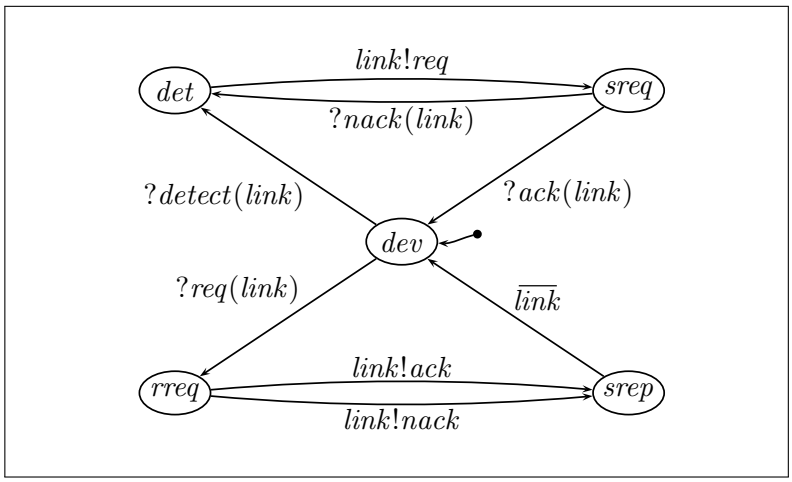


Figure 3.4: C_{ad} – the adhoc connection principle in terms of a DCS.

- *M is a finite set of message names,*
- *$M_X \subseteq M$ is a set of environment message names, and*
- *$P : \mathcal{T} \to \mathcal{P}(\mathcal{T}, M, M_X)$ is a mapping from a finite set of types $\mathcal{T}$ to DCS protocols over $\mathcal{T}$, M and M_X,*

is called a Dynamic Communication System (DCS). *The set of all Dynamic Communication Systems is denoted by $\mathcal{DCS}$.* ♢

We will illustrate the concepts of the DCS language by modelling a simple adhoc connection protocol as the Dynamic Communication System $\mathsf{C}_{\mathsf{ad}} := (M, M_X, P)$. For this system, we employ messages $M := \{detect, req, ack, nack\}$ where '*detect*' is the only environment messages, that is, $M_X := \{detect\}$. The DCS comprises only one single type '*d*', whose DCS protocol is given by $P(d) := (Q, A, F, C, succ)$ with

- states $Q := \{dev, det, sreq, rreq, srep\}$,
- initial state $A := \{dev\}$,
- no fragile states $F := \emptyset$,
- a single channel name $C := \{link\}$,
- and the transition relation *succ* as depicted in Figure 3.4.

Each new process starts being in state '*dev*'. In this state, it is sensible for receiving the environment message '*detect*' and the request message '*req*'. Upon receiving '*detect*', the process implicitly establishes a '*link*' channel to the sender of the message and proceeds to state '*det*'. From here, it sends a '*req*' message over the '*link*' channel to the just detected process, and enters state '*sreq*'. The

process then waits for an answer of his request, which may either be an error message '*nack*' or an positive reply '*ack*'. In the case of an error message, it will re-send the request by going back to state '*det*'. If it receives a positive reply, it will proceed to state '*dev*'.

If a process receives a request message '*req*', it stores the attached identity in its '*link*' channel and switches to state '*rreq*'. From here, it will nondeterministically reply with a '*nack*' or a '*ack*' message. In both cases, it enters state '*sreq*' and will reset its '*link*' channel when moving back to '*dev*'.

3.4.2 Semantics

The semantics of Dynamic Communication Systems is given by a translation to Dynamic Evolution Systems. Most of the constituents of a DCS protocol have a canonical counterpart in the DES world. For example, the states become unary predicates and the channels are represented by binary predicates. Transitions of the Dynamic Communication System, e.g. message communication or process creation, will be captured by corresponding evolution predicates.

The translation procedure itself largely depends on the underlying communication paradigm. For DCS, we assume an *asynchronous* communication mechanism, where the sending of messages is non-blocking and the messages are buffered on the receiver side. This buffer can faithfully be organised as a *set*, as the underlying transport protocols in distributed systems are typically *unreliable*, that is, neither the preservation of the order of the messages nor the proper delivery of the messages at all is guaranteed. Typical examples of such low-level protocols include the Internet Protocol (IP) [Pos81] and the User Datagram Protocol (UDP) [Pos80]. As a consequence, the communication protocol itself has to take care for the potential loss and overtaking of messages.

The set of pending incoming messages will be represented by binary predicates, where the name of the predicate corresponds to the name of the message. The fact that the interpretation of a message predicates 'm' yields true for a pair of processes (u_1, u_2) then encodes the fact that the message 'm' has been sent from process u_2 to process u_1.

Following the discussion above, we will start the translation of DCS to DES by defining the signature induced by a given Dynamic Communication System.

Definition 3.30 (DCS Signature) *Let* $\mathsf{C} = (M, M_X, P) \in \mathcal{DCS}$ *be a DCS. We define the* induced signature of C *as* $\mathcal{S}(\mathsf{C}) = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *where*

- *the set of variables comprises at least three elements* $\{x, y, z\} \subseteq \mathcal{X}$,

- *the state predicates comprises the states of all DCS protocols of* C*, i.e.*

$$\mathcal{P}_S := \{ q_t \mid t \in \mathsf{dom}(P), q \in Q_{P(t)} \},$$

- *the link predicates comprises the messages and the channels of all DCS Protocols of* C*, i.e.*

$$\mathcal{P}_L := M \cup \{ c_t \mid t \in \mathsf{dom}(P), c \in C_{P(t)} \}, \text{ and}$$

- *the evolution predicates denote messages communication, channel modifications and the creation, appearance and disappearance of typed processes*

$$\begin{aligned} \mathcal{P}_E := & \{\mathsf{snd}[m]/2, \mathsf{rcv}[m]/2 \mid m \in M\} \cup \\ & \{\mathsf{create}[t]/2, \mathsf{clr}[c_t]/1 \mid t \in \mathsf{dom}(P), c \in C_{P(t)}\} \cup \\ & \{\mathsf{appear}[t]/1, \mathsf{disappear}[t]/1 \mid t \in \mathsf{dom}(P)\}. \end{aligned}$$ ♢

The translation of DCS to DES comprises of two parts. One part deals with the translation of the behaviour that is given by the DCS protocols, and the second part realises the behaviour of the environment. In fact, each transition in each DCS protocol becomes a separate evolution rule. For the environment, there will be rules to let fresh process appear in one of their initial state, to remove process which are in one of their fragile states, and to send an environment message to alive processes.

Definition 3.31 (DES Semantics of DCS) *Let* $\mathsf{C} = (M, M_X, P) \in \mathcal{DCS}$ *be a Dynamic Communication System.*

We define the translation of a transition $(q, a, q) \in succ_{P(t)}$ *for some type* $t \in \mathsf{dom}(P)$ *to a set of evolution rules as follows.*

$$\begin{aligned} T((q, *^c_{t'}, q')_t) := & \qquad \text{(create action)} \\ & \{\mathsf{create}[t'](x,y) \bullet q_t(x) \wedge \oplus y \blacktriangleright \circledast y;\, q^0_{t'}!(y);\, c!(x,y);\, q'_t!(x) \mid q^0 \in A_{P(t')}\} \\ T((q, c_1!m, q')_t) := & \qquad \text{(send action)} \\ & \{\mathsf{snd}[m](x,y) \bullet q_t(x) \wedge c(x,y) \blacktriangleright m(y,z);\, q'_t!(x)\} \\ T((q, ?m(c), q')_t) := & \qquad \text{(receive action)} \\ & \{\mathsf{rcv}[m](x,y) \bullet q_t(x) \wedge m(x,y) \blacktriangleright c!(x,y); \neg m(x,y);\, q'_t!(x)\} \\ T((q, \bar{c}, q')_t) := & \qquad \text{(reset action)} \\ & \{\mathsf{clr}[c_t](x) \bullet q_t(x) \blacktriangleright \neg c(x);\, q'_t!(x)\} \end{aligned}$$

$$
\begin{aligned}
\mathsf{appear}[d](x) &\bullet \oplus x \blacktriangleright \circledast x;\ dev_d(x)\\
\mathsf{snd}[detect](x,y) &\bullet \circledcirc x \wedge \circledcirc y \wedge x \neq y \blacktriangleright detect(y,x)\\
\mathsf{rcv}[detect](x,y) &\bullet dev_d(x) \wedge detect(x,y) \blacktriangleright link_d!(x,y); \neg detect(x,y);\ det_d!(x)\\
\mathsf{snd}[req](x,y) &\bullet det_d(x) \wedge link_d(x,y) \blacktriangleright req(y,x);\ sreq_d!(x)\\
\mathsf{rcv}[req](x,y) &\bullet dev_d(x) \wedge req(x,y) \blacktriangleright link_d!(x,y); \neg req(x,y);\ rreq_d!(x)\\
\mathsf{snd}[nack](x,y) &\bullet rreq_d(x) \wedge link_d(x,y) \blacktriangleright nack(y,x);\ srep_d!(x)\\
\mathsf{snd}[ack](x,y) &\bullet rreq_d(x) \wedge link_d(x,y) \blacktriangleright ack(y,x);\ srep_d!(x)\\
\mathsf{rcv}[nack](x,y) &\bullet sreq_d(x) \wedge nack(x,y) \blacktriangleright link_d!(x,y); \neg nack(x,y);\ det_d!(x)\\
\mathsf{rcv}[ack](x,y) &\bullet sreq_d(x) \wedge ack(x,y) \blacktriangleright link_d!(x,y); \neg ack(x,y);\ dev_d!(x)\\
\mathsf{clr}[link_d](x) &\bullet srep_d(x) \blacktriangleright \neg link_d(x);\ dev_d!(x)
\end{aligned}
$$

Figure 3.5: $\mathsf{D}(\mathsf{C}_{\mathsf{ad}})$ – the semantics of a DCS as a set of evolution rules.

We define the environment rules of C *as*

$$
\begin{aligned}
&Env(\mathsf{C}) :=\\
&\{\mathsf{appear}[t](x) \bullet \oplus x \blacktriangleright \circledast x;\ q_t!(x) \mid t \in \mathsf{dom}(P), q \in A_{P(t)}\} \ \cup && \text{(appearance)}\\
&\{\mathsf{disappear}[t](x) \bullet q_t(x) \blacktriangleright \neg q_t(x); \otimes x \mid t \in \mathsf{dom}(P), q \in F_{P(t)}\} \ \cup && \text{(disappear.)}\\
&\{\mathsf{snd}[m](x,y) \bullet \circledcirc x \wedge \circledcirc y \wedge x \neq y \blacktriangleright m(y,x) \mid m \in M_X\} && \text{(env. message)}
\end{aligned}
$$

Combining both translations yields the DES semantics of C *as the following set of evolution rules:*

$$\mathsf{D}(\mathsf{C}) := Env(\mathsf{C}) \cup \bigcup_{t \in \mathsf{dom}(P)} \bigcup_{tr \in succ_{P(t)}} T(tr) \qquad \diamondsuit$$

For the DCS C_{ad} we obtain the signature $\mathcal{S}(\mathsf{C}_{\mathsf{ad}}) = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ with

- state predicates $\mathcal{P}_S := \{dev_d, det_d, sreq_d, rreq_d, srep_d\}$,
- link predicates $\mathcal{P}_L := \{detect, req, nack, ack, link_d\}$, and
- evolution predicates

$$\mathcal{P}_E := \{\mathsf{appear}[d], \mathsf{clr}[link_d]\} \cup \{\mathsf{snd}[m], \mathsf{rcv}[m] \mid m \in M\}$$

and the resulting DES $\mathsf{D}(\mathsf{C}_{\mathsf{ad}})$ as given in Figure 3.5. The first rule reflects the non-deterministic appearance of processes by the environment, that is, any dead process may appear being in the initial state '*dev*'. The second rule corresponds

to the sending of the environment message '*detect*' among two different alive process. The third rule models the reception of such an environment message, corresponding to the transition from state '*dev*' to '*det*' in the DCS protocol. This means, a process being in state '*dev*' and having an connection labelled '*detect*' to some process may change to state '*det*' and thereby establish the channel '*link*' to the message parameter identity and remove the pending message. Similar, the sending and receiving of internal messages takes place in the following six rules. The last rules corresponds to the removal of the '*link*' connection in the transition from state '*srep*' to '*dev*'.

We show an example run of $[\![\mathsf{D}(\mathsf{C}_{\mathsf{ad}})]\!]$ in Figure 3.6 on the following page, where two processes appear and perform a successful negotiation leading to two processes being in state 'dev_d' and having a directed '*link*' connection.

3.5 Related Work

In this section we compare our methods for modelling the considered class of dynamic systems with other existing approaches.

As the overall aim of this thesis is to devise refinement strategies for the spotlight abstraction technique, it is suggestive to employ a semantical domain where this abstraction is defined on. Otherwise, additional effort is required to transfer the abstraction mechanism to this new domain. In [Wes08], which is the main reference for the spotlight abstraction principle, a certain variant of labelled transition systems called "Evolving Topology Transition Systems" (ETTS) serve as the basic semantical structure. There, the states corresponds to *labelled multi graphs* and the transitions are labelled by a consistent *evolution annotation*. This annotation enables a very precise tracking of the life-cycle of processes, in particular one may observe the immediate re-use of process identities. We basically follow the approach of [Wes08] and use labelled transition systems as our semantical domain. However, we prefer to employ the basic concept of *logical structures* rather than labelled multi graphs. The usage of logical structures as a means to represent varying connection topologies is actually inspired by the seminal work on shape analysis in [SRW02]. Moreover, we modified the labelling of the transitions by annotating them with the set of processes involved in the corresponding evolution and the respective kind of evolution. Actually, this dedicated labelling will facilitate the refinement of the behaviour of the abstract process via temporal assumptions (cf. Section 6.1.2).

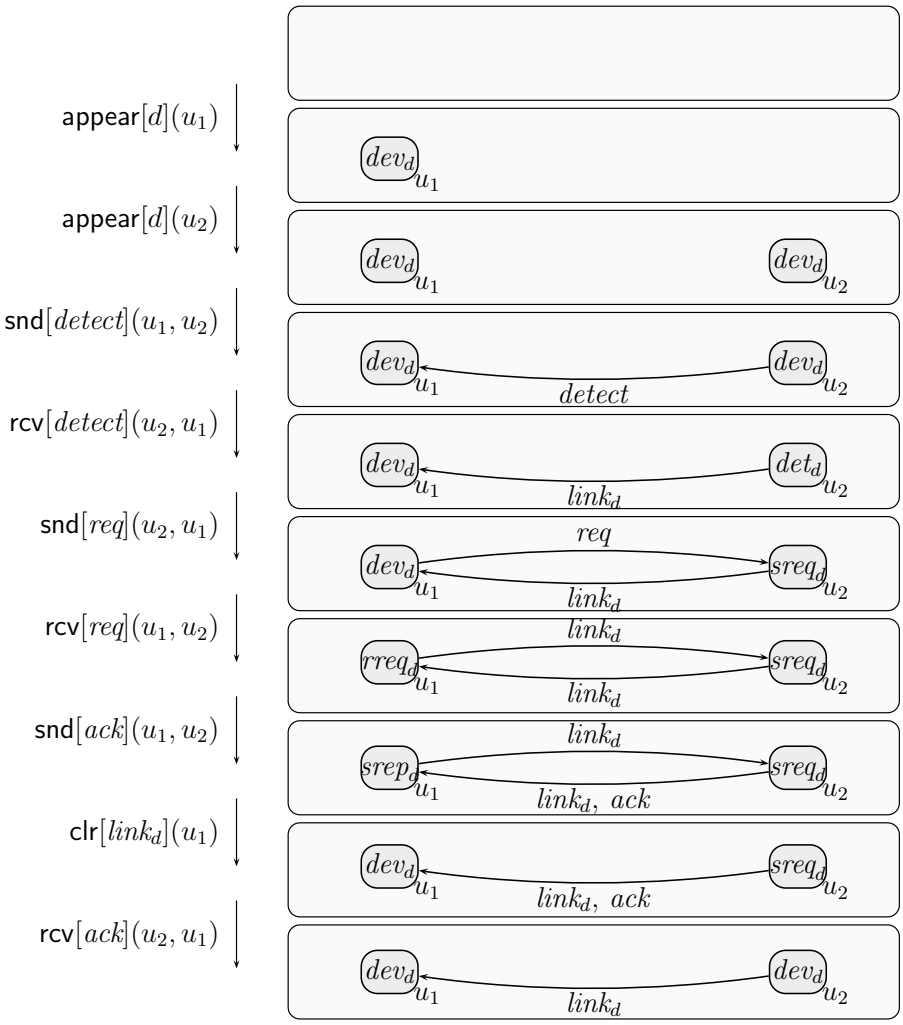


Figure 3.6: A prefix of a run in $[\![\mathsf{D}(\mathsf{C}_{\mathsf{ad}})]\!]$.

For a symbolic description of Structured Labelled Transition Systems, we devised a concise specification formalism called Dynamic Evolution Systems (DES). In this formalism, a set of rules (consisting of a guard and a sequence of statements) determine the possible evolutions of the system. This formalism is clearly related to graph transformation rules [Roz97], where a rule is made up of two disjoint graphs (left-hand side and right-hand side). Intuitively, the system may evolve under a graph transformation rule if the left-hand side of the rule can be matched somewhere in the current snapshot of the system. The resulting snapshot is then given by applying the transformation of the left-hand side graph to the right-hand side graph on the just matched substructure of the snapshot. In fact, graph transformation systems (GTS) have gained a lot of attention for

specifying dynamic systems, for example in [KK06, HPR06a, BW07, SWJ08]. However, the usage of GTS typically require a non-trivial machinery for hypergraph morphism and rewriting. For the application of spotlight abstraction, one would have to additionally take care that the identities of spotlight processes are not blurred under the inherent graph morphism steps. Note that GTS typically do not preserve dangling links. We believe that DES can actually serve as a light-weight alternative to graph transformation systems, in particular if not the complete specification power and -comfort of GTS is required.

Another prominent mechanism for defining dynamic systems comes from the domain of algebraic specifications, namely the π-calculus [MPW92, Mil99]. The expressive power of π stems from from the possibility of passing *names* among agents. This feature allows to model dynamic changes in the communication structure. Since π has no native concept of process identities, the transfer from a π specification to SLTS or ETTS is not obvious, and hence not addressed for the reasons given above. We will compare our verification approach with techniques for the analysis of π-calculus specifications in Section 5.3.

The verification of system with an arbitrary number of processes is also addressed in the area of *parameterized systems* [CGB86, ZP04], which represent the parallel composition of a finite set of identical processes. The task is then to prove properties of a system $P(N)$ for any number N of processes. However, these systems typically assume a fixed topology, for example a linear ring or a tree topology, and the behaviour of some process either depends solely on its direct connection neighbours or on the local states of *all* N processes. In general, the representation of dynamic creation/destruction and a varying connection topology would require a non-natural encoding in parameterized systems.

Evolving link structures naturally occurs in the execution of *pointer programs*, where heap objects are allocated and deallocated at runtime and pointers among heap objects may represent a kind of connection topology. However, heap objects are typically passive objects and the behaviour of the overall system is determined by the program in the stack. In Dynamic Evolution Systems, the processes themselves exhibit active behaviour and there is no global control flow which determines the scheduling and interconnection of processes. Nevertheless, pointer programs can of course be used to model our addressed class of systems, and we will elaborate on related analysis techniques in Section 5.3.

For the DCS language, we observe a relation to *dynamic I/O automata* [AL01], which are an extension of classical input/output automata [LT89] by the possibility of creating new automata at runtime. There is however no basic notion of connectivity in these systems, but an automaton may rather modify the set of events to which it is sensible for. Also, DCS can be seen to be the very core

of the *Unified Modelling Language* (UML) [OMG01, OMG07] with respect to object creation and interlinking. That is, DCS abstracts from many high-level constructs like hierarchies in state-charts, class attributes, class inheritance, different types of associations, and the like (which can all be compiled into a low-level kernel UML language [DJPV02]). In fact, one motivation for the development of the DCS language was to better understand the difficulties regarding the dynamic aspects of UML, which we encountered while working on a verification framework for UML models [STMW04, MRS+05].

Chapter 4

Requirement Specification for DES

This chapter addresses the need for an appropriate language for specifying requirements of the considered class of systems. As already mentioned, we employ a temporal logic that allows us to reason over the behaviour of the system over the time. As the underlying semantical domain (SLTS) represents the behaviour of individual processes, the language will comprise logical variables to denote process identities. The syntax is given in Subsection 4.1.

We will define a new three-valued satisfaction relation in Subsection 4.2. This definition is in particular tailored for the evaluation of a specification in spotlight abstractions of structured labelled transition systems. However, we will observe that we always obtain a definite value for the evaluation in SLTSs induced by the *concrete* semantics of Dynamic Evolution Systems (cf. Def. 3.25).

Subsection 4.3 discusses certain intricacies that arise due to the presence of appearing and disappearing processes, and sketches the relation of the graphical language of Live Sequence Charts to our specification language. Further related work is considered in Subsection 4.4.

4.1 Syntax

The syntax of our specification language is a rather mild extension of the formula language defined in Section 3.2, namely we basically add the classical temporal operators X, G, F and U from the linear time logic [MP92]. This extension,

however, has a considerable impact on its expressiveness as we are now able to reason about the evolution of the system over time, and not only about isolated snapshots. In particular, we may formulate safety properties stating that a certain configuration of processes is not reachable in any snapshot, or liveness properties that require to eventually reach some desired configuration. Formally, the requirement specification logic is defined as follows.

Definition 4.1 (Specification Logic) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature. A* specification over $\mathcal{S}$ *is generated by the grammar*

$$\phi ::= \psi \mid \phi{=}1 \mid \mathsf{X}\,\phi \mid \mathsf{G}\,\phi \mid \mathsf{F}\,\phi \mid \phi_1 \,\mathsf{U}\, \phi_2$$

where $\psi \in \mathit{Forms}(\mathcal{P})$ is a formula.

The set of all specifications over $\mathcal{S}$ is denoted by $\mathit{Specs}_{\mathcal{S}}$. ♢

The variables of a specification $\phi \in \mathit{Specs}_{\mathcal{S}}$ are denoted by $\mathit{vars}(\phi)$ and are defined inductively as usual as $\mathit{vars}(\phi = 1) = \mathit{vars}(\mathsf{X}\,\phi) = \mathit{vars}(\mathsf{G}\,\phi) = \mathit{vars}(\mathsf{F}\,\phi) := \mathit{vars}(\phi)$, $\mathit{vars}(\phi_1 \,\mathsf{U}\, \phi_2) := \mathit{vars}(\phi_1) \cup \mathit{vars}(\phi_2)$, and $\mathit{vars}(\psi)$ for a formula $\psi \in \mathit{Forms}(\mathcal{P})$ as defined on page 27.

This requirement specification language in particular covers the complete (non-temporal) formula language from Definition 3.11. Thus we may query the state-, the link- and the evolution *predicates* of the system. Additionally, we may ask for the *definite satisfaction* of a (sub-)formula via '$\phi = 1$'. This construct will be used for spotlight abstraction refinement via logical implication in Section 6.1.2. The constructs $\mathsf{X}\,\phi$ and $\phi_1 \,\mathsf{U}\, \phi_2$ denote the standard *temporal* operators from LTL, stating that ϕ holds in the next state and stating that ϕ_1 holds until ϕ_2 finally holds, respectively. The *finally* operator 'F' requires that some property ϕ eventually holds, the *globally* operator 'G' requires that some property ϕ holds from now on forever.

As in the formula language, there is no explicit quantification mechanism in our specification logic. Intuitively, the truth value of a specification is the validity of its universal closure in the set of runs induced by the underlying structured labelled transition system, this means there is an implicit universal quantification.

Note that we in particular import the notion $\circledcirc x$, stating that some process is currently alive, from the formula language. By the following two abbreviations

$$\odot x := \neg \circledcirc x \wedge \mathsf{X}\, \circledcirc x \qquad\qquad \ominus x := \circledcirc x \wedge \mathsf{X} \neg \circledcirc x$$

we allow to reason about the *life-cycle* of a process, that is, we may state that some process is about to newly appear by $\odot x$ or that some process will now disappear by $\ominus x$.

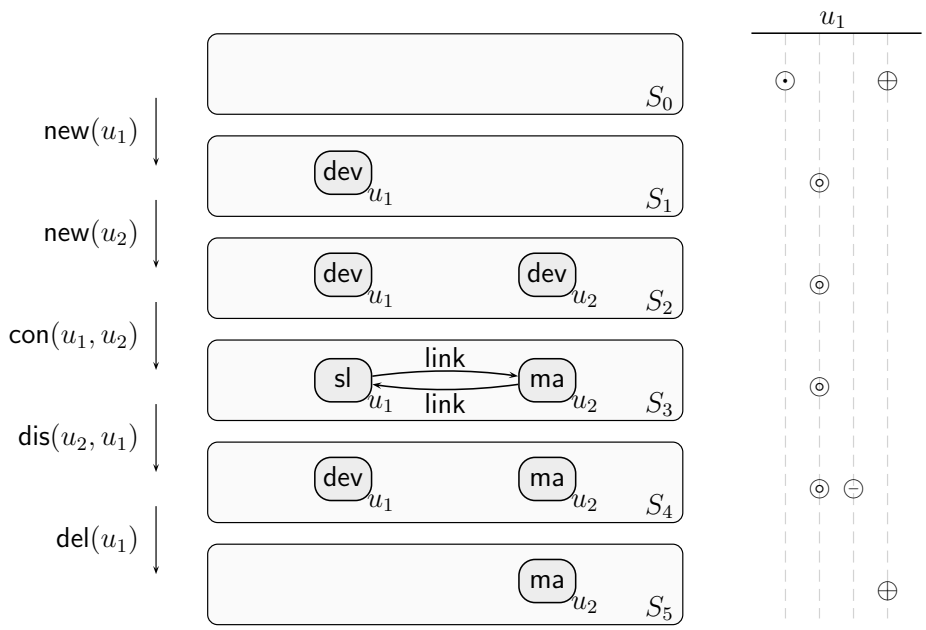


Figure 4.1: A prefix of a run in $[\![\mathsf{Ad}]\!]$, annotated with life-cycle properties.

We illustrate the concept of life-cycle queries and temporal operators on the adhoc networking system Ad. Figure 4.1 displays a prefix of the example run $\pi \in \mathit{Runs}([\![\mathsf{Ad}]\!])$ from page 40, but now each logical structure is annotated with the valid life-cycle properties for the identity u_1. That is, u_1 is dead and about to appear in S_0, is then alive from S_1 to S_4, disappears in S_4 and is dead in S_5.

Note that the proposed mechanisms to observe to birth and death of process may actually fail in certain situations. Consider for example the evolution rule

$$\mathsf{resurrect}(x) \bullet \circledcirc x \blacktriangleright \otimes x; \circledast x$$

which kills an existing processes and immediately creates it again. The death of x is not observable by '$\ominus x$' as x is alive in both the source and the target structure. To circumvent such problems, related specification formalism like VTL [YRSW06] or EvoCTL* [Wes08] use dedicated transition annotations which comprises the set of created and deleted process. Although we do not explicitly provide such an annotation, our specification language allows to observe such kind of "invisible" life-cycles by using atoms over evolution predicates $\mathcal{P}_E$, for example by stating '$\mathsf{F}\ \mathsf{resurrect}(x)$' to require that the process denoted by x finally resurrects.

We now consider a number of example specifications for our case study Ad.

1. *"A process is never both a master and a slave at the same time":*

$$\mathsf{G}\left(\ \neg(\mathsf{sl}(x) \wedge \mathsf{ma}(x))\ \right)$$

   This specification reasons purely about *state* predicates for a single variable x, stating that the two predicates 'sl' and 'ma' never become true simultaneously for any process. We expect this specification to hold for the Ad system.

2. *"A process will never become a slave":*

$$\mathsf{G}\left(\ \neg(\mathsf{sl}(x))\ \right)$$

   If we remove 'ma' from the previous specification, we require that the 'sl' predicate never becomes true. We expect a counterexample for this specification, which then witnesses how a process becomes a slave device.

3. *"A connection always established bidirectional links":*

$$\mathsf{G}\left(\ \mathsf{con}(x_1, x_2) \rightarrow (\mathsf{link}(x_1, x_2) \wedge \mathsf{link}(x_2, x_1))\ \right)$$

   This specification demonstrates how to reason about *evolution* predicates 'con' and *link* predicates 'link' of two logical variables. It guarantees a certain topological shape after each connection of any two processes.

4. *"A connected process does not disappear":*

$$\mathsf{G}\left(\ \mathsf{link}(x_1, x_2) \rightarrow \neg \ominus x_1\right)$$

   We also may combine predicates and *life-cycle* queries, stating that any process that is connected to some process will not disappear from the system.

5. *"Whenever some new process appears, it finally becomes a master or a slave":*

$$\mathsf{G}\left(\ \odot x \rightarrow \mathsf{F}\ (\mathsf{ma}(x) \vee \mathsf{sl}(x))\ \right)$$

   This requirement expresses a *liveness* property of some process, namely that whenever it appears, it eventually will connect to some other process (and hence become either a master of a slave device).

We will discuss further requirement specifications of different Dynamic Evolution Systems in the evaluation phase given in Chapter 7.

4.2 Three-Valued Semantics

Partly anticipating the subsequent chapter on abstraction techniques, we will define the satisfaction relation for the requirement specification language to (in particular) work on spotlight abstractions of Dynamic Evolution Systems. This means the definition is prepared to encounter the identity $\perp$ of the abstract process, both in the evaluation of state and link predicates in a logical structure and also as argument of an evolution ground atom on a transition label.

We will now shortly sketch the basic idea of the three-valued satisfaction relation, while the formal investigation follows in Chapter 5. Recall from the introduction that the abstract process serves as an over-approximative representation of any number of processes that are not in the spotlight. In general, this abstraction mechanism introduces runs in the abstract semantics that are not contained in the concrete semantics. However, as long as only spotlight processes are involved in the evolutions of the abstract run, the corresponding prefix is also a concrete run. In these cases, we can identify *definitive* violations of the specification also in an abstract run. However, as soon as the abstract process is involved in an evolution step, any subsequent violation may be spurious. The satisfaction relation accounts for this fact by distinguishing three possible outcomes, namely a definitive result '1' if the specification holds in the run, a definitive result '0' if the specification is violated in a concrete prefix of an run, and the indefinite result '1/2' in all other cases. Note that the definition can decide whether a prefix of a violating abstract run corresponds to a concrete prefix by making the temporal operators sensitive to the arguments of the observed evolution ground atoms. This intuition is formalised as follows.

Definition 4.2 (Specification Evaluation) *Let $\mathcal{S}$ be a signature, $I \subseteq Id^{\perp}$ a set of identities, $\mathsf{T} \in \mathcal{T}_{\mathcal{S}}(I)$ a SLTS over $\mathcal{S}$ and I, and $\phi \in Specs_{\mathcal{S}}$ a specification over $\mathcal{S}$. The* evaluation of ϕ *in a run*

$$\pi = ((L_i, S_i))_{i \in \mathbb{N}_0} \in Runs(\mathsf{T})$$

under a valuation $\mathcal{V} \in Vals_I(vars(\phi))$ is defined as $\pi[\phi]^0(\mathcal{V}) \in \mathbb{B}_3$ where

$$\begin{aligned}
\pi[\mathbf{tt}]^i(\mathcal{V}) &:= 1 \\
\pi[\circledcirc x]^i(\mathcal{V}) &:= S_i[\circledcirc x](\mathcal{V}) \\
\pi[x_1 = x_2]^i(\mathcal{V}) &:= S_i[x_1 = x_2](\mathcal{V}) \\
\pi[p_s(x)]^i(\mathcal{V}) &:= S_i[p_s(x)](\mathcal{V}) \\
\pi[p_l(x_1, x_2)]^i(\mathcal{V}) &:= S_i[p_l(x_1, x_2)](\mathcal{V}) \\
\pi[p_e(x_1, \ldots, x_{k_{p_e}})]^i(\mathcal{V}) &:= L_i = p_e(\mathcal{V}(x_1), \ldots, \mathcal{V}(x_{k_{p_e}}))
\end{aligned}$$

$$
\begin{aligned}
\pi[\neg\phi]^i(\mathcal{V}) &:= \neg\, \pi[\phi]^i(\mathcal{V}) \\
\pi[\phi_1 \wedge \phi_2]^i(\mathcal{V}) &:= \pi[\phi_1]^i(\mathcal{V}) \wedge \pi[\phi_2]^i(\mathcal{V}) \\
\pi[\phi{=}1]^i(\mathcal{V}) &:= \pi[\phi]^i(\mathcal{V}) = 1 \\
\pi[\mathsf{X}\,\phi]^i(\mathcal{V}) &:= \begin{cases} 1 & \textit{if } \pi[\phi]^{i+1}(\mathcal{V}) = 1 \\ 0 & \textit{if } \pi[\phi]^{i+1}(\mathcal{V}) = 0 \wedge \bot \notin A(L_{i+1}) \\ 1/2 & \textit{else} \end{cases} \\
\pi[\mathsf{G}\,\phi]^i(\mathcal{V}) &:= \begin{cases} 1 & \textit{if } \forall\, k \geq i : \pi[\phi]^k(\mathcal{V}) = 1 \\ 0 & \textit{if } \exists\, k \geq i : \big(\pi[\phi_1]^k(\mathcal{V}) = 0 \;\wedge \\ & \qquad \forall\, j \in \{i, \ldots, k\} : \bot \notin A(L_j)\big) \\ 1/2 & \textit{else} \end{cases} \\
\pi[\mathsf{F}\,\phi]^i(\mathcal{V}) &:= \begin{cases} 1 & \textit{if } \exists\, k \geq i : \pi[\phi]^k(\mathcal{V}) = 1 \\ 0 & \textit{if } \forall\, k \geq i : \big(\pi[\phi]^k(\mathcal{V}) = 0 \;\wedge \bot \notin A(L_k)\big) \\ 1/2 & \textit{else} \end{cases} \\
\pi[\phi_1\ \mathsf{U}\ \phi_2]^i(\mathcal{V}) &:= \begin{cases} 1 & \textit{if } \exists\, k \geq i : \big(\pi[\phi_2]^k(\mathcal{V}) = 1 \;\wedge \\ & \qquad \forall\, j \in \{i \ldots k\} : \pi[\phi_1]^j(\mathcal{V}) = 1\big) \\ 0 & \textit{if } \exists\, k \geq i : \big(\pi[\phi_1]^k(\mathcal{V}) = 0 \;\wedge \\ & \qquad \forall\, j \in \{i \ldots k\} : \pi[\phi_2]^j(\mathcal{V}) = 0 \wedge \bot \notin A(L_j)\big) \\ & \qquad\quad \vee\ \forall\, k \geq i : \pi[\phi_2]^k(\mathcal{V}) = 0 \wedge \bot \notin A(L_k) \\ 1/2 & \textit{else} \end{cases}
\end{aligned}
$$

where $p_s \in \mathcal{P}_S$, $p_l \in \mathcal{P}_L$, $p_e \in \mathcal{P}_E$, *and* $i \in \mathbb{N}_0$. $\diamondsuit$

A 'globally' formula '$\mathsf{G}\,\phi$' becomes 1 if ϕ globally holds, and becomes 0 if ϕ is violated somewhere in the future *and* the abstract identity $\bot$ is not involved until this point of violation. In all other cases, the 'globally' formula evaluates indefinite. Analogously, a 'finally' formula '$\mathsf{F}\,\phi$' becomes 1 if ϕ eventually holds, and becomes '0' if ϕ never holds *and* the abstract identity $\bot$ is never involved in any evolution. In all other cases, the 'finally' formula evaluates indefinite. The same principle is used for the evaluation of an 'until' specification. These definitions fit with our intuition that a counterexample can only be trusted if it occurs among the set of concrete identities. As soon as behaviour of the abstract identity yields a violating run, one remains inconclusive because the run may be spurious.

We follow the approach of [AS85] in order to formally distinguish between *safety* and *liveness* specifications (cf. Sect. 2.3). We call ϕ a safety specification if and only if every infinite run that does not satisfy ϕ contains a finite prefix

which cannot be extended to an infinite run satisfying the specification. On the other hand, a violation of a liveness specification can not be witnessed by a finite run, that is, every finite prefix of a run can be extended to an infinite run satisfying the specification. We formalise this intuition as follows.

Definition 4.3 (Safety and Liveness) *Let $\phi \in Specs_{\mathcal{S}}$ be a specification over a signature $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$. We call ϕ a* safety specification *iff*

$$\begin{aligned} &\forall\, \pi \in (E_{\mathcal{S}})^{\omega} \,.\, \pi[\phi]^0(\mathcal{V}) = 0 \implies \\ &\quad \exists\, p \in \mathit{Prefixes}(\pi) \,.\, (\forall\, \pi' \in (E_{\mathcal{S}})^{\omega} \,.\, p\pi'[\phi]^0(\mathcal{V}) = 0) \end{aligned}$$

and a liveness specification *iff*

$$\forall\, p \in (E_{\mathcal{S}})^{*}\, \exists\, \pi \in (E_{\mathcal{S}})^{\omega} \,.\, p\pi[\phi]^0(\mathcal{V}) = 1$$

where $E_{\mathcal{S}} = (\mathit{GroundAtoms}_{Id}(\mathcal{P}_E) \times \mathit{Strucs}^2_{\mathcal{S}}(Id))$. ♢

Note that this definition does not give a procedure to effectively decide whether a specification has safety or liveness character. A mechanical way to distinguish between safety and liveness via a translation to Büchi automata has been proposed in [AS87]. Typical specifications, however, can be syntactically identified, and a syntactic safety fragment of LTL has been proposed by [Sis94]. Further note that not every specification can be classified as being either safety or liveness. However, [AS85] shows that each specification can be decomposed into a safety and a liveness part such that we restrict our research to pure safety and liveness specifications in the following.

For linear time logic it is known that $\{\mathsf{X}, \mathsf{U}\}$ form an adequate set [HR00] of temporal connectives, because G and F can be defined in terms of the until operator as

$$\mathsf{F}\,\phi := \mathit{tt}\,\mathsf{U}\,\phi \quad \text{and} \quad \mathsf{G} := \neg\,\mathsf{F}\,\neg\phi.$$

In particular, F and G are duals of each other and X is dual with itself as $\neg\,\mathsf{X}\,\phi = \mathsf{X}\,\neg\phi$. While the F operator can also be defined via U in our specification language, the semantical distinction between concrete and abstract violations does not preserve the dualities of standard LTL. However, we are able to establish the following reasonable relations, stating that for each temporal operator a definitive violation entails a definitive violation of its dual, and a definitive satisfaction entails a possible satisfaction of its dual. In particular if a specification is known to evaluate to a definite value (cf. Rem. 4.7 below) then the classical dualities are preserved.

Remark 4.4 (Temporal Relations) *Let* $\mathsf{T} \in \mathcal{T}_{\mathcal{S}}(I)$ *a SLTS over signature* $\mathcal{S}$*,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification,* $\mathcal{V} \in \mathit{Vals}_{Id^{\perp}}(\mathit{vars}(\phi))$ *a valuation and* $\pi \in \mathit{Runs}(\mathsf{T})$ *a run. Then*

$$\begin{aligned} \pi[\mathsf{O}\phi]^i(\mathcal{V}) = 0 &\implies \pi[\neg\, \mathsf{O}^D \neg\phi]^i(\mathcal{V}) = 0 \\ \pi[\mathsf{O}\phi]^i(\mathcal{V}) = 1 &\implies \pi[\neg\, \mathsf{O}^D \neg\phi]^i(\mathcal{V}) \geq 1/2 \end{aligned}$$

for any position $i \in \mathbb{N}_0$ *and temporal operator* $\mathsf{O} \in \{\mathsf{F}, \mathsf{G}, \mathsf{X}\}$ *where* O^D *denotes its dual, defined as* $\mathsf{F}^D := \mathsf{G}$*,* $\mathsf{F}^D := \mathsf{G}$*,* $\mathsf{X}^D := \mathsf{X}$*.* ♢

Proof. The proof is given in the appendix (page 189). ■

We have defined and discussed the evaluation of a specification in a single run under a single valuation. We make now the transition to the general case, by defining the specification satisfaction of a structured labelled transition system as the minimal boolean value for all runs of the system under a given valuation.

Definition 4.5 (Specification Evaluation for SLTS) *Let* $\mathcal{S}$ *be a signature and* $I \subseteq Id^{\perp}$ *a set of identities.*

The evaluation of a specification $\phi \in \mathit{Specs}_{\mathcal{S}}$ *in a SLTS* $\mathsf{T} \in \mathcal{T}_{\mathcal{S}}(I)$ *under a valuation* $\mathcal{V} \in \mathit{Vals}_I(\mathit{vars}(\phi))$ *is defined as*

$$\mathsf{T}[\phi](\mathcal{V}) := \min\{\pi[\phi]^0(\mathcal{V}) \in \mathbb{B}_3 \mid \pi \in \mathit{Runs}(\mathsf{T})\}. \qquad ♢$$

When discussing the abstraction of Dynamic Evolution Systems in Section 5.2 we will establish a strong relation between the evaluation of a specification in the concrete and abstracted DES, namely that the abstract evaluation may only yield less precise results, but not wrong results. We already prepare this observation by establishing that the satisfaction of a requirements specification is weakly preserved under SLTS simulation according to Definition 3.16. The intuition is that the simulating transition system comprises less precise information than the original transition system. Hence if these information suffices to establish the satisfaction of a specification, then the original system will also satisfy the specification.

Lemma 4.6 (Simulation Preservation) *Let* $\mathcal{S}$ *be a signature,* $I_1, I_2 \subseteq Id$ *two sets of identities with* $I_2 \subseteq I_1$*,* $\mathsf{T}_1 \in \mathcal{T}_{\mathcal{S}}(I_1)$ *and* $\mathsf{T}_2 \in \mathcal{T}_{\mathcal{S}}(I_2^{\perp})$ *two structured labelled transition systems with* $\mathsf{T}_1 \preceq \mathsf{T}_2$*, and* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification. Then*

$$\mathsf{T}_2[\phi](e^{I_1}_{I_2}(\mathcal{V})) = 1 \implies \mathsf{T}_1[\phi](\mathcal{V}) = 1$$

for any valuation $\mathcal{V} \in \mathit{Vals}_{I_1}(\mathit{vars}(\phi))$*.* ♢

Proof. By Lemma 3.17. The proof is given in the appendix (page 189). ■

Definitive Evaluation

Note that a specification may only evaluate indefinite in the following cases:

Equality As the evaluation of equality '$x_1 = x_2$' is mapped to the evaluation of the corresponding equality expression (cf. Def. 3.11), we obtain the indefinite result iff both variables x_1 and x_2 denote the abstract identity $\perp$ (cf. Def. 3.12).

Predicates Also the evaluation of state and link predicates is mapped to the evaluation of the corresponding expressions (cf. Def. 3.11), thus the evolution becomes indefinite iff the interpretation of the corresponding predicate is indefinite in the logical structure (cf. Def. 3.12).

Life-cycle The abstract process is always considered to be possibly alive ($\circledcirc$).

Temporal operator The evaluation of a temporal formula becomes indefinite iff it is violated and there is an evolution within the violating part of the run where the abstract identity is involved.

We observe that two-valued structured labelled transition system over concrete identities *Id* always yield a definite evaluation of any specification. This is a direct consequence of the reasons for indefinite evaluations listed above, together with Remark 3.13 concerning the definite evaluations of formulas.

Remark 4.7 (Definite Specification Evaluation) *Let $\mathcal{S}$ be a signature, $\phi \in Specs_{\mathcal{S}}$ a specification, $I \subseteq Id$ a set of identities and $\mathsf{T} \in \mathcal{T}_{\mathcal{S}}(I)$ a structured labelled transition system. If T is two-valued, then ϕ always evaluates definite, that is,*

$$\mathsf{T} \in \mathcal{T}^2_{\mathcal{S}}(I) \implies \forall\, \pi \in Runs(\mathsf{T}) : \pi[\phi]^0(\mathcal{V}) \in \mathbb{B}$$

for any valuation $\mathcal{V} \in Vals_I(vars(\phi))$. ♢

Satisfaction Relation for Dynamic Evolution Systems

To conclude the investigation of the formal syntax and semantics of our requirement specification language, it remains to lift the satisfaction relation the DES language. As we are also interested in liveness properties of DES systems, we consider a set of compassion constraints (strong fairness) [Kwi89, PS08] for a set of identities F and a given set of evolution rules. Intuitively, these constraints ensure that an evolution that is enabled for F infinitely often is also executed infinitely often. By this we eliminate trivial violations of liveness properties by starvation. As we parametrise the fairness constraint by a given set of identities

we are in the end able to realise the idea of [Wes08] that spotlight abstraction is able to preserve local liveness properties within the spotlight part of the abstraction.

Definition 4.8 (Fair Runs) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$*, and* $I, F \subseteq Id$ *two sets of identities with* $F \subseteq I$*.*

A run $\pi = ((L_i, S_i))_{i \in \mathbb{N}_0} \in Runs([\![\mathsf{D}, S]\!]_I)$ *is* F-fair *if*

$$\forall\, r = (name, guard, stm) \in \mathsf{D}, \mathcal{V} \in Vals_F(vars(name)) : \\ \pi[guard, \mathcal{V}] = \infty \;\rightarrow\; \pi[name, stm, \mathcal{V}] = \infty$$

where

$$\pi[guard, \mathcal{V}] := |\{i \in \mathbb{N}_0 \mid \pi[guard]^i(\mathcal{V}) = 1\}| \\ \pi[name, stm, \mathcal{V}] := |\{i \in \mathbb{N}_0 \mid \pi[name]^i(\mathcal{V}) = 1 \wedge S_i = S_{i-1}\langle stm \rangle(\mathcal{V})\}|$$

denote how many times the rule r *is enabled and executed in* π*, respectively.*

The set of all F*-fair runs of* D_S *is denoted by* $FairRuns_F([\![\mathsf{D}, S]\!]_I)$*.* ♢

Note that the set F of fair process identities occur as the range of the considered valuation functions in the definition above. Hence, the fair evaluation of a specification in a DES under a given valuation function considers all runs where the process identities in the range of the valuation function are treated fair.

Definition 4.9 (Fair Specification Evaluation) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $I \subseteq Id$ *a set of identities, and* $S \in Strucs_{\mathcal{S}}(I)$ *an initial snapshot.*

The fair evaluation of a specification $\phi \in Specs_{\mathcal{S}}$ under a valuation $\mathcal{V} \in Vals_I(vars(\phi))$ *is defined as*

$$[\![\mathsf{D}, S]\!]_I[\phi](\mathcal{V}) := \min\{\pi[\phi]^0(\mathcal{V}) \in \mathbb{B} \mid \pi \in FairRuns_{\mathsf{ran}(\mathcal{V})}([\![\mathsf{D}, S]\!]_I)\}.$$ ♢

To evaluate a specification under the implicit universal quantification we evaluate it under all possible valuation of the logical variables of the specification. Recall that the concrete semantics of DES yields a two-valued structured labelled transition systems over (a subset of) Id (cf. Def. 3.25), thus by Remark 4.7 this evaluation always yields a definitive value.

Definition 4.10 (Specification Evaluation for DES) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $I \subseteq Id$ *a set of identities,* $S \in Strucs_{\mathcal{S}}(I)$ *an initial snapshot, and* $\phi \in Specs_{\mathcal{S}}$ *a specification.*

The evaluation of ϕ in D and S under I *is defined as*

$$\mathsf{D}^I_S[\![\phi]\!] \;:=\; \min\{\, [\![\mathsf{D}, S]\!]_I[\phi](\mathcal{V}) \in \mathbb{B} \mid \mathcal{V} \in \mathit{Vals}_I(\mathit{vars}(\phi))\}$$ ♢

For convenience, we write $\mathsf{D}_S \models_I \phi$ if $\mathsf{D}^I_S[\![\phi]\!] = 1$, and $\mathsf{D}_S \not\models_I \phi$ else.

As before, we omit the initial logical structure S if it coincides with the empty structure $(\emptyset, \iota_e)$. Also, we leave out the subscript I in the case $I = \mathit{Id}$.

4.3 Specification Intricacies

When defining the evaluation of formulas (cf. Def. 3.12), we already observed that the presence of dead processes yields some subtle difficulties. In particular, one can draw no conclusion concerning the aliveness of processes from the fact that some predicate evaluates to 0 (see the discussion on page 29). These issues become even more important for the design of *temporal* specifications. Consider a simple system where some process may issue a request to some other process which has to finally acknowledge this request. In-between, the first process should not raise any failure. One attempt to formalise this requirement is

$$\mathsf{req}(x, y) \rightarrow \big(\neg\mathsf{fail}(x)\ \mathsf{U}\ \mathsf{ack}(y, x)\big).$$

Note that this specification holds in particular in a run where the process denoted by x disappears before the acknowledgement is send to him. It is questionable whether this behaviour is valid in the sense of the informal specification given above. One possibility to solve these difficulties it to explicitly require the aliveness of processes. For the running example, we may write

$$\mathsf{req}(x, y) \rightarrow \big((\neg\mathsf{fail}(x) \wedge \circledcirc x)\ \mathsf{U}\ \mathsf{ack}(y, x)\big)$$

which no longer satisfies the run sketched above. We will mention other approaches to resolve this specification problem, for which the term *premature disappearance* has been coined in [Wes08], in the section on related work below.

The explicit annotation of life-cycle predicates is in particular meaningful when compiling from high-level specification mechanisms into our specification logic. We will illustrate this idea in terms of the language of *Live Sequence Charts* (LSC) [DH01], which is a formally rigorous variant of the well-known Message Sequence Charts (MSC) as standardised by the ITU in [IT99]. From the MSC language, an LSC inherits the typical constructs to graphically specify scenarios, that is, lifelines to denote processes, condition boxes to require certain system properties, and message arrows to denote communication among

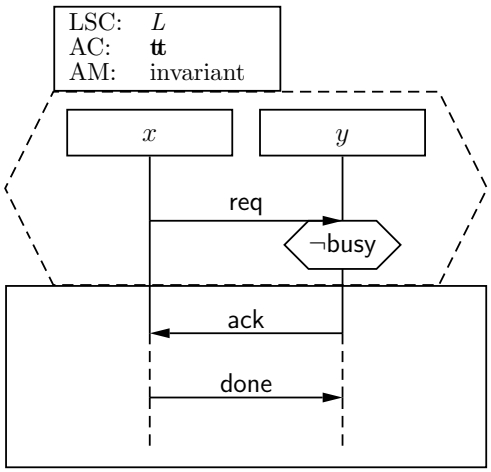


Figure 4.2: Specification in terms of a Live Sequence Chart.

processes. The main enhancement of LSCs in comparison to MSCs is that all LSC elements are equipped with a temperature, which may be *hot* or *cold*. This temperature annotation is used to define a formal semantics of an LSC in terms of a symbolic Büchi automaton [Klo03]. Intuitively, the resulting automaton accepts exactly those system runs that are supposed to satisfy the corresponding Live Sequence Chart.

In Figure 4.2 we give an extended variant of the specification discussed above in terms of an LSC. The LSC comprises three parts, namely a *header* (the box on the top), a *pre-chart* (the hexagonal with dashed lines), and a *main-chart* (the rectangle with solid lines). The header names the LSC with 'L' and provides an activation condition '**tt**' and an activation mode 'invariant'. An invariant LSC is supposed to hold in a run whenever the activation condition is satisfied. However, the *pre-chart* further restricts the activation of the LSC in the sense that the behaviour specified in the main-chart has only be observed after the behaviour given in the pre-chart has been observed.

The temperature of the whole LSC is hot, as indicated by the solid border of the main-chart. A hot LSC is also called universal LSC as it has to be satisfied by all runs of the system under consideration, in contrast to an cold (existential) LSC for which only one run has to satisfy the LSC. The temperature of the lifelines are hot (solid lines) up to the message arrow labelled by 'ack' after which they become cold (dashed lines). Hot lifelines require progress, that is, the subsequent element has finally to be observed in order to satisfy the LSC, while cold lifelines allow the runs to remain at this location forever. Summing up, the LSC specification in Figure 4.2 reads as

> "Whenever process x sends a message 'req' to process y, and y is not busy when receiving this message, then finally y answers with a 'ack' message to x. Afterwards, x may send 'done' back to y."

In a joint work [DTW06], we have observed that the LSC language in the sense of [Klo03] corresponds to a proper subset of first-order CTL*, that is, any LSC can be translated into this general temporal branching logic. Moreover, the fragment of universal LSCs translates into first-order LTL with outermost universal quantification, this means it can in particular be represented in terms of our specification logic as given in Definition 4.1. The basic structure of the resulting formulae is that of a nested chain of until expressions which correspond to the consecutive observation of the specified LSC elements. Following [KHP$^+$05], pre-charts can be treated by an implication between the pre-chart formula and the conjunction of the pre-chart formula with the main-chart formula.

Concerning the problem of premature disappearance as stated above, a natural interpretation of a *life*line is that the process denoted by this lifeline should be permanently alive until all required LSC elements have been observed. Note that this idea has already been mentioned in [Wes08]. Combining the translation schema from [DTW06] and the interpretation of a lifeline as described above, we obtain the following formula for the LSC from Figure 4.2

$$
\begin{aligned}
\phi(L) := \mathsf{G}\ \big(\ & (\mathsf{tt} \wedge \mathsf{req}(x,y) \wedge \neg\mathsf{busy}(y)) \rightarrow \\
& (\mathsf{tt} \wedge \mathsf{req}(x,y) \wedge \neg\mathsf{busy}(y) \wedge \circledcirc x \wedge \circledcirc y \wedge \\
& \quad \mathsf{X}\ (\ (\neg\mathsf{ack}(y,x) \wedge \circledcirc x \wedge \circledcirc y)\ \mathsf{U}\ (\ (\mathsf{ack}(y,x) \wedge \circledcirc x \wedge \circledcirc y) \wedge \\
& \quad\quad \mathsf{X}\ (\ (\neg\mathsf{done}(x,y) \wedge \circledcirc x \wedge \circledcirc y)\ \mathsf{W}\ (\ \mathsf{done}(x,y) \wedge \circledcirc x \wedge \circledcirc y)))))\ \big)
\end{aligned}
$$

which requires that after each satisfaction of the pre-chart, the required 'ack' message is observed. By the weak until operator, a 'done' answer may be observed later, but a run not exhibiting this message also satisfies the formula. The lifelines have been turned into alive predicates, such that any premature disappearance of participating processes yields a violation of the LSC.

4.4 Related Work

Our proposed specification language is inspired by a number of existing logics for specifying temporal requirements for systems with a varying number processes. One of the most basic approaches is the *Allocation Temporal Logic* ($\mathcal{A}\ell\ell$TL) [DRK02]. This logic reasons about pure allocation sequences, that is, only the birth, aliveness and death of processes is observable. Interestingly, the

model-checking problem for $\mathcal{A}\ell\ell$TL on HABAs (High-Level Allocational Büchi Automata) is decidable. This logic has been extended to $\mathcal{N}a\ell\ell$TL [DKR04] by providing means to reason about one fixed pointer name, that is, one may specify that two processes are connected via some pointer link. This allows for the specification of properties for e.g. linked data-structures in the heap. The *Evolution Temporal Logic* (VTL, but also known as ETL) [YRSW06] can be seen as a further extension of $\mathcal{N}a\ell\ell$TL, as it allows for the reasoning about general predicates concerning the underlying model. This in particular comprises local states of processes and pointer structures. Moreover, as the authors are interesting in the analysis of reachability in heap structures, they provide a transitive closure operator for binary predicates. In fact, VTL introduces the symbols $\odot$ and $\circledcirc$ to denote fresh and alive processes which we also use in our specification logic. In [BSTW06], we complemented the DCS language with a requirement specification logic called METT. This logic is basically VTL without transitive closure, but with explicit notations for specifying message communication.

Notably, all mentioned logics so far are *linear* time logics. EvoCTL* [Wes08] transfers the existing concepts to general branching time. EvoCTL* comes with a three-valued satisfaction relation in order to formally capture the problem of premature disappearance as discussed above. Note that this intended meaning of a three-valued result differs from our interpretation. While we use the indefinite result to indicate a possibly imprecise answer due to the employed abstraction, an indefinite evaluation of an EvoCTL* formula indicates a premature disappearance of a "relevant" process. Clearly, such a notion of relevance is not decidable in general, hence the satisfaction relation may conservatively become indefinite also for definite cases. However, there are syntactic criteria which ensure the definiteness of a specification evaluation.

A three-valued extension of temporal logic has been considered in [BG99] where a three-valued semantics of CTL in Kripke structures comprising partially unknown behaviour is given. A related tool is presented in [ECD$^+$03]. Refinement in this framework is addressed by [SG07], which employs the game-theoretic approach for CTL model-checking. These approaches however work on a given abstract transition system while our definitions and procedures are tailored to the specifics of spotlight abstractions of concrete systems.

Moreover, there exists several approaches to extend the *Object Constraints Language* (OCL) [OMG06] by temporal operators. The *Object-Based Temporal Logic* (BOTL) [DKR00] can be seen as a merge of OCL and $\mathcal{A}\ell\ell$TL. In [ZG03] and [BFS02], OCL expressions are used as basic propositions for the linear time logic and the μ-calculus, respectively. The work in [FM04] in particular addresses the combination of the temporal past operators with OCL. Seman-

tically, the proposed temporal extensions are typically limited to *one* life-cycle of the processes, that is, reappearance of processes is not considered and premature disappearance is treated by evaluating the overall specification to *false* (see the discussion above).

Besides these recent approaches, there is a large body of work in the domain of first-order modal logic, in particular by Barcan [Bar46], Kripke [Kri63] and Lewis [Lew68]. An excellent overview is given in the monograph of Fitting and Mendelssohn [FM98]. These works discuss in particular the impact of different choices regarding the underlying semantical domain, e.g. whether the individual worlds have a fixed or disjoint universe of objects. Note however that only the correspondents of the *globally* and *finally* operators are treated, but neither the general *until*- nor the *next* operator. In the joint work [BTW07b], we provide an detailed discussion how the research on first-order modal logic transfers to the specification of temporal requirements for the addressed class of systems.

The specification and verification of parameterised systems with respect to first-order temporal logic is addressed in [AJN+04] and [DFKL07]. The latter in particular treats monadic first-order logic. General aspects of first-order temporal specification logics are also discussed in a recent textbook [KM08].

To sum up, we observe that our specification logic is a proper subset of VTL (and thus of EvoCTL*), as we only provide outermost universal quantification of processes and we omit the transitive closure operator. We additionally allow for the observation of general *transitions* (or evolutions) by using atoms over evolution predicates $\mathcal{P}_E$.

Chapter 5

Analysis of Dynamic Evolution Systems

After having defined a computational model in Chapter 3 and a corresponding language for the specification of requirements in Chapter 4, we now address the question how to analyse whether a model satisfies its requirements.

We start in Section 5.1 by observing that this problem is in general not decidable, which is shown by a reduction from the reachability problem for Two-Counter Machines to Dynamic Evolution Systems under an unbounded number of processes. We address this problem by introducing finitary abstraction techniques for DES in the course of Section 5.2. Our abstraction strategy applies spotlight abstraction (cf. Sect. 5.2.1) and query reduction (cf. 5.2.2), which in combination allows for a reduction of both the model and the universally quantified specification to finite instances. We relate the verification results of the abstracted system to the original system by a new embedding theorem that is based on the information order of three-valued logic. This theorem combines the general soundness of the considered abstraction with its ability to preserve concrete counterexamples.

We conclude the investigation of the abstraction of Dynamic Evolution Systems by applying the presented technique to the running example and thereby motivate the need for *abstraction refinement* strategies. We provide an overview of related abstraction techniques in Section 5.3.

5.1 Undecidability Results

Given a Dynamic Evolution System D, an initial snapshot S and a specification ϕ, the *DES Model-Checking Problem* is to compute the satisfaction function $\mathsf{D}_S[\![\phi]\!]$ according to Def. 4.10. In the following we show that the DES Model-Checking Problem is undecidable, that is, there exists no algorithm that is able to compute the result of the satisfaction relation for arbitrary inputs of systems and specifications. This result is based on the observation that Dynamic Evolution Systems form a *turing-complete* language, such that any computation of a Turing Machine [Tur36] can also be performed by a corresponding Dynamic Evolution System. Turing-completeness of DES will be shown by encoding another turing-complete formalism, namely Two-Counter Machines [Min67], as a Dynamic Evolution System.

A Two-Counter Machine is a finite-state transition system equipped with two registers (or counters) where each register stores a natural number. A counter can be incremented, decremented and tested for zeroness. Without loss of generality, we assume that each decrement operation is guarded by a test for a positive value such that no counter becomes negative.

Definition 5.1 (Two-Counter Machine) *A Two-Counter Machine (2CM) is a tuple* $\mathsf{M} = (L, l_0, I)$ *where*

- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location, and*
- $I \subseteq L \times \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero}\} \times \{0,1\} \times L \times L$ *is a finite set of instructions.*

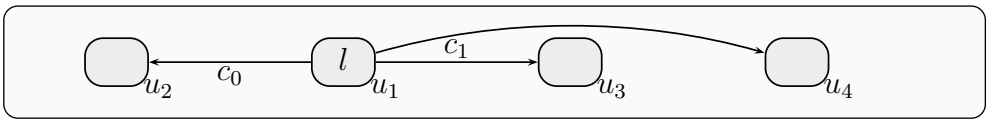
A configuration of M *is a tuple* $c = (l, K_0, K_1) \in L \times \mathbb{N}_0 \times \mathbb{N}_0$. *The configuration* $c' = (l', K_0', K_1')$ *of* M *is a* M*-successor of the configuration* $c = (l, K_0, K_1)$ *of* M, *denoted* $c \rightarrowtail_{\mathsf{M}} c'$, *if there is a instruction* $(l_1, \mathsf{op}, i, l_2^0, l_2^1) \in I$ *with*

- $l = l_1$
- $K'_{1-i} = K_{1-i}$
- $K_i' = \begin{cases} K_i + 1 & \text{if } \mathsf{op} = \mathsf{inc} \\ K_i - 1 & \text{if } \mathsf{op} = \mathsf{dec} \\ K_i & \text{if } \mathsf{op} = \mathsf{zero} \end{cases}$
- $l' = \begin{cases} l_2^0 & \text{if } \mathsf{op} \in \{\mathsf{inc}, \mathsf{dec}\} \text{ or } K_i = 0 \\ l_2^1 & \text{else} \end{cases}$

A location $l \in L$ *is reachable in* M, *denoted* $\mathsf{M} \rightarrowtail l$, *if* $(l_0, 0, 0) \rightarrowtail_{\mathsf{M}}^* (l, K_1, K_2)$ *for some* $K_1, K_2 \in \mathbb{N}_0$. ♢

It is known from [Min67] that the reachability problem for 2CMs is undecidable, that is, there is no algorithm that decides for arbitrary machines M and locations l whether $\mathsf{M} \rightarrowtail l$ or not. If we are able to encode (i.e. simulate) the behaviour of any 2CM by a corresponding Dynamic Evolution System, we transfer this result to the reachability problem for DES. As the model-checking problem in particular covers reachability questions, we have that the model-checking problem for DES is undecidable, too.

The encoding of a Two-Counter Machine to a DES is rather straightforward. The idea is to have one dedicated process which executes the instructions of the machine. The values for the two counters are stored by maintaining a corresponding number of connections to other "passive" processes. To this end, we use two links to distinguish between the two counters. For instance, the representation of a 2CM configuration $(l, 1, 2)$ in terms of a DES snapshot corresponds to a following structure:



The increment operation then translates to the creation of a new process, and the decrement operation to the removal of a corresponding connection link. Zero testing can be simulated by testing for the emptiness of the corresponding link. Formally, the translation is defined as follows.

Definition 5.2 (DES Encoding of 2CM) *Let* $\mathsf{M} = (L, l_0, I)$ *be a 2CM. We define the* encoding *of* M *as the Dynamic Evolution System* $\mathsf{D}(\mathsf{M})$ *over the signature* $(\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *with*

- *variables* $\mathcal{X} := \{x, x'\}$,
- *state predicates* $\mathcal{P}_S := L$,
- *link predicates* $\mathcal{P}_L := \{c_0, c_1\}$, *and*
- *evolution predicates* $\mathcal{P}_E = \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero}\}$

where $\mathsf{D}(\mathsf{M})$ *comprises the following set of rules:*

$$
\begin{aligned}
&\{\mathsf{inc}(x, x') \bullet l_1(x) \wedge \neg \circledcirc x' \blacktriangleright \circledast x'; c_i(x, x'); l_2^0!(x) \mid (l_1, \mathsf{inc}, i, l_2^0, l_2^1) \in I\} \cup \\
&\{\mathsf{dec}(x, x') \bullet l_1(x) \wedge c_i(x, x') \blacktriangleright \otimes x'; \neg c_i(x, x'); l_2^0!(x) \mid (l_1, \mathsf{dec}, i, l_2^0, l_2^1) \in I\} \cup \\
&\quad \{\mathsf{zero}(x) \bullet l_1(x) \wedge \neg c_i(x) \blacktriangleright l_2^0!(x), \\
&\quad\ \mathsf{zero}(x) \bullet l_1(x) \wedge c_i(x) \blacktriangleright l_2^1!(x) \mid (l_1, \mathsf{zero}, i, l_2^0, l_2^1) \in I\}.
\end{aligned}
$$

$\diamondsuit$

We establish the correctness of the translation by showing that a location in a Two-Counter Machine is reachable if and only if the active process in the encoding Dynamic Evolution System can obtain the corresponding location predicate. We query the reachability in the DES by stating that the corresponding predicate is globally not set. If this specification can be violated, we have that there exists a run which reaches a corresponding snapshot.

Lemma 5.3 (Correctness) *Let* $\mathsf{M} = (L, l_0, I)$ *be a Two-Counter Machine. Then*

$$\mathsf{M} \rightarrowtail l \iff \mathsf{D}(\mathsf{M})_S \not\models \mathsf{G}\,\neg l(x)$$

for $S := (\{u\}, \{l_0(u)\})$ *and any location* $l \in L$. ♢

Proof. Let $[\![\mathsf{D}(\mathsf{M}), S_0]\!] = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$. We define the *encoded configuration* of a logical structure as a function $cfg : \mathbf{S} \rightarrow L \times \mathbb{N}_0 \times \mathbb{N}_0$ with

$$cfg(S) = (l, K_0, K_1) \;:\Longleftrightarrow\; \exists\, u \in U : \iota(l)(u) = 1 \wedge K_0 = |S_{c_0}(u)| \wedge K_1 = |S_{c_1}(u)|$$

for some $S = (U, \iota) \in \mathbf{S}$ where $S_c(u) := \{u' \in U \mid \iota(c)((u, u')) = 1\}$ denotes the set of processes to which process $u \in U$ is connected to via c. The proof is based on the observation that relating the configuration of the Two-Counter Machine with the encoded configuration of the corresponding DES yields a bisimulation relation. The complete proof is given in the appendix (page 190). ■

Theorem 5.4 (DES model-checking is undecidable) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$, $S \in Strucs_{\mathcal{S}}(Id)$ *a logical structure and* $\phi \in Specs_{\mathcal{S}}$ *a specification over* $\mathcal{S}$.

There exists no general algorithm to compute the result of $\mathsf{D}_S[\![\phi]\!]$. ♢

Note that restricting the initial snapshot S_0 to the *empty snapshot* does not effect the undecidability result, as an additional evolution rule may be added in order to dynamically create processes which then executes the instructions.

We know that the DCS semantics uses only a subset of the DES language, in particular the construct to check for the emptiness of a channel is not available as a DCS action, and each channel comprises at most one identity at a time. Interestingly the expressiveness is not affected by these restrictions, and also the DCS language is turing-complete. The encoding however requires some more technical effort as the counter values have to be encoded by a linked list of processes with a corresponding length. The instructions are then carried out via communication, that is, the operations are forwarded to the head process of the corresponding list and a confirmation message is sent back to the main process. The formal encoding is given in the appendix on page 192.

Note that the encodings of a 2CM into the languages DES and DCS correspond to the notions of unbounded breadth and depth, respectively, in the π-calculus according to [Mey09]. Clearly, the correctness of the encodings rely on the fact that there is no upper bound on the maximal number of alive processes in the concrete semantics such that any value of an unbounded counter can be faithfully represented. In fact, as soon as the maximal number of process is bounded to some finite number, we obtain a finite state transition system for which the model-checking problem is decidable using standard techniques [CGP99]. But as the size of the resulting transition systems typically grows exponentially in the number of processes, it is may be helpful to employ abstraction techniques also for bounded approximations (cf. Def. 3.25).

5.2 Abstraction of Dynamic Systems

In this section we will formally define *spotlight abstraction* and *query reduction*. These techniques are able to reduce the analysis of a DES under any number of processes to its analysis under a typically rather small number of processes.

Spotlight abstraction will be introduced in Subsection 5.2.1 and query reduction in Subsection 5.2.2. While the first technique addresses the size of the induced structured labelled transition systems, the latter technique reduces the number of valuation to be considered according to the specification evaluation (cf. Def. 4.10) to a finite number. The combination of both techniques, which will be described in Subsection 5.2.3, then allows for the model-checking of the resulting finitary abstractions using standard procedures.

5.2.1 Spotlight Abstraction

The term *spotlight principle* has been coined by Wachter & Westphal in [WW07] and nicely resembles the intuition behind this abstraction technique, namely to

> *put a spotlight onto a subset of processes*
> *and abstract from those in the shadows.*

By this, the information concerning the spotlight processes are kept precise while any information about abstracted processes is dismissed by collapsing them into a single *abstract process* with identity $\perp$. More precisely, we have to distinguish between three classes of information, namely

1. information that affect spotlight processes only, i.e. interpretations of state or link predicates for spotlight processes,

2. information that affect abstracted processes only, i.e. interpretations of state or link predicates for processes outside of the spotlight, and
3. information that affect both spotlight and abstracted processes.

Clearly, the first class of information will be kept precise while the second class will be dismissed (i.e. set to $1/2$). The last class only applies to binary link predicates, that is, we have to define how links from spotlight to abstracted process and vice versa are represented in the abstract structure. We actually employ a rather coarse variant of the spotlight principle by setting all the corresponding interpretations to $1/2$. Hence we conservatively assume any link structure among concrete and abstracted processes to be possible. Note that the spotlight abstraction variants presented in [WW07, Wes08] preserve at least the *absence* of links from the spotlight into the shadows. We refrain from this feature as any impreciseness that may be introduced by dismissing this information during the abstraction is easily re-established by our refinement strategies. In return, we obtain an abstraction technique for which the abstract transition relation is almost trivially computable.

Definition 5.5 (Spotlight Abstraction) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $I' \subseteq Id$ a set of identities, and $S = (U, \iota) \in Strucs_{\mathcal{S}}(I')$ a logical structure.*

The spotlight abstraction of S under the set of spotlight identities $I \subseteq I'$ *is defined as*

$$\alpha_I(S) := (\ (U \cap I) \cup \{\bot\}, \alpha_I(\iota)\ ) \in Strucs_{\mathcal{S}}(I^{\bot})$$

where

$$\alpha_I(\iota)(p_s)(u) := \begin{cases} \iota(p_s)(u) & \text{if } u \in I \\ 1/2 & \text{else} \end{cases}$$

$$\alpha_I(\iota)(p_l)(u_1, u_2) := \begin{cases} \iota(p_l)(u_1, u_2) & \text{if } \{u_1, u_2\} \subseteq I \\ 1/2 & \text{else} \end{cases}$$

for predicates $p_s \in \mathcal{P}_S$ and $p_l \in \mathcal{P}_L$.

The spotlight abstraction of a ground atom $g \in GroundAtoms_{I'}(\mathcal{P})$ *under I, denoted $\alpha_I(g) \in GroundAtoms_{I^{\bot}}(\mathcal{P})$, replaces all identities from $I' \setminus I$ by the abstract identity, that is,*

$$\alpha_I(g) := g[I' \setminus I \mapsto \bot]. \qquad \diamondsuit$$

We illustrate spotlight abstraction for the logical structure D_{ad} (cf. page 23) from the running example in Figure 5.1. For a spotlight comprising only identity u_1 we end up with a structure comprising two processes, u_1 and $\bot$. All state

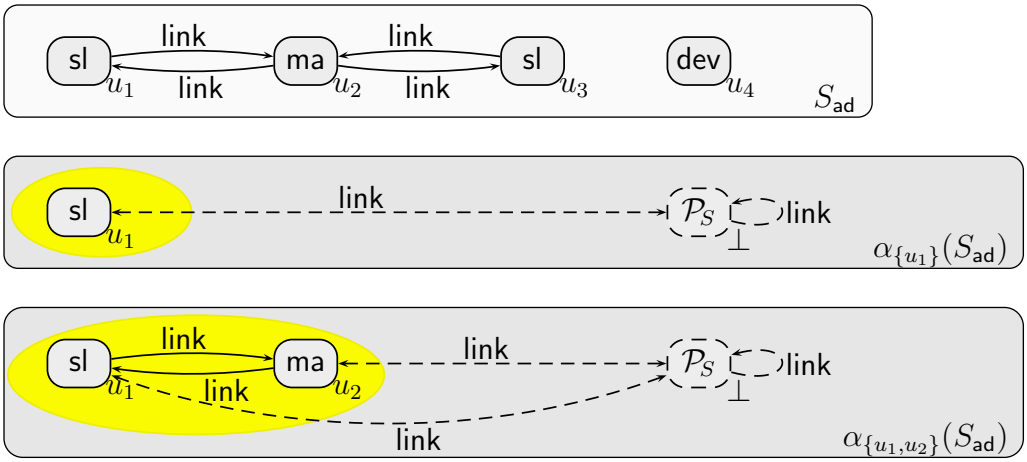


Figure 5.1: Spotlight Abstractions of S_{ad}

predicates $\mathcal{P}_S$ evaluate to $1/2$ for $\bot$ as well as the interpretation of the binary link predicate link for $(\bot, \bot)$. Intuitively, $\bot$ represents any number of concrete processes being in any state and having any inter-linking topology. Moreover, link evaluates to $1/2$ for $(\bot, u_1)$ and for $(u_1, \bot)$, representing any number of links between u_1 and processes $Id \setminus \{u_1\}$. The only information kept precise is the fact that u_1 is actually a slave. Note then when applying the abstraction under the spotlight $\{u_1, u_2\}$ we maintain a proper linking infrastructure between a master u_2 and a slave u_1. Typically, the abstraction becomes more precise when enlarging the spotlight. We will formalise this kind of increased precision in Section 6.1.1.

We will use spotlight abstraction to obtain a finite over-approximation of large structured labelled transition systems. To prepare the theorem that establishes the soundness of the abstraction we observe that spotlight abstraction actually provides us with a mechanism to obtain an embedded structure according to Definition 3.10. This observation in particular allow us to employ the preservation lemma (cf. Lemma 3.14) to show that a logical structure under spotlight abstraction weakly preserves the evaluation of formulas.

Lemma 5.6 (Embedding via Spotlight Abstraction) *Let $\mathcal{S}$ be a signature, $I' \subseteq Id$ a set of identities, and $S \in Strucs_{\mathcal{S}}(I')$ a logical structures.*

For any set of spotlight identities $I \subseteq I'$, the spotlight abstraction of S under I yields an embedded structure*, that is,*

$$S \Subset \alpha_I(S) \qquad \diamondsuit$$

Proof. The proof is given in the appendix (page 193). ■

The major benefit of the employed variant of spotlight abstraction is that the representation of the abstract process is *stateless* in the sense that the configuration of $\perp$ is independent of the configuration of the processes outside the spotlight. Thus, in the resulting abstract SLTS the configuration of the abstract process will be the same in every snapshot. Clearly, the drawback of this coarse abstraction is a large amount of spurious behaviour that we will eliminate by our refinement approaches to be described in Chapter 6. The advantage however is that we obtain the abstract transition system by a simple syntactical modification of the statements. The general principle is to ignore the effect of statements that affect the abstract process. Note that this in particular enables a straight-forward tool implementation of the abstraction technique, that is, the abstract semantics can be directly expressed in any standard programming language. We will present our implementation of spotlight abstraction and refinement in Section 7.1.

Definition 5.7 (Abstract Statement Execution) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, and $I \subseteq Id$ a set of spotlight identities.*

The abstract execution of a statement *is a function*

$$\cdot\langle\cdot\rangle^\sharp(\cdot) : Strucs_{\mathcal{S}}(I^\perp) \times Stms_{\mathcal{S}} \times Vals_{I^\perp}(\mathcal{X}) \to Strucs_{\mathcal{S}}(I^\perp)$$

which is defined inductively as

$$S\langle stm_1; stm_2\rangle^\sharp(\mathcal{V}) := S\langle stm_1\rangle^\sharp(\mathcal{V})\langle stm_2\rangle^\sharp(\mathcal{V})$$

$$S\langle stm(x_1,\ldots,x_k)\rangle^\sharp(\mathcal{V}) := \begin{cases} S\langle stm(x_1,\ldots,x_k)\rangle(\mathcal{V}) & \text{if } \{\mathcal{V}(x_1),\ldots,\mathcal{V}(x_k)\} \subseteq I \\ S & \text{else} \end{cases}$$

where $stm_1, stm_2, stm(x_1,\ldots,x_k) \in Stms_{\mathcal{S}}$ according to Def. 3.18. ♢

With this abstract version of executing a sequence of statements we directly obtain a notion of *abstract evolution*. The only two differences to the (concrete) evolution as defined in Def. 3.21 is on the one hand that the guard only needs to be *possibly* satisfied in the source structure, and on the other hand that we employ the abstract statement execution to obtain the target structure.

Definition 5.8 (Abstract Evolution) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $r = (name, guard, stm) \in EvoRule_{\mathcal{S}}$ an evolution rule over $\mathcal{S}$, and $I \subseteq Id$ a set of spotlight identities.*

Then r enables the abstract evolution of the logical structure $S \in Strucs_{\mathcal{S}}(I^\perp)$ into the logical structure $S' \in Strucs_{\mathcal{S}}(I^\perp)$ via the evolution ground atom $g \in GroundAtoms_{I^\perp}(\mathcal{P}_E)$ under I, *denoted*

$$S \xrightarrow[r,I]{g}{}^\sharp S'$$

if there is a valuation $\mathcal{V} \in \mathit{Vals}_{I^\perp}(\mathit{vars}(\mathit{name}))$ *such that the following three conditions are satisfied:*

1. *The source structure* S *possibly satisfies the guard under* $\mathcal{V}$*, that is,*

$$S[\![\mathit{guard}]\!](\mathcal{V}) \geq 1/2.$$

2. *The rule name under* $\mathcal{V}$ *yields the evolution ground atom* g*, that is,*

$$\mathit{name}[\mathcal{V}] = g.$$

3. *Abstractly executing the statements stm on* S *yields the target structure* S'*, that is,*

$$S' = S\langle\!|\mathit{stm}|\!\rangle^\sharp(\mathcal{V}).$$ ♢

We show that the abstract execution of statements preserves the effects of spotlight abstraction. This means each concrete evolution can be reflected by an abstract evolution of the corresponding spotlight abstractions of the source and target structures. This observation is the key to establish the weak preservation of the specification evaluation in Theorem 5.12 below.

Lemma 5.9 (Abstract Execution and Evolution) *Let* $\mathcal{S}$ *be a signature,* $I' \subseteq \mathit{Id}$ *a set of identities, and* $S, S' \in \mathit{Strucs}_{\mathcal{S}}(I')$ *two logical structures with*

$$S \xrightarrow[r,\, I']{g} S'$$

by some evolution rule $r \in \mathit{EvoRule}_{\mathcal{S}}$*. Then*

$$\alpha_I(S) \xrightarrow[r,\, I]{\alpha_I(g)}{}^\sharp\ \alpha_I(S')$$

for any set of spotlight identities $I \subseteq I'$*.* ♢

Proof. The proof is given in the appendix (page 194). ■

The *abstract semantics* of a Dynamic Evolution System is defined similar to the concrete semantics by collecting evolving structures in a structured labelled transition system. The states now become *three-valued* logical structures, the spotlight abstraction is applied to the initial states, and the abstract evolution according to Definition 5.8 is used to define the transition relation.

Definition 5.10 (Abstract Semantics of DES) *Let $\mathcal{S}$ be a signature, $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ a Dynamic Evolution System over $\mathcal{S}$, $I \subseteq Id$ a set of spotlight identities, and $S \in Strucs_{\mathcal{S}}(Id)$ a logical structure.*

The abstract semantics of D and S under I, denoted $[\![\mathsf{D}, S]\!]^{\sharp}_{I}$, is the structured labelled transition system $(\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow) \in \mathcal{T}_{\mathcal{S}}(I^{\perp})$ where

- *the states are three-valued logical structures over $\mathcal{S}$ and $I^{\perp}$, that is,*

$$\mathbf{S} := Strucs^{3}_{\mathcal{S}}(I^{\perp}),$$

- *the given structure S determines the set of initial states, that is,*

$$\mathbf{S}_0 := \{\alpha_I(\sigma(S)) \in Strucs_{\mathcal{S}}(I^{\perp}) \mid \sigma \in \Sigma_I\}$$

- *the labels are evolution ground atoms over $\mathcal{P}_E$ and $I^{\perp}$, that is,*

$$\mathbf{L} := GroundAtoms_{I^{\perp}}(\mathcal{P}_E),$$

- *and the abstract evolution determines the transition relation, that is,*

$$\rightarrow := \{(S, g, S') \in \mathbf{S} \times \mathbf{L} \times \mathbf{S} \mid S \xrightarrow[\mathsf{D}, I]{g}{}^{\sharp} S'\}. \qquad \diamondsuit$$

As for the concrete DES semantics, we will omit the initial structure S if it coincides with the empty structure $(\emptyset, \iota_e)$. For the adhoc networking case study, we present a prefix of a run of the abstract transition system for D_{ad} with only one concrete identity 'u_1' in the spotlight. In the following run $\pi^{\sharp} \in Runs([\![\mathsf{Ad}]\!]^{\sharp}_{\{u_1\}})$, the process u appears and connects with the abstract process.

$$\begin{aligned}
\pi^{\sharp} = \; & (\{\perp\}, (\{\}, \\
& \quad \{\mathsf{dev}(\perp), \mathsf{sl}(\perp), \mathsf{ma}(\perp), \mathsf{link}(\perp,\perp), \mathsf{link}(u_1,\perp), \mathsf{link}(\perp,u_1)\})), \\
(\mathsf{new}(u_1), \; & (\{u_1, \perp\}, (\{\mathsf{dev}(u_1)\}, \\
& \quad \{\mathsf{dev}(\perp), \mathsf{sl}(\perp), \mathsf{ma}(\perp), \mathsf{link}(\perp,\perp), \mathsf{link}(u_1,\perp), \mathsf{link}(\perp,u_1)\})), \\
(\mathsf{con}(u_1,\perp), \; & (\{u_1, \perp\}, (\{\mathsf{sl}(u_1)\}, \\
& \quad \{\mathsf{dev}(\perp), \mathsf{sl}(\perp), \mathsf{ma}(\perp), \mathsf{link}(\perp,\perp), \mathsf{link}(u_1,\perp), \mathsf{link}(\perp,u_1)\})),
\end{aligned}$$

Using the graphical notation for logical structures as introduced on page 23, we show an illustration of these evolution steps in Figure 5.2. As a consequence of Definition 5.10, the abstract process is present in every snapshot, and the interpretation of any predicate involving '$\perp$' becomes the indefinite value '$1/2$'. In fact, the set of ground atoms representing the indefinite values are identical in each snapshot, only the predicate interpretation that solely affect the spotlight process yield definitive values and vary over the time.

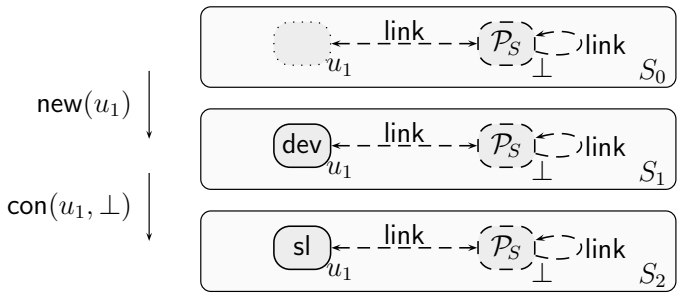


Figure 5.2: A prefix of a run of $[\![\mathsf{Ad}]\!]^\sharp_{\{u_1\}}$.

For the overall understanding of the abstraction principle and its refinement, it is important to see that a spotlight process can interact with the abstract process just like it would interact which some concrete process that has now been shadowed by the abstraction. However, the effects of these interactions are only observable within the spotlight. In particular, interactions that solely involve the abstract process will no longer be visible in the abstract semantics, that is, large portions of concrete runs are folded away by the employed abstraction.

The notion of a fair run in terms of compassion constraints as introduced in Definition 4.8 directly transfers to runs in the abstract transition system. In particular, we obtain the abstract fair specification evaluation analogously to Definition 4.9 by considering all *spotlight processes* to be fair. Note that this preservation of local fairness constraints for spotlight processes is a strong feature of the spotlight abstraction principle.

Definition 5.11 (Abstract Fair Specification Evaluation) *Let* $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_\mathcal{S}(\mathit{Id})$ *a logical structure, and* $I \subseteq \mathit{Id}$ *a set of spotlight identities.*

The abstract fair evaluation of a specification $\phi \in \mathit{Specs}_\mathcal{S}$ under a valuation $\mathcal{V} \in \mathit{Vals}_{I^\perp}(\mathit{vars}(\phi))$ *is defined as*

$$[\![\mathsf{D}, S]\!]^\sharp_I[\phi](\mathcal{V}) := \min\{\pi[\phi]^0(\mathcal{V}) \in \mathbb{B}_3 \mid \pi \in \mathit{FairRuns}_I([\![\mathsf{D}, S]\!]^\sharp_I)\} \qquad \diamondsuit$$

There is a clear demand for a strong correspondence between the concrete and the abstract semantics of a DES. Typically, one requires that any specification that is satisfied in the abstract system is also satisfied in the concrete, such that one does not obtain "false positives". Interestingly, the heterogeneous character of spotlight abstraction allows us to identify cases where also the opposite direction of implication holds, namely where the violation of a specification in the abstract system is also a violation of the concrete system [Tob08]. Intuitively, these are those cases where the violation happens completely within the

spotlight part of the abstraction. Together with the three-valued definition of specification satisfaction we obtain the following embedding theorem.

Theorem 5.12 (Specification Embedding) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $I' \subseteq Id$ *a set of identities,* $S \in Strucs_{\mathcal{S}}(I')$ *a logical structure, and* $\phi \in Specs_{\mathcal{S}}$ *a specification.*

Then for any valuation $\mathcal{V} \in Vals_{I'}(vars(\phi))$*, we have*

$$[\![\mathsf{D}, S]\!]_{I'}[\phi](\mathcal{V}) \;\sqsubseteq\; [\![\mathsf{D}, S]\!]^{\sharp}_{I}[\phi](\mathcal{V})$$

where $I := \mathsf{ran}(\mathcal{V})$ *determines the content of the spotlight.* ♢

Proof. The case for $[\![\mathsf{D}, S]\!]^{\sharp}_{I}[\phi](\mathcal{V}) = 1$ can be shown by establishing a simulation relation $[\![\mathsf{D}, S]\!]_{I'} \preceq [\![\mathsf{D}, S]\!]^{\sharp}_{I}$. The case for $[\![\mathsf{D}, S]\!]^{\sharp}_{I}[\phi](\mathcal{V}) = 0$ follows by the three-valued semantics of the specification satisfaction in combination with the locality of evolutions. The proof is given in the appendix (page 194). ■

Instantiating the above theorem with $I' = Id$ and $S = (\emptyset, \iota_e)$ yields

$$[\![\mathsf{D}]\!][\phi](\mathcal{V}) \;\sqsubseteq\; [\![\mathsf{D}]\!]^{\sharp}_{I}[\phi](\mathcal{V})$$

Thus whenever ϕ evaluates to 1 (resp. 0) in the abstract semantics $[\![\mathsf{D}]\!]^{\sharp}_{I}$ under $\mathcal{V}$, it also evaluates to 1 (resp. 0) in the concrete semantics $[\![\mathsf{D}]\!]$ under the same valuation $\mathcal{V}$. If ϕ evaluates to $1/2$ in $[\![\mathsf{D}]\!]^{\sharp}_{I}$, one remains inconclusive. This means the evaluation of the specification satisfaction in the abstract semantics may give less information than in the concrete, but it never gives any wrong information. In particular, we may obtain concrete counterexamples directly in the abstract semantics of DESs, namely if they occur within the spotlight.

So far we have not required that the set of spotlight identities I is actually a finite set. However, we easily obtain the following remark for finite spotlights.

Remark 5.13 (Finite Spotlight) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in Strucs_{\mathcal{S}}(Id)$ *a logical structure, and* $I \subset Id$ *a finite set of spotlight identities. Then* $[\![\mathsf{D}, S]\!]^{\sharp}_{I}$ *is a finite SLTS.* ♢

5.2.2 Query Reduction

Theorem 5.12 allows us to (approximatively) reduce the verification of large (or even infinite) state system to the verification of a finite state system, however yet only for a single valuation of the variables in the requirement specification. Recalling Definition 4.10 concerning the specification evaluation, we observe

that we actually have to analyse the specification under all possible valuations of the variables to identities from *Id*, yielding a potentially infinite number of verification tasks. To resolve this problem, we apply a technique called *query reduction* [ID96, Wes08] in order to reduce this number to a finite number of representative cases. We show that this reduction is possible as Dynamic Evolution Systems induces transition systems that are symmetric in identities [Wes08]. Intuitively, this notion states that the behaviour of the processes does not depend on their actual identity number, e.g. process u_1 behaves exactly the same as process u_{42} whenever they are in the same configuration.

Symmetric behaviour is formalised in terms of permutations of identities as defined in Def. 3.23 and 3.24. Basically following [Wes08], we define the notion of a transition system being symmetric in identities by requiring that any permutation of a transition is also a valid transition.

Definition 5.14 (Symmetric SLTS) *Let $\mathcal{S}$ be a signature, $I' \subseteq Id$ a set of identities and $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow) \in \mathcal{T}_{\mathcal{S}}(I')$ a structured labelled transition system.*

Then T is called symmetric in identities *iff for all permutations $\sigma \in \Sigma_{I'}$ both*

1. *the permutation of an initial state is also an initial state, that is,*

$$S_0 \in \mathbf{S}_0 \implies \sigma(S_0) \in \mathbf{S}_0$$

2. *and the permutation of a transition is also a transition, that is,*

$$(S, g, S') \in \rightarrow \implies (\sigma(S), \sigma(g), \sigma(S')) \in \rightarrow . \qquad \diamondsuit$$

The direct consequence of this definition is that symmetric SLTS exhibit symmetric runs. That is, each run where a particular set of identities reaches a certain configuration has corresponding runs where any permutation on this set of identities reaches the same, but permutated, configuration. In other words, the behaviour of processes does not depend on their identity.

We observe that Dynamic Evolution Systems *by construction* induces transition systems that are symmetric in identities. Hence, there is no need to further check for symmetric behaviour when using DES as a modelling language.

Lemma 5.15 (Symmetric DES) *Let $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ be a Dynamic Evolution System over signature $\mathcal{S}$, $I' \subseteq Id$ a set of identities, and $S \in Strucs_{\mathcal{S}}(I')$ a logical structure.*

Then $[\![\mathsf{D}, S]\!]_{I'}$ is symmetric in identities. $\diamondsuit$

Proof. The proof is given in the appendix (page 196). ■

As already sketched above, we benefit from this symmetric behaviour by reducing the number of valuations that have to be considered when checking for the satisfaction of the universal closure of a given specification (cf. Def. 4.10). Actually, it suffices to consider a finite valuation basis [Wes08] instead of the in general infinite set of valuations.

Definition 5.16 (Valuation Basis) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature and $I' \subseteq Id$ a set of identities. A subset of valuations*

$$\mathit{ValBasis}(\mathcal{X}) \subseteq \mathit{Vals}_{I'}(\mathcal{X})$$

is called a valuation basis of $\mathit{Vals}_{I'}(\mathcal{X})$ *iff each valuation of $\mathit{Vals}_{I'}(\mathcal{X})$ is a permutation of a valuation of $\mathit{ValBasis}(\mathcal{X})$, that is,*

$$\forall\, \mathcal{V} \in \mathit{Vals}_{I'}(\mathcal{X})\ \exists\, \sigma \in \Sigma_{I'}\ \exists\, \mathcal{V}_0 \in \mathit{ValBasis}(\mathcal{X}) : \sigma(\mathcal{V}_0) = \mathcal{V} \qquad \diamondsuit$$

For an example of a valuation basis, consider two variables $X = \{x_1, x_2\}$. The valuations of variables X to identities Id is the infinite set of functions

$$\mathit{Vals}_{Id}(X) = \{[x_1 \mapsto u, x_2 \mapsto u'] \mid u, u' \in Id\}.$$

A valuation basis for $\mathit{Vals}_{Id}(X)$ are the two functions

$$\mathit{ValBasis}(X) := \{[x_1 \mapsto u_1, x_2 \mapsto u_1], [x_1 \mapsto u_1, x_2 \mapsto u_2]\}$$

where $u_1, u_2 \in Id$ are two identities with $u_1 \neq u_2$. Obviously, each valuation in $\mathit{Vals}_{Id}(X)$ can be obtained by a permutation of one of the elements of $\mathit{ValBasis}(X)$.

Having established the symmetric behaviour for Dynamic Evolution Systems in Lemma 5.15 above, we may replace the set of valuation functions used in the definition of the evaluation of a specification in Definition 4.10 by a finite valuation basis as follows.

Lemma 5.17 (Query Reduction) *Let $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ be a Dynamic Evolution System over signature $\mathcal{S}$, $I' \subseteq Id$ a set of identities, and $S \in \mathit{Strucs}_{\mathcal{S}}(I')$ a logical structure.*

For any specification $\phi \in \mathit{Specs}_{\mathcal{S}}$, there exists a finite valuation basis

$$\mathit{ValBasis}(\mathit{vars}(\phi)) \subseteq \mathit{Vals}_{I'}(\mathit{vars}(\phi))$$

sufficient to compute the satisfaction relation of ϕ for D and S under I', i.e.

$$\mathsf{D}^{I'}_S[\![\phi]\!] = \min\{\, [\![\mathsf{D}, S]\!]_{I'}[\phi](\mathcal{V}) \in \mathbb{B} \mid \mathcal{V} \in \mathit{ValBasis}(\mathit{vars}(\phi))\}. \qquad \diamondsuit$$

Proof. By a straight adaption of the proof given in [Wes08], Section A.3. ■

5.2.3 Model Checking

In this section, we observe that the combination of spotlight abstraction and query reduction allows us to apply standard model-checking techniques in order to obtain an approximative value for the specification satisfaction relation according to Definition 4.10. Note that the computed value in general can only be an approximation by the undecidability results from Section 5.1, however it is approximative in terms of the information order of three-valued logic, that is, the result may be indefinite but not wrong.

A valuable integration of spotlight abstraction and query reduction can be obtained by first applying query reduction to the specification and then using the range of the valuation functions as the contents of the spotlight under which the transition system is abstracted. In other words, we first generate a finite set of valuation functions via query reduction and then analyse the requirement separately under each of these valuation functions whereby the spotlight is chosen as the set of identities denoted by the actually considered valuation. Formally, this procedure yields the abstract specification satisfaction relation as follows.

Definition 5.18 (Abstract Specification Evaluation for DES) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure, and* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification.*

The abstract evaluation of ϕ in D and S *is defined as*

$$\mathsf{D}^{\sharp}_{S}[\![\phi]\!] \;:=\; \min\{\, [\![\mathsf{D}, S]\!]^{\sharp}_{\mathsf{ran}(\mathcal{V})}[\phi](\mathcal{V}) \in \mathbb{B}_3 \mid \mathcal{V} \in \mathit{ValBasis}(\mathit{vars}(\phi))\}.$$

where $\mathit{ValBasis}(\mathit{vars}(\phi))$ *is a finite valuation basis of* $\mathit{Vals}_{\mathit{Id}}(\mathit{vars}(\phi))$*.* ♢

Having defined the (concrete) evaluation of a specification in Def. 3.25 and the abstract evaluation above, we can lift the embedding theorem 5.12 to the analysis of the universal closure of a requirement specification for a given Dynamic Evolution System.

Lemma 5.19 (DES Embedding) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure, and* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification. Then*

$$\mathsf{D}_{S}[\![\phi]\!] \;\sqsubseteq\; \mathsf{D}^{\sharp}_{S}[\![\phi]\!] \qquad ♢$$

Proof. We prove the more general case in Lemma 6.3. ■

As a specification only comprises finitely many variables, each of the finitely many valuations yields a verification task for a finite state transition system.

These tasks can now be solved by standard model-checking techniques. For example, we consider the specification

$$\phi_4 := \mathsf{G}\left(\mathsf{link}(x_1, x_2) \rightarrow \neg \ominus x_1 \right)$$

from page 56, stating that a connected process does not disappear from the adhoc networking system D_{ad}. As already observed above, a finite valuation basis for $vars(\phi) = \{x_1, x_2\}$ comprises the two valuations

$$\mathcal{V}_1 := [x_1 \mapsto u_1, x_2 \mapsto u_1]$$
$$\mathcal{V}_2 := [x_1 \mapsto u_1, x_2 \mapsto u_2]$$

where $u_1, u_2 \in Id$ with $u_1 \neq u_2$. According to Definition 5.18, we compute $\mathsf{Ad}^\sharp[\![\phi]\!]$ as the minimum of the two values for

$$[\![\mathsf{Ad}]\!]^\sharp_{\{u_1\}}[\phi_4](\mathcal{V}_1) \quad := b_1$$
$$[\![\mathsf{Ad}]\!]^\sharp_{\{u_1,u_2\}}[\phi_4](\mathcal{V}_2) \quad := b_2$$

Note that by Definition 5.18, we only keep the process u_1 when analysing the case for $\mathcal{V}_1$ where both x_1 and x_2 map to process u_1, and we put the spotlight on both u_1 and u_2 for the valuation $\mathcal{V}_2$ where both variables denote different processes. For the first case, we obtain $b_1 = 1$, basically as the premise $\mathsf{link}(x_1, x_2)$ never becomes true when x_1 and x_2 denote same process. For the second case, we obtain $b_2 = 1/2$, as there is the following run π_4 in the abstract semantics $[\![\mathsf{Ad}]\!]^\sharp_{\{u_1,u_2\}}$ that indicates a possible violation of the specification.

$$\begin{aligned}
\pi_4 = {} & \alpha_{\{u_1,u_2\}}((\emptyset, \iota_e)), \\
(\mathsf{new}(u_1), & \alpha_{\{u_1,u_2\}}((\{u_1\}, (\{\mathsf{dev}(u_1)\}, \emptyset))), \\
(\mathsf{new}(u_2), & \alpha_{\{u_1,u_2\}}((\{u_1,u_2\}, (\{\mathsf{dev}(u_1), \mathsf{dev}(u_2)\}, \emptyset))), \\
(\mathsf{con}(u_1,u_2), & \alpha_{\{u_1,u_2\}}((\{u_1,u_2\}, (\{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1,u_2), \mathsf{link}(u_2,u_1)\}, \emptyset))), \\
(\mathsf{dis}(\bot,u_1), & \alpha_{\{u_1,u_2\}}((\{u_1,u_2\}, (\{\mathsf{dev}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1,u_2), \mathsf{link}(u_2,u_1)\}, \emptyset))), \\
(\mathsf{del}(u_1), & \alpha_{\{u_1,u_2\}}((\{u_2\}, (\{\mathsf{ma}(u_2), \mathsf{link}(u_1,u_2), \mathsf{link}(u_2,u_1)\}, \emptyset))), \ldots
\end{aligned}$$

We illustrate this run graphically in Figure 5.3. It shows that two processes u_1 and u_2 appear and connect themselves in snapshot S_3. By the over-approximative representation of the summary process $\bot$, the evolution labelled $\mathsf{dis}(\bot, u_1)$ is enabled for S_3, such that process $\bot$ disconnects process u_1, leading to u_1 being a free device in snapshot S_4. Note that u_2 still has the 'link' connection to u_1. Now the last rule of Ad is enabled and the connected process u_1 disappears from the system in snapshot S_5. This run is by Definition 4.2

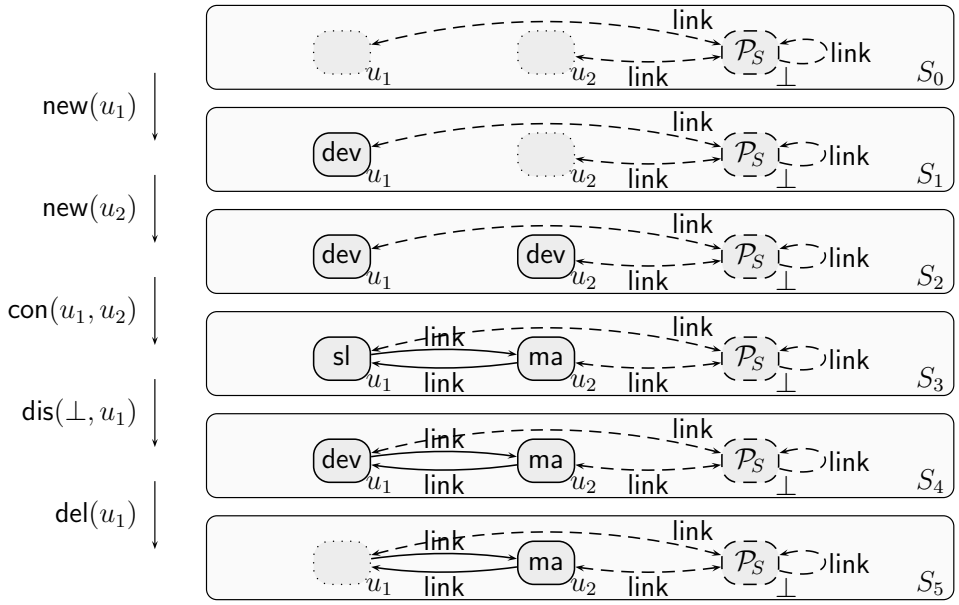


Figure 5.3: A run of $[\![\mathsf{Ad}]\!]^\sharp_{\{u_1,u_2\}}$, indicating a possible violation of ϕ_4.

however only a *possible* violation as the abstract process $\bot$ is involved in the evolution to snapshot S_4.

By this abstract run we obtain $\mathsf{Ad}^\sharp[\![\phi]\!] = 1/2$, and yet cannot conclude anything about $\mathsf{Ad}[\![\phi]\!]$. In particular, we do not know whether the abstract run π_4 corresponds to a definite violation in $[\![\mathsf{Ad}]\!]$, or whether it represents spurious behaviour introduced by the spotlight abstraction.

We may manually argue the adhoc networking example was designed to obey a "proper disconnection property", that is, a process that has just been disconnected does not have any outgoing links, formally represented as

$$\mathsf{Ad} \models \mathsf{G}\,(\mathsf{dis}(x, y) \rightarrow \neg\mathsf{link}(y))$$

Obviously, the run π_4 does not respect this invariant, and hence represents spurious behaviour of $\bot$. As we will demonstrate in the following, this line of reasoning allows us to actually *refine* spotlight abstraction. However, this approach opens up three non-trivial subtasks, namely

1. devise candidates for invariant system properties,
2. establish that these candidates are in fact invariants of the system, and
3. use the invariants to suppress spurious behaviour of the abstract process.

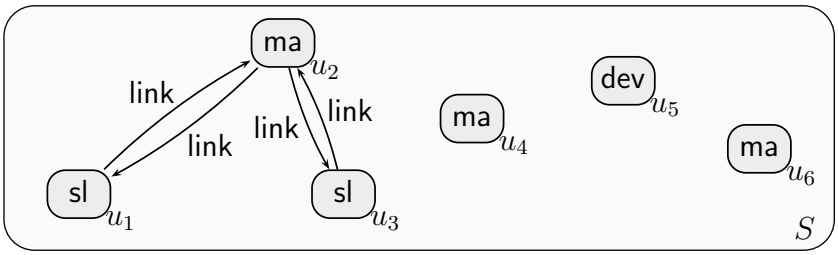
Actually, none of these problems are easy to solve. Firstly, finding candidates typically requires an overall "understanding" of the system and is thus is hard

to automate. Secondly, proving that a property is an invariant of the system in general reduces to a model-checking problem and hence again requires to employ abstraction techniques. Thirdly, refining the behaviour of the abstract process ought to effectively suppress spurious behaviour while maintaining the advantage of a stateless representation. We will address all three problems in the following Chapter 6 on automatic refinement of spotlight abstraction.

5.3 Related Work

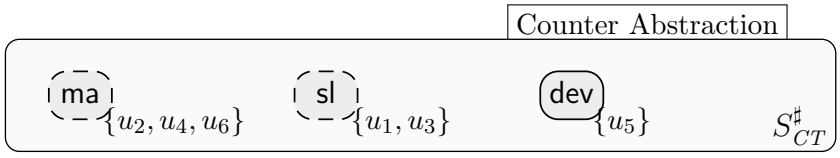
In this section, we compare the spotlight abstraction mechanism with several other abstraction techniques that in some sense *summarise processes*. The overall principle of summarising abstractions is to count the number of those processes which coincides on a certain property, whereby this notion of "property" has different instantiations. The counter values are then cut off at a certain number K, that is, the largest value $K+1$ denotes that there are *more than* K processes. This cutoff leads to a summarisation of all sets of processes exceeding the counter K, and the abstraction is thereby in particular be able to reduce even infinite structures to finite ones. It has been shown in [WW07] that spotlight abstraction can be seen as a particular instance of the canonical abstraction framework (see below).

The underlying ideas of the different abstractions is best explained on a concrete example. We consider the following snapshot of the adhoc network cases study, comprising six processes in configurations as given in the following figure.



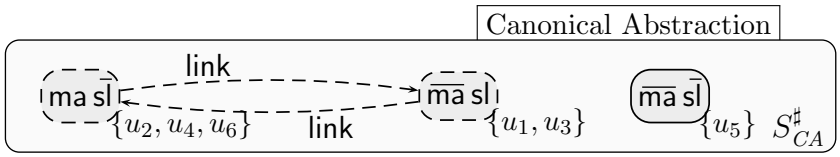
One of the earliest approach in the domain of summarising abstractions is the *Counter Abstraction* [Lub84, GS92, PXZ02] which is typically applied for parameterized systems. The basic idea is to count the number of those processes which are in the same local state. For the running example we count the number of free devices, of slaves and of masters. Under a cutoff of 1, we obtain the following abstraction of the snapshot from above. The values of the counters are given by the border of the nodes. A dotted border indicates value 0, a solid

border indicates 1 and a dashed border indicates "more than 1". For illustration purposes we index each summary node by the set of concrete identities which abstract to this node.



We obtain the information that the original structure comprises more than one master and slaves nodes and exactly one device node. Note that the links among the processes are not taken into account such that all kind of topology information gets lost. This makes classical counter abstraction inappropriate for our addressed class of systems.

There is a large body of work in the area of *Shape Analysis* which aims at a precise and finite characterisation of the overall shape of heap allocated data-structures. Probably the most prominent approach defines the canonically abstraction framework [SRW02] which tags a subset of unary predicates as *abstraction predicates*. These abstraction predicates induce an equivalence relation on processes by merging processes which are indistinguishable under these predicates. All predicates that do not serve as abstraction predicates (i.e. in particular binary predicates) will keep their definite values whenever possible. However, by the merge of processes into summary processes under the given abstraction predicates it is often necessary to revert to the indefinite value in order to obtain a sound abstraction. Using '$\{\mathsf{ma}, \mathsf{sl}\}$' as the set of abstraction predicates yields the following structure.

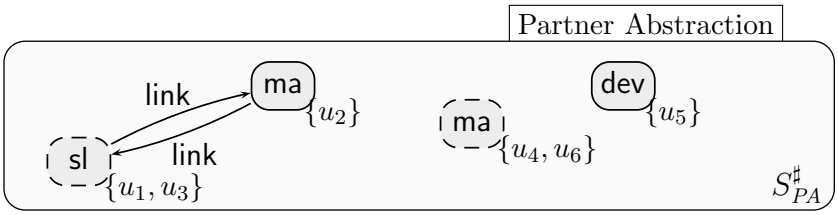


Regarding the local states of the processes, we obtain similar information as in the counter abstraction given above. For the summary nodes of the masters and slaves, the 'link' connection obtains the indefinite value in both directions, because the abstraction has to represent cases where the summarised nodes actually have a connection in the original structure, but also cases where there is no such connection in the original structure. The rightmost node representing the free device soundly maintains the information of not having any outgoing

and incoming links. A refinement of the abstraction can be achieved by adding more abstraction predicates, however only *unary* predicates may be used in order to enhance the precision of the abstraction.

This kind of shape analysis has been extensively used to analyse heap structures, for example in [LAS00, LARSW00, MYRS05]. It has also been applied to establish temporal properties for multi-threaded Java programs in [Yah01] and [YRSW06], respectively. Also, there exists promising approaches in symbolic variants [PW05, BCC+07] of shape analysis that in particular allow for an automatic refinement of the abstraction.

An extension of canonically abstraction that is tailored for the analysis of communication topologies as present in the considered class of dynamic systems is given in [Bau06, BW07]. The proposed *Partner Abstraction* principle on the one hand takes binary predicates into account for abstraction, and on the other hand employs a two-staged variant of the abstraction. The basic principle is to first perform an abstraction step local to each strongly connected component by merging those nodes which are in the same local states *and* who have the same neighbourship, that is, the same set of outgoing and incoming links to neighbours being the same local states. In a second step, isomorphic components are summarised. When applying this principle we obtain the following abstract structure.

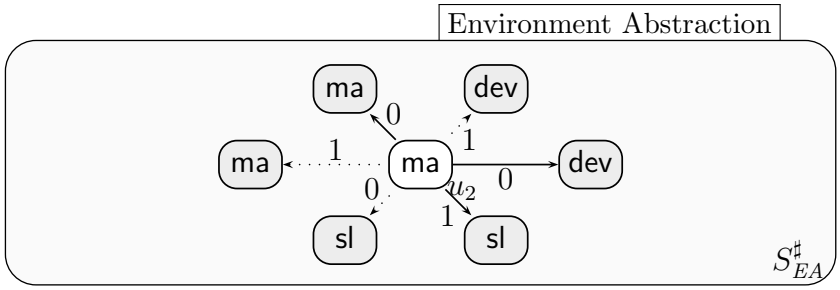


The two slave devices u_1 and u_3 have been summarised in the first step as they are "partner equivalent" as sketched above. In the second step, the two masters u_4 and u_6 are summarised as they represent isomorphic connected components.

The partner abstraction techniques is able to establish a conservative approximation of the set of possible topologies. This has been demonstrated for the car platooning case study (see Section 7.2.4) in [Bau06, BTW07a]. Under certain restrictions the approach is guaranteed to compute an *exact* approximation of the possible topologies. However, the worst-case complexity of computing the transition system under this abstraction mechanism is triple-exponentially [Bau06].

A different view on topologies is taken in the approach of *Environment Abstraction* [CTV06, CTV08], which is advertised as a combination of predicate

and counter abstraction. There, the abstraction is performed from the viewpoint of a single process, the so-called reference process. The abstraction preserves the local state of the reference process and its relationship in terms of binary connections to all other processes. That is, is maintains how many processes in the environment of the reference processes exists to which the reference process has a certain connection. For example, using the master 'u_2' as the reference process leads to the following abstract structure.



In this representation, arcs are labelled by the counter value and the style of the arc determines where this counter value is true (solid line) or false (dotted line). The structure represents that there are 1 or more slaves connected to the reference process u_2, and no other masters or free devices. Note that only the identity of the reference process is kept while the environment processes are anonymised by the abstraction.

In [CTV08], the environment abstraction approach has been applied to the verification of semaphore based distributed algorithms and to several classical mutual exclusion protocols in the area of parameterized verification. Note that the property to be analysed via environment abstraction may comprise at most two variables.

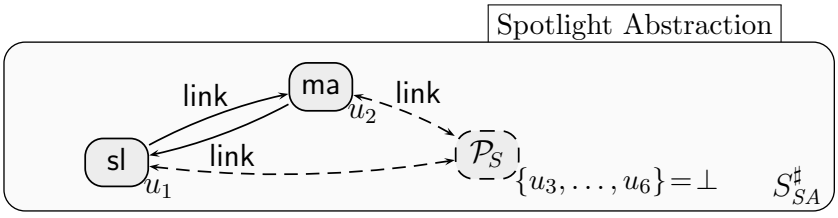
The method of *Indexed Predicate Abstraction* [LB04, LB07] generalised predicate abstraction [GS97] to work on predicates comprising free variables. The task is to compute a boolean combination of these predicates such that the universal closure of this formula holds in every reachable state of the system. The employed abstraction function maps a concrete snapshot to the set of abstract snapshots under the different valuations of the free variables. Initiating this abstraction principle on the indexed predicates '$\mathsf{ma}(x)$', '$\mathsf{sl}(x)$' and '$\mathsf{link}(x, y)$' we obtain 6^2 abstract states for the different valuations of $\{x, y\}$ to $\{u_1, \ldots, u_6\}$. Each row in the following tables represents one element of the abstract states under one of the possible valuations.

x	y	$\mathsf{ma}(x)$	$\mathsf{sl}(x)$	$\mathsf{link}(x,y)$
u_2	u_1	1	0	1
u_2	u_2	1	0	0
u_2	u_3	1	0	1
u_2	u_4	1	0	0
u_2	u_5	1	0	0
u_4	u_4	0	1	0
u_5	u_5	0	0	0
...	...	...	...	...

From the full set of abstract states, one can e.g. conclude that links are always bidirectional, that is $\forall\, x, y\ .\ \mathsf{link}(x,y) \rightarrow \mathsf{link}(y,x)$ is an invariant of the system.

The approach has been demonstrated to work on cache coherence protocols and adhoc networking routing protocols in [Lah04]. For dynamic systems in our setting, we anticipate that the same class of invariants can also be obtained by evaluating the set of abstract topologies as computed by partner abstraction. Moreover, partner abstraction will likely preserve more properties as it takes the overall neighbour-ship into account whereas the focus of indexed predicate abstraction is limited to the range of the actual valuation function.

For completeness, we also give the resulting snapshot under *Spotlight Abstraction* according to Def. 5.5 for processes u_1 and u_2.



One central difference to the related abstraction techniques is that spotlight abstraction allows for a very efficient computation of the abstract transition system. Indeed, only the evolution of the finite spotlight processes have to be taken into account (cf. Def. 5.7) as the representation of the abstract part of the structure is identical in each state. In contrast, all related analysis techniques require a non-trivial computation of the abstract transition relation.

Another point where spotlight abstraction is unique is that it preserves the process identities for the spotlight processes. In contrast, the abstraction mechanisms above suffer from the problem of *identity blurring* [Wac05], that is, processes may migrate from one abstract node to another abstract node due

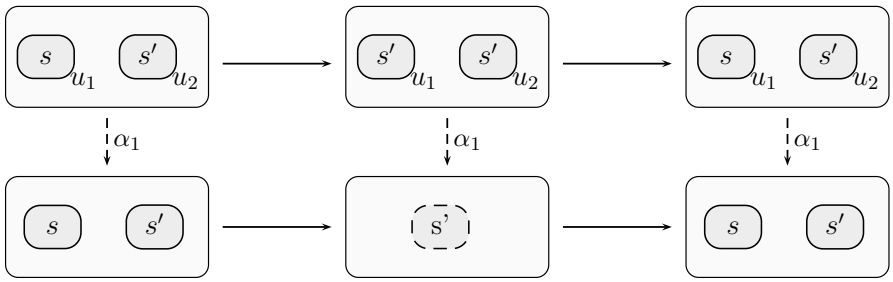


Figure 5.4: Identity blurring under counter abstraction.

to their evolutions. Consider for example the evolutions of the three concrete snapshots in the upper part of Figure 5.4. The property $s(x) \rightarrow \mathsf{X}\,\mathsf{X}\,s(x)$ is lost for example under counter abstraction as we cannot tell which process from the summary node s' evolves into which successor node. Identity blurring in general inhibits the analysis of full temporal properties where configurations of a dedicated process in different snapshots are related by temporal operators. This affects both temporal safety and liveness properties, e.g. in the requirement

$$\mathsf{req}(x, y) \rightarrow \big(\neg\mathsf{fail}(x)\ \mathsf{U}\ \mathsf{ack}(y, x)\big)$$

from page 63, the processes denoted by '$\mathsf{ack}(y, x)$' should be the same processes that have exchanged the 'req' message before. Note that despite abstracting from concrete process identities the important class of *invariant safety properties*, that is, specifications of the form '$\mathsf{G}\,\phi$' where 'ϕ' contains no temporal operators, can still be treated.

It is known that liveness properties are typically not preserved under finitary abstractions, independently of the specific abstraction technique. In the approach of [YRSW06] mentioned above, identities and liveness properties are partly maintained by a dedicated augmentation of the abstract states. This basically follows the general approach of adding some variant of *progress monitoring* to the abstraction as described in [KP99]. We integrated the idea of [Wes08] by which fairness constraints and liveness properties local to the spotlight are preserved under the abstraction. This allows us to establish *local* liveness properties without the need for additional augmentation. Our refinement strategy will additionally exploit this fact by increasing the scope of the fairness constraints by spotlight extension (cf. Sect. 6.1.1 below). We will demonstrate the analysis of liveness properties by our approach for the considered case studies in Chapter 7.

Chapter 6

Spotlight Abstraction Refinement

The spotlight embedding theorem 5.12 ensures that any definite result for the abstract evaluation of a specification immediately transfers to the same value for its concrete evaluation. We however remain inconclusive whenever we obtain the indefinite result. For this case we will now devise a method to transfer this indefinite result '$1/2$' into a definite result, that is, to either '0' or '1'. The method is based on an iterative refinement of the employed abstraction.

Our general strategy for spotlight abstraction refinement uses two complementary kinds of refinement, namely *spotlight extension* and *shadow refinement*, which are explained in Section 6.1. The realisation of this strategy by exploiting the information contained in abstract counterexamples is then described in Section 6.2. It turns out that the key problem is the identification of *spurious* counterexamples. We show that the problem is undecidable in general and

we provide a sound but incomplete method that is able to solve the problem in many cases. In Section 6.3 we describe an additional technique to obtain refinements for Dynamic Communication Systems by statically deriving communication dependencies in a given DCS protocol. We describe related work in Section 6.4 where we discuss related refinement strategies as well as further existing approaches to analyse the considered class of dynamic systems.

6.1 Refinement Strategies

We recall and illustrate the basic principles and properties of the employed spotlight abstraction in Figure 6.1. With respect to the concrete semantics of a Dynamic Evolution System D_S in the form of the infinite SLTS $[\![\mathsf{D}, S]\!]$, we compute the abstract semantics $[\![\mathsf{D}, S]\!]^\sharp_I$ for a finite spotlight $I \subseteq Id$ according to Chapter 5. In general, this abstract system is a proper over-approximation of the original system such that any suitable concretisation operator on the abstract system yields a system comprising more behaviour than $[\![\mathsf{D}, S]\!]$. We gave an example of such an over-approximative behaviour in Section 5.2.3 in terms of a spurious run of the abstract SLTS.

Due to the heterogeneous character of spotlight abstraction in the sense that different parts of the system are abstracted with different degrees of precision, we obtain a certain part of the abstract semantics that exactly corresponds to the same part within the concrete semantics, namely $[\![\mathsf{D}, S]\!]_I$. Intuitively, this transition system comprises exactly those evolutions where only process identities of I are involved. We have established that this transition system is an under-approximation of $[\![\mathsf{D}, S]\!]$ in Lemma 3.26, hence any specification violation within this transition system yields a definite violation also in the concrete transition system $[\![\mathsf{D}, S]\!]$. Note that our three-valued satisfaction relation (cf. Def. 4.2) exploits this property by monitoring whether the violating counterexample remains within $[\![\mathsf{D}, S]\!]_I$. Combining this feature with the soundness of the abstraction according to Lemma 4.6 provides us with the spotlight embedding theorem 5.12.

One natural way to increase the precision of the abstraction is hence to *extend the spotlight* to some set $I' \supset I$. By this, also counterexamples employing concrete processes from $I' \setminus I$ are identifiable as concrete counterexamples directly in the abstract semantics. Our strategy to obtain a meaningful enlargement of the spotlight will be driven by the analysis of abstract counterexamples. These counterexamples comprise evolutions involving the abstract process identity, and each occurrence of this process identity will increase the spotlight by one

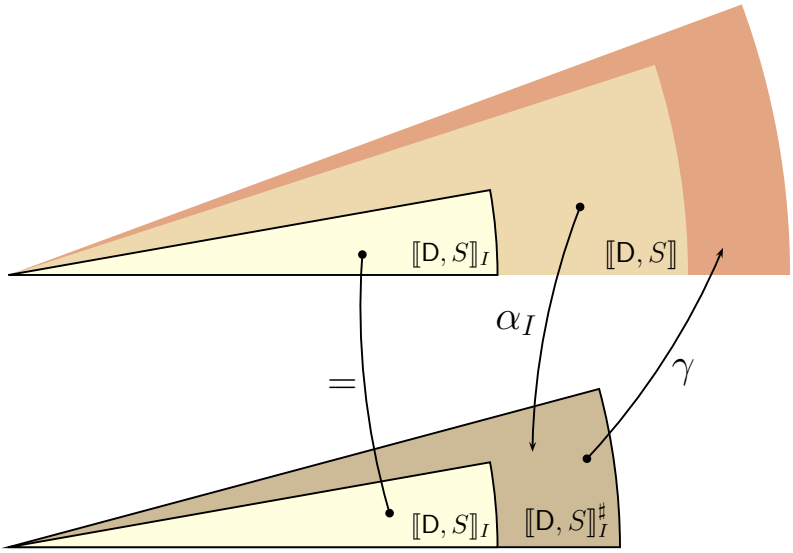


Figure 6.1: Spotlight Abstraction and Refinement

concrete process identity. Intuitively, we try to replay the abstract counterexample with purely concrete processes under the enlarged spotlight. Note that the size of the spotlight is completely determined by the variables of the specification (cf. Def. 5.18). So in order to obtain the desired spotlight extension we generate a specification comprising the right number of variables. By this we do not have to modify the abstraction mechanism itself and may reuse the established procedure of spotlight abstraction, query reduction and model-checking. This in particular leads to a very compact refinement algorithm as given on page 111 below.

By the translation of abstract counterexamples to specifications we obtain a validation mechanism for abstract counterexamples. The validation task reduces to checking whether the negation of the resulting specification holds in each run of the abstract system under the enlarged spotlight. The analysis of this counterexample specification can have three outcomes again. Firstly, it can be '0' and we obtain a concrete counterexample which demonstrates the feasibility of the abstract counterexample under an enlarged spotlight. We then may immediately stop our verification procedure. Secondly, we can obtain the result '1', stating that the behaviour of the abstract counterexample is not possible with concrete processes. While this does not imply that the original specification is satisfied (as there may exist other counterexamples), it allows us to refine the abstraction by eliminating those behaviour that has just been identified to be spurious from the abstraction. This elimination phase is called *shadow refinement* and will be explained below. Thirdly, we may obtain the result '½'

and thereby obtain another abstract counterexample. In this case, we initiate a further extension of the spotlight by taking also the abstract evolutions of the new abstract counterexample into account. This extension procedure will be repeated until a definitive answer has been obtained.

We so far have identified two kinds of refinement for spotlight abstraction, namely spotlight extension and shadow refinement, which we explain in more detail in the two following sections. Notably, both refinement strategies preserve the advantage of a stateless representation of the abstract process. In particular the shadow refinement will not increase the precision of the *configuration* of the abstract process itself but will rather remove those abstract *evolutions* that have been identified to be spurious under certain spotlight configurations.

6.1.1 Spotlight Extension

As sketched above, we can increase the precision of the abstraction by extending the spotlight. By this, more concrete behaviour of the original systems is precisely represented in the abstract semantics. Hence, the main purpose of spotlight extension is to transfer abstract counterexamples into concrete counterexample.

Following Definition 5.10 the size of the spotlight is completely determined by the variables of the specification. In particular, the spotlight is at most as large as the number of variables. The extension of the spotlight is hence driven by introducing fresh variables to a specification. Clearly, the new specification should relate to the original specification in the sense that any violation under the enlarged specification yields a violation of the original specification. This leads to the following definition of spotlight extension.

Definition 6.1 (Spotlight Extension) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a DES over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure and* $\phi, \phi' \in \mathit{Specs}_{\mathcal{S}}$ *two specifications.*

Then ϕ' *is called a* spotlight extension *of* ϕ *for* D *if* $\mathit{vars}(\phi') \supset \mathit{vars}(\phi)$ *and* $\mathsf{D}_S[\![\phi']\!] = 0$ *implies that* $\mathsf{D}_S[\![\phi]\!] = 0$. ♢

Our refinement procedure in Section 6.2 will automatically construct new specifications from abstract counterexamples that yield spotlight extensions of the original specification. Note that extending the spotlight also extends the scope of the fairness constraints as all spotlight processes are required to adhere to the compassion constraints induced by the system description (cf. Def. 5.11).

6.1.2 Shadow Refinement

The shadow refinement of spotlight abstraction will be driven by successively adding assumptions regarding the behaviour (i.e. the evolutions) of the abstract process $\perp$. This approach basically follows the strategy of analysing an open system under certain assumptions concerning the environment – the so-called assume-guarantee paradigm [Pnu85]. Here, a specification consists of a pair $\langle \theta, \phi \rangle$ where ϕ describes a property of the system that has to be guaranteed whenever the environment obeys the assumption θ. When both θ and ϕ are linear temporal logic formulae, the pair can be rewritten to a single formula via implication, that is, to $\theta \rightarrow \phi$. For branching time logics, one has to revert to modular verification techniques [GL94, Jos93].

In this sense, we aim at proving the original specification for the spotlight part of the abstraction in an open environment that is represented by the abstract process. The behaviour of the abstract process may then be refined by adding assumptions for the DES under consideration. Hence, the main purpose of shadow refinement is to establish the validity of a specification under a refined behaviour of the abstract process. To be able to reason about behaviour of the abstract process, we need to declare a subset of variables that actually denote the abstract process. We do so by extending Definition 5.18 as follows.

Definition 6.2 (Extended Abstract Specification Satisfaction) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(I)$ *a logical structure, and* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification.*

The extended abstract evaluation of ϕ in D and S under X *is defined as*

$$\mathsf{D}^{\sharp}_{S,X}[\![\phi]\!] \;:=\; \min\{\, [\![\mathsf{D}, S]\!]^{\sharp}_{\mathsf{ran}(\mathcal{V})}[\phi](\mathcal{V} \cdot \mathcal{V}_X) \in \mathbb{B}_3 \mid \mathcal{V} \in \mathit{ValBasis}(X')\}$$

where

- $X \dot{\cup} X' = \mathit{vars}(\phi)$ *is a disjoint partitioning of the variables of* ϕ*,*
- $\mathit{ValBasis}(X')$ *is a finite valuation basis of* $\mathit{Vals}_{Id}(X')$*, and*
- $\mathcal{V}_X(x) := \perp$ *if and only if* $x \in X$*.* $\diamondsuit$

In particular, we obtain

$$\mathsf{D}^{\sharp}_{S,\emptyset}[\![\phi]\!] \;=\; \mathsf{D}^{\sharp}_{S}[\![\phi]\!]$$

that is, if no variable in ϕ denotes $\perp$, Definition 6.2 coincides with Definition 5.18. In general however, the result of $\mathsf{D}^{\sharp}_{S,X}[\![\phi]\!]$ tends to become indefinite if variables denote the abstract process as its predicate interpretations yield $1/2$ by the definition of spotlight abstraction (cf. Def. 5.5). We obtain the following generalisation of the Embedding Lemma 5.19.

Lemma 6.3 (Extended Embedding) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$, $S \in Strucs_{\mathcal{S}}(Id)$ *a logical structure and* $\phi \in Specs_{\mathcal{S}}$ *a specification. Then*

$$\mathsf{D}_S[\![\phi]\!] \sqsubseteq \mathsf{D}^{\sharp}_{S,X}[\![\phi]\!]$$

for any set of variables $X \subseteq vars(\phi)$. ♢

Proof. The proof is given in the appendix (page 196). ■

We observe that both the assumptions obtained by spurious counterexamples as well as the refinement based on the DCS protocol analysis to be described in Section 6.3 form a proper subset of our general specification language. We call this subset the language of *evolution constraints* as these assumptions constraint the execution of evolutions. We give the formal definition and some examples below.

Definition 6.4 (Evolution Constraint) *Let* $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *be a signature. An* evolution constraint over $\mathcal{S}$ *is generated by the grammar*

$$\theta ::= \mathbf{ff} \mid a \rightarrow \psi \vee \theta_1 \mid \mathsf{X}\, a \rightarrow \psi \vee \theta_1 \mid \mathsf{G}\, \theta_1$$

where $a \in Atoms_{\mathcal{X}}(\mathcal{P}_E)$ *is an atom over evolution predicates* $\mathcal{P}_E$ *and* $\psi \in Forms(\mathcal{P}_{SL})$ *is a formula over state and link predicates* $\mathcal{P}_{SL}$.

The set of all evolution constraints over $\mathcal{S}$ *is denoted by* $EvoC_{\mathcal{S}}$. ♢

Let us consider some examples of evolution constraints over $\mathcal{S}_{\mathsf{ad}}$.

1. $\mathsf{G}\,\big(\mathsf{dis}(x,y) \rightarrow \mathsf{dev}(y) \wedge \neg\mathsf{link}(y,x)\big)$

   This evolution constraint expresses that always after process x disconnects process y, the process y becomes a free device and has no connection link to (its former master) x.

2. $\mathsf{G}\,\big(\mathsf{dis}(x,y) \rightarrow \mathsf{dev}(y) \wedge \neg\mathsf{link}(y)\big)$

   This evolution constraints strengthens the previous example as it requires that a disconnected process y has no connection to *any* process (and hence in particular not to process x).

3. $\mathsf{G}\,\big(\mathsf{X}\,\mathsf{dis}(x,y) \rightarrow \mathsf{dis}^{+}(x,y)\big)$

   Evolution constraints of the form $\mathsf{X}\,a \rightarrow a^{+}$ require a positive counter value for a evolutions. These counter values are updated by observing those evolutions that affect a. We postpone the formal treatment of these kind of evolution constraints to Section 6.3.

So far, the language of evolution constraints provides us with a syntax to formulate *candidates* for the Dynamic Evolution System under consideration. These candidates become evolution constraints for a Dynamic Evolution System with respect to a specification. This allows us to obtain evolution constraints tailored for the actual specification under consideration, although the candidate might not be a valid evolution constraint for *every* specification.

Definition 6.5 (Evolution Constraints for DES) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification, and* $\theta \in \mathit{EvoC}_{\mathcal{S}}$ *an evolution constraint.*

Then θ *is an* evolution constraint for D_S and ϕ *if each run of* D_S *satisfies* θ *or* ϕ*, that is, if* $\mathsf{D}_S \models \theta \vee \phi$ *holds.* ♢

In the case of $\phi = \mathbf{ff}$, we simply say that θ is an evolution constraint for D_S. If we re-write the above condition to $\neg\phi \rightarrow \theta$ we observe that the satisfaction of the evolution constraint is only relevant for paths where the specification is *violated*. This is meaningful as evolution constrains will be used to eliminate spurious violations of the original specification.

We are now able to use an evolution constraint θ for refining the abstract behaviour by setting a subset of evolution variables $\mathit{evovars}(\theta)$ to denote the abstract process and using θ as premise to the specification. Note that by Definition 4.2 an abstract run violating the evolution constraint then however only yields a *possible* violation as the abstract process is involved in at least one evolution label. But as $1/2 \rightarrow b$ yields $1/2$ for any $b \in \mathbb{B}_3$, using 'θ' as an assumption does not lead to an effective refinement.

However, if θ is an evolution constraint for D_S and ϕ we may actually require a definite satisfaction of the evolution constraint. We express this by requiring '$\theta{=}1$' as a premise, which becomes *definitely* violated when the abstract process exhibits evolutions that do not adhere to the constraints given in θ, turning the implication $0 \rightarrow b$ to 1 for any $b \in \mathbb{B}_3$. We formally characterise shadow refinement as follows.

Theorem 6.6 (Shadow Refinement) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a dynamic system over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification, and* $\theta \in \mathit{EvoC}_{\mathcal{S}}$ *an evolution constraint for* D *and* ϕ*. Then*

$$\mathsf{D}_S[\![\phi]\!] \;\sqsubseteq\; \mathsf{D}^{\sharp}_{S,X}[\![\,(\theta{=}1) \rightarrow \phi\,]\!]$$

for any set of variables $X \subseteq \mathit{evovars}(\theta)$*.* ♢

Proof. The proof is given in the appendix (page 197). ■

To demonstrate shadow refinement, we reconsider the specification

$$\phi_4 = \mathsf{G}\,\big(\,\mathsf{link}(x_1, x_2) \rightarrow \neg\ominus x_1\,\big)$$

for the adhoc networking example Ad. As observed above, this property yields a possible violation in the abstract semantics $[\![\mathsf{Ad}]\!]^\sharp_{\{u_1,u_2\}}$ by the abstract run π_4 shown in Figure 5.3, that is, we have $\mathsf{Ad}^\sharp[\![\phi_4]\!] = 1/2$. Given that "proper disconnection" (cf. page 85)

$$\theta_4 = \mathsf{G}\,\big(\mathsf{dis}(b, x_1) \rightarrow \neg\mathsf{link}(x_1)\big)$$

where b is a fresh variable that will be bound to the abstract process identity below, is indeed an evolution constraint for Ad, we observe that under the valuation $\mathcal{V} = [b \mapsto \bot, x_1 \mapsto u_1]$ we obtain

$$\pi_4[\theta_4 = 1]^0(\mathcal{V}) = 0,$$

and in consequence shadow refinement with

$$\mathsf{Ad}^\sharp_{\{b\}}[\![(\theta_4 = 1) \rightarrow \phi_4]\!]$$

yields the value 1 as there is no other run violating the specification. The constraint θ_4 requires that each evolution '$\mathsf{dis}(b, x_1)$' yields a logical structure where the process denoted by variable x_1 is fully disconnected. Hence, the abstract run π_4 violating this constraint is dismissed by the premise $\theta_4 = 1$. By Theorem 6.6 we may then conclude that the infinite state system Ad satisfies ϕ_4, that is,

$$\mathsf{Ad} \models \phi_4.$$

Clearly, we yet have to separately establish that θ_4 is indeed an evolution constraint for Ad and ϕ_4 in order to use θ_4 for a sound refinement. As already sketched, the validation of abstract counterexample via spotlight extension will provide us with a method to automatically obtain valid evolution constraints. This fruitful combination of spotlight extension and shadow refinement is the topic of the next section.

6.2 Counterexample Guided Refinement

By using abstract counterexamples as a means to drive the refinement of the abstraction we follow the general principle of *counterexample-guided abstraction refinement* (CEGAR), which has been initially presented in [Kur94] and has been adapted to the verification of general ACTL* specifications in [CGJ$^+$00].

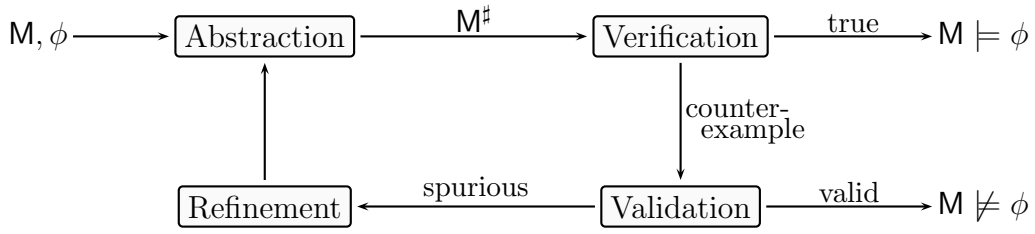


Figure 6.2: Counterexample Guided Abstraction Refinement.

This approach is also known as the *abstract-check-refine* paradigm [HJMS02]. The basic procedure as sketched in Figure 6.2 starts by applying a suitable abstraction technique to the model M, yielding the abstract model $\mathsf{M}^\sharp$. This model is then verified against the specification ϕ. If the outcome is true, the abstraction was already precise enough to establish the validity of the specification, and the refinement loop stops. Else the obtained abstract counterexample is validated, that is, it is checked whether the counterexample corresponds to a concrete counterexample of the original model M. If this is the case, the specification is shown to be violated and the refinement loop stops. If the counterexample however is identified to be spurious, this information is used to refine the abstraction, yielding a refined abstract model. Then the next iteration of the loop is entered and the refined abstract model is verified against the specification. This procedure is iterated until it stops by one of the possibilities described above.

Note that none of the boxes in Figure 6.2 are bound to a specific technique. By this, the framework is applicable to different abstraction and verification techniques, and hence domain-specific validation and refinement techniques have to be used in order to obtain an effective refinement procedure. Indeed, the framework has been applied to many abstraction techniques for quite different classes of systems, for example for hybrid systems [CFH$^+$03, Seg07], C programs [BMMR01, BHJM07], FIFO queues [LW05], pushdown system [EKS06], and graph transformation systems [KK06]. Here, we instantiate the framework for dynamic evolution systems under spotlight abstraction.

Counterexamples

A counterexample is a system run that witnesses the violation of a given temporal specification. From Def. 4.3 we have that actually a finite prefix of a run is sufficient to demonstrate the violation of a safety specification, while for liveness specifications an infinite run is necessary. However, for finite state systems,

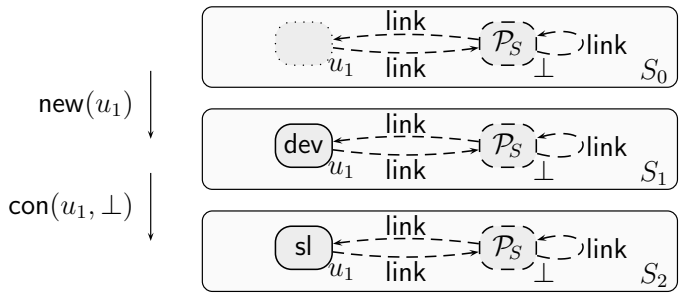


Figure 6.3: A counterexample in $Cex(\mathsf{Ad}, \mathsf{G}\,\neg\mathsf{sl}(x))$ with $[x \mapsto u_1]$.

it is known that this infinite run can be finitely represented in a lasso-shaped form [VW86], that is, a finite prefix with a looping suffix.

Definition 6.7 (Counterexample) *Let $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ be a DES over signature $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$, $I \subset Id$ a finite set of identities, $S \in Strucs_\mathcal{S}(I)$ a logical structure and $\phi \in Specs_\mathcal{S}$ a specification.*

A counterexample for a safety specification ϕ for D and S *is a tuple*

$$\delta = (\bar{\pi}, \mathcal{V})$$

where $\bar{\pi}$ is a prefix of a run $\pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in Runs([\![\mathsf{D}, S]\!]_I) \cup Runs([\![\mathsf{D}, S]\!]_I^\sharp)$ and $\mathcal{V} \in Vals_{Id^\perp}(vars(\phi))$ is a valuation such that $\bar{\pi}w[\phi](\mathcal{V}) \leq 1/2$ for all sequences $w \in (GroundAtoms_I(\mathcal{P}_E) \times Strucs_\mathcal{S}(I))^\omega$.

A counterexample for a liveness specification ϕ for D and S *is a tuple*

$$\delta = (\bar{\pi}, \mathcal{V})$$

where $\bar{\pi}$ is a prefix of a run $\pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in Runs([\![\mathsf{D}, S]\!]_I) \cup Runs([\![\mathsf{D}, S]\!]_I^\sharp)$ and there exists a position $l \in \{0, \ldots, \mathsf{len}(\bar{\pi})\}$ and a valuation $\mathcal{V} \in Vals_{Id^\perp}(vars(\phi))$ such that $\bar{\pi}\big((L_l, S_l) \ldots (L_{\mathsf{len}(\bar{\pi})}, S_{\mathsf{len}(\bar{\pi})})\big)^\omega[\phi](\mathcal{V}) \leq 1/2$.

We call δ an abstract counterexample *if $\pi[\phi](\mathcal{V}) = 1/2$, and a* concrete counterexample *if $\pi[\phi](\mathcal{V}) = 0$. The set of all counterexamples for ϕ in D and S is denoted by $Cex(\mathsf{D}_S, \phi)$.*

By $\bot(\delta) := \{i \in \mathsf{len}(\bar{\pi}) \mid \bot \in A(L_i)\}$ we denote the positions of the abstract evolutions in an abstract counterexample. ♢

As an example, we consider the abstract run $\pi^\sharp \in Runs([\![\mathsf{Ad}]\!]^\sharp_{\{u_1\}})$ from page 78 in Figure 6.3. Under the valuation $\mathcal{V} := [x \mapsto u_1]$ the prefix of this run is a counterexample for

$$\phi_2 := \mathsf{G}\ \big(\neg\mathsf{sl}(x)\big)$$

as u_1 becomes a follower in snapshot S_2, that is, $S_2[\mathsf{sl}(x)](\mathcal{V}) = 1$. Note that this run represents an *abstract* counterexample for the requirement as the abstract process is involved in the evolution leading to this snapshot S_2, that is, we obtain $\pi^\sharp[\mathsf{G}(\neg\mathsf{sl}(x))]^0(\mathcal{V}) = 1/2$ by Definition 4.2.

By this run, we have that $\mathsf{Ad}^\sharp[\![\phi_2]\!] = 1/2$, and we remain inconclusive about the value of $\mathsf{Ad}[\![\phi_2]\!]$. In particular, we do not know whether $\pi^\sharp$ corresponds to a concrete run in $[\![\mathsf{Ad}]\!]$ or whether it represents spurious behaviour introduced by the abstraction.

To answer this question we need to define the *concretisation* of a counterexample, which is a run in the *concrete* semantics that also violates the specification and that is in some sense related to the counterexample. We intentionally use a rather weak notion of relationship between the counterexample and its concretisation. In fact, we basically only require that all evolutions of the abstract process also occur in the same order in the concretisation run. Additionally, these concretised evolutions must lead to the same configuration of the involved processes. In particular, we do not require that the concrete evolutions of the counterexample are contained in the concretisation run. Moreover, as the abstract process represents an arbitrary number of concrete processes and hence may abstractly summarise an arbitrary chain of evolution steps, we do not impose any correspondence between the lengths of the counterexample and its concretisations.

The general strategy of counterexample concretisation is hence to focus on the critical points in an abstract counterexample, namely the evolutions of the abstract process. Actually, this liberal notion of concretisation is the key to obtain effective evolution constraints below.

Definition 6.8 (Counterexample Concretisation) *Let $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ a Dynamic Evolution System over $\mathcal{S}$, $S \in Strucs_\mathcal{S}(Id)$ a logical structure, $\phi \in Specs_\mathcal{S}$ a specification, and $\delta = (((L_i, S_i))_{0\leq i\leq n}, \mathcal{V}) \in Cex(\mathsf{D}_S, \phi)$ an abstract counterexample.*

A run $\pi \in Runs([\![\mathsf{D}, S]\!])$ is a concretisation of δ, *denoted $\delta \prec \pi$, if*

1. *the run π is a concrete counterexample for ϕ, that is,*

$$\pi[\phi]^0(\mathcal{V}) = 0,$$

2. *and there exists a monotone function $f : \bot(\delta) \to \mathbb{N}_0$ such that*

$$\forall\, i \in \mathsf{dom}(f) : L'_{f(i)}[Id \setminus I \mapsto \bot] = L_i \ \wedge\ \alpha_I(S'_{f(i)}\!\downarrow_{A(L_i)}) = S_i\!\downarrow_{A(L_i)}$$

where $I := \mathsf{ran}(\mathcal{V}) \setminus \{\bot\}$ and $\pi = ((L'_i, S'_i))_{i\in\mathbb{N}_0}$.

The set of all concretisation of δ is $\gamma(\delta) := \{\pi \in Runs([\![\mathsf{D}, S]\!]) \mid \delta \prec \pi\}$. ♢

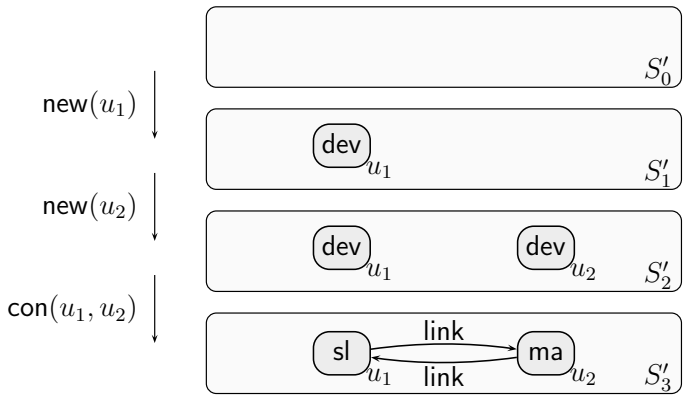


Figure 6.4: A concretisation run for $(\pi^\sharp, \mathcal{V}) \in \mathit{Cex}(\mathsf{Ad}, \phi_2)$.

For the abstract counterexample $(\pi^\sharp, \mathcal{V})$ from above, it is easy to come up with a concretisation run $\pi \in \mathit{Runs}(\llbracket \mathsf{Ad} \rrbracket)$, namely

$$\begin{aligned}
\pi = (\emptyset, \iota_e),& \\
(\ \mathsf{new}(u_1),\ (\{u_1\}, (\{\mathsf{dev}(u_1)\}, \emptyset)\ ),& \\
(\ \mathsf{new}(u_2),\ (\{u_1, u_2\}, (\{\mathsf{dev}(u_1), \mathsf{dev}(u_2)\}, \emptyset)\ ),& \\
(\ \mathsf{con}(u_1, u_2),\ (\{u_1, u_2\}, (\{\mathsf{sl}(u_1), \mathsf{ma}(u_2), \mathsf{link}(u_1, u_2), \mathsf{link}(u_2, u_1)\}, \emptyset)\ ),&
\end{aligned}$$

represented graphically in Figure 6.4. According to Definition 6.8, we have that $\pi[\phi_2]^0(\mathcal{V}) = 0$ and we observe the required relation between the abstract and the concrete counterexample as follows. We have $\bot(\pi^\sharp) = \{2\}$, that is, the abstract identity is only involved in the evolution to snapshot S_2. We then may construct the required correspondence function f with $f(2) = 3$ because

- replacing every identity except for u_1 by the abstract identity in the concrete evolution $L'_3 = \mathsf{con}(u_1, u_2)$ yields L_2, that is,

$$L'_3[\mathit{Id} \setminus \{u_1\} \mapsto \bot] = \mathsf{con}(u_1, \bot) = L_2$$

- and the concrete snapshot S'_3 (cf. Fig. 6.4) under spotlight abstraction with $\{u_1\}$ yields the abstract snapshot S_2 (cf. Fig. 5.2) focussing on u_1:

$$\begin{aligned}
&\alpha_{\{u_1\}}(S'_3\rangle\!\rangle_{\{u_1\}}) \\
&= (\{u_1, \bot), (\{\mathsf{sl}(u_1)\}, \{\mathsf{dev}(\bot), \mathsf{sl}(\bot), \mathsf{ma}(\bot)\}, \mathsf{link}(\bot, \bot), \mathsf{link}(\bot, u_1), \mathsf{link}(u_1, \bot)\}) \\
&= S_2\rangle\!\rangle_{\{u_1\}}
\end{aligned}$$

We visualise this relationship in Figure 6.5, which shows that the concrete process u_2 takes over the behaviour of the abstract process as given in the abstract counterexample $(\pi^\sharp, \mathcal{V})$. Note that in this small example, the focus of

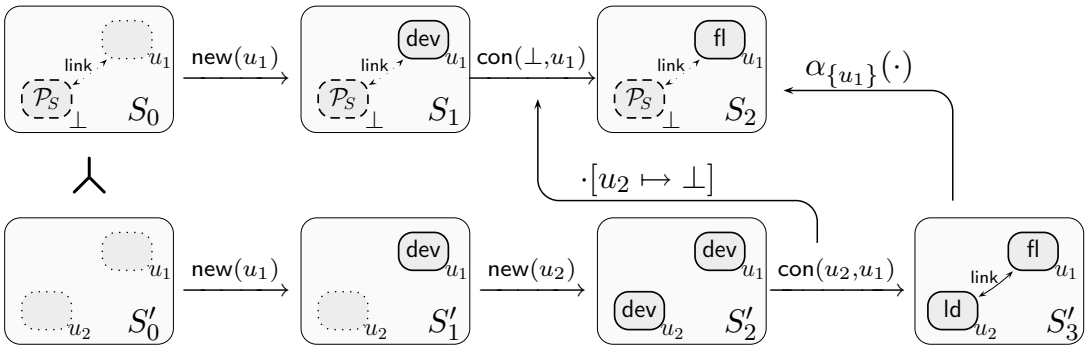


Figure 6.5: Concretisation of the counterexample $(\pi^\sharp, \mathcal{V})$ with $f(2) = 3$.

the abstract snapshot S_2 on $A(L_2)$ has no effect, however for counterexamples involving more concrete processes it allows us to concentrate on those processes that are actually influenced by the considered evolution.

Having defined the concretisation of counterexamples, we obtain a natural notion of a counterexample being *spurious* as follows.

Definition 6.9 (Spurious Counterexample) *Let* $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in Strucs_\mathcal{S}(Id)$ *a logical structure and* $\phi \in Specs_\mathcal{S}$ *a specification.*

An abstract counterexample $\delta \in Cex(\mathsf{D}_S, \phi)$ *is called* spurious*, denoted* $\digamma(\delta)$*, if it has no concretisations, i.e. if* $\gamma(\delta) = \emptyset$*.* $\diamondsuit$

For the abstract counterexample $(\pi^\sharp, \mathcal{V})$ from above, we have manually established that this counterexample is not spurious by identifying a concretisation run according to Definition 6.8. We will automate this reasoning by turning the problem of identifying spurious counterexamples into a model-checking problem.

6.2.1 Identifying Spurious Counterexamples

As already sketched, we devise a translation of an abstract counterexample to a specification in order to validate the counterexample. To this end, we establish a correspondence between the satisfiability of the obtained specification and the question whether the given counterexample is spurious. The translation of an abstract counterexample is based on two (rather technical) subtasks, namely on

1. turning an evolution ground atom into an atom (in Def. 6.10), and
2. turning a logical structure into a formula (in Def. 6.11).

We will illustrate these subtasks and the overall translation schema in terms of the abstract counterexample $(\pi^\sharp, \mathcal{V}) \in \mathit{Cex}(\mathsf{Ad}, \phi_2)$ from above.

Definition 6.10 (Atom of an Evolution Ground Atom) *For a signature $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$, a valuation $\mathcal{V} \in \mathit{Vals}_I(\mathcal{X})$ for a set of identities $I \subseteq \mathit{Id}^\perp$, and an evolution ground atom $g = p(u_1, \ldots, u_{k_p}) \in \mathit{GroundAtoms}_I(\mathcal{P}_E)$, the* atom of g under $\mathcal{V}$ and $i \in \mathbb{N}$*, denoted $\mathit{evo}(g, \mathcal{V}, i)$, is defined as*

$$\mathit{evo}(p(u_1, \ldots, u_{k_p}), \mathcal{V}, i) := p(\nu_{i,1}(\mathcal{V}, u_1), \ldots, \nu_{i,k_p}(\mathcal{V}, u_{k_p}))$$

where

$$\nu_{i,j}(\mathcal{V}, u) := \begin{cases} \mathcal{V}_u & \text{if } u \neq \perp \\ x_{i,j} \in \mathcal{X} \setminus \mathsf{dom}(\mathcal{V}) & \text{else} \end{cases}$$

for an identity $u \in I$ and numbers $i, j \in \mathbb{N}$ and where

$$\mathcal{V}_u := \max\{x \in \mathsf{dom}(\mathcal{V}) \mid \mathcal{V}(x) = u\}$$

determines a variable that maps to identity $u \in I$ under $\mathcal{V}$. ♢

Turning a ground atom into an atom translates each identity to a logical variable. For concrete processes, we select a unique variable from the domain of the valuation function. Note that this selection is well-defined as the set of variables is totally ordered. For each occurrence of the abstract process we introduce a fresh variable. The argument $i \in \mathbb{N}$ is used to guarantee distinct variables when more than one evolution ground atom has to be translated during the counterexample translation defined below. Regarding the evolution $L_2 = \mathsf{con}(u_1, \perp)$ in the abstract counterexample $(\pi^\sharp, \mathcal{V})$, we obtain the atom

$$\mathit{evo}(\mathsf{con}(u_1, \perp), \mathcal{V}, 1) \;=\; \mathsf{con}(x, x_{1,2})$$

with $x_{1,2} \in \mathcal{X} \setminus \mathsf{dom}(\mathcal{V})$ being a fresh variable.

Definition 6.11 (Formula of a log. Structure) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $I \subseteq \mathit{Id}^\perp$ a set of identities, $S = (U, (\iota_{\mathbf{1}}, \iota_{1/\mathbf{2}})) \in \mathit{Strucs}_{\mathcal{S}}(I)$ a logical structure and $\mathcal{V} \in \mathit{Vals}_I(X)$ a valuation for a set of variables $X \subseteq \mathcal{X}$.*

The formula of S under $\mathcal{V}$*, denoted $\mathit{expr}(S, \mathcal{V})$, is defined as*

$$\mathit{expr}(S, \mathcal{V}) := \bigwedge_{u \in I', p_s \in \mathcal{P}_S} \Big(\mathit{vital}(u, \mathcal{V}) \wedge \mathit{atom}(p_s, u, \mathcal{V}) \wedge \bigwedge_{u' \in I', p_l \in \mathcal{P}_L} \mathit{atom}(p_l, u, u', \mathcal{V}) \Big)$$

where $I' := I \setminus \{\perp\}$ and

$$vital(u_1, \mathcal{V}) := \begin{cases} \circledcirc \mathcal{V}_{u_1} & \text{if } u_1 \in U \\ \oplus \mathcal{V}_{u_1} & \text{if } u_1 \notin U \end{cases}$$

$$atom(p, u_1, \ldots, u_{k_p}, \mathcal{V}) := \begin{cases} p(\mathcal{V}_{u_1}, \ldots, \mathcal{V}_{u_{k_p}}) & \text{if } p(u_1, \ldots, u_{k_p}) \in \iota_{\mathbf{1}} \\ \mathbf{tt} & \text{if } p(u_1, \ldots, u_{k_p}) \in \iota_{1/\mathbf{2}} \\ \neg p(\mathcal{V}_{u_1}, \ldots, \mathcal{V}_{u_{k_p}}) & \text{else} \end{cases}$$

for predicates $p \in \mathcal{P}_{SL}$ *and identities* $u_1, \ldots, u_{k_p} \in I'$. ♢

By Definition 6.11 we obtain a formula as a conjunction over all state and link predicates for all elements in the domain of the logical structure excluding the abstract process. For example, regarding the logical structure $S_2 =$

$$(\{u_1, \bot\}, (\{\mathsf{sl}(u_1)\},\ \{\mathsf{dev}(\bot), \mathsf{sl}(\bot), \mathsf{ma}(\bot), \mathsf{link}(\bot, \bot)), \mathsf{link}(\bot, u_1), \mathsf{link}(u_1, \bot)\})$$

from Figure 5.2 and the valuation $\mathcal{V} = [x \mapsto u_1]$. We obtain the formula

$$expr(S_2, \mathcal{V}) = \overbrace{\circledcirc x}^{vital(u_1,\mathcal{V})} \wedge \overbrace{\neg \mathsf{dev}(x)}^{atom(\mathsf{dev},u_1,\mathcal{V})} \wedge \overbrace{\mathsf{sl}(x)}^{atom(\mathsf{sl},u_1,\mathcal{V})} \wedge \overbrace{\neg \mathsf{ma}(x)}^{atom(\mathsf{ma},u_1,\mathcal{V})} \wedge \overbrace{\neg \mathsf{link}(x, x)}^{atom(\mathsf{link},u_1,u_1,\mathcal{V})}$$

As in the Ad system at most one unary predicate holds for each process, we may reduce the expression in order to enhance its readability to

$$expr(S_2, \mathcal{V}) = \ \circledcirc x \wedge \mathsf{sl}(x) \wedge \neg \mathsf{link}(x, x)$$

which characterises an alive slave device that has no connection to itself.

We observe that the formula of a logical structure precisely captures its *definite* information, and in particular the formula of the spotlight abstraction under a set of identities I of a structure S_2 is guaranteed to hold for a larger structure S_1 whenever both agree on a common set of focussed identities F. This property is formalised in the following lemma, which will be used to establish the correspondence between the evaluation of the counterexample specification and its validity in Theorem 6.14 below.

Lemma 6.12 (Spotlight Formula) *Let* $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *be a signature,* $I_1, I_2 \subseteq Id$ *two sets of identities with* $I_2 \subseteq I_1$.

For any two structures $S_1 \in Structs_{\mathcal{S}}(I_1)$ *and* $S_2 \in Structs_{\mathcal{S}}(I_2^{\perp})$, *we have*

$$S_2\rangle\!\rangle_F = \alpha_{I_2}(S_1\rangle\!\rangle_F) \iff S_1[expr(S_2\rangle\!\rangle_F, \mathcal{V})](\mathcal{V}) = 1$$

for a valuation $\mathcal{V} \in Vals_{I_2}(\mathcal{X})$ *and focused identities* $F \subseteq \mathsf{ran}(\mathcal{V})$. ♢

Proof. The proof is given in the appendix (page 197). ■

We combine both translation schemes from Definitions 6.10 and 6.11 and obtain the specification of an abstract counterexample as a nested chain of 'F' expressions as follows.

Definition 6.13 (Counterexample Specification) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a DES over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure and*

$$\delta = (((L_i, S_i))_{0 \leq i \leq n}, \mathcal{V}) \in \mathit{Cex}(\mathsf{D}_S, \phi)$$

an abstract counterexample for a specification $\phi \in \mathit{Specs}_{\mathcal{S}}$*.*

We define the counterexample specification *of* δ *inductively as* $\varphi(\delta) := \varphi(\delta)^1$ *where*

$$\varphi(\delta)^i := \begin{cases} \mathsf{F}\left(\mathit{evo}(L_i, \mathcal{V}, i) \wedge \mathit{expr}(S_i\rangle_{A(L_i)}, \mathcal{V}) \wedge (\varphi(\delta)^{i+1})\right) & \text{if } i \in \bot(\delta) \\ \varphi(\delta)^{i+1} & \text{if } i \notin \bot(\delta) \wedge i \leq n \\ \mathbf{tt} & \text{else} \end{cases}$$

By $\mathit{fresh}(\varphi(\delta)) := \mathit{vars}(\varphi(\delta)) \setminus \mathsf{dom}(\mathcal{V})$ *we denote the new variables in* $\varphi(\delta)$*.* $\diamondsuit$

For the abstract counterexample $(\pi^\sharp, \mathcal{V})$ we obtain the counterexample specification

$$\begin{aligned} \varphi((\pi^\sharp, \mathcal{V})) &= \mathsf{F}\left(\mathit{evo}(L_2, \mathcal{V}, 2) \wedge \mathit{expr}(S_2\rangle_{A(L_2)}, \mathcal{V}) \wedge \mathbf{tt}\right) \\ &= \mathsf{F}\left(\mathsf{con}(x, x_{2,2}) \wedge \circledcirc x \wedge \mathsf{sl}(x) \wedge \neg \mathsf{link}(x, x)\right) \end{aligned}$$

with $\mathit{vars}(\varphi) = \{x, x_{2,2}\}$ and $\mathit{fresh}(\varphi) = \{x_{2,2}\}$. This specification requires to finally observe the abstract evolution L_2 leading to S_2, this time however by *concrete* processes denoted by $x_{2,2}$ and x. If there is a run in Ad satisfying the counterexample specification *and* violating the initial specification, we have obtained a concretisation run and the counterexample is not spurious. Otherwise, if no concrete run exists that both satisfies the counterexample formula and violates the specification, then the counterexample is spurious. Hence, we actually claim the negation of the conjuncted specifications in order to obtain a concrete counterexample (if one exists), leading to the following method for validating abstract counterexamples.

Theorem 6.14 (Counterexample Validation) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification and* $\delta \in \mathit{Cex}(\mathsf{D}_S, \phi)$ *an abstract counterexample. Then*

$$\mathit{F}(\delta) \iff \mathsf{D}_S[\![\neg(\varphi(\delta) \wedge \neg\phi)]\!] \qquad \diamondsuit$$

Proof. By Lemma 6.12. The proof is given in the appendix (page 198). ■

This result in particular shows that the validation of abstract counterexamples corresponds to a verification problem for Dynamic Evolution System. By the undecidability of this problem (cf. Thm. 5.4) we obtain the following corollary.

Corollary 6.15 (Spuriousity is undecidable) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification and* $\delta \in \mathit{Cex}(\mathsf{D}_S, \phi)$ *an abstract counterexample.*

It is undecidable whether δ *is spurious or not.* ♢

Actually, this negative result is not very surprising given the fact that the abstract process may represent the behaviour of any number of concrete processes. Hence, to validate whether the behaviour of the abstract is valid may in general require to analyse the behaviour of this potentially unbounded number of processes.

A natural way to address this undecidability result is to again apply spotlight abstraction in order to reduce the new verification task to a computable problem. Regarding the running counterexample, we have to compute the result of $\mathsf{Ad}[\![\neg(\varphi((\pi^\sharp, \mathcal{V})) \wedge \neg\phi_2)]\!]$, that is check whether

$$\mathsf{Ad} \models \neg\big(\,\mathsf{F}(\mathsf{con}(x, x_{2,2}) \wedge \circledcirc x \wedge \mathsf{sl}(x) \wedge \neg\mathsf{link}(x, x)) \wedge \neg\,\mathsf{G}(\neg\mathsf{sl}(x))\big)$$

holds. For this specification, the spotlight is increased to at most two distinct process identities, say $\{u_1, u_2\}$, and we obtain a concrete counterexample in $[\![\mathsf{Ad}]\!]^\sharp_{\{u_1,u_2\}}$. Indeed, this counterexample coincides with the concretisation run π that we manually established on page 104 (cf. Fig. 6.4). We thus have established by Theorem 6.14 that $\pi^\sharp$ is not spurious and thereby simultaneously obtained a concrete counterexample for Ad violating ϕ_2 by Lemma 5.19.

As just mentioned, the usage of the counterexample specification leads to an enlarged spotlight as each occurrence of abstract identity $\perp$ by a fresh variable in the translation of an evolution ground atom to an atom in Definition 6.10. Formally, the method of counterexample validation as presented in Theorem 6.14 yields a spotlight extension according to Definition 6.1.

Remark 6.16 (Validation by Spotlight Extension) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification and* $\delta \in \mathit{Cex}(\mathsf{D}_S, \phi)$ *an abstract counterexample.*

Then $\neg(\varphi(\delta) \wedge \neg\phi)$ *is a spotlight extension of* ϕ *for* D *and* S*.* ♢

Proof. We have $\mathit{vars}(\neg(\varphi(\delta) \wedge \neg\phi))) \supset \mathit{vars}(\phi)$ by Definition 6.13 and we obtain $\mathsf{D}_S[\![\neg(\varphi(\delta) \wedge \neg\phi)]\!] = 0 \implies \mathsf{D}_S[\![\phi]\!] = 0$ by Definition 4.2. ■

In the above example, we have established that the counterexample is not spurious. In the case that we identify a *spurious counterexample*, an important benefit of the validation according to Theorem 6.14 is that we automatically obtain an *evolution constraint* according to Definition 6.5. Note that this perfectly fits with the overall strategy of counterexamples guided abstraction refinement (cf. Fig. 6.2), that is, a spurious counterexamples triggers and determines the abstraction refinement for the next iteration. We will give an example for the following remark below.

Remark 6.17 (Evolution Constraints for Spurious Counterexamples) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in Strucs_{\mathcal{S}}(Id)$ *a logical structure,* $\phi \in Specs_{\mathcal{S}}$ *and* $\delta \in Cex(\mathsf{D}_S, \phi)$ *an abstract counterexample.*

If δ *is spurious, then* $\neg\varphi(\delta)$ *is an evolution constraint for* D *and* ϕ*.* $\diamondsuit$

Proof. We observe $\neg\varphi(\delta) \in EvoC_{\mathcal{S}}$ via simple syntactical transformations. As δ is spurious, we have by Theorem 6.14 that $\mathsf{D}_S[\![\neg\varphi(\delta) \vee \phi]\!] = 1$. This implies that $\neg\varphi(\delta)$ is an evolution constraint for D_S and ϕ by Definition 6.5. ■

In general, the application of spotlight abstraction in order to validate an abstract counterexample may not give a definitive result, such that we must to prepared to obtain another abstract counterexample, indicating only a *possible* validation of the first counterexample. Clearly, we may use the same technique of validation again, thereby gradually increasing the size of the spotlight. This indicates the need for a *two-staged* variant of counterexample guided abstraction refinement, because the validation of abstract counterexamples in general requires an abstraction refinement loop on its own. We will describe our instantiation of the CEGAR loop in the next Section, thereby integrating all the results given above in one concise framework.

6.2.2 Abstraction Refinement Loop

In order to instantiate the framework of counterexample guided abstraction refinement (cf. Fig. 6.2) for spotlight abstraction, we first of all need to integrate the fact that we work with a *three-valued* satisfaction relation. Additionally, we implement a *two-staged* variant of the loop following the discussion above. Algorithm 1 below addresses these issues by recursively calling itself. It compactly represents our approach of *Counterexample Guided Spotlight Abstraction Refinement* (CEGSAR), and in particular facilitates the implementation of a verification tool as we will demonstrate in Section 7.1.

The algorithm takes a Dynamic Evolution System D, an initial snapshot S and a specification ϕ, and initialises the evolution constraints θ to $\mathbf{tt}$ and the set

Algorithm 1 $\mathsf{cegsar}(\mathsf{D}, S, \phi)$ returns $\mathbb{B}$

```
1: let θ := tt
2: let X := ∅
3: let b := D♯_S[[φ]]
4: while b = 1/2 do
5:    let δ ∈ Cex(D_S, θ → φ)
6:    if cegsar(D, S, ¬φ(δ) ∨ φ) then
7:       let X := X ∪ fresh(φ(δ))
8:       let θ := θ ∧ (¬φ(δ)=1)
9:       let b := D♯_{S,X}[[θ → φ]]
10:   else
11:      b := 0
12:   end if
13: end while
14: return b
```

of variables X denoting the abstract process to the empty set in lines 1 and 2, respectively. It computes the abstract evaluation of ϕ in D_S by standard model-checking techniques (cf. Sect. 5.2.3). If the result is a definitive value (i.e. either 1 or 0), the algorithm directly returns this value in line 14. If the result is $1/2$ we enter the counterexample validation phase. This phase is implemented by a new instance of the algorithm by calling $\mathsf{cegsar}(\mathsf{D}, S, \neg\varphi(\delta) \vee \phi)$ in line 6. If this call returns with value 1, we enlarge the set X by the fresh variables of the counterexample formula in line 7, and integrate the evolution constraint for refinement in line 8. The next iteration of the refinement loop is performed in line 9 by computing the abstract satisfaction relation for $\mathsf{D}^\sharp_{S,X}$ with respect to $\theta \to \phi$. Otherwise, if the counterexample validation returns 0 and a concrete counterexample has been identified, we pass this return value to our caller.

The different phases of the algorithm are graphically represented in Figure 6.6, which shows an extension of the general approach of CEGAR as given in Fig. 6.2. We observe that we obtain an additional refinement loop in vertical direction, which accounts for the fact that the validation of abstract counterexamples requires a separate verification task. In each validation stage, a new instance of the classical abstraction refinement loop is performed, as indicated by the gray area. As soon as an abstract counterexample is identified to be not spurious, the procedure may terminate with a concrete counterexample (①). If the counterexample is spurious (②), the corresponding evolution constraint is used for refinement (③). If the counterexample cannot be definitely analysed under the given spotlight, a refined validation phase is performed (④).

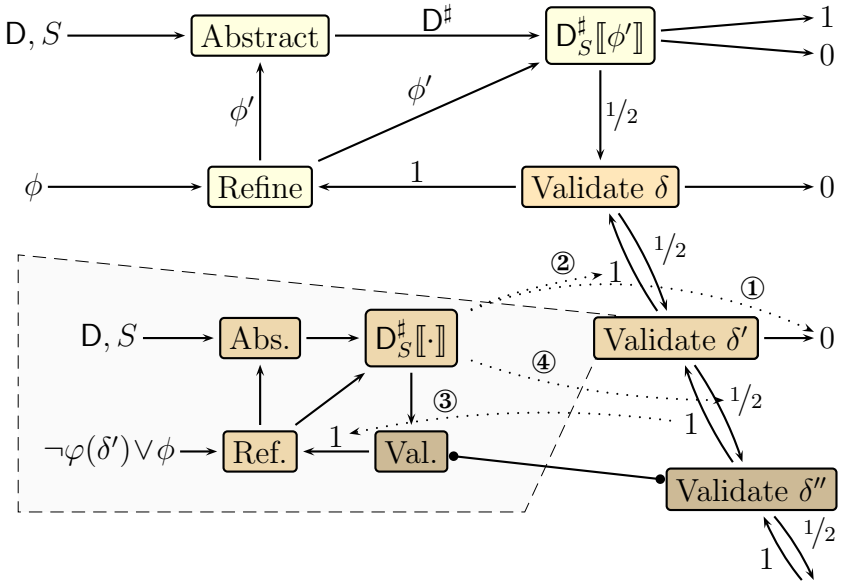


Figure 6.6: Counterexample Guided Spotlight Abstraction Refinement.

As already noted above, the vertical refinement loop performs a gradually enlargement of the spotlight (*spotlight extension*), while the horizontal refinement loops iteratively integrate evolution constraints for *shadow refinement*. The algorithm is shown to be correct by the following theorem.

Theorem 6.18 (Correctness of CEGSAR) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure and* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification. Then*

$$\mathsf{D}_S[\![\phi]\!] \;=\; \mathsf{cegsar}(\mathsf{D}, S, \phi) \qquad \diamondsuit$$

Proof. By induction over the number of iterations of the algorithm. The induction base corresponds to the case $b \in \mathbb{B}$ in line 3, for which we have $\mathsf{cegsar}(\mathsf{D}, S, \phi) = b = \mathsf{D}_S[\![\phi]\!]$ by Lem. 6.3 (Spotlight Embedding).

In the i-th iteration, we have that θ is an evolution constraint for D and ϕ as $F(\delta) \iff \mathsf{cegsar}(\mathsf{D}, S, \neg\varphi(\delta) \vee \phi)$ by Thm. 6.14 (Counterexample Validation) and the induction hypothesis. Hence $b'' = \mathsf{D}^\sharp_{S,X}[\![(\theta \wedge (\neg\varphi(\delta) = 1)) \to \phi]\!]$ in line 9 entails $\mathsf{cegsar}(\mathsf{D}, S, \phi) = \mathsf{D}_S[\![\phi]\!]$ by Thm. 6.6 (Shadow Refinement) if $b'' \in \mathbb{B}$. If $\mathsf{cegsar}(\mathsf{D}, S, \neg\varphi(\delta) \vee \phi) = 0$, we have $\mathsf{cegsar}(\mathsf{D}, S, \phi) = 0 = \mathsf{D}_S[\![\phi]\!]$ by Thm. 6.14 (Counterexample Validation). ■

For an illustration of the CEGSAR iterations for the running ad-hoc networking case study Ad we reconsider the specification

$$\phi_4 = \; \mathsf{G}\,\big(\; \mathsf{link}(x_1, x_2) \to \neg\ominus x_1 \;\big)$$

from page 56. Following Algorithm 1, we start by computing $\mathsf{Ad}^\sharp_\emptyset[\![\phi_4]\!]$ for which we, as discussed on page 84, obtain the result $1/2$ by the abstract counterexample $\langle \pi_4, \mathcal{V} \rangle$ (cf. Fig. 5.3) with $\mathcal{V} = [x_1 \mapsto u_1, \mathcal{V}_2 \mapsto u_2]$. We hence enter the counterexample validation phase by calling

$$\mathsf{cegsar}(\mathsf{Ad}, \neg\varphi(\pi_4) \vee \phi_4)$$

where the counterexample formula of π_4 is

$$\begin{aligned}
\varphi((\pi_4, \mathcal{V})) &= \text{[Def. 6.13 (Counterexample Specification)]} \\
&\quad \mathsf{F}\left(\mathit{evo}(L_4, \mathcal{V}) \wedge \mathit{expr}(S_4)\!\downarrow_{\{u_1\}}, \mathcal{V}) \wedge \mathbf{tt}\right) \\
&= \text{[Def. 3.7 (Focus)]} \\
&\quad \mathsf{F}\left(\mathit{evo}(\mathsf{dis}(\bot, u_1), \mathcal{V}) \wedge \mathit{expr}((\{u_1\}, (\{\mathsf{dev}(u_1), \mathsf{link}(u_1, u_2)\}, \iota_{1/2})))\right) \\
&= \text{[Def. 6.10, 6.11 (Translations)]} \\
&\quad \mathsf{F}\left(\mathsf{dis}(x_{4,1}, x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1, x_1) \wedge \mathsf{link}(x_1, x_2)\right)
\end{aligned}$$

which states that x_1 is finally disconnected by $x_{4,1}$ while remaining a link connection to some process x_2. Calling $\mathsf{cegsar}(\mathsf{Ad}, \neg\varphi(\pi_4) \vee \phi_4)$ leads to the verification task

$$\mathsf{Ad}^\sharp_\emptyset[\![\neg\, \mathsf{F}\left(\mathsf{dis}(x_{4,1}, x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1, x_1) \wedge \mathsf{link}(x_1, x_2)\right) \vee \phi_4]\!]$$

that is, the counterexample validation is performed on a maximum spotlight size of three processes.

We postpone the discussion of the above verification task to page 115, and for now assume that it yields 1, that is, the counterexample $(\pi_4, \mathcal{V})$ is indeed identified to be spurious. The call of $\mathsf{cegsar}(\mathsf{Ad}, \neg\varphi(\pi_4) \vee \phi_4)$ hence returns 1 to its caller which enters the refinement phase by

- setting $X := \{x_{4,1}\}$ in line 7,
- setting $\theta := \mathbf{tt} \wedge \neg\varphi(\pi_4) = 1$ in line 8, and

computing $\mathsf{Ad}^\sharp_X[\![\theta \to \phi]\!]$ in line 9, which translates to

$$\begin{aligned}
\mathsf{Ad}^\sharp_{\{x_{4,1}\}}[\![\mathsf{G}\,(\,\mathsf{dis}(x_{4,1}, x_1) \to \neg(\circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1, x_1) \wedge \mathsf{link}(x_1, x_2))\,) = 1 \\
\to \mathsf{G}\,(\,\mathsf{link}(x_1, x_2) \to \neg\ominus x_1\,)]\!]
\end{aligned}$$

by a syntactical transformation of $\neg\varphi(\pi_4)$ to the evolution constraint language. By this, all evolutions of the abstract process that in some point in time disconnects some process x_1, which is currently connected to some other process x_2, are eliminated. This refined verification task yields 1 and algorithm 1 terminates in line 14. We then may conclude by Theorem 6.18 that $\mathsf{Ad} \models \phi_4$, that is, the specification ϕ_4 is satisfied for the concrete semantics of Ad.

6.2.3 Progress Property

For obtaining an effective refinement in the CEGAR framework, one desires a certain notion of *progress*, namely progress in the sense that each iteration excludes a new source for spurious behaviour. For refining reactive systems it is usually not viable to exclude a *single* spurious counterexample per iteration, because infinitely many "similar" counterexamples exists, which for example only differ in their number of idle steps or which exhibit a differently interleaved execution of concurrent processes. Similar problems exists when considering loops in counterexamples [Dam03], where naive refinement strategies will only exclude one concrete number of unrolling of the loop per iteration.

In our setting of CEGSAR, we actually can investigate two different notions of progress, namely the classical progress of removing spurious behaviour in the refinement loop (*refinement progress*) and additionally the progress in terms of the validation of counterexamples (*validation progress*). Interestingly, we may easily establish refinement progress while we substantially have to improve the translation of counterexamples to specifications (cf. Def. 6.13) in order to obtain validation progress.

To investigate *refinement progress*, we consider a slight modification of algorithm 1 by replacing the validation call in line 6 by a general test for spuriousness, that is, by $F(\delta)$. We refer to this modification by algorithm 1b. We now may investigate properties of our refinement loop under the assumption that the spuriousity of counterexample is decidable, e.g. by some oracle. We observe that the obtained evolution constraints in fact exclude an infinite *class of spurious counterexamples*, namely all runs where the abstract process exhibits the corresponding spurious evolutions, independently of their exact position in the run and independently of the behaviour of the concrete processes in-between these evolutions. This ensures termination of algorithm 1b as follows.

Lemma 6.19 (Refinement Progress) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in Strucs_{\mathcal{S}}(Id)$ *a logical structure, and* $\phi \in Specs_{\mathcal{S}}$ *a specification.*

Then algorithm 1b terminates *and returns* $\mathsf{D}_S[\![\phi]\!]$. ♢

Proof. The proof is given in the appendix (page 198). ■

As in the worst case at most one evolution to a spurious configuration is eliminated per iteration, the number of required iterations is exponentially in both the size of the system and the content of the spotlight. Our practical evaluation in Chapter 7 however shows that the shadow refinement loop for

typical systems already terminates after two or three iterations. Note that this guaranteed termination of this refinement phase does not contradict the general undecidability result given in Theorem 5.4, as Lemma 6.19 is only valid under the assumption that the validation of abstract counterexamples is decidable. In fact, the validation of abstract counterexamples turns out to be the key problem in our CEGSAR framework. To illustrate the difficulties for obtaining a general notion of *validation progress*, we reconsider the validation of π_4 as discussed on page 113. There, we deliberately postponed the verification of

$$\mathsf{Ad}^\sharp_\emptyset[\![\neg\,\mathsf{F}\,\big(\mathsf{dis}(x_{4,1},x_1)\wedge\circledcirc x_1\wedge\mathsf{dev}(x_1)\wedge\neg\mathsf{link}(x_1,x_1)\wedge\mathsf{link}(x_1,x_2)\big)\vee\phi_4]\!]$$

as this task does not yield the desired value 1 (and thus shows that π_4 is indeed spurious), but rather yields the indefinite value $1/2$. This is because

$$\varphi((\pi_4,\mathcal{V}))=\mathsf{F}\,\big(\mathsf{dis}(x_{4,1},x_1)\wedge\circledcirc x_1\wedge\mathsf{dev}(x_1)\wedge\neg\mathsf{link}(x_1,x_1)\wedge\mathsf{link}(x_1,x_2)\big)$$

is possibly satisfied by the following run $\pi'_4\in \mathit{Runs}([\![\mathsf{Ad}]\!]^\sharp_I)$ with $I=\{u_1,u_2,u_3\}$.

$$\begin{aligned}
&\pi'_4=\alpha_I((\emptyset,\iota_e)),\\
&(\ \mathsf{new}(u_1),\ \alpha_I((\{u_1\},(\{\mathsf{dev}(u_1)\},\emptyset)\ )),\\
&(\ \mathsf{new}(u_2),\ \alpha_I((\{u_1,u_2\},(\{\mathsf{dev}(u_1),\mathsf{dev}(u_2)\},\emptyset)\ )),\\
&(\ \mathsf{new}(u_3),\ \alpha_I((I,(\{\mathsf{dev}(u_1),\mathsf{dev}(u_2),\mathsf{dev}(u_3)\},\emptyset)\ )),\\
&(\ \mathsf{con}(u_1,u_2),\ \alpha_I((I,(\{\mathsf{sl}(u_1),\mathsf{ma}(u_2),\mathsf{dev}(u_3),\mathsf{link}(\{u_1,u_2\})\},\emptyset)\ )),\\
&(\ \mathsf{dis}(\bot,u_1),\ \alpha_I((I,(\{\mathsf{dev}(u_1),\mathsf{ma}(u_2),\mathsf{dev}(u_3),\mathsf{link}(\{u_1,u_2\})\},\emptyset)\ )),\\
&(\ \mathsf{con}(u_1,u_3),\ \alpha_I((I,(\{\mathsf{sl}(u_1),\mathsf{ma}(u_2),\mathsf{ma}(u_3),\mathsf{link}(\{u_1,u_2\}),\mathsf{link}(\{u_1,u_3\})\},\emptyset)\ )),\\
&(\ \mathsf{dis}(u_3,u_1),\ \alpha_I((I,(\{\mathsf{dev}(u_1),\mathsf{ma}(u_2),\mathsf{ma}(u_3),\mathsf{link}(\{u_1,u_2\})\},\emptyset)\ )),\\
&(\ \mathsf{del}(u_1),\ \alpha_I((\{u_2,u_3\},(\{\mathsf{ma}(u_2),\mathsf{ma}(u_3),\mathsf{link}(\{u_1,u_2\})\},\emptyset)\ )),\ \ldots
\end{aligned}$$

We observe that $\varphi((\pi_4,\mathcal{V}))$ is possibly satisfied under the valuation

$$\mathcal{V}'=[x_1\mapsto u_1,x_2\mapsto u_2,x_{4,1}\mapsto u_3]$$

due to the concrete evolution $\mathsf{dis}(u_3,u_1)$, which corresponds to a disconnection of two concrete processes whereby u_1 keeps a link connection to u_2. However, the reason for u_1 having this connection is the disconnect evolution $\mathsf{dis}(\bot,u_1)$ two steps before. Roughly speaking, the abstract process "plays the same trick again". For validating the run π'_4 we obtain the counterexample specification

$$\varphi((\pi'_4,\mathcal{V}'))=\mathsf{F}\,\big(\mathsf{dis}(x_{5,1},x_1)\wedge\circledcirc x_1\wedge\mathsf{dev}(x_1)\wedge\neg\mathsf{link}(x_1,x_1)\wedge\mathsf{link}(x_1,x_2)\big)$$

which coincides with $\varphi((\pi_4,\mathcal{V}))$ up to renaming. Clearly, using this specification for validation would lead to an infinite chain of validation attempts.

Summing up, we have observed that the counterexample specifications according to Definition 6.13 are well-suited for shadow refinement but are too weak for an effective validation of abstract counterexamples. To improve this, we again exploit the inherent symmetry of Dynamic Evolution Systems (cf. Sect. 5.2.2). The basic idea is to concentrate the search for a concrete counterexample to a minimal one, in the sense that only the earliest possible violation of the negation of a counterexample specification is considered. To this end, we strengthen the counterexample specification by claiming that no symmetric evolution happens before the concretisation of the abstract evolution is observed in a run satisfying the counterexample specification. Formally, we define the minimal violation of a formula (i.e. a non-temporal specification) as follows.

Definition 6.20 (Minimal Violation) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Forms}(\mathcal{P})$ *a formula, and* $[\![\mathsf{D}, S]\!] = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *the concrete semantics of* D *and* S*.*

The pair $(L', S') \in \mathbf{L} \times \mathbf{S}$ *is a called a* minimal violation of ϕ from $S \in \mathbf{S}$*, denoted*

$$S \dashrightarrow^{\phi}\!\!\bullet\ (L', S')$$

if the following three conditions holds:

1. (L', S') *is a violation of* ϕ*, i.e. there exists a valuation* $\mathcal{V} \in \mathit{Vals}_{\mathit{Id}}(\mathit{vars}(\phi))$ *such that*
$$(L', S')[\phi]^0(\mathcal{V}) = 0,$$
2. S' *is reachable from* S *by* L'*, i.e. there exists a run*
$$\pi = ((L_i, S_i))_{i \in \mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!])$$
with $S = S_i$ *and* $(L', S') = (L_j, S_j)$ *for some* $i, j \in \mathbb{N}_0$ *with* $i \leq j$*, and*
3. *there is no further violation of* ϕ *in the run from* S *to* S'*, i.e.*
$$\pi[\phi]^k(\mathcal{V}') = 1,$$
for all $i \leq k < j$ *and for all valuations* $\mathcal{V}' \in \mathit{Vals}_{\mathit{Id}}(\mathit{vars}(\phi))$*.* $\diamondsuit$

A minimal violation requires that no other processes are in a configuration violating the specification before the minimal violation is observed. In other words, no symmetric configuration of the minimal violation exists before. We are able to establish that whenever there exists a run that comprises a violating configuration, then there exists a minimal violation in this run.

Lemma 6.21 (Existence of Minimal Violations) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in Strucs_{\mathcal{S}}(I)$ *a logical structure,* $\phi \in Forms(\mathcal{P})$ *a formula, and* $[\![\mathsf{D}, S]\!] = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ *the concrete semantics of* D *and* S*.*

If some violation of ϕ *is reachable from state* $S \in \mathbf{S}$*, then there* exists a minimal violation of ϕ from S*, that is,*

$$S \xrightarrow{L'} S' \implies \exists\, (L'', S'') : S \xrightarrow{\phi}\bullet\ (L'', S'')$$

with $(L', S'), (L'', S'') \in \mathbf{L} \times \mathbf{S}$ *being violations of* ϕ*.* ♢

Proof. The proof is given in the appendix (page 199). ■

This results allows us to limit the search for a concrete run violating the negation of the counterexamples specification to a run comprising only minimal violations. If no such minimal concretisation run exist, no concretisation run exist at all. To be able to formalise the search for minimal concretisation in terms of a specification we need to modify a formula by permuting its variables. Recall from Definition 3.23 that a permutation is a bijective function on a given set. Permuting a formula results in the conjunction of all the variants where (as subset of) its variables is permuted. We will give an example below.

Definition 6.22 (Formula Permutation) *Let* $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *be a signature and* $\psi \in Forms(\mathcal{P})$ *a formula.*

For a set of permutations $\Sigma \subseteq \Sigma_{\mathcal{X}}$ *on the set of variables, we define the* permutated conjunction of ψ under Σ *as*

$$\Sigma(\psi) := \bigwedge_{\sigma \in \Sigma} \sigma(\psi)$$

where the permutation of a formula is defined canonically as

$$\sigma(\psi) := \psi[x_1 \mapsto \sigma(x_1)] \ldots [x_n \mapsto \sigma(x_n)]$$

for $vars(\psi) = \{x_1, \ldots, x_n\}$*.* ♢

This notion allows us to formalise the search for minimal violations by turning the finally operators of a counterexample specification to until operators, whereby the expression on the left hand side of the untils is set to the permutation of the negation of the expression of the right hand side. Intuitively, this schema filters out all symmetry violations before a minimal violation is observed.

Lemma 6.23 (Specification Contraction) *Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature, $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ a Dynamic Evolution System over $\mathcal{S}$, $S \in Strucs_\mathcal{S}(Id)$ a logical structure, and $\phi \in Forms(P)$ a formula. Then*

$$\mathsf{D}_S \models \neg\, \mathsf{F}\, \phi \iff \mathsf{D}_S \models \neg\big(\Sigma(\neg\phi)\ \mathsf{U}\ \phi\big)$$

where $\Sigma \subseteq \Sigma_{vars(\psi)}$ is a set of permutations of $vars(\psi)$. ♢

Proof. By Lemma 6.21. The proof is given in the appendix (page 200). ■

We generalise the idea of this contraction lemma to the general case of counterexample specification comprising a nested chain of finally operators. In this strict variant of a counterexample specification, we employ 'U' operators rather than 'F' operators and set the left hand side of each until expression to the conjunction of the negated right hand side under the permutation of all variables in the right hand side expression. The fresh variables in the left hand side will be bound to the abstract process identity when using the strict counterexample specification for validation. By this we effectively eliminate validation loops which stems from the fact that the abstract process performs the same evolutions which are currently to be validated (see the example given on page 115).

Definition 6.24 (Strict Counterexample Specification) *Let $\mathsf{D} \in \mathcal{D}_\mathcal{S}$ be a Dynamic Evolution System over signature $\mathcal{S}$, $S \in Strucs_\mathcal{S}(Id)$ a logical structure and $\delta = (((L_i, S_i))_{0 \le i \le n}, \mathcal{V}) \in Cex(\mathsf{D}_S, \phi)$ an abstract counterexample for some specification $\phi \in Specs_\mathcal{S}$.*

We define the strict counterexample formula *of δ as $\varphi_\mathsf{U}(\delta) := \varphi_\mathsf{U}(\delta)^1$ where*

$$\varphi_\mathsf{U}(\delta)^i := \begin{cases} \big((\Sigma_{X_i}(\neg\tau(L_i, S_i, \mathcal{V}, n{+}1)))\ \mathsf{U} \\ \qquad \tau(L_i, S_i, \mathcal{V}, i) \wedge (\varphi_\mathsf{U}(\delta)^{i+1})\big) & \text{if } i \in \bot(\delta) \\ \varphi_\mathsf{U}(\delta)^{i+1} & \text{if } i \notin \bot(\delta) \wedge i \le n \\ \mathsf{tt} & \text{else} \end{cases}$$

where

$$\tau(L, S, \mathcal{V}, i) := (evo(L, \mathcal{V}, i) \wedge expr(S\rangle\!\rangle_{A(L)}, \mathcal{V}))$$

is the translation of label L and structure S under valuation $\mathcal{V}$ at position i as introduced in Definition 6.13, and $X_i := vars(\tau(L, S, \mathcal{V}, i))$ denotes the corresponding set of variables for each $i \in \bot(\delta)$. ♢

To ease understanding of this translation, let us construct the strict counterexample formula of $(\pi_4, \mathcal{V})$ from page 100. As described on page 113, the (non-strict) counterexample specification is

$$\varphi(\delta)((\pi_4,\mathcal{V})) = \mathsf{F}\,\big(\underbrace{\mathsf{dis}(x_{4,1},x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1,x_1) \wedge \mathsf{link}(x_1,x_2)}_{\tau(L_4,S_4,\mathcal{V},4)}\big)$$

with $X_4 = \mathit{vars}(\tau(L_4,S_4,\mathcal{V},4)) = \{x_{4,1},x_1,x_2\}$.

The permutations on X_4 are

$$\begin{aligned}\Sigma_{X_4} = \{\ & [x_1 \mapsto x_1, x_2 \mapsto x_2, x_{4,1} \mapsto x_{4,1}],\\ & [x_1 \mapsto x_1, x_2 \mapsto x_{4,1}, x_{4,1} \mapsto x_2],\\ & [x_1 \mapsto x_2, x_2 \mapsto x_1, x_{4,1} \mapsto x_{4,1}],\\ & [x_1 \mapsto x_2, x_2 \mapsto x_{4,1}, x_{4,1} \mapsto x_1],\\ & [x_1 \mapsto x_{4,1}, x_2 \mapsto x_2, x_{4,1} \mapsto x_1],\\ & [x_1 \mapsto x_{4,1}, x_2 \mapsto x_1, x_{4,1} \mapsto x_2]\ \}\end{aligned}$$

and hence we obtain

$$\begin{aligned}\varphi_{\mathsf{U}}((\pi_4,\mathcal{V})) =& \big(\Sigma_{X_4}(\neg(\tau(L_4,S_4,\mathcal{V},6)))\ \mathsf{U}\ \tau(L_4,S_4,\mathcal{V},4)\big)\\ =& \big(\neg(\mathsf{dis}(x_{6,1},x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1,x_1) \wedge \mathsf{link}(x_1,x_2)) \wedge\\ & \neg(\mathsf{dis}(x_{6,1},x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1,x_1) \wedge \mathsf{link}(x_1,x_{4,1})) \wedge\\ & \neg(\mathsf{dis}(x_{6,1},x_2) \wedge \circledcirc x_2 \wedge \mathsf{dev}(x_2) \wedge \neg\mathsf{link}(x_2,x_2) \wedge \mathsf{link}(x_2,x_1)) \wedge\\ & \neg(\mathsf{dis}(x_{6,1},x_2) \wedge \circledcirc x_2 \wedge \mathsf{dev}(x_2) \wedge \neg\mathsf{link}(x_2,x_2) \wedge \mathsf{link}(x_2,x_{4,1})) \wedge\\ & \neg(\mathsf{dis}(x_{6,1},x_{4,1}) \wedge \circledcirc x_{4,1} \wedge \mathsf{dev}(x_{4,1}) \wedge \neg\mathsf{link}(x_{4,1},x_{4,1}) \wedge \mathsf{link}(x_{4,1},x_2)) \wedge\\ & \neg(\mathsf{dis}(x_{6,1},x_{4,1}) \wedge \circledcirc x_{4,1} \wedge \mathsf{dev}(x_{4,1}) \wedge \neg\mathsf{link}(x_{4,1},x_{4,1}) \wedge \mathsf{link}(x_{4,1},x_1))\big)\\ & \mathsf{U}\ (\mathsf{dis}(x_{4,1},x_1) \wedge \circledcirc x_1 \wedge \mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1,x_1) \wedge \mathsf{link}(x_1,x_2))\end{aligned}$$

with $\mathit{fresh}(\varphi_{\mathsf{U}}((\pi_4,\mathcal{V}))) = \{x_{6,1}\}$.

We may use the strict variant of a counterexample specification for validation by the following theorem.

Lemma 6.25 (Strict Counterexample Validation) *Let* D *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification and* $\delta \in \mathit{Cex}(\mathsf{D}_S,\phi)$ *an abstract counterexample. Then*

$$\mathit{F}(\delta) \iff \mathsf{D}_S[\![\neg(\varphi_{\mathsf{U}}(\delta) \wedge \neg\phi)]\!] \qquad \diamondsuit$$

Proof. The proof is given in the appendix (page 201). ■

Returning to the running example, we observe that the strict counterexample specification for the validation of $(\pi_4,\mathcal{V})$ identifies it to be spurious as

$$\mathsf{Ad}^{\sharp}_{\{x_{6,1}\}}[\![\neg\varphi_{\mathsf{U}}((\pi_4,\mathcal{V})) \vee \phi_4]\!] = 1$$

and we can use the (non-strict) counterexample specification of $(\pi_4, \mathcal{V})$ to obtain an evolution constraint for refinement as described on page 113.

In algorithm 2 given below we integrate the strict variant of the counterexample translation for counterexample validation into the CEGSAR algorithm 1 given on page 111. The main modification is to use the strict counterexample specification in the validation phase, which requires to pass the set of fresh variables to the validation instance of the algorithm. The correctness of this algorithm follows analogously to the proof of Theorem 6.18 using Lemma 6.25.

Algorithm 2 $\mathsf{cegsar}(\mathsf{D}, S, \phi, F)$ returns $\mathbb{B}$

1: let $\theta := \mathbf{tt}$
2: let $X := F$
3: let $b := \mathsf{D}^{\sharp}_{S,X}[\![\phi]\!]$
4: **while** $b = 1/2$ **do**
5: let $\delta \in \mathit{Cex}(\mathsf{D}_S, \theta \rightarrow \phi)$
6: **if** $\mathsf{cegsar}(\mathsf{D}, S, \neg\varphi_{\mathsf{U}}(\delta) \vee \phi, \mathit{fresh}(\varphi_{\mathsf{U}}(\delta))$ **then**
7: let $X := X \cup \mathit{fresh}(\varphi(\delta))$
8: let $\theta := \theta \wedge (\neg\varphi(\delta) = 1)$
9: let $b := \mathsf{D}^{\sharp}_{S,X}[\![\theta \rightarrow \phi]\!]$
10: **else**
11: $b := 0$
12: **end if**
13: **end while**
14: return b

From the above discussions we observe that $\mathsf{cegsar}(\mathsf{Ad}, (\emptyset, \iota_e), \phi_4, \emptyset)$ terminates after one iteration with the value 1, that is, we have established

$$\mathsf{Ad} \models \mathsf{G}\,\big(\ \mathsf{link}(x_1, x_2) \rightarrow \neg \ominus x_1\ \big).$$

We will describe a tool implementation that demonstrates the feasibility of our refinement algorithm 2 in Chapter 7. There we in particular describe the individual verification tasks of the discussed specifications for the running adhoc networking example in more detail.

We can use the strict variant of counterexample validation in order to establish a notion of validation progress in the sense that an abstract counterexample δ', which has been obtained for the validation validation of an abstract counterexample δ, does not comprise (or cover) this counterexample δ. We formalise this notion of coverage as follows.

Definition 6.26 (Counterexample Coverage) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure, and*

$$\delta_1 = (((L_i^1, S_i^1))_{0 \leq i \leq n_1}, \mathcal{V}_1) \in \mathit{Cex}(\mathsf{D}_S, \phi_1)$$
$$\delta_2 = (((L_i^2, S_i^2))_{0 \leq i \leq n_2}, \mathcal{V}_2) \in \mathit{Cex}(\mathsf{D}_S, \phi_2)$$

two abstract counterexamples for specifications $\phi_1, \phi_2 \in \mathit{Specs}_{\mathcal{S}}$*, respectively.*

We say that δ_1 *covers* δ_2*, denoted* $\delta_1 \gg \delta_2$*, if there exists a permutation* $\sigma \in \Sigma_{\mathsf{ran}(\mathcal{V}_2)\setminus\{\perp\}}$ *and an element* $a_i \in \{a_1, \ldots, a_n\} = \perp(\delta_1)$ *such that*

$$(\sigma(L_{a_i}^2), \sigma(S_{a_i}^2)) = (L_j^1, S_j^1)$$

for

$$j \in \begin{cases} \{0, \ldots, f(a_1)\} & \text{if } i = 1 \\ \{f(a_{i-1}), \ldots, f(a_i)\} & \text{if } i > 1 \end{cases}$$

where f *is the concretisation mapping from* δ_2 *to* δ_1 *according to Def. 6.8.* ♢

We have encountered such an undesirable coverage by our attempt to validate the counterexample $\delta = (\pi_4, \mathcal{V})$ by the non-strict counterexample specification on page 115. For this validation task we have obtained the abstract counterexample $\delta' = (\pi_4', \mathcal{V}')$ which covers δ as $(L_4, S_4) = (L_5', S_5')$ and $f(4) = 7$. We observe that the covering counterexample δ' represents a "pumped" version of δ as some permutated variant of the abstract evolution is observed in δ' before the actual concretisation of this abstract evolution takes place. Clearly, this pumping can be repeated ad infimum as the concretisation of the new variant of the abstract evolution can be concretised by the same pattern. We observe that the strict variant of counterexample excludes such effects as follows.

Lemma 6.27 (Counterexample Discoverage) *Let* $\mathsf{D} \in \mathcal{D}_{\mathcal{S}}$ *be a Dynamic Evolution System over signature* $\mathcal{S}$*,* $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id})$ *a logical structure,* $\phi \in \mathit{Specs}_{\mathcal{S}}$ *a specification, and* $\delta \in \mathit{Cex}(\mathsf{D}_S, \phi)$ *an abstract counterexample.*

If the strict validation of δ *yields a counterexample* δ'*, this counterexample does not cover* δ*, that is,*

$$\delta' \in \mathit{Cex}(\mathsf{D}_S, \neg(\varphi_{\mathsf{U}}(\delta) \wedge \neg\phi)) \implies \delta' \not\gg \delta$$ ♢

Proof. The proof is given in the appendix (page 202). ■

A practical evaluation of our approach on realistic case studies has shown that the presented pumping schema occurs in many of these systems, which we now can effectively suppress by our *strict* variant of counterexample validation. We present the evaluation results in Chapter 7.

Complexity of the Approach

The worst-case complexity of verifying LTL specifications is known to be linear in the size of the transition system and exponentially in the size of the specification [CGP99, HR00]. For characterising the complexity of our approach it is hence of interest how the refinement influences these two parameters.

From Definitions 6.13 and 6.24, we have that the size of the obtained *specifications* corresponds to the number of abstract evolutions in the given abstract counterexample. However, the specifications themselves exhibit a certain structure, namely a nested chain of until operators. The resulting Büchi automaton for such kind of specifications only require a linear number of states and do not trigger the worst-case scenario of yielding an exponential number of states [SB00]. Note that this observation holds in particular for the strict variant of the counterexample specification.

By spotlight extension, each occurrence of the abstract process identity in an evolution of the abstract counterexample contributes one concrete process in the validation of abstract counterexample. This in general leads to an exponential grow of the resulting *transition system* of the model. In fact, we observe in Chapter 7 that a large size of the spotlight inhibits an efficient verification of large systems. To limit the increase of the spotlight in practice we propose a modification of the translation which we call the *tail heuristic*. The basic idea is that the translation operation when applied in algorithm 2 at a recursion level of n only considers the last n abstract evolutions of the counterexample to be validated. In particular, the validation attempt for the first abstract counterexample considers only the last abstract evolution. As there is no bound on the recursion depth of the algorithm, each number of abstract evolutions will finally be captured by this approach. This heuristic turns out be be rather effective in practice (cf. Chapter 7), intuitively as the back-most evolutions typically have more impact on the violating part of the counterexample. If these evolutions can already be shown to be spurious one obtains small evolution constraints and in particular avoids the validation under a too large spotlight. In the case of genuine abstract counterexamples this heuristic may however lead to additional validation attempts under intermediate spotlight sizes.

6.3 Communication Based Refinement

As the semantics of Dynamic Communication Systems is given by a translation into the DES language, the abstraction refinement procedure as described

in the previous section is in particular applicable to the DCS language. In this section, we propose an alternative method for obtaining evolution constraints for Dynamic Communication Systems that exploits the correlationship of message communication as given by some DCS protocol. These additional evolution constraints will be called communication constraints, and they can be easily combined with the overall refinement loop. In particular, if the available communication constraints are not precise enough to concretise an indefinite evaluation result we automatically fall back to the standard refinement loop based on the analysis of abstract counterexamples.

The basic idea of communication constraints is based on the thesis that we can characterise for each process in a given system snapshot a set of messages that this process may receive, and this characterisation can be established purely in terms of the evolution (i.e. communication) steps that led to this snapshot. To obtain communication constraints we hence drop our principle of having a stateless representation of the abstract process to a certain amount. We now maintain a special kind of information of those processes that are summarised by the abstract process, namely information about the currently valid message communication between the concrete processes and the abstracted processes.

We will given the intuition of our approach in more detail in Section 6.3.1 and provide the theoretical foundation for determining the dependencies among messages in Section 6.3.2. In Section 6.3.3 we perform an augmentation of the SLTSs by counters that determine valid message interferences and we describe how these information may be used to obtain communication constraints.

6.3.1 Intuition

To demonstrate the basic idea of spotlight abstraction refinement by communication constraints we reconsider the adhoc DCS example in Figure 3.4. This example has been introduced in Section 3.4 to illustrate the syntax and semantics of Dynamic Communication Systems.

A desirable property of this protocol is that the reception of a request message implies the existence of a connection from the sender to the recipient, that is

$$\phi_r := \mathsf{G}\left(\mathsf{rcv}[req](x_1, x_2) \rightarrow link_d(x_2, x_1)\right).$$

We expect this property to be true for the example system, intuitively as each request (‘*req*’) has to be handled by the recipient before any new communication happens. By this, the senders waits being connected in state ‘*det*’ for the answer of his request.

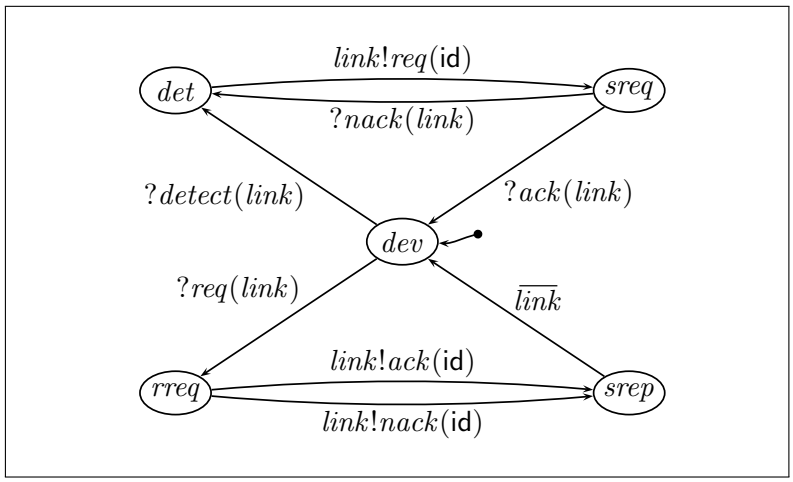


Figure 6.7: Dynamic Communication System C_{ad} (cf. Sect. 3.4).

Under abstraction with spotlight $I = \{u_1, u_2\}$, we however obtain the abstract counterexample $\delta_r = (\pi_r, \mathcal{V}_r) \in \mathit{Cex}(\mathsf{D}(\mathsf{C}_{\mathsf{ad}}), \phi_r)$ with $\mathcal{V}_r = [x_1 \mapsto u_1, x_2 \mapsto u_2]$

$$
\begin{aligned}
\pi_r = \alpha_I((\emptyset, \iota_e)),\\
(\ \mathsf{appear}[d](u_1),\ \alpha_I((\{u_1\}, (\{\mathit{dev}(u_1)\}, \emptyset)\ )),\\
(\ \mathsf{appear}[d](u_2),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{dev}(u_2)\}, \emptyset)\ )),\\
(\ \mathsf{snd}[\mathit{detect}](u_1, u_2),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{dev}(u_2), \mathit{detect}(u_2, u_1)\}, \emptyset)\ )),\\
(\ \mathsf{rcv}[\mathit{detect}](u_2, u_1),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{det}(u_2), \mathit{link}(u_2, u_1)\}, \emptyset)\ )),\\
(\ \mathsf{snd}[\mathit{req}](u_2, u_1),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{sreq}(u_2), \mathit{link}(u_2, u_1), \mathit{req}(u_1, u_2)\}, \emptyset)\ )),\\
(\ \mathsf{snd}[\mathit{ack}](\bot, u_2),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{dev}(u_2), \mathit{link}(u_1, u_2), \mathit{req}(u_1, u_2)\}, \emptyset)\ )),\\
(\ \mathsf{rcv}[\mathit{ack}](u_2, \bot),\ \alpha_I((I, (\{\mathit{dev}(u_1), \mathit{dev}(u_2), \mathit{link}(u_2, \bot), \mathit{req}(u_1, u_2)\}, \emptyset)\ )),\\
(\ \mathsf{rcv}[\mathit{req}](u_1, u_2),\ \alpha_I((I, (\{\mathit{rreq}(u_1), \mathit{dev}(u_2), \mathit{link}(u_2, \bot), \mathit{link}(u_1, u_2)\}, \emptyset)\ )),
\end{aligned}
$$

where u_2 detects u_1 and sends a request. Then however the abstract process $\bot$ interferes by sending an '*ack*' message to u_2. The reception of this message deletes the '*link*' connection of u_2 to u_1, and hence the subsequent reception of '*req*' by u_1 leads to a contradiction of the specification ϕ_r. We observe that the abstract process violates the informal assumption given above, namely it issues a reply to some process which is currently negotiating with some third process. In particular, there was no corresponding request message sent from u_1 to $\bot$.

Communication constraints are able to suppress spurious interferences like the one discussed above. As already sketched, the idea is to maintain information about legal message communication between the concrete and the abstract process and then use constraints to limit the system evolutions to *valid message*

communication. The maintained information is tailored to the DCS language as it is on the one hand directly computable from the syntax of a DCS protocol and can be easily integrated in the resulting abstract SLTS. On the other hand, this information allows us to exclude spurious message interferences to a large amount, and these interferences are a major source of spurious behaviour in spotlight abstractions of communicating systems as demonstrated in [DW05, BTW07a, Tob07].

We illustrate the usage of communication constraints for the running DCS example $\mathsf{C_{ad}}$. For this protocol we can statically derive a relationship between the sending and reception of certain messages. For example, we derive the information that sending a '*req*' message enables the recipient to reply with either an '*ack*' or a '*nack*' message. Moreover, the reception of a '*req*' message is actually *necessary* to become able to send an '*ack*' or a '*nack*' message, as only this reception established the '*link*' channel in the transition from state '*dev*' to '*sreq*' over which the reply messages are sent. For refining the abstract behaviour, we can thus soundly suppress the sending of '*ack*' and '*nack*' from the abstract process to some concrete process unless this concrete process has itself send a request to the abstract process. To this end, we keep for each message a *counter* indicating how many of these messages may be sent from the current configuration. These counters are updated based on the static information of communication dependencies and the observed evolutions of the processes. We then construct communication constraints, which limit the reception of messages to a non-zero value of the corresponding message counter.

6.3.2 Message Dependencies

To formalise the algorithm that statically analyses a given DCS protocol with respect to its communication dependencies we need to introduce some technical notations before.

Definition 6.28 (Transition Notations) *Let* $\mathsf{P} = (Q, A, F, C, succ)$ *be a DCS Protocol over messages* M*, environment messages* M_X *and types* $\mathcal{T}$*.*

We define the following notations for a transition $tr = (q, a, q') \in succ$*:*

- *By* $src(tr) := q$ *and* $dst(tr) := q'$ *we denote the* source and destination state *of tr, respectively.*
- *By* $kind(tr) \in \{\mathsf{create}, \mathsf{snd}, \mathsf{rcv}, \mathsf{clr}\}$ *we denote the* kind *of tr, defined as*

$$kind((q, *_t^c, q')) := \mathsf{create} \qquad kind((q, c_1!m, q')) := \mathsf{snd}$$
$$kind((q, \bar{c}, q')) := \mathsf{clr} \qquad kind((q, ?m(c), q')) := \mathsf{rcv}$$

- *By* $chan(tr) \in C$ *we denote the* channel *of tr, defined as*

$$chan((q, *_t^c, q')) := c \qquad chan((q, c!m, q')) := c$$
$$chan((q, \bar{c}, q')) := c \qquad chan((q, ?m(c), q')) := c$$

- *By* $msg(tr) \in \dot{M \cup}\{\_\}$ *we denote the* message *of tr, defined as*

$$msg((q, *_t^c, q')) := \_ \qquad msg((q, c_1!m, q')) := m$$
$$msg((q, \bar{c}, q')) := \_ \qquad msg((q, ?m(c), q')) := m \qquad \diamondsuit$$

Let us recall the four different DCS actions (cf. Sect. 3.4.1, Def. 3.27). A create action creates a new process and updates the given channel to comprise the new process identity. Analogously, a receive action modifies the given channel to comprise the received process identity. Note that both actions will remove the current content of the channel before assigning the new identity to it. A reset action clears the affected channel, and a send action sends a message to the process denoted by a non-empty channel.

Now given a path of transitions in a DCS protocol we can determine the send actions in this path over a given channel up to the next modification of this channel in this path. This information will serve as the basic ingredient for computing the communication dependencies of a DCS protocol. Intuitively, whenever a process acquires a new process identity and stores it in one of its channel, the possible sendings up to the next modification of this channel will be enabled.

Definition 6.29 (Transition Path Sendings) *Let* $\mathsf{P} = (Q, A, F, C, succ)$ *be a DCS Protocol over messages M, environment messages* M_X *and types* $\mathcal{T}$*.*

A transition path in P *is a finite or infinite sequence of transitions*

$$\tau = tr_0, tr_1, \ldots \in succ^* \cup succ^\omega$$

such that $dst(tr_i) = src(tr_{i+1})$ *for all* $0 \leq i < \mathsf{len}(\tau)$*. The set of all transition paths of* P *is denoted by* $Paths_\mathsf{P}$*, and the set of all transition paths of* P *starting in state* $q \in Q$ *is defined as* $Paths_\mathsf{P}(q) := \{\tau = tr_0, tr_1, \ldots \in Paths_\mathsf{P} \mid src(tr_0) = q\}$*.*

For a transition path $\tau \in Paths_\mathsf{P}$ *we determine* the number of send actions of a message $m \in M$ over an unmodified channel $c \in C$ *as*

$$Sends_\mathsf{P}(\tau, m, c) := |\,\{i \in \{0, \ldots, \mathsf{len}(\tau)\} \mid kind(tr_i) = \mathsf{snd} \wedge msg(tr_i) = m \wedge chan(tr_i) = c \wedge \forall\, 0 \leq j < i : (chan(tr_j) = c \rightarrow kind(tr_j) = \mathsf{snd})\}\,|. \quad \diamondsuit$$

In the following definitions, we lift this notion in order to obtain a characterisation of message dependencies. The first step is to consider the possible sendings from a given state in a DCS protocol. As there may be multiple transition paths originating from this state we will conservatively enable the set of all possible message sendings. Moreover, as there may be multiple send action for the same message we keep track of the maximal number of sendings for all outgoing paths. Note that this number may in particular be infinite in the case of loops in a transition path.

Definition 6.30 (Enabled Messages) *Let* $\mathsf{P} = (Q, A, F, C, succ)$ *be a DCS Protocol over messages* M*, environment messages* M_X *and types* $\mathcal{T}$*.*

We define the enabled messages *of a process in state* $q \in Q$ *over a channel* $c \in C$ *as a function*

$$\triangleright_{\mathsf{P}} : (Q \times C) \to (2^{M \setminus M_X} \times \mathbb{N}_0^\infty)$$

with

$$\begin{aligned}\triangleright_{\mathsf{P}}((q,c)) := (&\{m \in M \setminus M_X \mid \exists\, \tau \in Paths_{\mathsf{P}}(q) : Sends_{\mathsf{P}}(\tau, m, c) > 0\},\\ &\max\{\Sigma_{m \in M} Sends_{\mathsf{P}}(\tau, m, c) \in \mathbb{N}_0^\infty \mid \tau \in Paths_{\mathsf{P}}(q)\})\end{aligned}$$

For an enabled set $\triangleright_{\mathsf{P}}((q,c)) = (E, n)$ *we use*

$$\triangleright_{\mathsf{P}}^{\mathsf{M}}((q,c)) := E$$
$$\triangleright_{\mathsf{P}}^{\mathsf{N}}((q,c)) := n$$

to denote the enabled messages and corresponding number, respectively. ♢

Note that Definition 6.30 does not provide an effective algorithm to compute the enabled messages, basically as $Paths_{\mathsf{P}}(q)$ may be an infinite set due to loops in the successor relation. However, as a loop immediately leads to an infinite number of replies for the affected set of messages, it actually suffice to employ a depth-first search through the successor graph in order to visit all simple paths, with additional bookkeeping whether some state has already been visited. This search can be done in $O(|Q| + |succ|)$ for a given state and channel name.

For demonstrating the concept of enables messages, we give the corresponding sets for the adhoc DCS example from Fig. 3.4. As this DCS protocol comprises five states and one channel name, we obtain the following five sets:

$$\begin{aligned}\triangleright_d(dev, link) &= (\emptyset, 0)\\ \triangleright_d(det, link) &= (\{req\}, 1)\\ \triangleright_d(sreq, link) &= (\emptyset, 0)\\ \triangleright_d(rreq, link) &= (\{ack, nack\}, 1)\\ \triangleright_d(srep, link) &= (\emptyset, 0)\end{aligned}$$

There is clearly a tradeoff between the *precision* of the characterisation of messages dependencies and the cost for its *representation* We decided to over-approximate the number of sendings of a certain message m in a transition path by the maximal number of sendings of *any* message in this path. This allows us to assign counter values to sets of messages rather than to each message individually. Note that this characterisation still preserves the fact that either an '*ack*' *or* a '*nack*' message is enabled in state '*rreq*' but not both, as the maximal number of sendings is computed to be '1' for each outgoing path. The alternative to increase the counter to both messages individually would have the drawback that the decrement of one counter would not affect the counter of the other message, which hence remains spuriously enabled.

By considering the two possibilities of acquiring new process identities, namely the reception of messages and creation of processes, we obtain the desired characterisation of message dependencies. The creation of a new process under a channel 'c' enables the sending of those messages that are possible for 'c' in the destination state of the create action. Alternatively, the sending of a message 'm' enables the sending of those messages that are possible after any receive action of 'm'.

Definition 6.31 (Reply Messages) *Let* $\mathsf{P} = (Q, A, F, C, succ)$ *be a DCS Protocol over messages* M*, environment messages* M_X *and types* $\mathcal{T}$*.*

We define the reply messages *as a function*

$$\lhd_{\mathsf{P}} : M \cup \{\star\} \to (2^{M \setminus M_X} \times \mathbb{N}_0^\infty)$$

with

$$\lhd_{\mathsf{P}}(\star) := \Big(\bigcup_{(q, *_t^c, q') \in succ} \rhd_{\mathsf{P}}^{\mathsf{M}}(q', c),\ \max\{ \rhd_{\mathsf{P}}^{\mathsf{N}}(q', c) \mid (q, *_t^c, q') \in succ\} \Big)$$

$$\lhd_{\mathsf{P}}(m) := \Big(\bigcup_{tr \in succ(m)} \rhd_{\mathsf{P}}^{\mathsf{M}}(dst(tr), chan(tr)),\ \max\{ \rhd_{\mathsf{P}}^{\mathsf{N}}(dst(tr), chan(tr)) \mid tr \in succ(m)\} \Big)$$

for $m \in M$*, where* $succ(m) \subseteq succ$ *denotes those transitions where* m *is received, that is,* $succ(m) := \{tr \in succ \mid kind(tr) = \mathsf{rcv} \wedge msg(tr) = m\}$. ♢

For the running example we obtain

$$\lhd_d(detect) = (\{req\}, 1) \qquad and \qquad \lhd_d(req) = (\{ack, nack\}, 1)$$

and empty set of reply messages for the create action and all other messages. Note that these relations precisely capture the basic communication aspects of the protocol, that is,

- after receiving a '*detect*' message the process may send a request '*req*',
- after receiving a '*req*' message the process may answer by either an '*ack*' or a '*nack*' message,
- and elsewhere no sending of non-environment messages it possible.

We exploit this characterisation of message dependencies by monitoring the send and create evolutions in order to approximate the message receptions that are possible in the future. This monitoring will be carried out by maintaining counters for valid message receptions. A sending of a message 'm' will increase the counter for the set of reply messages '$\lhd^{\mathsf{M}}_{\mathsf{P}}(m)$' by the corresponding number '$\lhd^{\mathsf{N}}_{\mathsf{P}}(m)$'. Likewise, the creation of a new process will increase the counter for possible message sendings to the new process. A reception of a message m will then decrease the counter of a corresponding set of messages, that is, for a set of messages E with $m \in E$. To keep the characterisation finite we will only count up to a finite bound and fall back to uncertainty if the counter exceeds this bound. That is, we employ a variant of counter abstraction (cf. Sect. 5.3) on top of spotlight abstraction.

6.3.3 Message Counting

By finite counting we denote the counting that is precise up to a certain bound and that collapses all values exceeding this bound to the value ∞.

Definition 6.32 (Finite Counting) *We call $K \in \mathbb{N}$ a cutoff value, and define the K-cutoff set $\mathbb{N}_K$ as $\{0, \ldots, K, \infty\}$.*

*For an integer $Z \in \mathbb{Z}$, we define its K-*cutoff *as*

$$Z|_K := \begin{cases} 0 & \text{if } Z < 0 \\ Z & \text{if } Z \in \mathbb{N}_K \\ \infty & \text{if } Z > K. \end{cases}$$

The addition and subtraction operation for $\mathbb{N}_K$ is defined as

$$N_1 \oplus_K N_2 := (N_1 + N_2)|_K$$
$$N_1 \ominus_K N_2 := (N_1 - N_2)|_K$$

where $N_1, N_2 \in \mathbb{N}_K$. ♢

The valid message interferences are represented by counters such that each set of messages that is possibly enabled in a given DCS protocol is mapped to a finite number under a given cutoff value. Each pair of identities is then equipped with such a counter.

Definition 6.33 (Message Counter) *Let* $\mathsf{C} = (M, M_X, P) \in \mathcal{DCS}$ *be a Dynamic Communication System and* $K \in \mathbb{N}_K$ *a cutoff.*

The message counter of C for K *is a function*

$$\Pi_K^{\mathsf{C}} : 2^{M \setminus M_X} \rightharpoonup \mathbb{N}_K$$

with $\mathsf{dom}(\Pi_{\mathsf{C}}) := \{\triangleright_{\mathsf{P}}^{\mathsf{M}}((q,c)) \subseteq M \setminus M_X \mid \mathsf{P} \in \mathsf{dom}(P), q \in Q_{\mathsf{P}}, c \in C_{\mathsf{P}}\}$.

For a set of identities $I \subseteq Id^{\perp}$*, we define the* process message counter of C and K *as a function*

$$\Pi : \left(I^2 \times \mathsf{dom}(\Pi_K^{\mathsf{C}})\right) \to \mathbb{N}_K$$

where the update operator *is defined by function modification as*

$$\Pi\langle (u_1, u_2, E) \otimes_K N \rangle := \Pi[((u_1, u_2), E) \mapsto \Pi((u_1, u_2,), E) \otimes_K N]$$

for $u_1, u_2 \in I$*,* $E \in \mathsf{dom}(\Pi_K^{\mathsf{C}})$*,* $\otimes \in \{\oplus, \ominus\}$*, and* $N \in \mathbb{N}_K$*.* ♢

We now exploit the semantics of the different DCS actions to obtain a precise updating of the counters. To this end, we extend an Structured Labelled Transition System by process message counters as introduced above. All counters are initialised to the value zero. The transition relation of the original SLTS is then refined to update the counters for the involved processes. Note that this counter augmentation of an SLTS is possible both for the concrete and the abstract semantics of a DCS model.

Definition 6.34 (Counter Augmentation) *Let* $\mathsf{C} \in \mathcal{DCS}$ *be a Dynamic Communication System,* $K \in \mathbb{N}_K$ *a cutoff, and* $\mathsf{T} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \to)$ *a SLTS.*

The K-augmentation of T for C and K *is* $\mathsf{T}_K^{\mathsf{C}} := (\mathbf{S}^K, \mathbf{S}_0^K, \mathbf{L}^K, \to^K)$ *with*

- *states* $\mathbf{S}^K := \mathbf{S} \times [I^2 \to [\Pi_K^{\mathsf{C}}]]$
- $\mathbf{S}_0^K := \{(S_0, ((u_1, u_2), (E, 0)) \in \mathbf{S}^K \mid S_0 \in \mathbf{S}_0, u_1, u_2 \in I, E \in \mathsf{dom}(\Pi_K^{\mathsf{C}})\}$
- *labels* $\mathbf{L}^K := \mathbf{L}$*, and*
- *transition relation*

$$succ^K := \{(\,(S, \Pi), L, (S', \Pi')\,) \in \mathbf{S}^K \times \mathbf{L}^K \times \mathbf{S}^K \mid (S, L, S') \in succ \wedge$$
$$\Pi' = \begin{cases} \Pi\langle (u_2, u_1, \triangleleft_{t(u_1)}^{\mathsf{M}}(\star)) \oplus_K \triangleleft_{t(u_1)}^{\mathsf{N}}(\star) \rangle & \text{if } L = \mathsf{create}[t](u_1, u_2) \\ \Pi\langle (u_1, u_2, \triangleleft_{t(u_2)}^{\mathsf{M}}(m)) \oplus_K \triangleleft_{t(u_2)}^{\mathsf{N}}(m) \rangle & \text{if } L = \mathsf{snd}[m](u_1, u_2) \\ \Pi\langle (u_1, u_2, E) \ominus_K 1 \rangle & \text{if } L = \mathsf{rcv}[m](u_1, u_2) \wedge \\ & \exists\, E \in \mathsf{dom}(\Pi_K^{\mathsf{C}}) : m \in E) \\ \Pi & \text{else } \} \end{cases}$$

where $u_1, u_2 \in I$*,* $m \in M$*, and* $t \in \mathsf{dom}(P)$*.* ♢

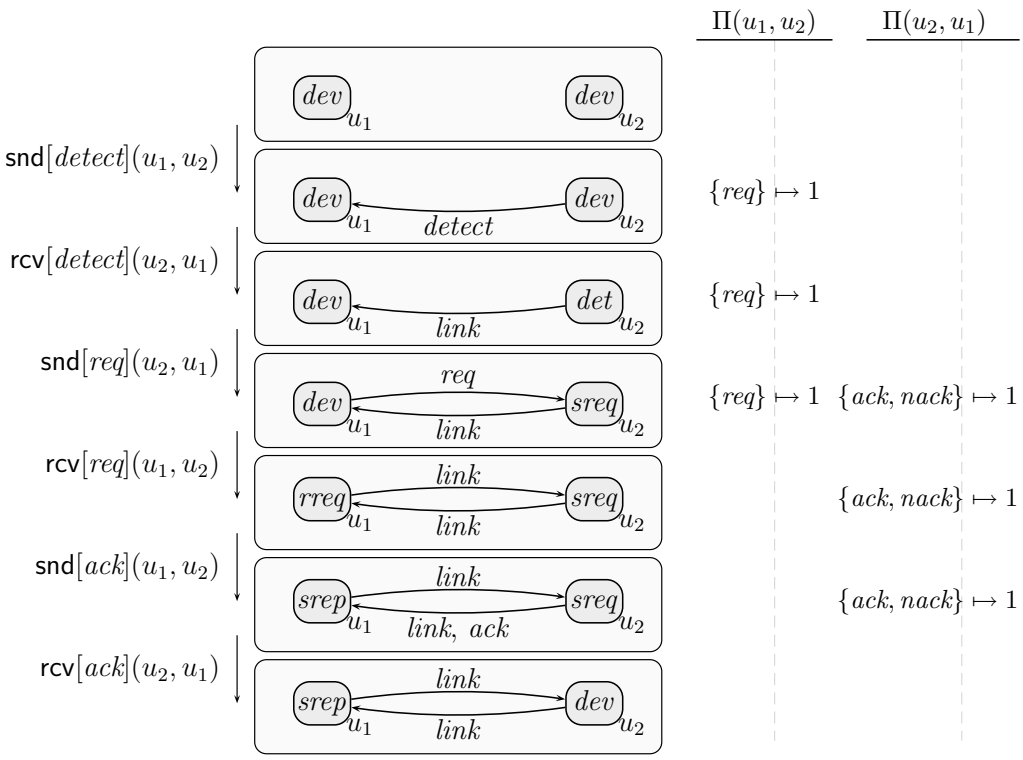


Figure 6.8: A fragment of a run in $[\![\mathsf{D}(\mathsf{C}_{\mathsf{ad}})]\!]_1$.

We illustrate the updating of process message counters by a run in the 1-augmentation of the concrete semantics of C_{ad} in Figure 6.8. For sake of readability this run is annotated by *positive* counter values only. We observe that the sending of the environment message '*detect*' with parameter u_1 to process u_2 increases the counter for valid '*req*' receptions for the pair (u_1, u_2) from zero to one. This counter value is kept set until the '*req*' message is received by u_1. In the evolution before, the sending of '*req*' of u_2 to u_1 increases counter for the set of reply messages $\{ack, nack\}$ from zero to one. Again, this counter is decremented when a corresponding message, in this case an '*ack*', is received.

To establish the soundness of the abstract refinement by communication constraints we show that each reception of a message implies the existence of a positive counter value for a corresponding set of reply messages.

Lemma 6.35 (Positive Counter) *Let $\mathsf{C} \in \mathcal{DCS}$ be a Dynamic Communication System, $I \subseteq Id$ a set of identities, and $[\![\mathsf{D}(\mathsf{C})]\!]_{I_K^{\mathsf{C}}} = (\mathbf{S}^K, \mathbf{S}_0^K, \mathbf{L}^K, \rightarrow^K)$ the concrete semantics of C under I with counter augmentation.*

Then

$$\forall\,((S,\Pi),\mathsf{rcv}[m](u_1,u_2),(S',\Pi')) \in succ^K :\\ \exists\, E \in \mathsf{dom}(\Pi^{\mathsf{C}}_K) : m \in E \wedge \Pi((u_1,u_2),E) > 1$$

for each non-environment message $m \in M \setminus M_X$. ♢

Proof. The proof is given in the appendix (page 202). ■

To use the property of message counter in order to obtain evolution constraints we introduce a mechanism to reason over the values of the augmented counter in our specification language. It suffices to state that a counter value for a set of identities and a set of enabled messages is *positive*.

Definition 6.36 (Counter Formula) *Let* $\mathsf{C} \in \mathcal{DCS}$ *a Dynamic Evolution System,* $\mathcal{S}(\mathsf{C}) = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ *its signature,* $I \subseteq Id^{\perp}$ *a set of identities, and* $K \in \mathbb{N}$ *a cutoff.*

A counter formula *is of the form* $E(x,y)$ *where* $E \in \mathsf{dom}(\Pi^{\mathsf{C}}_K)$ *and* $x, y \in \mathcal{X}$.

The positive counter formula of a message $m \in M \setminus M_X$ *is defined as*

$$m^+(x,y) := \bigvee_{E \in \mathsf{dom}(\Pi^{\mathsf{C}}_K), m \in E} E(x,y)$$

Given a run $\pi \in Runs([\![\mathsf{D}(\mathsf{C})]\!]_I{}^{\mathsf{C}}_K)$ *we define the evaluation of a counter formula at position* $i \in \mathbb{N}_0$ *under a valuation* $\mathcal{V} \in Vals_I(\mathcal{X})(\{x,y\})$ *as*

$$\pi[E(x,y)]^i(\mathcal{V}) := \Pi_i((\mathcal{V}(x),\mathcal{V}(y)),E) > 1$$ ♢

We combine this notion of counter formulas with Lemma 6.35 in order to obtain the desired set of communication constraints for a given DCS model.

Theorem 6.37 (Communication Constraints) *Let* $\mathsf{C} = (M, M_X, P) \in \mathcal{DCS}$ *a Dynamic Evolution System and* $K \in \mathbb{N}$ *a cutoff.*

For any message $m \in M \setminus M_X$, *the specification*

$$\varphi_{\mathsf{C}}(m) := \mathsf{G}\left(\mathsf{X}\,\mathsf{rcv}[m](x,y) \rightarrow m^+(x,y)\right)$$

is an evolution constraint for C_K. ♢

Proof. By Lemma 6.35. The proof is given in the appendix (page 203). ■

A communication constraint can be used for spotlight abstraction refinement by binding one of its variable to a concrete spotlight identity and the other to

the abstract process identity. By this, the communication between these two processes is limited to valid communication with respect to the derived set of reply messages.

For the abstract run π_r given in the introduction of this section on page 124 we observe that the communication constraint '$\varphi_{\mathsf{C}_{\mathsf{ad}}}(ack)$' is violated under the valuation $\mathcal{V}_r = [x \mapsto u_2, y \mapsto \bot]$ as the reception of '*ack*' by u_2 from $\bot$ is not enabled by a prior sending of a '*req*' message from u_2 to $\bot$. Hence, this reception represents a spurious message interference which is effectively removed by applying the corresponding communication constraint.

We will demonstrate the suitability of using communication constraints for spotlight abstraction refinement on a small DCS model in the next subsection and on the larger case study of a car platooning system in Section 7.2.4.

6.3.4 Discussion

It is easy to see the refinement based on communication constraints is in general not complete in the sense that every spurious message interferences can be eliminated via communication constraints. The first kind of uncertainty stems from the cutoff parameter K. Whenever a counter value exceeds this bound it will become ∞ and will remain at this value. There is not yet a method to identify a sufficient cutoff parameter for a given DCS protocol, however experiments show that already a cutoff at 1 or 2 effectively refines the abstraction (cf. Sect. 7.2.4). This effect relates to the notion of *communication slackness* as introduced in [DJ02], observing that typical communication patterns exhibits certain dialogues depending on local modes of the communication partners. In particular it is typically not the case that one partner keeps sending messages without waiting for acknowledgements from the receiver.

The second kind of impreciseness stems from the fact that the dependency relation is in general only an over-approximation of the exact correlationship of messages. We will discuss this issue on the DCS model given in Figure 6.9. It comprises four states s_0 to s_3, one channel 'c' and three messages $\{a, b, e\}$ where 'e' is an environment message. By Definition 6.31 we obtain the communication dependencies for this protocol as

$$\triangleleft(e) = (\{a\}, \infty) \qquad \textit{and} \qquad \triangleleft\,(a) = (\{b\}, \infty)$$

stating that receiving an (environment) message 'e' enables the sending of any number of 'a' messages, and receiving an 'a' enables the process to reply with any number of 'b' messages.

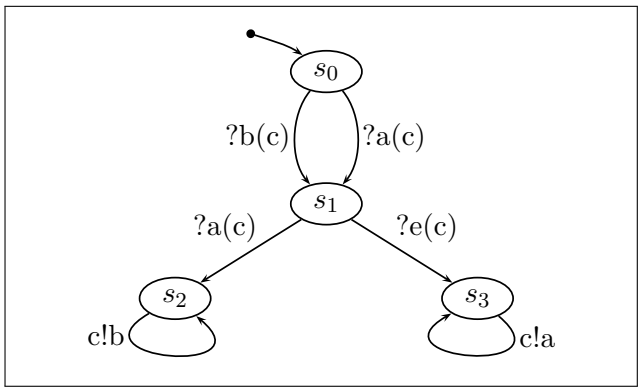


Figure 6.9: C_{ab} – a DCS protocol with unreachable states.

From the initial state 's_0' there are two possibility to reach the next state 's_1', either by receiving an 'a' or a 'b'. The sending of both messages is possibly enabled by the sets of reply messages as computed above, the protocol itself, however, exhibits a cyclic dependency as both send transitions are only reachable from state 's_1'.

Under a spotlight abstraction comprising one concrete process u the abstract process is able to send an 'a' or 'b' message to u, basically as each state predicate is possibly satisfied for the abstract process. The task of abstraction refinement is now to identify and eliminate these spurious interferences. In this example, the communication constraints are able to eliminate any spurious reception of 'b' messages for the spotlight process as it requires the sending of an 'a' before, which is not possible for any concrete process.

The spurious reception of 'a' however can not be handled by communication constraints in isolation. Intuitively, the dependency analysis cannot decide whether the corresponding reception transition of 'e' is actually reachable in the protocol. In this case, the CEGSAR procedure steps in and identifies the abstract reception of 'a' leading to state 's_1' as being spurious and generates an according evolution constraints. In combination with the communication constraint for 'b', both constraints are then sufficient to prove that 's_1' is actually not reachable for any concrete process.

This example demonstrates an interesting synergy effect among both refinement approaches. On the one hand, the CEGSAR approach is able to eliminate spurious behaviour that is not identifiable by communication constraints. On the other hand, many spurious message interferences can be treated by communication constraints such that the number of abstract counterexamples to be validated by the CEGSAR algorithm is reduced. This in particular avoids to

enlarge the spotlight for counterexample validation and in return decreases the overall analysis time. Indeed, not using any kind of communication constraints for the example above forces the CEGSAR algorithm to refute three abstract counterexample under a maximal spotlight of three, in contrast to one abstract counterexample under a spotlight of two in the discussion above. We will provide the detailed listing of the running times for the different combinations of refinement strategies in the evaluation chapter on page 143.

6.4 Related Work

In this section we discuss two sorts of related work, namely firstly existing methods to refine the spotlight abstraction technique and secondly the relation of our overall verification approach by spotlight abstraction refinement in contrast to other existing analysis techniques for the addressed class of dynamic systems.

Spotlight Abstraction Refinement

The spotlight principle is used in [CM02, MC05] for the analysis of telecommunication protocols involving an unbounded number of participants. As the authors are able to establish the considered requirements already in the initial abstraction, refinement strategies are left out as further work.

In [DW05, Wes06] the spotlight abstraction principle is applied to the verification of UML models against Life Sequence Charts ([DH01], cf. Sect. 4.3). For abstraction refinement the paper proposes to transfer the idea of non-interference lemmata from [McM01] to the domain of communicating systems. The general form of these lemmata is characterised as *"If some entity sends something to me, then it is allowed to do so."*. Similar to our approach, these lemmata are then used as an assumption for the proof of the original specification. The lemmata themselves are derived from the static information given in terms of UML class diagrams, and hence exclude any communication that is not possible under the given association relations. Any runtime aspects of the infinite state system under consideration are however not considered.

In a joint work [BTW07a] we have proposed to use the analysis method of [Bau06] for spotlight abstraction refinement. Recall from Sect. 5.3 that [Bau06] is able to compute an over-approximation of the possible communication topologies of the system under consideration. The result is given as a set of so-called abstract clusters comprising summary nodes, and each concrete topology is an instance of some combination of these clusters. The basic idea is to restrict the

behaviour of the abstract process to the valid communication topologies. The theoretical and technical difficulties of this approach stems from the fact that two abstract topologies representations, namely abstract clusters and spotlight abstracted structures, have to be combined. Interestingly, the solution presented in [BTW07a] can be seen as a further source for evolution constraint as the behaviour of the abstract process is limited to those evolution that result in topologies adhering to some combination of abstract clusters. However, the overall refinement approach has no immediate potential for iteration, that is, if the obtained abstract clusters are not precise enough to establish the desired property one remains inconclusive.

Our refinement approaches as published in [Tob07] and [Tob08] are presented and extended in Sections 6.3 and 6.2, respectively. The analysis of communication dependencies in distributed systems is mainly addressed in the area of compiler optimisation [Tse95, PA01], and for systems with a static communication topology, e.g. in [GL84, Hua90, LS94]. The work in [ABJ98, BJNT00] computes the possible content of unbounded FIFO queues in terms of regular languages, and in particular treat the effect of loops in control structures. An abstract-interpretation based approach of characterising queue expressions is given in [GJJ06]. Note that our analysis does not address the content of the message queues explicitly but rather computes the effect of send actions in terms of valid reply messages. A similar idea is used by the authors of [DJ02, Jos06] who propose a symbolic representation of the queue by marking enabled transitions on the receiver side. We are not aware of other approaches to actually use communication dependencies for the refinement of summarising abstractions in the sense of Section 5.3.

Our proposal to refine by spotlight extension (cf. Def. 6.1) can actually be seen as the automatic variant of case-split [McM00]. The principle of case-splitting is to manually refine the specification by introducing new quantifiers. This in consequence increases the number of concretely represented processes. [Wes08] observes that the heuristics for creating case-splits following [McM00] are not directly applicable to system with dynamic topologies. Our approach to analyse abstract counterexamples in order to automatically obtain an extended spotlight is one solution for this problem.

Proving parts of a systems separately under assumptions of the system environment is in particular employed in the approach of thread-modular verification [FFQ02, MPR07]. By this methodology the complexity is reduced from one large verification task to several smaller verification tasks on subcomponents of the system. In our setting these “components” are determined by the content of the spotlight and the set of evolution constraints serve as assumptions.

Analysis of DES-like Systems

The system models of [HPR06b, Pen08] are given as extended graph programs, which are graph transformation rules with application conditions, nondeterministic choice, sequential composition, conditional execution and iteration constructs. Requirements specification are given as graph conditions describing structural graph properties. The correctness of a graph program with respect to a pre- and postcondition is then shown by constructing the weakest precondition for the postcondition and deciding whether the precondition implies the postcondition. The corresponding tool implementation EnforCe [AHPZ07] has been evaluated on the basis of railroad systems and access-control models.

The Augur2 tool [KK08] abstracts hyperedge graph transformation rules to petri graphs, which are then analysed based on coverability graphs and backward reachability algorithms. Correctness specifications are given as regular expressions, which characterise forbidden graph paths. The usage of monadic second-order logic for graphs is under development. The refinement of the abstraction procedure is guided by abstract counterexamples [KK06] or by increasing the unfolding depth. The case studies considered for this approach stems from the networking domain, such as firewall systems or the private/public servers case study (cf. Sect. 7.2.2).

As already mentioned above, [Bau06, BW07] performs an abstract execution of graph transformation rules with negative application conditions in order to to compute a sound over-approximation of the reachable graph structures. The resulting structures may then be analysed for invariant structural properties, for example given as OCL constraints as demonstrated in [BDTW07], or for abstraction refinement as described above. The underlying partner abstraction principle implemented in the Hiralysis tool is tailored to distributed communication systems as it preserves relevant information of the "communication neighbourhood" for each process. The approach is in particular able to handle the merge manoeuvre of the car platooning case study (cf. Sect. 7.2.4).

An abstraction principle tailored to adhoc networking systems is described in [SWJ08]. The corresponding GBT tool checks whether any undesirable graph configuration is reachable from an initial graph by means of backward reachability algorithms. The approach is able to establish loop freedom for a non-trivial adhoc routing protocol called DYMO.

Other tools for graph transformation systems like CheckVML [Var04] or Groove [Ren03] handle only a finite number of nodes.

Note that all approaches presented so far are only interested in reachability problems for different kinds of structural patterns. While this allows the user to

find unwanted system configuration and likewise to show the existence of desired configurations, no statement about temporal relations of different configurations can be made. In particular, the behaviour of individual processes over time can not be traced nor queried.

The work of [MKS08, Mey09] considers finite control processes, a subset of the π-calculus [Mil99], for modelling dynamically reconfigurable systems and devises a translation into bounded petri nets, which may then be analysed with respect to temporal specification by well-established verification techniques. More efficient unfolding-based techniques for safe petri nets are applicable when restricting finite control processes to so-called safe processes. The PETRUCHIO tool is shown to be more efficient in terms of runtime and memory consumption than earlier approach for model-checking π-specifications like the mobility workbench [VM94] or HAL [FGMP03].

The system model of [GMRT06] is given by a set of active process classes, and the behaviour of each class is specified by a set of sequence diagrams. Instances of these classes are summarised by a variant of counter abstraction under a behavioural equivalence relation, leading to a symbolic execution algorithm. A method to identify spurious traces is provided. As the focus of this work is rather on the efficient simulation of large systems than on verification, no explicit specification logic and no abstraction refinement procedure has been addressed in the paper.

Verification using shape analysis techniques ([SRW02], cf. Sect. 5.3) is mainly applied in the context of Java programs, e.g. using the TVLA tool [LAS00]. In [Yah01] the verification of invariant properties for concurrent Java programs is presented. The efficiency and precision of this procedure is improved in [YR04] by decomposing the verification problem into a set of verification subproblems. In [YRS01, YRSW06] a first-order extension of LTL is evaluated on sets of abstract traces of a given Java program. It allows to establish mutual exclusion properties of concurrent threads and liveness guarantees such as that every request is eventually followed by a thread creation.

Research on symbolic variants [PW05] of shape analysis techniques led to the implementation of the BOHNE tool [WKZ+06]. The basic idea is to characterise heap structures by first-order expressions instead of using an explicit encoding. The tool is in particular able to establish some of the topology invariants computed by [Bau06] for a Java encoding of the car platooning case study.

The verification of parameterized systems has been addressed by a large number of approaches, for example based an network invariants [KPSZ02], counter abstraction [PXZ02], regular model-checking [ADHR07, BHV04] or quotient

symmetry reductions [GS99, ID99]. These techniques are however bounded to the specifics of the underlying system languages, which typically assume a static set of processes and a global visibility of their internal states.

The existing verification frameworks for the UML language assume a finite upper bound on the number of simultaneously alive objects. Typical examples include vUML [PL99], UMLAUT [JHGP99], Hugo/RT [KMR02], ObjectCheck [XLB02], and RUVE [STMW04]. An integration of spotlight abstraction into the RUVE framework is demonstrated in [Wes06], however the overhead of the UML meta-model leads to a high verification complexity which as of yet renders this combination inapplicable for larger models.

Most of the discussed approaches consider full temporal properties of the system, however only for anonymous processes. This means that entities in different snapshots of a system run may not be identified to actually denote the same process. Only [YRSW06] maintains an explicit evolution relation along the transitions in order to trace individual process behaviour over time. The scalability of this approach is however unclear and not yet practically evaluated. As discussed in Section 5.3, the spotlight principle preserves the identities of the spotlight processes over the time. By this, even liveness properties among these processes can be established if a fair scheduling of the concrete processes is assumed and any relevant spurious interferences from the abstract process is eliminated. This exactly is the task of spotlight abstraction refinement and we will demonstrate the feasibility of our approach in Section 7.2.

Chapter 7

Evaluation

In this chapter we demonstrate the feasibility of our approach by analysing a number of case studies from the addressed class of systems. To this end, we have conducted a tool implementation of the abstraction and refinement procedure as developed in the previous chapters. In its basic mode, the command line tool takes a textual description of the DES model and the specification and performs the recursive refinement loop according to algorithm 2. Counterexamples are presented graphically using an external visualisation tool. Moreover, the algorithms and translations related to the DCS language exist as a compilation frontend. We will discuss the tool and its architecture in more detail in Section 7.1.

In Section 7.2 we present a number of case studies for which formal models have been generated. For each of the case studies a number of formal requirement specifications are investigated and the performance of our tool implementation is reported. The case studies are selected in order to cover a wide range of the addressed class of systems and they are in particular not restricted to a typical shape of connection topologies. Most of the case studies have been investigated by other researchers before and we compare our results to these existing verification approaches. We are however not aware of any related approach that is able to treat all the case studies and specifications that we consider here.

7.1 Tool Support

Our proposed verification approach for Dynamic Evolution Systems has been realised by the SARMC tool – the **S**potlight **A**bstraction **R**efinement **M**odel-**C**hecker. The DES model is given in an XML format and the specification in an ASCII text-file according to the syntax as given in Definition 4.1. As an example, we present the XML version of the running adhoc networking example 'Ad' together with some samples of the tool output in Appendix B.

The SARMC implements the abstract semantics of DESs according to Definition 5.10 via a translation from the XML model into a low-level verification language called SMI [Bro99]. Moreover, it automates the translation of abstract counterexamples to specifications according to Definitions 6.13 and 6.24, and it triggers the recursive refinement loops according to algorithm 2. We also implemented the translation from the DCS to the DES language according to Definition 3.31 including the static protocol analysis in order to compute communication constraints as described in Section 6.3. By this, the whole range of abstraction and refinement strategies that have been proposed in the previous chapters can be practically evaluated using the SARMC tool.

The verification tasks themselves are carried out by the VIS model-checker in the version 2.1 [Gro96], which is a freely available analysis engine supporting the efficient verification of CTL and LTL specifications for finite-state transition systems. Note that the VIS tool is not aware of the *three-valued* satisfaction relation (cf. Def. 4.2) but rather implements the classical two-valued LTL semantics [MP92] which we will denote by '$\mathsf{D}, S \models_2 \phi$' for a Dynamic Evolution System D, an initial snapshot S and a specification ϕ. Recall that the main purpose of our new satisfaction relation is to distinguish between concrete and abstract counterexamples. In fact, we obtain the classical satisfaction relation from our Definition 4.10 by the following mapping.

$$\mathsf{D}^\sharp, S \models_2 \phi \;=\; \begin{cases} \mathit{true} & \text{if } \mathsf{D}^\sharp_S[\![\phi]\!] = 1 \\ \mathit{false} & \text{if } \mathsf{D}^\sharp_S[\![\phi]\!] \in \{0, 1/2\} \end{cases}$$

That is, the '$\models_2$' relation collapses both abstract and concrete counterexamples into the result '*false*'. On the other hand, we can use such an unspecific counterexample to recover the three-valued semantics from the classical semantics by setting

$$\mathsf{D}^\sharp_S[\![\phi]\!] \;=\; \begin{cases} 1 & \mathit{if}\ \mathsf{D}^\sharp, S \models_2 \phi \\ 0 & \mathit{if\ not}\ \mathsf{D}^\sharp, S \models_2 \phi \text{ and } \forall\, 0 \le i \le n : \bot \notin A(L_i) \\ 1/2 & \mathit{else} \end{cases}$$

where $((L_i, S_i))_{0 \leq i \leq n}$ is the counterexample for the case that $\mathsf{D}, S \models_2 \phi$ does not hold. By this, we postpone the distinction between abstract and concrete counterexamples to a simple syntactical check, which is also implemented in the SARMC tool. Note that this mapping schema allows us to employ any of the available and highly optimised model-checking engines in order to compute our abstract three-valued satisfaction relation.

In its default mode, the SARMC tool activates the tail heuristic (cf. page 122) for the validation of counterexamples. We evaluate and discuss the effect of this heuristic on a representative case study presented in Section 7.2.2 below.

7.2 Case Studies

We consider four case studies from different domains to evaluate our verification approach. Section 7.2.1 will provide the evaluation data for the adhoc networking case study, which was introduced as the running example for this thesis in Section 1.2. In Section 7.2.2 we investigate a case study from the domain of access control systems. This example has been originally been introduced in [KK06] and represents an unbounded system of servers which are capable of spawning new processes. We are able to re-establish the known safety properties of this system and in addition investigate full temporal specifications. Section 7.2.3 models the Automated Rail Cars System [HG97] as a Dynamic Evolution System and demonstrate that our approach is in particular able to treat the handing-over of processes. Finally, a DCS model of the Car Platooning [HESV91] case study is presented and analysed in Section 7.2.4. Requirements for this case study are in particular given in terms of an LSC.

All experiments have been performed on a Linux host equipped with 16 GBytes of RAM and an Intel® Xeon CPU with a frequency of 3 GHz.

Notations

We start our evaluation phase by reconsidering the DCS example 'C_{ab}' from Figure 6.9 that was used to demonstrate the combination of communication and evolution constraints. On the basis of this example we will also explain our notations for technical data like running times and memory consumption. The discussion on page 133 indicates that no process is able to reach state 's_1', that is, the specification

$$\phi := \mathsf{G}\left(\neg s_1(x)\right)$$

is expected to hold for the DCS model.

According to algorithm 2, the SARMC tool performs the steps given in the following table in order to establish this property. In the tables, the first column denotes the verification task which has been initiated by the tool. The second column gives the maximal size of the spotlight that is induced by the corresponding specification, that is, by its number of variables. The third column list the (three-valued) result of the verification task, and if the result is not '1' the name of the obtained counterexample is given. The last two columns show the running time and the memory consumption of the actual task. Note that the times to actually compute the employed evolution and communication constraints are in the range of some milliseconds for all considered cases studies and are hence not explicitly listed. The last row combines the outcomes of the individual verification task by summing up the overall verification times and by listing the maximum amount of required memory.

task	spot	result	time	memory
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\phi]\!]$	1	$1/2$ $(\delta^\sharp_1)$	0.2 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi]\!]$	2	$1/2$ $(\delta^\sharp_2)$	0.5 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi)]\!]$	3	1	2.1 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\neg\varphi^t(\delta^\sharp_2) \rightarrow (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi)]\!]$	2	1	0.6 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\neg\varphi^t(\delta^\sharp_1) \rightarrow \phi)]\!]$	1	$1/2$ $(\delta^\sharp_3)$	0.2 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_3) \vee \phi)]\!]$	2	1	0.3 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![(\neg\varphi^t(\delta^\sharp_1) \wedge \varphi^t(\delta^\sharp_3)) \rightarrow \phi)]\!]$	1	1	0.2 s	2 MB
		$\mathsf{C}_{\mathsf{ab}} \models \phi$	4.1 s	2 MB

In the table above, the first row corresponds to the analysis of the specification in the initial spotlight abstraction, which yields an abstract counterexample $\delta^\sharp_1$. The next task performs the validation of this counterexample which again yields an abstract counterexample $\delta^\sharp_2$. This run is identified as being spurious in the third row and the corresponding evolution constraints is used to validate the first counterexample $\delta^\sharp_1$ in the task in row four. Now $\delta^\sharp_1$ is identified to be spurious and the original specification is analysed under the corresponding evolution constraint in row five. This yields another abstract counterexample $\delta^\sharp_3$ that is invalidated and used for a further refinement such that the original specification can be established by the last verification task.

The evolution constraints for the first and third counterexample remove the spurious receptions of a 'b' and 'a' message, respectively. As discussed on page 133 we can reduce the overall verification time by incorporating the communication constraint for 'b' which we will denote by $\varphi(b)$.

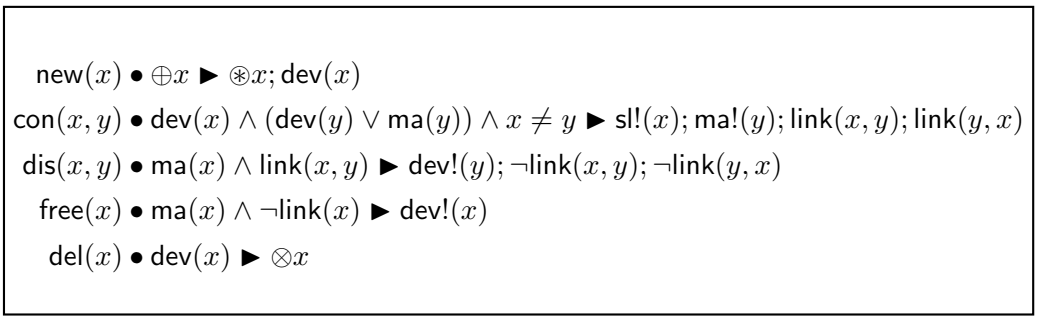


Figure 7.1: The adhoc networking DES 'Ad'.

task	spot	result	time	memory
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\varphi(b) \rightarrow \phi]\!]$	1	$1/2$ $(\delta^\sharp_3)$	0.3 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![\varphi(b) \rightarrow (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_3) \vee \phi)]\!]$	2	1	0.7 s	2 MB
$\mathsf{C}^\sharp_{\mathsf{ab}}[\![(\varphi(b) \wedge \neg\varphi^t(\delta^\sharp_3)) \rightarrow \phi)]\!]$	1	1	0.5 s	2 MB
		$\mathsf{C}_{\mathsf{ab}} \models \phi$	1.5 s	2 MB

As a consequence, the spurious reception of 'b' messages no longer occurs as abstract counterexample and hence only the reception of an 'a' message has to be removed by a counterexample analysis. This reduces the overall verification time from 4.1 seconds to 1.5 seconds. Note that however the running times of individual tasks are slightly larger than before as the representation of the counter values increase the overall state space of the system.

7.2.1 AdHoc Networking

We present the analysis results of the example specifications for the adhoc networking system Ad, which was used as the running example in this thesis. Recall that this example comprises the five evolution rules given in Figure 7.1, which generate an arbitrary number of arbitrarily large star-shaped topologies of devices. The rules are explained in detail on page 35.

Firstly, we reconsider the specification

$$\phi_2 := \mathsf{G}\ \big(\neg\mathsf{sl}(x)\big)$$

which has been discussed on page 102. It takes `SARMC` less than a second to discover a concrete counterexample for this specification under a spotlight comprising two processes. The two steps of the underlying abstraction refinement algorithm and their results are listed in the following table.

task	spot	result	time	memory
$\mathsf{Ad}^\sharp[\![\phi_2]\!]$	1	$1/2$ $(\delta_2^\sharp)$	0.2 s	2 MB
$\mathsf{Ad}^\sharp[\![\neg\varphi^t_{\mathsf{U}}(\delta_2^\sharp) \vee \phi_2]\!]$	2	0 (δ_2)	0.4 s	2 MB
		$\mathsf{Ad} \not\models \phi_2$	0.6 s	2 MB

The strict counterexample specification for the counterexample $\delta_2^\sharp$ (cf. Fig. 6.3), which has been generated by SARMC according to Definition 6.24 and which leads to the identification of the concrete counterexample δ_2 (cf. Fig. 6.4), is

$$\begin{aligned}\varphi^t_{\mathsf{U}}(\delta_2^\sharp) = &\neg(\mathsf{con}(x,b) \wedge \circledcirc x \wedge \neg\mathsf{dev}(x) \wedge \mathsf{sl}(x) \wedge \neg\mathsf{ma}(x) \wedge \neg\mathsf{link}(x,x))\\ &\neg(\mathsf{con}(x_1,b) \wedge \circledcirc x_1 \wedge \neg\mathsf{dev}(x_1) \wedge \mathsf{sl}(x_1) \wedge \neg\mathsf{ma}(x_1) \wedge \neg\mathsf{link}(x_1,x_1))\\ &\mathsf{U}\ \big(\mathsf{con}(x,x_1) \wedge \circledcirc x \wedge \neg\mathsf{dev}(x) \wedge \mathsf{sl}(x) \wedge \neg\mathsf{ma}(x) \wedge \neg\mathsf{link}(x,x)\big)\end{aligned}$$

In fact, the run δ_2 from Figure 6.4 presents a definite violation of the negation of this specification under a spotlight comprising two processes $\{u_1, u_2\}$ for the valuation

$$[b \mapsto \bot, x \mapsto u_1, x_1 \mapsto u_2].$$

Whenever SARMC identifies a concrete counterexample, it will represent it graphically to the user by invoking a so-called trace-viewer program. A screenshot of this tool showing the counterexample 'δ_2' is given in Figure 7.2.

Secondly, we reconsider the specification

$$\phi_4 := \mathsf{G}\,\big(\,\mathsf{link}(x_1,x_2) \rightarrow \neg\ominus x_1\,\big)$$

which has discussed on page 84ff. It takes less than 2 seconds for SARMC to establish that this specification holds for Ad. The individual verification tasks are listed in the following table.

task	spot	result	time	memory
$\mathsf{Ad}^\sharp[\![\phi_4]\!]$	2	$1/2$ $(\delta_4^\sharp)$	0.4 s	2 MB
$\mathsf{Ad}^\sharp[\![\neg\varphi^t_{\mathsf{U}}(\delta_4^\sharp) \vee \phi_4]\!]$	3	1	0.9 s	2 MB
$\mathsf{Ad}^\sharp[\![\neg\varphi^t(\delta_4^\sharp) \rightarrow \phi_4]\!]$	2	1	0.4 s	2 MB
		$\mathsf{Ad} \models \phi_4$	1.7 s	2 MB

The initial verification task yields an abstract counterexample $(\delta_4^\sharp)$, which we already discussed on page 85 (cf. Fig. 5.3). The second task corresponds to the validation phase. To this end, the strict formula of the abstract counterexample is computed (the resulting formula is given on page 119) and analysed. As the

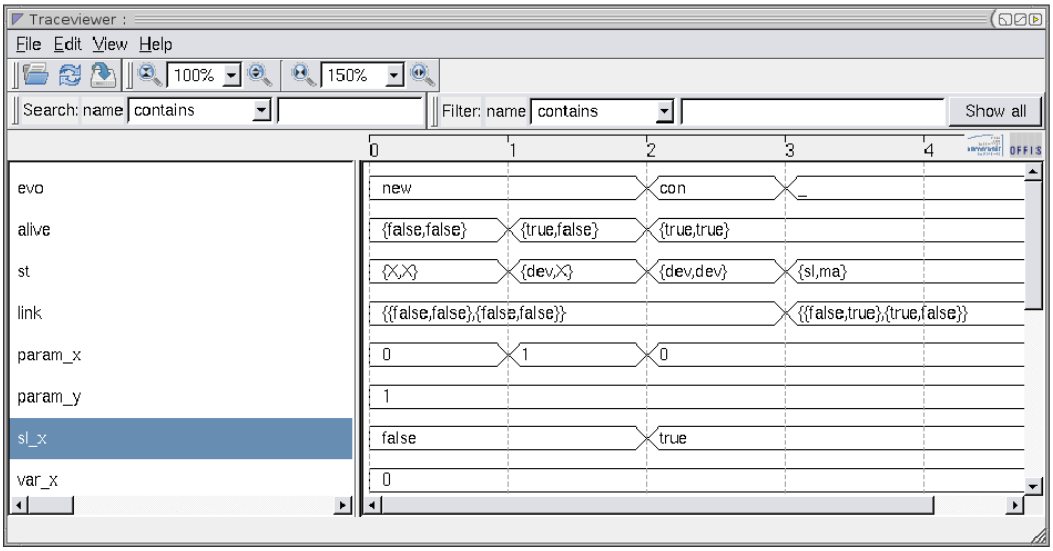


Figure 7.2: The trace-viewer tool showing a concrete counterexample.

counterexample formula introduces a fresh variable, the size of the spotlight is automatically increased from two to three. The verification yields '1' by which we obtain the evolution constraint

$$\neg\varphi^t(\delta_4^\sharp) := \mathsf{G}\left(\mathsf{dis}(x_{4,1}, x_1) \rightarrow \neg(\mathsf{dev}(x_1) \wedge \neg\mathsf{link}(x_1, x_1) \wedge \mathsf{link}(x_1, x_2))\right)$$

Using this evolution constraint for abstraction refinement yields '1' in the third verification task, by which the initial specification ϕ_4 is shown to be correct.

7.2.2 Public/Private Servers

The "Public/Private Servers" case study has been introduced in [KK06] in order to evaluate their verification approach for Graph Transformation Systems (see the discussion on related work in Sect. 6.4). This example is also considered by other researchers in [SWJ08].

The system initially comprise one private server, and an arbitrary number of public servers may be created during runtime. Any (private or public) server may establish connections to other public servers. A server is also able to spawn new processes. More specific, a public server may create external processes and a private server may create internal processes. These processes can then move along existing server connections to other servers. At some point in time, the private server can decide to turn itself into a public server.

$$\mathsf{newpub}(x) \bullet \oplus x \blacktriangleright \circledast x; \mathsf{pub}(x)$$
$$\mathsf{pub2pubc}(x) \bullet \mathsf{pub}(x) \blacktriangleright \mathsf{pubc!}(x)$$
$$\mathsf{prv2prvc}(x) \bullet \mathsf{prv}(x) \blacktriangleright \mathsf{prvc!}(x)$$
$$\mathsf{conpubpubc}(x,y) \bullet \mathsf{pub}(x) \wedge \mathsf{pubc}(y) \wedge x \neq y \blacktriangleright \mathsf{con}(x,y); \mathsf{pub!}(y)$$
$$\mathsf{conprvpubc}(x,y) \bullet \mathsf{prv}(x) \wedge \mathsf{pubc}(y) \wedge x \neq y \blacktriangleright \mathsf{con}(x,y); \mathsf{pub!}(y)$$
$$\mathsf{conprvprvc}(x,y) \bullet \mathsf{prv}(x) \wedge \mathsf{prvc}(y) \wedge x \neq y \blacktriangleright \mathsf{con}(x,y); \mathsf{prv!}(y)$$
$$\mathsf{newext}(x,y) \bullet \mathsf{prv}(x) \wedge \oplus y \blacktriangleright \circledast y; \mathsf{int}(y); \mathsf{acc}(x,y)$$
$$\mathsf{newint}(x,y) \bullet \mathsf{pub}(x) \wedge \oplus y \blacktriangleright \circledast y; \mathsf{ext}(y); \mathsf{acc}(x,y)$$
$$\mathsf{crossint}(x,y,z) \bullet \mathsf{int}(x) \wedge \mathsf{acc}(x,y) \wedge \mathsf{con}(y,z) \blacktriangleright \neg\mathsf{acc}(x,y); \mathsf{acc}(x,z)$$
$$\mathsf{crossext}(x,y,z) \bullet \mathsf{ext}(x) \wedge \mathsf{acc}(x,y) \wedge \mathsf{con}(y,z) \blacktriangleright \neg\mathsf{acc}(x,y); \mathsf{acc}(x,z)$$
$$\mathsf{prv2pub}(x) \bullet \mathsf{prv}(x) \blacktriangleright \mathsf{pub!}(x)$$

Figure 7.3: The "Public/Private Servers" case study as DES '**prvpub**'.

Figure 7.2.2 shows the straight-forward translation of the case study given in terms of graph transformation rules into the DES language. The system comprises eleven evolution rules which we will briefly describe. The first rule corresponds to the ability of creating arbitrarily many public servers, that is, any dead process may appear and become a public server. The next two rules set servers into a "connecting mode" by switching from '**pub**' to **pubc** (for public servers) and from **prv** to **prvc** (for private servers). Any server in this connection mode may then by connected by some other server, such that both private and public servers may connect to some public server being in state '**pubc**' (rules four and five). Also, a private server may connect to some other private server being in state '**prvc**'. The next two rules allow the system to create internal and external process by private and public servers, respectively. These processes may transition from the server to which they are connected via '**acc**' to other servers following some '**con**' connection. Note that these two rules do not restrict the kind of the servers. Finally, the last rule transforms the private server into a public one. The initial snapshot for the system comprises exactly one private server, that is, we set $S := (\{u_1\}, (\{\mathsf{prv}(u_1)\}, \{\}))$.

In [KK06], two safety invariants are specified as "bad graph patterns" and verified for the case study. Informally, the two specifications are given as

(NC) No connection will ever be created from a public to a private server.

(EP) External process will never access private servers.

and we may formulate them in our specification language as follows:

$$\phi_{\mathsf{NC}} := \mathsf{G}\,\neg\big(\mathsf{pub}(x) \wedge \mathsf{prv}(y) \wedge \mathsf{con}(x,y)\big)$$
$$\phi_{\mathsf{EP}} := \mathsf{G}\,\neg\big(\mathsf{ext}(x) \wedge \mathsf{prv}(y) \wedge \mathsf{acc}(x,y)\big)$$

The first specification ϕ_{NC} is verified by `SARMC` in the very first iteration in less than one second. Hence, no refinement was necessary to establish this property. The second specification ϕ_{EP} requires more effort, as shown in the following table.

task	spot	result	time	memory
$\mathsf{prvpub}^\sharp_S[\![\phi_{\mathsf{EP}}]\!]$	2	$1/2$ $(\delta^\sharp_1)$	0.5 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_{\mathsf{EP}}]\!]$	3	1	1.8 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t(\delta^\sharp_1) \rightarrow \phi_{\mathsf{EP}}]\!]$	2	$1/2$ $(\delta^\sharp_2)$	0.6 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee \phi_{\mathsf{EP}}]\!]$	3	1	2.0 s	3 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t(\delta^\sharp_1) \wedge \neg\varphi^t(\delta^\sharp_2) \rightarrow \phi_{\mathsf{EP}}]\!]$	2	1	0.6 s	2 MB
		$\mathsf{prvpub}_S \models \phi_{\mathsf{EP}}$	5.5 s	3 MB

Two abstract counterexamples have been obtained. In $\delta^\sharp_1$, there is an abstract evolution $\mathsf{crossext}(u_2, \bot, u_1)$ leading to a configuration with

$$\{\mathsf{ext}(u_2), \mathsf{prv}(u_1), \mathsf{acc}(u_2, u_1)\}$$

This evolution is contained in the abstract system as '$\mathsf{acc}(u_2, \bot)$' and '$\mathsf{con}(\bot, u_1)$' are always possibly satisfied. The abstract evolution in $\delta^\sharp_2$ is '$\mathsf{conprvprvc}(\bot, u_1)$', stemming from the fact that '$\mathsf{prv}(\bot)$' is possibly satisfied.

Both abstract counterexample are directly identified to be spurious by our validation strategy. The two resulting evolution constraints are the following:

$$\neg\varphi^t(\delta^\sharp_1) = \mathsf{G}\,\big(\mathsf{crossext}(x,b,y) \rightarrow \neg(\mathsf{ext}(x) \wedge \mathsf{prv}(y) \wedge \mathsf{acc}(x,y) \wedge \ldots)\big)$$
$$\neg\varphi^t(\delta^\sharp_2) = \mathsf{G}\,\big(\mathsf{conprvprvc}(b,y) \rightarrow \neg(\mathsf{prv}(y) \wedge \ldots)\big)$$

To ease readability, we use dots to abstract from those terms that are actually not relevant for effectiveness of the evolution constraints. The first constraint states that a '$\mathsf{crossext}$' evolution cannot lead to a snapshot where an external process has access to a private server. The second evolution constraint removes the evolution that connects two private servers, which is clearly spurious as there is at most one private server in the system. Both constrains in combination are sufficient to establish the property in the last verification task. Notably,

both evolution constraints represent very natural and intuitive invariants of the systems.

The graph based approaches from [KK06, SWJ08] are able to establish both properties in about one second, that is, they are faster on ϕ_{EP} than our approach. In contrast, we are not limited to invariant specifications but are also able to establish full temporal properties, for example that some external process never becomes an internal process, i.e.

$$\phi_p := \mathsf{G}\left(\mathsf{int}(x) \rightarrow \neg\, \mathsf{F}\, \mathsf{ext}(x)\right)$$

The `SARMC` tool proves this specification in less than a second. Clearly, this property depends on the fact that processes do not disappear from the system. For the extended DES $\mathsf{prvpubD}$ defined as

$$\mathsf{prvpubD} := \mathsf{prvpub} \cup \{\mathsf{destroy}(x) \bullet \mathsf{int}(x) \vee \mathsf{ext}(x) \blacktriangleright \otimes x\}$$

we obtain a concrete counterexample as follows.

task	spot	result	time	memory
$\mathsf{prvpubD}^\sharp_S[\![\phi_p]\!]$	1	$1/2$ $(\delta^\sharp_1)$	0.3 s	2 MB
$\mathsf{prvpubD}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_p]\!]$	2	$1/2$ $(\delta^\sharp_2)$	0.7 s	2 MB
$\mathsf{prvpubD}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_p)]\!]$	3	0 (δ_3)	4.6 s	3 MB
		$\mathsf{prvpubD}_S \not\models \phi_p$	5.6 s	3 MB

We see that the validation of the first abstract counterexample yields another abstract counterexample. The validation of this counterexample leads to the detection of a concrete counterexample. Likewise, the size of the spotlight is gradually extended in each validation phase until we have three concrete processes, say u_1, u_2 and u_3, in the spotlight under which a definite violation can be observed with the following evolutions:

$$\mathsf{newpub}(u_2) \rightarrow \mathsf{newint}(u_2, u_3) \rightarrow \mathsf{destroy}(u_3) \rightarrow \mathsf{newext}(u_1, u_3)$$

The violating run reuses the process identity u_3 to create a new internal process after it has disappeared from the system. If we now require that the transformation of some internal to an external process is only forbidden *during the lifetime* of the process by stating

$$\mathsf{G}\left(\mathsf{int}(x) \rightarrow \neg(\circledcirc x\ \mathsf{U}\ \mathsf{ext}(x))\right)$$

we are able to establish this property also for $\mathsf{prvpubD}$ in less than a second.

The last experiment on the public/private server case study evaluates the effect of the tail heuristics. The specification is

$$\phi_w := \mathsf{G}\,\neg\big(\mathsf{ext}(x) \wedge \mathsf{int}(y)\big)$$

stating that never both an external and an internal process exists. We expect this specification to be false and search for a concrete counterexample that witnesses the existence of such processes. The verification tasks of `SARMC` using the (default) tail heuristics are as follows.

task	spot	result	time	memory
$\mathsf{prvpub}^\sharp_S[\![\phi_w]\!]$	2	1/2 ($\delta^\sharp_1$)	0.4 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_w]\!]$	3	1/2 ($\delta^\sharp_2$)	2.7 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_w)]\!]$	4	0 (δ_3)	20.1 s	3 MB
		$\mathsf{prvpub}_S \not\models \phi_w$	23.2 s	3 MB

The first abstract counterexample comprises two abstract evolutions, namely $\mathsf{newint}(\bot, u_1)$ and $\mathsf{newext}(\bot, u_2)$. Due to the tail heuristics, only the last evolution is validated in the second verification task under a spotlight of three. This task yields another abstract counterexample comprising the '$\mathsf{newint}(\bot, u_1)$' evolution, which is successfully validated in the next verification under a spotlight of four, yielding a concrete counterexample.

If we disable the tail heuristics, *both* abstract evolutions of $\delta^\sharp_1$ are validating directly in the first validation phase, leading to the same concrete counterexample as follows.

task	spot	result	time	memory
$\mathsf{prvpub}^\sharp_S[\![\phi_w]\!]$	2	1/2 ($\delta^\sharp_1$)	0.4 s	2 MB
$\mathsf{prvpub}^\sharp_S[\![\neg\varphi_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_w]\!]$	4	0 (δ_3)	16.1 s	3 MB
		$\mathsf{prvpub}_S \not\models \phi_w$	16.5 s	3 MB

We observe that the validation phase under a spotlight of three is skipped, and `SARMC` directly increases the spotlight from two to four. Clearly, this reduces the overall verification time. It hence is advisable to disable the tail heuristics when expecting a counterexample for the specification under consideration. By this, all abstract evolutions are considered and validated in the subsequent verification task, thereby potentially saving intermediate indefinite validation results as in the example above.

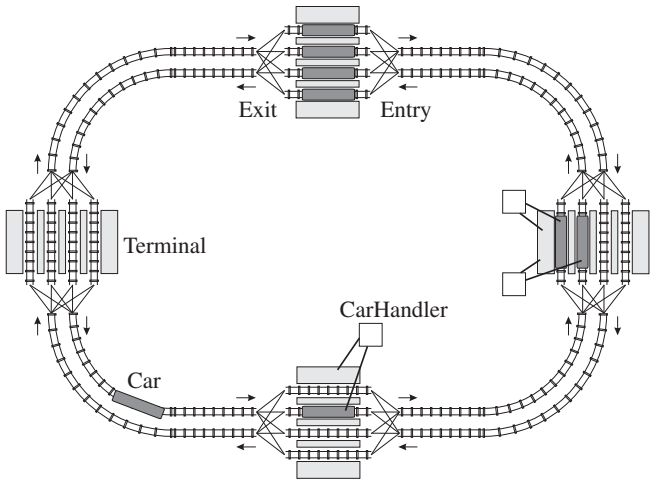


Figure 7.4: Automated Rail Cars System [HG97]

In contrast we observe that disabling the tail heuristic for the verification of ϕ_{EP} increases the overall running time to about 8 minutes, basically as the validation of the complete abstract counterexamples requires a spotlight size of five processes. In general the performance benefits of having the tail heuristic enabled outweigh its disadvantage of potentially forcing intermediate extension steps of the spotlight.

7.2.3 Automated Rail Cars System

The *Automated Rail Cars System* (ARCS) has been introduced in [HG97] as an illustrating example for modelling executable systems using UML state-charts. It comprises a cyclic path of two railways, one for clockwise, the other for counter-clockwise travelling of passengers in railcars. Several terminals are located along the railways where passengers can enter and leave a railcar. See Figure 7.4 for a graphical illustration of the system.

Each terminal has a number of platforms where railcars are waiting for their employment. Whenever a railcar approaches a terminal, the terminal has to allocate a platform for the approaching railcar. For this purpose, the terminal is creating a so-called CarHandler object which then becomes responsible for the negotiation between the railcar and the resources of the terminal.

This procedure employs a typical interaction pattern that can often be found in distributed communication systems, namely the *handing-over* of processes. In this pattern, some process p has connections to two different processes p_1 and p_2 and uses this relationship to introduce process p_2 to process p_3. In other

$$
\begin{aligned}
\mathsf{newcar}(x) &\bullet \oplus x \blacktriangleright \circledast x; \mathsf{cruise}(x)\\
\mathsf{detect}(x,y) &\bullet \mathsf{cruise}(x) \wedge \mathsf{term}(y) \blacktriangleright \mathsf{appr!}(x); \mathsf{req}(y,x)\\
\mathsf{handover}(x,y,z) &\bullet \mathsf{appr}(x) \wedge \mathsf{term}(y) \wedge \mathsf{req}(y,x) \wedge \oplus z \blacktriangleright \circledast z; \mathsf{carh}(z); \mathsf{hnd}(z,x); \neg\mathsf{req}(y,x)\\
\mathsf{busy}(x,y) &\bullet \mathsf{carh}(x) \wedge \mathsf{appr}(y) \wedge \mathsf{hnd}(x,y) \blacktriangleright \mathsf{cruise!}(y); \neg\mathsf{hnd}(z,x)\\
\mathsf{free}(x,y) &\bullet \mathsf{carh}(x) \wedge \mathsf{appr}(y) \wedge \mathsf{hnd}(x,y) \blacktriangleright \mathsf{park!}(y); \neg\mathsf{hnd}(z,x)\\
\mathsf{leave}(x,y) &\bullet \mathsf{park}(x) \wedge \mathsf{carh}(y) \wedge \mathsf{hnd}(x,y) \blacktriangleright \mathsf{cruise!}(x); \neg\mathsf{hnd}(x,q); \otimes y
\end{aligned}
$$

Figure 7.5: The "ARCS" case study as DES 'arcs'.

words, process p hands over p_2 to p_3. In the terms of the case study, a terminal that has received a request from a car creates a new handler process and hands over the identity of the car to the new car handler.

We will demonstrate that our abstraction refinement approach is able to treat handover patterns on the basis of the DES implementation of the ARCS case study as given in Figure 7.2.3. We focus on a single terminal which an arbitrary number cars may request to enter. The six evolution rules read as follows. The first rule creates fresh cars which may detect the terminal process via the second rule, thereby establishing a 'req' link from the terminal to the car. This request triggers the third rule which in turn creates a new car handler process and hands over the identity of the requesting car to the new handler by establishing a 'hnd' connection among them. The car handler may non-deterministically decide to either disallow the car to enter the terminal by the fourth rule called 'busy' or to grant the car's request which in turn enters its parking mode by the fifth rule called 'free'. A parking car then may leave the terminal by the last rule whereby the corresponding handler process is destroyed.

We begin the analysis of the arcs system by considering a simple reachability property, namely $\phi_1 := \mathsf{G}\left(\neg\mathsf{park}(x)\right)$ for which we expect a counterexample that demonstrates how a car can enter its parking mode. The SARMC tool performs the following verification steps by which the spotlight is gradually increased until a concrete counterexample can be identified.

task	spot	result	time	memory
$\mathsf{arcs}^\sharp_S[\![\phi_1]\!]$	1	$1/2$ $(\delta^\sharp_1)$	0.4 s	2 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_2]\!]$	2	$1/2$ $(\delta^\sharp_2)$	0.5 s	2 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_2)]\!]$	4	0 (δ_3)	5.1 s	3 MB
		$\mathsf{arcs}_S \not\models \phi_1$	6.0 s	3 MB

The next requirements addresses the handover procedure. A natural requirement is that a car is not handed over twice. More precisely, after a car that has been handed over it is not handed over again while it is in the approaching mode. In order to concisely formulate this requirement we make use of a particular feature of SARMC that allows us to actually provide less variables for an evolution atom than the arity of the corresponding predicate. By this, the non-specified variables become *don't care values* as the evaluation of the term with respect to an evolution ground atom only considers the given variables. For example, the specification

$$\phi_2 := \mathsf{G}\,\big(\ \mathsf{handover}(x,y) \rightarrow \neg(\mathsf{appr}(x)\ \mathsf{U}\ \mathsf{X}\,\mathsf{handover}(x))\ \big)$$

requires that whenever process x is handed by process y to some anonymous handler, then it is afterwards not handed over again (by some anonymous terminal to some anonymous handler) while it is approaching. We exploit the fact that the actual handler processes are not relevant for the specification. In return, we start with a smaller size of the spotlight as the specification comprises fewer variables.

task	spot	result	time	memory
$\mathsf{arcs}^\sharp_S[\![\phi_2]\!]$	2	$1/2$ $(\delta^\sharp_1)$	0.5 s	2 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_2]\!]$	3	$1/2$ $(\delta^\sharp_2)$	1.8 s	2 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_2) \vee (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_2)]\!]$	5	1	58.1 s	7 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t(\delta^\sharp_2) \rightarrow (\neg\varphi^t_{\mathsf{U}}(\delta^\sharp_1) \vee \phi_2)]\!]$	3	1	1.9 s	3 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t(\delta^\sharp_1) \rightarrow \phi_2]\!]$	2	1	0.6 s	2 MB
		$\mathsf{arcs}_S \models \phi_2$	62.9 s	7 MB

The tool needs a recursion depth of two in order to identify the second abstract counterexample $\delta^\sharp_2$ (which is indeed an abstract counterexample for the validation of the first abstract counterexample $\delta^\sharp_1$) as spurious. Under the corresponding evolution constraint, the fourth verification task then shows $\delta^\sharp_1$ to be spurious. The resulting evolution constraint for arcs and ϕ_2 is

$$\neg\varphi^t(\delta^\sharp_1) = \mathsf{G}\ \big(\mathsf{handover}(x,y,b) \rightarrow \neg(\mathsf{appr}(x) \wedge \mathsf{term}(y)\ldots)\big)$$

stating that on all concrete runs where ϕ_2 is violated there is no handover of the approaching car x to some other handler b that is represented by the abstract process (cf. Def. 6.4). This constraint sufficiently refines the behaviour of the abstract process such that the original specification can be established.

We are also interesting in liveness properties of the systems such as that an entering car finally dismisses its car handler, formalised as

$$\phi_3 := \mathsf{G}\left(\mathsf{free}(x,y) \rightarrow \mathsf{F} \ominus y\right)$$

Following Definition 4.9 we consider the processes that are denoted by the variables in the specification to be fair, in the sense that each evolution that is constantly enabled for these processes is finally executed. To establish the liveness property under spotlight abstraction we need to eliminate one spurious counterexample that corresponds to that fact that the entering car assumes the abstract identity to be its handler and kills this process when leaving the terminal by rule '$\mathsf{leave}(x,y)$'. In consequence, the real handler keeps being alive. `SARMC` rules out this spurious behaviour by the following steps:

task	spot	result	time	memory
$\mathsf{arcs}^\sharp_S[\![\phi_3]\!]$	2	$1/2$ $(\delta^\sharp)$	0.6 s	2 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp) \vee \phi_3]\!]$	3	1	1.6 s	3 MB
$\mathsf{arcs}^\sharp_S[\![\neg\varphi^t(\delta^\sharp) \rightarrow \phi_3]\!]$	2	1	0.6 s	2 MB
		$\mathsf{arcs}_S \models \phi_3$	2.8 s	3 MB

7.2.4 Car Platooning

The *Car Platooning* case study according to [HESV91] has gained a lot of attention both in industrial and academic projects. We will discuss the relevant approaches below. The overall aim is to increase both the safety and the capacity of highways. To this end, individual cars are supposed to form coordinated groups of interlinked cars driving with a reduced safety distance. Car platooning has several advantages:

- The highway capacity is substantially increased as a lot of space is no longer wasted as safety distances.
- The reduction of aerodynamic resistance within a platoon decreases the consumption of fuel and hence the emission of carbon dioxide.
- The on-board electronics can react faster to breaking manoeuvres than a human driver can, thus hopefully leading to fewer accidents.

The car at the front of a platooning group is called the *leader*, the other cars in this group are *followers*. A car that is not yet involved in a platoon is called a *free agent*. To dynamically form platoons on a highway, individual cars approaching a car platoon may join the platoon after a negotiation phase.

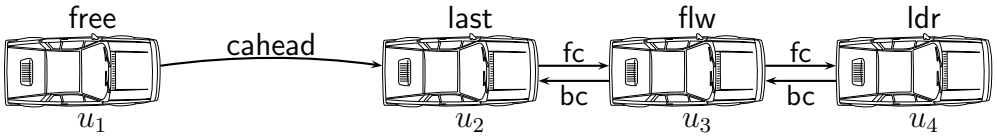


Figure 7.6: A snapshot of the car platooning DCS model.

On the other hand, a car may leave its platoon, e.g. if it is about to reach its destination exit. Hence, the two elementary manoeuvres in order to realise the platooning system are *merge* and *split*.

As already mentioned there already exists a number of approaches to formally specify and analyse this case study. Notably, all approaches that we are aware of fall short in certain aspects. The PATH project[1] designed and analysed car platooning systems, their models [HESV91, Var93] however did not properly respect the dynamic and distributed characteristics of the case study as only a fixed communication topology among at most three cars was considered. Currently, the Car2Car Consortium[2] is establishing an open standard for car communication systems, based on wireless components. So far, they have not designed any formal models of their approach. Academic work on the car platooning case study investigated only the merge operation in isolation, for example in [Bau06, Mey06, BTW07a, Wes08]. Clearly, these restrictions hides problems that may occur for example when not properly disconnected cars try to re-merge with different platoons, leading to inconsistent connection topologies. Note that one of the formal models of [Bau06] integrates a split operation in a purely synchronous variant where both cars disconnects in one atomic step. This does however not properly reflect the uncoupled nature of the car platooning manoeuvres for distributed implementations of the protocols.

DCS model

Our DCS model of the car platooning system comprises in particular a split mechanism, which is initiated by the leader of the platoon. We assume four basic car modes: a car can be a free agent ('free'), a leader ('ldr'), a follower ('flw') or the last follower of a platoon ('last'). A snapshot of the system is shown in Figure 7.2.4, where the three cars u_2, u_3, u_4 form a platoon. The communication topology is organised as a doubly-linked list, such that the car in *front* in reachable under the 'fc' channel and the *back* car via the 'bc' channel. This interlinking schema accounts for the fact that the sending radius of the

[1] `http://www.path.berkeley.edu`
[2] `http://www.car-to-car.org`

communication hardware of a car is typically limited. In Figure 7.2.4, the free car u_1 is about to detect the car 'u_2' driving in front, which is represented by the pending environment message 'cahead'. The reception of this message will trigger the merge negotiation phase as described below.

The behaviour of a car is given as a DCS protocol in Figure 7.7. Both free agents and leaders a sensible for detecting new cars driving in front of them. This recognition is modelled by a car-ahead message ('cahead') sent by the environment. The reception of such a message triggers the sending of a merge request message ('merge') to the detected car. If this message is acknowledged ('mack') by the recipient the back car proceeds to its new mode. The leader of a platoon can leave the platoon by sending a split message ('split') to its back car. After this split request has been acknowledged, the former leader clears its 'bc' channel and becomes a free car. A split request is acknowledged by sending a 'sack' message, and the new mode of the sender depends on the cars' former position in the platoon, that is, a car in mode 'last' becomes a free car while a car in mode 'flw' becomes the new leader of the platoon.

We obtain the following characterisation of the message dependencies of the DCS protocol in terms of reply messages according to Section 6.3.

$$\begin{aligned} \lhd_{\mathsf{cars}}(\mathsf{cahead}) &= (\{\mathsf{merge}\}, 1) & \qquad \lhd_{\mathsf{cars}}(\mathsf{mack}) &= (\{\}, 0) \\ \lhd_{\mathsf{cars}}(\mathsf{merge}) &= (\{\mathsf{mack}, \mathsf{split}\}, 2) & \qquad \lhd_{\mathsf{cars}}(\mathsf{sack}) &= (\{\}, 0) \\ \lhd_{\mathsf{cars}}(\mathsf{split}) &= (\{\mathsf{sack}\}, 1) \end{aligned}$$

These sets concisely capture the basic negation principles of both the merge and the split manoeuvres as modelled by the DCS protocol cars. The reply messages of the 'merge' message suggest to employ cutoff value of at most 2.

Analysis Results

The first requirement excludes a circular connection among platoon leaders by stating that no two leaders recognise each other to be their leaders. We formally express this property as

$$\phi_1 := \mathsf{G}\left((\mathsf{ldr}(x) \wedge \mathsf{ldr}(y)) \rightarrow \neg(\mathsf{fc}(x, y) \wedge \mathsf{fc}(y, x))\right)$$

This property has also been investigated for the merge protocol in [BSTW06] where the communication behaviour of the abstract process was refined by an assumption that has manually been derived from the DCS protocol. In our approach the communication constraint '$\varphi(\mathsf{split})$' with a cutoff of 1 suffice to establish this property under a spotlight of two processes within 39.8 seconds.

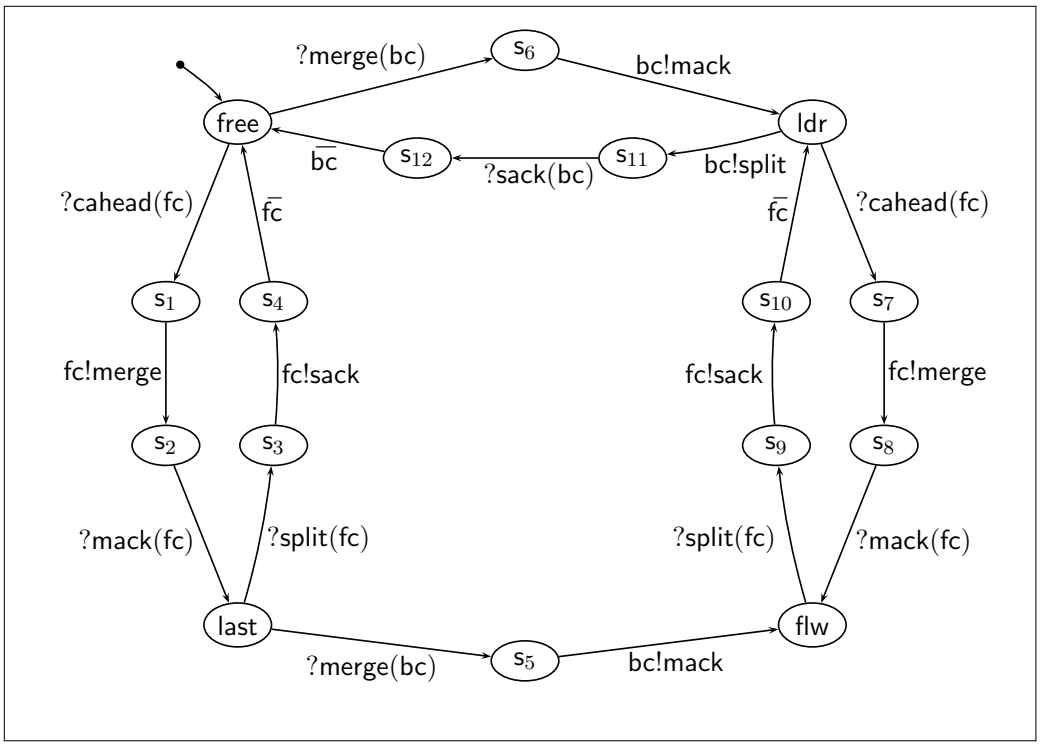


Figure 7.7: The "Car Platooning" case study as DCS 'cars'.

Intuitively, this constraints eliminates the spurious interferences of 'sack' message leading to illegal topologies comprising *connected* free agent cars. Using the full set of communication constraints as computed above increases the verification time to 70.3 seconds.

The protocol is designed such that each merge that has been accepted will be acknowledged to the requester afterwards. We can formally establish this property by verifying the following specification.

$$\phi_2 := \mathsf{G}\left(\mathsf{rcv}[\mathsf{merge}](x,y) \rightarrow \mathsf{F}\,\mathsf{snd}[\mathsf{mack}](x,y)\right)$$

SARMC establishes this response property within 92.4 seconds in the initial spotlight abstraction under the set communication constraints with a cutoff of 2.

The next specification for the platooning system addresses the split mechanism and is given in terms of the Live Sequence Chart in Figure 7.2.4. This LSC requires that whenever a merge is acknowledged by a 'mack' message, the recipient of this message will keep a 'fc' connection to the sender unless it receives a 'split' message. As the lifeline fragment is dashed ("cold") after the reception of the first message, the split message is actually not required to finally occur. However, if it occurs the connection topology must adhere to the

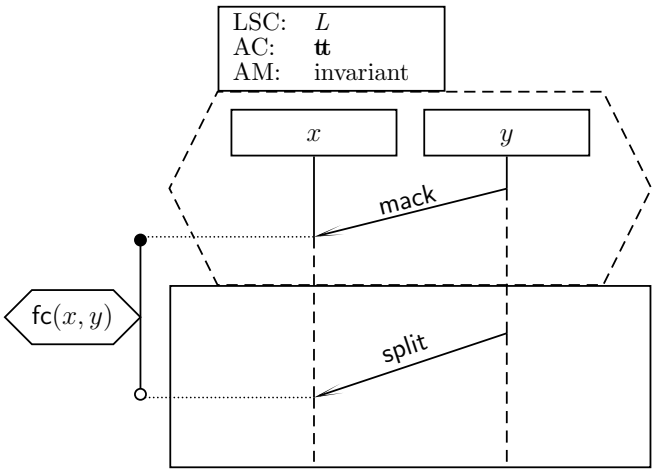


Figure 7.8: The LSC specification for the DCS 'cars'.

local invariant, that is, '$\mathsf{fc}(x, y)$' must be satisfied up to but excluding the point in time of receiving the 'split' message.

As both possible orderings between the sending of 'split' and the reception of 'mack' have to be considered, the LSC requirement translates (cf. Sect. 4.3) to a rather complex temporal logic formula as follows.

$$
\begin{aligned}
&\mathsf{G}\,\big(\,(\neg\mathsf{snd}[\mathsf{mack}](y,x)\;\mathsf{U}\;\mathsf{snd}[\mathsf{mack}](y,x) \wedge \mathsf{X}\,\neg\mathsf{rcv}[\mathsf{mack}](x,y)\;\mathsf{U}\;\mathsf{rcv}[\mathsf{mack}](x,y))\\
&\quad\rightarrow\\
&\quad(\neg\mathsf{snd}[\mathsf{mack}](y,x)\;\mathsf{U}\;\mathsf{snd}[\mathsf{mack}](y,x) \wedge \mathsf{X}\,\neg\mathsf{rcv}[\mathsf{mack}](x,y) \wedge \neg\mathsf{snd}[\mathsf{split}](y,x)\;\mathsf{U}\\
&\quad(\mathsf{snd}[\mathsf{split}](y,x) \wedge \mathsf{X}\,(\neg\mathsf{rcv}[\mathsf{mack}](x,y)\;\mathsf{U}\;\mathsf{rcv}[\mathsf{mack}](x,y) \wedge \mathsf{fc}(x,y) \wedge\\
&\qquad\mathsf{X}\,(\neg\mathsf{rcv}[\mathsf{split}](x,y) \wedge \mathsf{fc}(x,y)\;\mathsf{W}\;\mathsf{rcv}[\mathsf{split}](x,y))))\\
&\quad\vee\\
&\quad(\mathsf{rcv}[\mathsf{mack}](x,y) \wedge \mathsf{fc}(x,y) \wedge \mathsf{X}\,((\neg\mathsf{snd}[\mathsf{split}](y,x) \wedge \mathsf{fc}(x,y)\;\mathsf{W}\\
&\qquad\mathsf{snd}[\mathsf{split}](y,x) \wedge \mathsf{fc}(x,y) \wedge \mathsf{X}\,(\neg\mathsf{rcv}[\mathsf{split}](x,y) \wedge \mathsf{fc}(x,y)\;\mathsf{W}\;\mathsf{rcv}[\mathsf{split}](x,y)))))))\big)
\end{aligned}
$$

SARMC is able to establish this property in about 29 minutes using 382 MBytes of memory in the initial spotlight abstraction under the set communication constraints with a cutoff of 2.

We observe that exploiting the communication constraints for refinement is actually *necessary* to establish the desired properties in a reasonable time. Using the pure CEGSAR approach for the analysis of this case study leads to validation attempts of abstract counterexamples under a spotlight comprising up to 5 processes. These validation tasks do not finish within 24 hours. We conjecture that the increased state space needed for representing the configura-

tion of each car process in combination with the interleaving of asynchronous communication triggers the classical state-explosion problem whenever the size of the spotlight exceeds a certain number of given processes. This observation substantiates the need of auxiliary sources of evolution constraints derived from the specific modelling language. Communication constraints are one particular example for the DCS language.

Note that the CEGSAR loop is still required and suitable to identify concrete counterexamples, for example to witness the reachability of certain topology configurations. For example, the requirement $\phi_f := \mathsf{G}\left(\neg\mathsf{flw}(x)\right)$ cannot be disproved by communication constraints in isolation such that the CEGSAR iteration takes over and detects a concrete counterexample as follows.

task	spot	result	time	memory
$\mathsf{cars}^\sharp[\![\phi_f]\!]$	1	$1/2$ $(\delta^\sharp)$	6.1 s	4 MB
$\mathsf{cars}^\sharp[\![\neg\varphi^t_{\mathsf{U}}(\delta^\sharp) \vee \phi_f]\!]$	2	0 (δ)	39.1 s	11 MB
		$\mathsf{cars} \not\models \phi_f$	45.2 s	11 MB

7.3 Discussion

We have reported on a number of successful verification tasks using the spotlight abstraction refinement procedure. The analysis of the car platooning system, however, indicates that the size of the system under consideration is currently the limiting factor for the applicability of the `SARMC` tool.

To discuss the tool limits in more detail we have to distinguish between two notions of "system size", namely on the one hand the *number of predicates* in the underlying signature in order to describe the configurations and the evolutions of the processes, and on the other hand the *number of processes* to be considered. By the spotlight abstraction principle, the number of processes in the spotlight is completely determined by the *specification*. By this, we can in principle analyse very large numbers of processes (and even infinitely many) as long as the specification addresses only a small number of process configurations. For example, our approach easily handles a variant of the ARCS case study where several terminals are located on a large track, basically as the query reduction principle allows us to focus on small representative cases. The size in terms of the number of predicates is not reducible by our approach, that is, each process inside of the spotlight is completely represented by all its predicates. This fact inhibits the efficient verification of e.g. the platooning case study under spotlight sizes larger than four. To enhance the applicability of the `SARMC` tool it is

hence important to reduce the verification complexity by integrating orthogonal reduction techniques. We discuss some ideas in the concluding section 8.2.

Note that the proposed *tail heuristic* and the usage of *communication constraints* are two particular possibilities in order to reduce to required sizes for the spotlight. From our evaluation phase we observe that both techniques should be enabled in the default mode of the tool. The performance benefits of the tail heuristic more than compensate its disadvantage of potentially forcing intermediate extension steps of the spotlight in all the considered cases. Similarly, the ability to suppress spurious counterexamples *directly* by exploiting communication constraints outweigh the disadvantage of increasing the number of predicates in order to represent the counter values.

Chapter 8

Conclusion

"Exit Light, enter Night."
Metallica (american rock band)

B. Westphal concludes his thesis on fundamental properties of the spotlight abstraction principle by stating that future work on

> *"the topic of refinement is most prominent"* ([Wes08], page 302).

Our work makes a significant step into this research direction by turning existing heuristics for manually refining the abstract transition system into *automatic* procedures. In this chapter, we review our thesis by summarising its main contributions in Section 8.1 and we provide an outlook on future research directions related to our topic in Section 8.2.

8.1 Summary

To be able to concisely describe our refinement approach we first introduced adequate languages to obtain formal models of the addressed class of systems. In general we took a more light-weight modelling and specification approach compared to [Wes08]. The reason is that our focus was not the investigation of the basic properties of spotlight abstraction but rather on devising refinement strategies that extends the applicability of the abstracting technique for the falsification and verification of a useful class of systems and specifications. To this end, we presented in Chapter 3 a clean semantical model which allows us to focus on the basic features of the addressed class of systems, namely the evolutions of a dynamically varying number of interconnected processes. For a symbolic description of possible evolutions we introduced the low-level modelling language of Dynamic Evolution Systems (DES). This language is a contribution on its own as it provides an intuitive characterisation of the

non-trivial behaviour of the corresponding system. We additionally considered the more high-level language of Dynamic Communication Systems (DCS) that allows the system designer to focus on the behaviour of a single process. We integrated this language into our framework by devising a translation from DCS to DES.

A good requirement specification logic for the addressed class of system is a non-trivial problem on its own [BTW07b]. In Chapter 4, we introduced the powerful class of outermost universally quantified first-order linear time logic, which is a proper subclass of the specification logic EvoCTL* as introduced in [Wes08]. The main contribution of this chapter is the new three-valued characterisation of the satisfaction relation for this kind of specification logic. This definition is the key for a concise presentation of the abstraction and refinement principles in the subsequent chapters. We demonstrated the suitability of our specification logic by a number of example specification and by relating the important class of universal LSC specifications to our logic.

We started our investigation of a suitable verification procedure in Chapter 5 by observing that the presence of an unbounded number of interlinked processes renders even simple reachability problems undecidable. We showed this by an encoding of two-counter-machines in terms of DES and DCS systems. We then applied the spotlight abstraction and query reduction principles to our descriptions of dynamic systems and specifications, and we devised a generalised soundness theorem of the abstraction technique in terms of an embedding relation exploiting the three-valued definition of the specification satisfaction relation. This embedding theorem contributes the first formal characterisation of the fact that the spotlight abstraction principle exactly preserves the behaviour of the processes within the spotlight.

The central contribution of Chapter 6 is the iterative refinement algorithm for spotlight abstractions, which instantiates the classical CEGAR framework in a non-usual two-staged manner. The additional second stage accounts for the fact that we identified the *validation* of abstract counterexample as the key problem. In fact, we showed that the validation of counterexamples under spotlight abstraction requires a proper reachability analysis of the underlying system model, which renders the validation problem undecidable for the typical domain of infinite-process systems. Our iterative refinement procedure is based on two complementary kinds of refinement, namely spotlight extension and shadow refinement. Spotlight extension provides us with a sound but necessarily incomplete method of counterexample validation and shadow refinement exploits the information of a spurious counterexample for an effective refinement of the behaviour of the abstract process. Both kinds of refinements were seamlessly

integrated in a general approach for the analysis of dynamic evolution systems via counterexample-guided spotlight abstraction refinement (CEGSAR).

In terms of the DCS language we demonstrate how to statically derive systems invariants in order to improve the proposed refinement algorithm. To this end, we formalised the notion of message dependencies and applied a counter augmentation to the resulting transition system in order to obtain a new form of evolution constraints, the so-called communication constraints. In general, we observe that evolution constraints serves as a general instrument for refining the behaviour of the abstract process under spotlight abstraction, in particular as also the related work in [BTW07a] can be formulated in this framework.

We have practically evaluated our verification approach on a number of realistic case studies. The considered examples cover a wide range of the application domain and are in particular not restricted to a certain class of topology shapes. For example, the adhoc networking system induces star-shaped topologies, the ARCS case studies employs the hand-over of processes and the car platooning system is realised as a linked list of processes. The evaluation phase demonstrates the suitability of our approach. We were able to automatically reestablish a number of properties for which manually intervention was necessary before (e.g. for the car platooning system). Moreover, we proved new properties, including full temporal properties, which were not considered before (e.g. for the existing Public/Private Server case study). An important side effect of our refinement loop is its ability for *falsification*, that is, we are now able to automatically obtain concrete counterexamples under spotlight abstraction, while existing strategies for refinement are tailored to the verification of properties. This is an important contribution as model-checking techniques are often used to prove the reachability of desired system configuration by so-called "drive-to-property" specifications. This is achieved by a falsification of the negation of the configuration description. For larger system our approach still suffers from the state-explosion problem, which we tackle only indirectly by a gradual increase of the the size of the spotlight (in particular by considering the tail heuristic). We discuss some solutions to this problem in the section on future work below.

8.2 Perspectives

Summing up, our work has turned the sound technique of spotlight abstraction into a proper verification instrument that is applicable to a large class of systems comprising a dynamically evolving set of processes. We conclude by

discussing open problems and pointing to further work related to our approach. We elaborate on the four categories efficiency, specification, scope and theory.

Efficiency The main limitation of our approach lies in the combinatorial explosion of the state- and transition-representation for larger models and sizes of the spotlight. The two ways to tackle this problem are obvious, either speed-up the verification procedure itself and/or reduce the required size of the spotlight.

We observed that the symbolic model-checking engine of VIS is not able to efficiently handle the interleaving semantics of processes in combination with asynchronous communication. A natural approach is to consider other verification engines which are tailored to the analysis of communication protocols like variants of the SPIN [Hol04] model-checker. SPIN comes with native support for communication channels and employs a partial order reduction method [God90] in order to eliminate redundant order of interleavings. We supervised a number of experiments [Rak06, Amm07] on the verification of DCS models using SPIN, which indicate that despite these optimisation techniques the explicit state-space representation inhibits the exploration of DCS models like the car platooning system for more than 3 concrete processes with SPIN. There are recent trends to combine symbolic state-space representations with partial-order reduction [KGS06, WYKG08], which are certainly worth to evaluate once corresponding tools are available.

As our abstract transition system is defined in terms of three-valued logic it might be suggestive to employ data-structures for verification that natively deals with the indefinite logical value. This includes the usage of ternary decision diagrams [Sas97] and related tools [LAS00, CDE01, ECD+03]. However, as the indefinite value is purely used to capture the stateless representation of the abstract process we do not expect significant performance gains.

A different approach to reduce the complexity problems is to minimise the size of the spotlight under which an abstract counterexample is to be validated. The tail heuristic in order to validate only a certain part of the counterexample is a first step into this direction. Another approach would be to identify subgroups of abstract evolution in the counterexample that, if they correspond to genuine behaviour at all, have to be executed by the same set of processes. If this is the case we do not need to introduce fresh variables for each occurrence of the abstract process but may rather re-use existing variables. This in return allows to validate under a reduced spotlight. We conjecture that static information like the message dependencies in DCS protocols will allow us to identify related abstract evolutions, as for example the abstract reply of an abstract request can be soundly validated by increasing the spotlight by one rather than by two.

Specification A natural continuation of our work with respect to the requirement specification logic is to consider quantification over paths. We are in particular interested to adapt our approach for the alternating-time temporal logic (ATL) [AHK02] as it allows for the selection of relevant paths based on winning strategies of individual processes. For example, while the universal path quantification in LTL will not establish that any two adjacent process always eventually build a platoon, one may refine this query in ATL by stating

$$\langle\langle \emptyset \rangle\rangle \, \mathsf{G} \, \big(\mathsf{near}(x_1, x_2) \rightarrow \langle\langle \{x_1, x_2\} \rangle\rangle \, \mathsf{F} \, \mathsf{platoon}(x_1, x_2) \big)$$

where $\langle\langle X \rangle\rangle$ is a path quantifier which ranges over all computations where the processes denoted by variables X can actually resolve their internal nondeterminism in a way such that the requirement is satisfied independently of the interaction of other processes. A counterexample-guided refinement algorithm for two-player games is presented in [HJM03]. In this sense, spotlight abstraction addresses the abstraction of an unbounded number of players where the size of the spotlight determines the number of concrete players and the abstract player acts as the system environment.

Scope We observed that our abstraction and refinement technique is able to handle quite different "topological shapes" like stars and lists. It would be interesting to see if this result can be carried over to even more complex datastructures. One example of particular research interest are processes which communicate over unbounded FIFO queues [BZ83]. The idea is to model the content of the queue as a linked list of processes such that each message in the queue corresponds to one of these processes. The push and pop operations then translate to corresponding evolution rules on this list. The effect of the spotlight abstraction refinement strategy would be that a sufficient part of the unbounded queue automatically moves into the spotlight in order to prove or disprove a given specification.

Theory As we are working on a turing-complete formalism our verification algorithm yields a necessarily incomplete solution of the problem. One obvious way to obtain a decidable sub-language is to restrict the set of available identities to a finite set. Then, the abstraction purely serves as a method to reduce the complexity of the verification tasks by only considering a subset of identities concretely. However, there is currently no systematic way to obtain a sufficient number of identities under which every relevant system behaviour is preserved.

It is known that most verification problems for unbounded FIFO queues are undecidable, too. In this context, [AJ96] shows that the reachability prob-

lem becomes decidable when considering unreliable channels, the so-called lossy queues. This observation has a strong practical impact as many real-world channels are in fact unreliable and have to be treated accordingly in the overlying protocol. We believe that "lossyness" may also be a valid assumption in our addressed class of evolving systems as process may disappear quite unexpectedly. It is thus be worth to investigate whether general decidability results can be obtained for the class of *lossy Dynamic Evolution Systems*. This task involves the establishment of a well-quasi order [FS01] on the resulting transition system by incorporating ideas from the graph minor theory [DHiK05]. First approaches in this direction are presented in [JK08].

Bibliography

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *LNCS*, pages 305–318. Springer, 1998.

[ADHR07] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 721–736. Springer, 2007.

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[AHPZ07] Karl Azab, Annegret Habel, Karl-Heinz Pennemann, and Christian Zuckschwerdt. ENFORCe: A system for ensuring formal correctness of high-level programs. In *Proc. 3rd International Workshop on Graph Based Tools (GraBaTs'06)*, volume 1, pages 82–93. Electronic Communications of EASST, 2007.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying Programs with Unreliable Channels. *Inf. Comput.*, 127(2):91–101, 1996.

[AJN+04] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular Model Checking for LTL(MSO). In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 348–360. Springer, 2004.

[AL01] Paul C. Attie and Nancy A. Lynch. Dynamic Input/Output Automata: A Formal Model for Dynamic Systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *LNCS*, pages 137–151. Springer, 2001.

[Amm07] Christian Ammann. Verifikation von DCS-Beschreibungen mit dem Modelchecker SPIN, August 2007. Individuelles Projekt (Studienarbeit), Carl von Ossietzky Universität Oldenburg.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[Bac08] Peter Backes. An Interface between XML-coded DCS protocols and the hiralysis representation of Graph Transformation Systems. Master's thesis, Universität des Saarlandes, 2008.

[Bar46] Ruth C. Barcan. A functional calculus of first order based on strict implication. *Journal of Symbolic Logic*, 11:1–16, 1946.

[Bau06] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität des Saarlandes, 2006.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990.

[BDTW07] Jörg Bauer, Werner Damm, Tobe Toben, and Bernd Westphal. Verification and Synthesis of OCL Constraints Via Topology Analysis. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *AGTIVE*, volume 5088 of *LNCS*, pages 361–376. Springer, 2007.

[BFS02] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL Using Observational Mu-Calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[BG99] Glenn Bruns and Patrice Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *LNCS*, pages 274–287. Springer, 1999.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract

Regular Model Checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 372–386. Springer, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular Model Checking. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.

[Bro99] Udo Brockmeyer. *Verifikation von STATEMATE Designs*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Germany, 1999.

[BSTW06] Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and Verification of Dynamic Communication Systems. In *ACSD*, pages 189–200. IEEE Computer Society, 2006.

[BTW07a] Jörg Bauer, Tobe Toben, and Bernd Westphal. Mind the Shapes: Abstraction Refinement Via Topology Invariants. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *LNCS*, pages 35–50. Springer, 2007.

[BTW07b] Jörg Bauer, Tobe Toben, and Bernd Westphal. The Temporal Logic of Appearance and Disappearance. Reports of SFB TR 14 AVACS 24, AVACS, June 2007. ISSN: 1860-9821, http://www.avacs.org.

[BW07] Jörg Bauer and Reinhard Wilhelm. Static Analysis of Dynamic Communication Systems by Partner Abstraction. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 249–264. Springer, 2007.

[BZ83] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[CD89] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[CDE01] Marsha Chechik, Benet Devereux, and Steve M. Easterbrook. Implementing a Multi-valued Symbolic Model Checker. In Tiziana

Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 404–419. Springer, 2001.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFH+03] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Olaf Stursberg, and Michael Theobald. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *LNCS*, pages 192–207. Springer, 2003.

[CGB86] Edmund M. Clarke, Orna Grumberg, and Michael C. Browne. Reasoning About Networks With Many Identical Finite-State Processes. In *PODC*, pages 240–248, 1986.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CM02] Muffy Calder and Alice Miller. Automatic verification of any number of concurrent, communicating processes. In *ASE*, pages 227–230. IEEE Computer Society, 2002.

[CTV06] 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 *LNCS*, pages 126–141. Springer, 2006.

[CTV08] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 33–47. Springer, 2008.

[Dam03] Dennis Dams. Comparing Abstraction Refinement Algorithms. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

[DFKL07] Clare Dixon, Michael Fisher, Boris Konev, and Alexei Lisitsa. Efficient First-Order Temporal Logic for Infinite-State Systems. *CoRR*, abs/cs/0702036, 2007.

[DH01] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[DHiK05] Erik D. Demaine, Mohammad Taghi Hajiaghayi, and Ken ichi Kawarabayashi. Algorithmic Graph Minor Theory: Decomposition, Approximation, and Coloring. In *FOCS*, pages 637–646. IEEE Computer Society, 2005.

[DJ02] Werner Damm and Bengt Jonsson. Eliminating Queues from RT UML Model Representations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, volume 2469 of *LNCS*, pages 375–394. Springer, 2002.

[DJPV02] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 2852 of *LNCS*, pages 71–98. Springer, 2002.

[DKR00] 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, *FMOODS*, volume 177 of *IFIP Conference Proceedings*, pages 285–304. Kluwer, 2000.

[DKR04] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Who is Pointing When to Whom? In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004.

[DRK02] Dino Distefano, Arend Rensink, and Joost-Pieter Katoen. Model Checking Birth and Death. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *IFIP TCS*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer, 2002.

[DTW06] Werner Damm, Tobe Toben, and Bernd Westphal. On the Expressive Power of Live Sequence Charts. In Thomas W. Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 225–246. Springer, 2006.

[DW05] Werner Damm and Bernd Westphal. Live and let die: LSC based verification of UML models. *Sci. Comput. Program.*, 55(1-3):117–159, 2005.

[ECD+03] Steve Easterbrook, Marsha Chechik, Benet Devereux, Arie Gurfinkel, Albert Lai, Victor Petrovykh, Anya Tafliovich, and Christopher Thompson-Walsh. χchek: a model checker for multi-valued reasoning. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 804–805, Washington, DC, USA, 2003. IEEE Computer Society.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 489–503. Springer, 2006.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[FFQ02] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-Modular Verification for Shared-Memory Programs. In Daniel Le Métayer, editor, *ESOP*, volume 2305 of *LNCS*, pages 262–277. Springer, 2002.

[FGMP03] Gian Luigi Ferrari, Stefania Gnesi, Ugo Montanari, and Marco Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003.

[FJL00] M. Frodigh, P. Johansson, and P. Larsson. Wireless Ad Hoc Networking: The Art of Networking without a Network. *Ericsson Review*, 4, 2000.

[FM98] Melvin Fitting and Richard L. Mendelsohn. *First Order Modal Logic*. Kluwer, 1998.

[FM04] Stephan Flake and Wolfgang Müller. Past- and Future-Oriented Time-Bounded Temporal Properties with OCL. In *SEFM*, pages 154–163. IEEE Computer Society, 2004.

[FS01] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[GJJ06] Tristan Le Gall, Bertrand Jeannet, and Thierry Jéron. Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues. In Michael Johnson and Varmo Vene, editors, *AMAST*, volume 4019 of *LNCS*, pages 204–219. Springer, 2006.

[GL84] Eitan M. Gurari and Ten Hwang Lai. Deadlock detection in communicating finite state machines. *SIGACT News*, 15(4):63–64, 1984.

[GL94] Orna Grumberg and David E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[GMRT06] Ankit Goel, Sun Meng, Abhik Roychoudhury, and P. S. Thiagarajan. Interacting Process Classes. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 302–311. ACM, 2006.

[God90] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *CAV*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.

[Gro96] The VIS Group. VIS: a System for Verification and Synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. CAV*, number 1102 in LNCS, pages 428–432. Springer, 1996.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about Systems with Many Processes. *J. ACM*, 39(3):675–735, 1992.

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GS99] Viktor Gyuris and A. Prasad Sistla. On-the-Fly Model Checking Under Fairness that Exploits Symmetry. *Formal Methods in System Design*, 15(3):217–238, 1999.

[Haa98] J. Haartsen. Bluetooth – the universal radio interface for adhoc, wireless connectivity. *Ericsson Review*, 3, 1998.

[HESV91] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. The Design of Platoon Maneuver Protocols for IVHS. PATH Report UCB-ITS-PRR-91-6, University of California, Berkeley, April 1991.

[HG97] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HJM03] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-Guided Control. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *ICALP*, volume 2719 of *LNCS*, pages 886–902. Springer, 2003.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[HLP01] Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Trans. Software Eng.*, 27(8):749–765, 2001.

[Hol94] Gerard J. Holzmann. The Theory and Practice of A Formal Method: NewCoRe. In *IFIP Congress (1)*, pages 35–44, 1994.

[Hol04] Gerald J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.

[HPR06a] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest Preconditions for High-Level Programs. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *LNCS*, pages 445–460. Springer, 2006.

[HPR06b] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 445–460. Springer-Verlag, 2006.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[Hua90] S. T. Huang. A distributed deadlock detection algorithm for csp-like communication. *ACM Trans. Program. Lang. Syst.*, 12(1):102–122, 1990.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[ID99] C. Norris Ip and David L. Dill. Verifying Systems with Replicated Components in Murϕ. *Formal Methods in System Design*, 14(3):273–310, 1999.

[IT99] ITU-T. *ITU-T Rec. Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.

[JHGP99] Jean-Marc Jézéquel, Wai-Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[JK08] Salil Joshi and Barbara König. Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 214–226. Springer, 2008.

[Jos93] Bernhard Josko. Modular Specification and Verification of Reactive Systems, 1993. Habilitation thesis.

[Jos06] Henning Jost. Eliminating FIFO Message Queues From Dynamic Communication Systems. Master's thesis, Carl von Ossietzky Universität Oldenburg, December 2006.

[KGS06] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 286–299. Springer, 2006.

[KHP+05] Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. Temporal Logic for Scenario-Based Specifications. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 445–460. Springer, 2005.

[KK06] Barbara König and Vitali Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 197–211. Springer, 2006.

[KK08] Barbara König and Vitali Kozioura. Augur 2 - A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electr. Notes Theor. Comput. Sci.*, 211:201–210, 2008.

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Bibliotheca Mathematica. North-Holland, Amsterdam, 1952.

[Klo03] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Germany, 2003.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking - Timed UML State Machines and Collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, volume 2469 of *LNCS*, pages 395–416. Springer, 2002.

[KP99] Yonit Kesten and Amir Pnueli. Verifying Liveness by Augmented

Abstraction. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *LNCS*, pages 141–156. Springer, 1999.

[KP00] 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.

[KPSZ02] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck. Network Invariants in Action. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *LNCS*, pages 101–115. Springer, 2002.

[Kri63] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[KTWW06] 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 *LNCS*, pages 219–233. Springer, 2006.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes : the Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994.

[Kwi89] M. Z. Kwiatkowska. Survey of fairness notions. *Inf. Softw. Technol.*, 31(7):371–386, 1989.

[Lah04] Shuvendu Lahiri. *Unbounded System Verification using Decision Procedure and Predicate Abstraction*. PhD thesis, Carnegie Mellon University, September 2004.

[Lam80] Leslie Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 1980)*, pages 174–185, New York, NY, USA, 1980. ACM.

[LARSW00] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A System for Implementing Static Analyses. In Jens Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.

[LB04] Shuvendu K. Lahiri and Randal E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In Bernhard Steffen

and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 267–281. Springer, 2004.

[LB07] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[Lew68] David Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968.

[Low96] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.

[LS94] Peter B. Ladkin and Barbara B. Simons. Static analysis of multiway synchronization. In John E. Botsford, Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Kelly A. Lyons, Jacob Slonim, and J. Howard Johnson, editors, *CASCON*, page 39. IBM, 1994.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lub84] Boris D. Lubachevsky. An Approach to Automating the Verification of Compact Parallel Coordination Programs I. *Acta Inf.*, 21:125–169, 1984.

[LW05] Stefan Leue and Wei Wei. Counterexample-Based Refinement for a Boundedness Test for CFSM Languages. In Patrice Godefroid, editor, *SPIN*, volume 3639 of *LNCS*, pages 58–74. Springer, 2005.

[Mai00] Monika Maidl. The Common Fragment of CTL and LTL. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 643–652, 2000.

[Mar95] Robert Cecil Martin. *Designing object-oriented C++ applications: using the Booch method*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[MC05] Alice Miller and Muffy Calder. A Generic Approach for the Automatic Verification of Featured, Parameterised Systems. In Stephan Reiff-Marganiec and Mark Ryan, editors, *FIW*, pages 217–235. IOS Press, 2005.

[McM99] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *CHARME*, volume 1703 of *LNCS*, pages 219–234. Springer, 1999.

[McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[McM01] Kenneth L. McMillan. *Getting Started with SMV*. Cadence Design Systems, March 2001.

[Mey06] Roland Meyer. Modelling and Specifying Mobile Systems, July 2006. Presentation Slides. http://dsrg.mff.cuni.cz/teaching/seminars/2006-03-07-Meyer-PiCalculus.pdf.

[Mey09] Roland Meyer. *Structural Stationarity in the π-Calculus*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Germany, 2009.

[Mil71] Robin Milner. An Algebraic Definition of Simulation between Programs. In *Proceedings of the 2^{nd} Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society Press, London, 1971.

[Mil99] Robin Milner. *The Pi Calculus*. CU Press, 1999.

[Min67] Marvin Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.

[MKS08] Roland Meyer, Victor Khomenko, and Tim Strazny. A Practical Approach to Verification of Mobile Systems Using Net Unfoldings. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *LNCS*, pages 327–347. Springer, 2008.

[MP92] Zohar Manna and Armir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, Berlin, January 1992.

[MPR07] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise Thread-Modular Verification. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 218–232. Springer, 2007.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *IC*, 100:1–77, 1992.

[MRS+05] Christian Mrugalla, Oliver Robbe, Ingo Schinz, Tobe Toben, and Bernd Westphal. Formal Verification of a Sensor Voting and Monitoring UML Model. In Siv Hilde Houmb, Jan Jürjens, and Robert France, editors, *CSDUML*, Fredrikstad, Norway, September 2005. Technische Universität München.

[MYRS05] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.

[OMG01] OMG. Unified Modeling Language Specification. Technical Report 1.4-UML-01-09-67, OMG, September 2001.

[OMG06] OMG. Object Constraint Language, version 2.0. Technical Report formal/06-05-01, OMG, 2006.

[OMG07] OMG. Unified Modeling Language: Superstructure, version 2.1.1. Technical Report formal/07-02-05, OMG, February 2007.

[PA01] Santosh Pande and Dharma P. Agrawal. *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, volume 1808 of *LNCS*. Springer, 2001.

[Pen08] Karl-Heinz Pennemann. Development of correct graph transformation systems – Preliminary abstract. In *Graph Transformations (ICGT'08)*, volume 5214 of *LNCS*, pages 508–510. Springer-Verlag, 2008.

[PL99] Ivan Porres Paltor and Johan Lilius. vUML: A tool for verifying UML models. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, USA, October 1977. IEEE.

[Pnu85] Armir Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, pages 123–144, 1985.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), 1980.

[Pos81] J. Postel. Internet Protocol. RFC 791 (Standard), 1981.

[PS08] Amir Pnueli and Yaniv Sa'ar. All You Need Is Compassion. In Francesco Logozzo, Doron Peled, and Lenore D. Zuck, editors, *VMCAI*, volume 4905 of *LNCS*, pages 233–247. Springer, 2008.

[PW05] Andreas Podelski and Thomas Wies. Boolean Heaps. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, ∞)-Counter Abstraction. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002.

[Rak06] Jan Rakow. Verification of Dynamic Communication Systems. Master's thesis, Carl von Ossietzky Universität Oldenburg, April 2006.

[Ren03] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[SAH+00] Jørgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jørn Lind-Nielsen, Kim Guldstrand Larsen, Gerd Behrmann, Kåre J. Kristoffersen, Arne Skou, Henrik Leerberg, and Niels Bo Theilgaard. Practical Verification of Embedded Software. *IEEE Computer*, 33(5):68–75, 2000.

[Sas97] Tsutomu Sasao. Ternary decision diagrams: Survey. In *Proc. of ISMVL'97*, pages 241–250, 1997.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi Automata from LTL Formulae. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

[Sch00] Johann M. P. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag, 2000.

[Seg07] Marc Segelken. Abstraction and Counterexample-Guided Construction of *mega* -Automata for Model Checking of Step-Discrete Linear Hybrid Models. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 433–448. Springer, 2007.

[SG07] Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Logic*, 9(1):1, 2007.

[Sis94] A. Prasad Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[STMW04] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *SEFM*, pages 174–183. IEEE Computer Society, 2004.

[Str00] Bjarne Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional, February 2000.

[SWJ08] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 18–32. Springer, 2008.

[Tob07] Tobe Toben. Non-Interference Properties for Data-Type Reduction of Communicating Systems. In Jim Davies and Jeremy Gibbons, editors, *IFM*, volume 4591 of *LNCS*, pages 619–638. Springer, 2007.

[Tob08] Tobe Toben. Counterexample Guided Spotlight Abstraction Refinement. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *FORTE*, volume 5048 of *LNCS*, pages 21–36. Springer, 2008.

[Tse95] Chau-Wen Tseng. Communication Analysis for Shared and Distributed Memory Machines. In *Proc. of the 7^{th} IEEE Symposium on Parallel and Distributed Processing*, San Antonio, USA, October 1995.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Second Series*, 42:230–265, 1936.

[UNI02] UNISIG. Subset 026-ch. 3; vers. 2.2.2 (srs), March 2002. `http://www.aeif.org/`.

[Var93] Pravin Varaiya. Smart cars on smart roads: problems of control. *IEEE Transactions on Automatic Control*, 38(2):195–207, February 1993.

[Var01] Moshe Y. Vardi. Branching vs. Linear Time: Final Showdown. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.

[Var02] Moshe Y. Vardi. Model Checking: A Complexity-Theoretic Perspective (invited talk). *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[Var04] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench - A Tool for the π-Calculus. In David L. Dill, editor, *CAV*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[Wac05] Björn Wachter. Checking Universally Quantified Temporal Properties with three-valued Analysis. Master's thesis, Universität des Saarlandes, March 2005.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, February 1991.

[Wes06] Bernd Westphal. LSC Verification for UML Models with Unbounded Creation and Destruction. *Electr. Notes Theor. Comput. Sci.*, 144(3):133–145, 2006.

[Wes08] Bernd Westphal. *Specification and Verification of Dynamic Topology Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Germany, 2008.

[WKZ+06] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin C. Rinard. On Verifying Complex Properties using Symbolic Shape Analysis. *CoRR*, abs/cs/0609104, 2006.

[WT06] Bernd Westphal and Tobe Toben. The Good, the Bad and the Ugly: Well-Formedness of Live Sequence Charts. In Luciano Baresi and Reiko Heckel, editors, *FASE*, volume 3922 of *LNCS*, pages 230–246. Springer, 2006.

[WW07] Björn Wachter and Bernd Westphal. The Spotlight Principle. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *LNCS*, pages 182–198. Springer, 2007.

[WYKG08] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 382–396. Springer, 2008.

[XLB02] Fei Xie, Vladimir Levin, and James C. Browne. ObjectCheck: A Model Checking Tool for Executable Object-Oriented Software System Designs. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE*, volume 2306 of *LNCS*, pages 331–335. Springer, 2002.

[Yah01] Eran Yahav. Verifying Safety Properties of Concurrent Java Programs using 3-valued Logic. In *POPL*, pages 27–40, 2001.

[YR04] Eran Yahav and G. Ramalingam. Verifying Safety Properties using Separation and Heterogeneous Abstractions. In William Pugh and Craig Chambers, editors, *PLDI*, pages 25–34. ACM, 2004.

[YRS01] Eran Yahav, Thomas Reps, and Mooly Sagiv. LTL Model Checking for Systems with Unbounded Number of Dynamically Created Threads and Objects. Technical Report TR-1424, Computer Sciences Department, Univ. of Wisconsin, Madison, WI, March 2001.

[YRSW06] Eran Yahav, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. *Logic Journal of the IGPL*, 14(5):755–783, 2006.

[ZG03] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *LNCS*, pages 351–357. Springer, 2003.

[ZP04] Lenore D. Zuck and Amir Pnueli. Model Checking and Abstraction to the Aid of Parameterized Systems. *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.

Appendix A

Proofs

Proof of Lemma 3.14 (page 30)

- The claim is trivially satisfied for the formula '$\mathbf{tt}$' as $S[\mathbf{tt}](\mathcal{V}) = 1$ for any structure $S \in \mathit{Strucs}_{\mathcal{S}}(\mathit{Id}^{\perp})$ and valuation $\mathcal{V} \in \mathit{Vals}_{\mathit{Id}^{\perp}}(\mathcal{X})$ by Def. 3.12.
- $S_2[\circledcirc x](e^{I_1}_{I_2}(\mathcal{V})) = 1$

  $\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
  $e^{I_1}_{I_2}(\mathcal{V}(x)) \in U_2 \setminus \{\perp\}$

  $\Rightarrow$ [Def. 3.10 (Structure Embedding)]
  $e^{I_1}_{I_2}(\mathcal{V}(x)) = \mathcal{V}(x) \wedge \mathcal{V}(x) \in U_1$

  $\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
  $S_1[\circledcirc x](\mathcal{V}) = 1$
- $S_2[\circledcirc x](\mathcal{V}) = 0$

  Analogously to the previous case.
- $S_2[p_s(x)](e^{I_1}_{I_2}(\mathcal{V})) = b$ for $b \in \mathbb{B}$

  $\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
  $\iota_2(p_s)(e^{I_1}_{I_2}(\mathcal{V}(x))) = b$

  $\Rightarrow$ [Def. 3.10 (Structure Embedding)]
  $\iota_1(p_s)(\mathcal{V}(x)) = b$

  $\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
  $S_1[p_s(x)](\mathcal{V}) = b$
- The cases for $p_l(x_1, x_2)$ and $p_l(x_1)$ follow analogously to the previous case.
- $S_2[x_1 = x_2](e^{I_1}_{I_2}(\mathcal{V})) = 1$

  $\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
  $e^{I_1}_{I_2}(\mathcal{V}(x_1)) = e^{I_1}_{I_2}(\mathcal{V}(x_2)) = u$ with $u \in I_2 \setminus \{\perp\}$

  $\Rightarrow$ [Def. 3.8 (Embedding Function)]
  $\mathcal{V}(x_1) = \mathcal{V}(x_2) = u$ and $u \in I_1$

$\Rightarrow$ [Def. 3.12 (Formula Evaluation)]
$S_1[x_1 = x_2](\mathcal{V}) = 1$

- $S_2[x_1 = x_2](\mathcal{V}) = 0$

  Analogously to the previous case.
- The proofs for $\neg\psi$ and $\psi_1 \wedge \psi_2$ follow directly by the three-valued semantics of the corresponding operator for negation and conjunction (cf. Rem. 2.4).

■

Proof of Lemma 3.17 (page 32)

Induction Base.

As $\mathsf{T}_1 \preceq \mathsf{T}_2$, we have that for each initial state $S_0^1 \in \mathbf{S}_0^1$ there exists an initial state $S_0^2 \in \mathbf{S}_0^2$ with $S_0^1 \sim S_0^2$, hence $S_0^1 \Subset S_0^2$. For the initial label, we set $L_0^2 = e_{I_2}^{I_1}(L_0^1)$ which entails $L_0^1 \Subset L_0^2$ by Def. 3.9.

Induction Step.

Let $L_i^1 \in \mathbf{L}^1$, $L_i^2 \in \mathbf{L}^2$, $S_i^1 \in \mathbf{S}^2$, $S_i^2 \in \mathbf{S}^2$ with $L_i^1 \Subset L_i^2$ and $S_i^1 \Subset S_i^2$ for some $i \in \mathbb{N}$. As $\mathsf{T}_1 \preceq \mathsf{T}_2$, we have that for $L_{i+1}^1 \in \mathbf{L}^1$ and $S_{i+1}^1 \in \mathbf{S}^1$ there exist $L_{i+1}^2 \in \mathbf{L}^2$ and $S_{i+1}^2 \in \mathbf{S}^2$ with $L_{i+1}^1 \sim L_{i+1}^2$ and $S_{i+1}^1 \sim S_{i+1}^2$, hence $L_{i+1}^1 \Subset L_{i+1}^2$ and $S_{i+1}^1 \Subset S_{i+1}^2$.

■

Proof of Remark 3.22 (page 38)

$S \xrightarrow[r,\,I]{g} S'$

$\iff$ [Def. 3.21 (Evolution)]

$\exists\, \mathcal{V} \in \mathit{Vals}_I(\mathit{vars}(\mathit{name}))$ such that the conditions of Def. 3.21 are satisfied

$\iff$ $[\mathit{vars}(\mathit{name}) = \mathit{vars}(\mathit{guard}) \cup \mathit{vars}(\mathit{stm})]$

$\exists\, \mathcal{V} \in \mathit{Vals}_{A(g)}(\mathit{vars}(\mathit{name}))$ such that

1. $S\rangle\!\rangle_{A(g)}[\![\mathit{guard}]\!](\mathcal{V}) = 1$
2. $\mathit{name}[\mathcal{V}] = g$
3. $S'\rangle\!\rangle_{A(g)} = S\rangle\!\rangle_{A(g)}\langle\mathit{stm}\rangle(\mathcal{V})$

$\iff$ [Def. 3.21 (Evolution)]

$S\rangle\!\rangle_{A(g)} \xrightarrow[r,\,I]{g} S'\rangle\!\rangle_{A(g)}$

■

Proof of Remark 4.4 (page 60)

Let $\mathsf{O} = \mathsf{F}$.

Case 1:

$\pi[\mathsf{F}\,\phi]^i(\mathcal{V}) = 0$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\forall\, k \geq i : \pi[\phi]^i(\mathcal{V}) = 0$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\forall\, k \geq i : \pi[\neg\phi]^i(\mathcal{V}) = 1$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\pi[\mathsf{G}\,\neg\phi]^i(\mathcal{V}) = 1$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\pi[\neg\,\mathsf{G}\,\neg\phi]^i(\mathcal{V}) = 0$

Case 2:

$\pi[\mathsf{F}\,\phi]^i(\mathcal{V}) = 1$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\exists\, k \geq i : \pi[\phi]^i(\mathcal{V}) = 1$
$=$ [logical transformation]
$\neg(\forall\, k \geq i : \pi[\neg\phi]^i(\mathcal{V}) = 1)$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\neg(\pi[\mathsf{G}\,\neg\phi]^i(\mathcal{V}) = 1)$
$\Rightarrow$ [logical transformation]
$\pi[\mathsf{G}\,\neg\phi]^i(\mathcal{V}) \leq 1/2$
$\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
$\pi[\neg\,\mathsf{G}\,\neg\phi]^i(\mathcal{V}) \geq 1/2$

The cases for $\mathsf{O} = \mathsf{G}$ and $\mathsf{O} = \mathsf{X}$ follow analogously. ■

Proof of Lemma 4.6 (page 60)

We show the contraposition, that is, $\mathsf{T}_1[\phi](\mathcal{V}) \leq 1/2 \implies \mathsf{T}_2[\phi](e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$.

By the premise there is a run $\pi_1 = ((L_i^1, S_i^1))_{i\in\mathbb{N}_0} \in \mathit{Runs}(\mathsf{T}_1)$ with $\pi_1[\phi]^0(\mathcal{V}) \leq 1/2$.

As $\mathsf{T}_1 \preceq \mathsf{T}_2$, we can apply Lemma 3.17 to obtain a corresponding run $\pi_2 = ((L_i^2, S_i^2))_{i\in\mathbb{N}_0} \in \mathit{Runs}(\mathsf{T}_2)$ with $L_i^1 \Subset L_i^2$ and $S_i^1 \Subset S_i^2$ for all $i \in \mathbb{N}_0$.

We show by induction over the structure of ϕ that this implies $\pi_2[\phi]^0(e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$, which entails $\mathsf{T}_2[\phi](e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$ by Def. 4.5.

- The cases for $\phi \in \mathit{Forms}(\mathcal{P}_{SL})$ follow directly by Lemma 3.14.

- For an evolution atom $a_e = p_e(x_1, \ldots, x_{k_{p_e}}) \in \mathit{Atoms}_{\mathcal{S}}(\mathcal{P}_E)$, we have that $\pi_1[a_e]^i(\mathcal{V}) \leq 1/2$ implies that $L_i^1 \neq a_e[\mathcal{V}]$ by Def. 4.2 (Specification Satisfaction). With $L_i^1 \Subset L_i^2$ we have $L_i^2 \neq a_e[e_{I_2}^{I_1}(\mathcal{V})]$ by Def. 3.9 (Ground Atom Embedding), hence $\pi_2[a_e]^i(e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$.
- $\pi_1[\mathsf{X}\,\phi_2]^i(\mathcal{V}) \leq 1/2$
  $\Rightarrow$ [Def. 4.2 (Specification Evaluation)]
  $\pi_1[\phi]^{i+1}(\mathcal{V}) \leq 1/2 \vee \bot \in A(L_{i+1}^1)$
  $\Rightarrow$ [$L_i^1 \Subset L_i^2$ for all $i \in \mathbb{N}_0$]
  $\pi_2[\phi]^{i+1}(e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2 \vee \bot \in A(L_{i+1}^2)$
  $\Rightarrow$ [Def. 4.2 (Specification Satisfaction)]
  $\pi_2[\mathsf{X}\,\phi_2]^i(e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$
- $\pi_1[\phi_1\,\mathsf{U}\,\phi_2]^i(\mathcal{V}) \leq 1/2$
  $\Rightarrow$ [Def. 4.2 (Specification Satisfaction)]

$$\begin{aligned}
&\forall\, k \geq i : \big(\pi_1[\phi_2]^k(\mathcal{V}) \leq 1/2 \vee \exists\, j \in \{i \ldots k\} : \pi_1[\phi_1]^j(\mathcal{V}) \leq 1/2\big) \\
\vee\ &\exists\, k \geq i : \big(\pi_1[\phi_1]^k(\mathcal{V}) = 0 \wedge \forall\, j \in \{i \ldots k\} : \pi_1[\phi_2]^j(\mathcal{V}) = 0 \wedge \bot \notin A(L_j)\big) \\
\vee\ &\forall\, k \geq i : \pi_1[\phi_2]^k(\mathcal{V}) = 0 \wedge \bot \notin A(L_k) \qquad (\star)
\end{aligned}$$

  As $L_i^1 \Subset L_i^2$ have $\bot \notin A(L_i^1) \implies \bot \notin A(L_i^2)$ for all $i \in \mathbb{N}_0$, hence $(\star)$ implies

$$\begin{aligned}
&\forall\, k \geq i : \big(\pi_2[\phi_2]^k(\mathcal{V}) \leq 1/2 \vee \exists\, j \in \{i \ldots k\} : \pi_2[\phi_1]^j(\mathcal{V}) \leq 1/2\big) \\
\vee\ &\exists\, k \geq i : \big(\pi_2[\phi_1]^k(\mathcal{V}) = 0 \wedge \forall\, j \in \{i \ldots k\} : \pi_2[\phi_2]^j(\mathcal{V}) = 0 \wedge \bot \notin A(L_j)\big) \\
\vee\ &\forall\, k \geq i : \pi_2[\phi_2]^k(\mathcal{V}) = 0 \wedge \bot \notin A(L_k)
\end{aligned}$$

  which entails $\pi_2[\phi_1\,\mathsf{U}\,\phi_2]^i(e_{I_2}^{I_1}(\mathcal{V})) \leq 1/2$ by Def. 4.2 (Specification Satisfaction).
- $\mathsf{F}\,\phi$ and $\mathsf{G}\,\phi$ are special cases of $\phi_1\,\mathsf{U}\,\phi_2$.

■

Proof of Lemma 5.3 (page 72)

We observe that a Two-Counter Machine M induces a transition system $\mathsf{T}(\mathsf{M}) = (\mathbf{S}^1, \mathbf{S}_0^1, \mathbf{L}^1, \rightarrow)$ where the states $\mathbf{S}^1 := L \times \mathbb{N}_0 \times \mathbb{N}_0$ are configurations of M, the initial state is $\mathbf{S}_0^1 := \{(l_0, 0, 0)\}$, and the transition relation is the M-successor relation, i.e. $\rightarrow := \rightarrowtail_{\mathsf{M}}$, whereby each transition is labelled by the underlying operation from $L^1 := \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero}\}$.

Let $\mathsf{D}(\mathsf{M})$ be the Dynamic Evolution System over signature $(\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ that encodes M according the Def. 5.2 (2CM Encoding), and $[\![\mathsf{D}(\mathsf{M}), S_0]\!] =$

$(\mathbf{S}^2, \mathbf{S}_0^2, \mathbf{L}^2, \rightsquigarrow)$ its semantics according to Def. 3.25 (.) We define the *encoded configuration* of a logical structure as a function $cfg : \mathbf{S}^2 \to L \times \mathbb{N}_0 \times \mathbb{N}_0$ with

$$cfg(S) = (l, K_0, K_1) \ :\Longleftrightarrow \ \exists\, u \in U : \iota(l)(u) = 1 \wedge K_0 = |S_{c_0}(u)| \wedge K_1 = |S_{c_1}(u)|$$

for some $S = (U, \iota) \in \mathbf{S}$ where $S_c(u) := \{u' \in U \mid \iota(c)((u, u')) = 1\}$ denotes the set of processes to which process $u \in U$ is connected to via $c \in \mathcal{P}_L$.

We define the relation $\sim \subseteq (\mathbf{S}^1 \cup \mathbf{L}^1) \times (\mathbf{S}^2 \cup \mathbf{L}^2)$ as

$$S_1 \sim S_2 :\Longleftrightarrow \ S_1 = cfg(S_2)$$
$$L_1 \sim L_2 :\Longleftrightarrow \ L_1 = p_e(L_2)$$

where $p_e(L)$ denote the evolution predicate of a transition label $L \in \mathbf{L}^2$.

We show that both $\sim$ and $\sim^{-1}$ are simulation relations according to Def. 3.16 by distinguishing between the three operations of a 2CM. We only discuss the treatment of the first counter as the second counter can be shown analogously.

'inc': Let $S_1 = (l_1, K_1, K_2) \overset{\mathsf{inc}}{\to} (l_2^0, K_1{+}1, K_2)$ by an instruction $(l_1, \mathsf{inc}, 0, l_2^0, l_2^1) \in I$. From the construction in Def. 5.2, there is a corresponding evolution rule

$$\mathsf{inc}(x, x') \bullet l_1(x) \wedge \neg \circledcirc x' \blacktriangleright \circledast x'; c_1(x, x'); l_2^0!(x) \in \mathsf{D}(\mathsf{M}). \tag{A.1}$$

As $S_1 \sim S_2$, that is, $S_1 = cfg(S_2)$, we have that its guard is satisfiable in S_2, i.e. $S_2[l_1(x) \wedge \neg \circledcirc x'](\mathcal{V}) = 1$ for some valuation $\mathcal{V} \in Vals_{Id}(\mathcal{X})$. Then there is an evolution $S_2 \overset{L_2}{\rightsquigarrow} S_2'$ with $p_e(L_2) = \mathsf{inc}$ and $cfg(S_2') = (l_2^0, K_1 + 1, K_2)$ by the actions of rule (A.1), i.e. $S_1' \sim S_2'$.

For $\sim^{-1}$, we assume that $S_2 \overset{L_2}{\rightsquigarrow} S_2'$ with $cfg(S_2) = (l_1, K_1, K_2)$ and $p_e(L_2) = \mathsf{inc}$. This implies the existence of the evolution rule (A.1) in $\mathsf{D}(\mathsf{M})$ such that $cfg(S_2') = (L^0 t_2, K_1 + 1, K_2)$, and, by the construction in Def. 5.2, a corresponding instruction $(l_1, \mathsf{inc}, 0, l_2^0, l_2^1) \in I$. As $S_1 \sim S_2$, that is, $S_1 = (l_1, K_1, K_2)$, there exists a configuration $S_1' = (l_2^0, K_1{+}1, K_2) \in \mathbf{S}^1$ with $S_1 \overset{\mathsf{inc}}{\to} S_1'$, hence $S_2' \sim^{-1} S_1'$.

'dec': The decrement case can be handled analogously to the increment instruction by observing that the evolution rule

$$\mathsf{dec}(x, x') \bullet l_1(x) \wedge c_1(x, x') \blacktriangleright \otimes x'; \neg c_1(x, x'); l_2^0!(x) \in \mathsf{D}(\mathsf{M}) \tag{A.2}$$

corresponding to a decrement instruction $(l_1, \mathsf{dec}, 0, l_2) \in I$ ensures that any positive value of the first counter in the encoded configurations is exactly decreased by one and the successor state l_2^0 is entered.

'zero': Let $S_1 = (l_1, 0, K_2) \overset{\mathsf{zero}}{\to} (l_2^0, 0, K_2)$ by an instruction $(l_1, \mathsf{zero}, 0, l_2^0, l_2^1) \in I$. The corresponding rule according to the construction in Def. 5.2

$$\mathsf{zero}(x) \bullet l_1(x) \wedge \neg c_0(x) \blacktriangleright l_2^0!(x) \in \mathsf{D}(\mathsf{M}) \tag{A.3}$$

is enabled in any structure $S_2 \in \mathbf{S}^2$ where $cfg(S_2) = (l_1, 0, K_2)$ for some $K_2 \in \mathbb{N}_0$ (cf. Def. 3.12). Then there is a transition $S_2 \overset{L_2}{\rightsquigarrow} S_2'$ with $p_e(L_2) = \mathsf{zero}$ and $cfg(S_2') = (l_2^0, 0, K_2)$, hence $S_1' \sim S_2'$. For the opposite direction of simulation, we have that $S_2 \overset{L_2}{\rightsquigarrow} S_2'$ with $p_e(L_2) = \mathsf{zero}$ by rule (A.3) ensures the existence of a zero instruction $(l_1, \mathsf{zero}, 0, l_2^0, l_2^1) \in I$ such that for any configuration $S_1 = (l_1, 0, K_2)$ we have $S_1 \overset{\mathsf{zero}}{\rightarrow} (l_2^0, 0, K_2) =: S_1'$, hence $S_2' \sim^{-1} S_1'$.

Analogously, a negative test for zeroness is simulated by the evolution rule

$$\mathsf{zero}(x) \bullet l_1(x) \wedge c_0(x) \blacktriangleright l_2^1!(x) \in \mathsf{D}(\mathsf{M}) \tag{A.4}$$

By $S_0 := (\{u\}, \{l_0(u)\}, \emptyset)$, we have that $cfg(S_0^2) = (l_0, 0, 0)$ for any $S_0^2 \in \mathbf{S}_0^2$, hence $S_0^1 \sim S_0^2$ and $S_0^2 \sim^{-1} S_0^1$ for all initial states $S_0^1 \in \mathbf{S}_0^1$ and $S_0^2 \in \mathbf{S}_0^2$.

If $\mathsf{D}(\mathsf{M}), S_0 \not\models \mathsf{G}\,\neg l(x)$, we have that $S_0^2 \dashrightarrow S$ with $cfg(S) = (l, K_1, K_2)$ for some values $K_1, K_2 \in \mathbb{N}_0$ and $S_0^2 \in \mathbf{S}_0^2$. Using the simulation relation $\sim^{-1}$ from above, we have that by by Lemma 3.17 there exists a corresponding run in $\mathsf{T}(\mathsf{M})$, that is, $\mathsf{M} \rightarrowtail l$. Analogously, $\mathsf{M} \rightarrowtail l$ entails that $\mathsf{D}(\mathsf{M}), S_0 \not\models \mathsf{G}\,\neg l(x)$ by $\sim$.

■

Encoding of a Two-Counter-Machine as DCS (page 72)

Let $\mathsf{M} = (L, l_0, I)$ be a 2CM. We construct the DCS $\mathsf{C}(\mathsf{M}) = (M, M_X, P)$ with messages $M = \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero}, \mathsf{last}, \mathsf{done}\}$ and $M_X = \emptyset$.

A configuration (l, K_0, K_1) of the 2CM is encoded by $K_0 + K_1 + 3$ processes where one process is in state l and each counter K_i is represented by a linked list of $K_i + 1$ processes. For convenience, we employ two DCS protocols $\{\mathsf{m}, \mathsf{c}\} = \mathsf{dom}(P)$, where the first protocol will initiate the commands and mimic the transition of the 2CM while instances of the second protocol will serve as elements of the counter lists. The behaviour of the latter protocol is thus responsible to extend and shrink the interlinked list of processes.

We set $P(\mathsf{m}) = (Q_\mathsf{m}, A_\mathsf{m}, F_\mathsf{m}, C_\mathsf{m}, succ_\mathsf{m})$ with

- states $Q_\mathsf{m} := L \cup I$, $A_\mathsf{m} = \{l_0\}$, $F_\mathsf{m} = \emptyset$,
- channels $C_\mathsf{m} = \{c_0, c_1\}$, and
- transition relation

$$\begin{aligned} succ_\mathsf{m} := {} & \{(l_0, *_\mathsf{c}^{c_0}, l_0), (l_0, *_\mathsf{c}^{c_1}, l_0)\} \cup \\ & \{(l_1, c_i!\mathsf{op}, inst), (inst, ?\mathsf{done}, l_2^0) \mid inst = (l_1, \mathsf{op}, i, l_2^0, l_2^1) \in I\} \cup \\ & \{(inst, ?\mathsf{last}, l_2^1) \mid inst = (l_1, i, \mathsf{zero}, l_2^0, l_2^1) \in I\} \end{aligned}$$

and $P(\mathsf{c}) = (Q_\mathsf{c}, A_\mathsf{c}, F_\mathsf{c}, C_\mathsf{c}, succ_\mathsf{c})$ with

- states $Q_{\mathsf{c}} := \{\mathsf{head}, \mathsf{tail}, \mathsf{tail}_{\mathsf{op}}, \mathsf{head}_{\mathsf{op}}, \mathsf{tail}_{\mathsf{inc}_2} \mid \mathsf{op} \in \{\mathsf{inc}, \mathsf{dec}, \mathsf{zero}\}\}$
- initial and fragile states , $A_{\mathsf{c}} = \{\mathsf{head}\}$ and $F_{\mathsf{c}} = \emptyset$,
- channels $C_{\mathsf{c}} = \{\mathsf{next}, \mathsf{prev}\}$, and
- transition relation

$$\begin{aligned} succ_{\mathsf{c}} := {} & \{(\mathsf{tail}, ?\mathsf{op}(\mathsf{prev}), \mathsf{tail}_{\mathsf{op}}), (\mathsf{tail}_{\mathsf{op}}, \mathsf{next}!\mathsf{op}, \mathsf{tail}) \mid \mathsf{op} \in \{\mathsf{inc}, \mathsf{dec}\}\} \cup \\ & \{(\mathsf{tail}, ?\mathsf{done}(\mathsf{next}), \mathsf{tail}_{\mathsf{done}}), (\mathsf{tail}_{\mathsf{done}}, \mathsf{prev}!\mathsf{done}, \mathsf{tail})\} \cup \\ & \{(\mathsf{head}, ?\mathsf{inc}(\mathsf{prev}), \mathsf{head}_{\mathsf{inc}}), (\mathsf{head}_{\mathsf{inc}}, *^{\mathsf{next}}_{\mathsf{head}}, \mathsf{head}_{\mathsf{inc}_2}), (\mathsf{head}_{\mathsf{inc}_2}, \mathsf{prev}!\mathsf{done}, \mathsf{tail})\} \cup \\ & \{(\mathsf{head}, ?\mathsf{dec}(\mathsf{prev}), \mathsf{head}_{\mathsf{dec}}), (\mathsf{head}_{\mathsf{dec}}, \mathsf{prev}!\mathsf{last}, \mathsf{head})\} \cup \\ & \{(\mathsf{tail}, ?\mathsf{last}(\mathsf{next}), \mathsf{tail}_{\mathsf{last}}), (\mathsf{tail}_{\mathsf{last}}, \mathsf{prev}!\mathsf{done}, \mathsf{head})\} \\ & \{(\mathsf{tail}, ?\mathsf{zero}(\mathsf{prev}), \mathsf{tail}_{\mathsf{zero}}), (\mathsf{tail}_{\mathsf{zero}}, \mathsf{prev}!\mathsf{last}, \mathsf{tail})\} \\ & \{(\mathsf{head}, ?\mathsf{zero}(\mathsf{prev}), \mathsf{head}_{\mathsf{zero}}), (\mathsf{head}_{\mathsf{zero}}, \mathsf{prev}!\mathsf{done}, \mathsf{head})\}. \end{aligned}$$

We define the encoded configuration of a snapshot of the DCS $\mathsf{C}(\mathsf{M})$ as

$$\begin{aligned} cfg((U,\iota)) = (l, K_0, K_1) \ :\Longleftrightarrow \ & \exists\, u \in U : \iota(l)(u) = 1 \wedge \forall\, i \in \{0,1\} : \\ & \exists\, u_{i_0}, \ldots, u_{i_{K_i}} \in U \,\forall\, k \in \{0, \ldots, K_{i-1}\} : \\ & \iota(\mathsf{tail})(u_{i_k}) = 1 \wedge \iota(\mathsf{next})(u_{i_k}, u_{i_{k+1}}) = 1 \wedge \\ & \iota(\mathsf{head})(u_{i_{K_i}}) = 1 \wedge \iota(c_i)(u, u_{i_0}) = 1 \end{aligned}$$

The correctness of the encoding follows analogously to the proof of Lemma 5.3 by observing that the evolutions of an encoded configuration simulates the instructions of M in a corresponding configuration.

■

Proof of Lemma 5.6 (page 75)

Let $\alpha_I(S) = (U^\sharp, \iota^\sharp)$ be the spotlight abstraction of a logical structure $S = (U, \iota) \in \mathit{Strucs}_{\mathcal{S}}(I')$ under $I \subseteq I'$.

By Def. 5.5, we have that $U^\sharp = (U \cap I) \cup \{\bot\}$, hence $u \in U \iff e_I^{I'}(u) \in U^\sharp$ for all $u \in I$, that is, $\alpha_I(S)$ preserves aliveness.

To show that $\iota^\sharp$ over-approximates ι we only have to consider the case where $\iota^\sharp$ yields a definitive value. If $\iota^\sharp(p)(e_I^{I'}(u_1), \ldots, e_I^{I'}(u_{p_k})) \in \mathbb{B}$ we have

$$\{e_I^{I'}(u_1), \ldots, e_I^{I'}(u_{p_k})\} \subseteq I$$

and hence

$$\iota^\sharp(p)(e_I^{I'}(u_1), \ldots, e_I^{I'}(u_{p_k})) = \iota(p)(u_1, \ldots, u_{p_k})$$

by Def. 5.5 (Spotlight Abstraction) for some predicate $p \in \mathcal{P}_{SL}$.

■

Proof of Lemma 5.9 (page 77)

Let $S \xrightarrow[r, I']{g} S'$ by the evolution rule $r = (name, guard, stm) \in EvoRule_{\mathcal{S}}$.

Then by Def. 3.21, there exists a valuation $\mathcal{V} \in Vals_{I'}(vars(name))$ such that

1. $S[\![guard]\!](\mathcal{V}) = 1$,
2. $g = name[\mathcal{V}]$, and
3. $S' = S\langle stm\rangle(\mathcal{V})$.

By Lemma 5.6 we have that $S \Subset \alpha_I(S)$, hence $\alpha_I(S)[\![guard]\!](e_I^{I'}(\mathcal{V})) \geq 1/2$ $(\star)$ by Lemma 3.14, that is, the rule r is enabled in $\alpha_I(S)$.

We set $S_1^\sharp = \alpha_I(S)$, $S_2^\sharp = \alpha_I(S')$, and $S_{stm}^\sharp = S_1^\sharp\langle stm\rangle^\sharp(e_I^{I'}(\mathcal{V}))$, and show that $S_2^\sharp = S_{stm}^\sharp$, that is,

$$\alpha_I(S') = S_1^\sharp\langle stm\rangle^\sharp(e_I^{I'}(\mathcal{V})). \qquad (\star\star)$$

- '$stm = \mathsf{skip}$'. Then $S' = S$ by Def. 3.20 and $S_{stm}^\sharp = S_1^\sharp$ by Def. 5.7, hence $S_2^\sharp = S_{stm}^\sharp$.
- '$stm = stm(x_1, \ldots, x_k)$'.

  Case 1: $\{\mathcal{V}(x_1), \ldots, \mathcal{V}(x_k)\} \subseteq I$.

  Then $S\langle stm\rangle^\sharp(\mathcal{V}) = S\langle stm\rangle(\mathcal{V})$ by Def. 5.7 (Abstract Statement Execution), and $S\Downarrow_{\mathsf{ran}(\mathcal{V})} = S_1^\sharp\Downarrow_{\mathsf{ran}(\mathcal{V})}$ by Def. 5.5 (Spotlight Abstraction). This entails $S'\Downarrow_{\mathsf{ran}(\mathcal{V})} = S_{stm}^\sharp\Downarrow_{\mathsf{ran}(\mathcal{V})}$, thus $S_2^\sharp = S_{stm}^\sharp$.

  Case 2: $\{\mathcal{V}(x_1), \ldots, \mathcal{V}(x_k)\} \supset I$.

  Then $S\langle stm\rangle^\sharp(\mathcal{V}) = S$ by Def. 5.7 (Abstract Statement Execution), and $S_1^\sharp = S_2^\sharp$ by Def. 5.5 (Spotlight Abstraction). This entails $S_2^\sharp = S_{stm}^\sharp$.

As $name[e_I^{I'}(\mathcal{V})] = \alpha_I(g)$ by Def. 5.5, we have with $(\star)$ and $(\star\star)$ that

$$\alpha_I(S) \xrightarrow[r, I]{\alpha_I(g)}{}^\sharp \alpha_I(S')$$

by Definition 5.8 (Abstract Evolution).

■

Proof of Theorem 5.12 (page 80)

Note that $I = \mathsf{ran}(\mathcal{V})$ such that both the abstract and the concrete evaluation of the specification employs the same set of fairness constraints according to Definitions 4.9 and 5.11.

Case 1: $[\![\mathsf{D}, S]\!]^\sharp_I[\phi](\mathcal{V}) = 1$

Let $[\![\mathsf{D}, S]\!]_{I'} := (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \to)$ and $[\![\mathsf{D}, S]\!]^\sharp_I = (\mathbf{S}^\sharp, \mathbf{S}^\sharp_0, \mathbf{L}^\sharp, \rightsquigarrow)$.

We show that $[\![\mathsf{D}, S]\!]_{I'} \preceq [\![\mathsf{D}, S]\!]^\sharp_I$ for the spotlight $I \subseteq I'$ by the relation $\sim \subseteq (\mathbf{S} \times \mathbf{S}^\sharp) \cup (\mathbf{L} \times \mathbf{L}^\sharp)$ defined as

$$R \sim R^\sharp :\Longleftrightarrow\ R^\sharp = \alpha_I(R).$$

This setting entails that $L \sim L^\sharp$ implies $L \Subset L^\sharp$ for any label $L \in \mathbf{L}$ by Def. 3.9, and $S \sim S^\sharp$ implies $S \Subset S^\sharp$ for any state $S \in \mathbf{S}$ by Lem. 5.6.

Now consider two states $S, S' \in \mathbf{S}$ and a label $L' \in \mathbf{L}$ with $S \xrightarrow{L} S'$, that is,

$$S \xrightarrow[\mathsf{D},\, I']{L} S'.$$

By Lemma 5.9 this entails

$$\alpha_I(S) \xrightarrow[r,\, I]{\alpha_I(L)}{}^\sharp\ \alpha_I(S')$$

thus $\alpha_I(S) \overset{\alpha_I(g)}{\rightsquigarrow} \alpha_I(S')$ by Def. 5.10 (Abstract DES Semantics). As $S' \sim\ \alpha_I(S')$ and $L \sim \alpha_I(L)$, we have that $\sim$ is a simulation relation according to Def. 3.16.

For all initial states $S_0 \in \mathbf{S}_0$ there exist a state $S^\sharp = \alpha_I(S_0) \in \mathbf{S}^\sharp_0$ by Def. 5.10 (Abstract DES Semantics).

This entails $[\![\mathsf{D}, S]\!]_{I'} \preceq [\![\mathsf{D}, S]\!]^\sharp_I$, and Lemma 4.6 ensures $[\![\mathsf{D}, S]\!]_{I'}[\phi](\mathcal{V}) = 1$.

Case 2: $[\![\mathsf{D}, S]\!]^\sharp_I[\phi](\mathcal{V}^\sharp) = 0$

By the definition of the three-valued satisfaction relation (cf. Def. 4.5), there exists a run $\pi^\sharp = ((L_i, S_i))_{i \in \mathbb{N}_0} \in \mathit{FairRuns}_I([\![\mathsf{D}, S]\!]^\sharp_I)$ with

$$\pi^\sharp[\phi]^0(\mathcal{V}) = 0$$

and $\bot \notin A(L_i)$ for all $i \in \mathbb{N}_0$.

Following Remark 3.22 (Locality) we have

$$\pi^\sharp \in \mathit{Runs}([\![\mathsf{D}, S]\!]_J)$$

where $J := \bigcup_{i \in \mathbb{N}_0} A(L_i)$, i.e. the run is contained in the J-underapproximated semantics of D where J comprises all identities of the involved evolutions.

As $I \subseteq I'$ we have $J \subseteq I'$, thus by Lemma 3.26 (Under-approximation) there exists a run $\pi \in \mathit{Runs}([\![\mathsf{D}, S]\!]_{I'})$ with $\pi|_J = \pi^\sharp$, hence $\pi[\phi]^0(\mathcal{V}) = 0$. This entails $[\![\mathsf{D}, S]\!]_{I'}[\phi](\mathcal{V}) = 0$ by Def. 4.10.

■

Proof of Lemma 5.15 (page 81)

Let $[\![\mathsf{D}, S]\!]_{I'} = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow) \in \mathcal{T}_{\mathcal{S}}(I')$.

Requirement (1) of Def. 5.14 follows directly from the construction of the concrete semantics of DES as given in Def. 3.25.

Let $(S, g, S') \in \rightarrow$. Then by Def. 3.25 we have

$$S \xrightarrow[r,\, I']{g} S'$$

by some evolution rule $r = (\mathit{name}, \mathit{guard}, \mathit{stm}) \in \mathsf{D}$. From the definition of permutations in Def. 3.24 we obtain

1. $\sigma(S)[\![\mathit{guard}]\!](\sigma(\mathcal{V})) = 1$,
2. $\sigma(g) = \mathit{name}[\sigma(\mathcal{V})]$, and
3. $\sigma(S') = \sigma(S)\langle\!\langle \mathit{stm} \rangle\!\rangle(\sigma(\mathcal{V}))$.

This implies $\sigma(S) \xrightarrow[r,\, I']{\sigma(g)} \sigma(S')$ by Def. 3.21, thus $(\sigma(S), \sigma(g), \sigma(S')) \in \rightarrow$.

■

Proof of Lemma 6.3 (page 98)

$\mathsf{D}_S[\![\phi]\!]$

= [Def. 4.10 (Concrete Semantics)]
$\min\{[\![\mathsf{D}, S]\!][\phi](\mathcal{V}) \in \mathbb{B} \mid \mathcal{V} \in \mathit{Vals}_{\mathit{Id}}(\mathit{vars}(\phi))\}$

= [Lem. 5.17 (Query Reduction)]
$\min\{[\![\mathsf{D}, S]\!][\phi](\mathcal{V}) \in \mathbb{B} \mid \mathcal{V} \in \mathit{ValBasis}(\mathit{vars}(\phi))\} \quad (\star)$

Let $X' := \mathit{vars}(\phi) \setminus X$ be the disjoint partitioning induced by the given set of variables X, and let $I := \mathsf{ran}(\mathcal{V}|_{X'})$ for some valuation $\mathcal{V} \in \mathit{ValBasis}(\mathit{vars}(\phi))$. We apply Thm. 5.12 to obtain

$$[\![\mathsf{D}, S]\!][\phi](\mathcal{V}) \;\sqsubseteq\; [\![\mathsf{D}, S]\!]^\sharp_I[\phi](\mathcal{V}^\sharp)$$

where $\mathcal{V}^\sharp := \mathcal{V}|_{X'} \cdot \mathcal{V}_X$ with $\mathcal{V}_X(x) := \bot$ for each $x \in X$. Thus we obtain

$$(\star) \sqsubseteq \min\{\, [\![\mathsf{D}, S]\!]^\sharp_{\mathsf{ran}(\mathcal{V})}[\phi](\mathcal{V}^\sharp) \in \mathbb{B}_3 \mid \mathcal{V} \in \mathit{ValBasis}(X')\}$$

which entails $(\star) \sqsubseteq \mathsf{D}^\sharp_{S,X}[\![\phi]\!]$ by Def. 5.18.

■

Proof of Theorem 6.6 (page 99)

Case 1: $\mathsf{D}^\sharp_{S,X}[\![(\theta{=}1) \to \phi]\!] = 0$
$\Rightarrow$ [Def. 6.2 (Abstract Satisfaction), Def. 4.2 (Specification Satisfaction)]
$\exists\, \mathcal{V}' \in \mathcal{V}{\cdot}\mathcal{V}_X\ \exists\, \pi^\sharp \in \mathit{Runs}([\![\mathsf{D}, S]\!]^\sharp_{\mathsf{ran}(\mathcal{V})}) : \pi^\sharp[\theta]^0(\mathcal{V}') = 1 \wedge \pi^\sharp[\phi]^0(\mathcal{V}) = 0$
$\Rightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\mathsf{D}^\sharp_S[\![\phi]\!] = 0$
$\Rightarrow$ [Lemma 5.19 (Embedding)]
$\mathsf{D}^I_S[\![\phi]\!] = 0$

Case 2: $\mathsf{D}^\sharp_{S,X}[\![(\theta{=}1) \to \phi]\!] = 1$
$\Rightarrow$ [Def. 6.2 (Abstract Satisfaction)]
$\forall\, \mathcal{V}' \in \mathcal{V}{\cdot}\mathcal{V}_X\ \forall\, \pi^\sharp \in \mathit{Runs}([\![\mathsf{D}, S]\!]^\sharp_{\mathsf{ran}(\mathcal{V})}) : \pi^\sharp[\theta{=}1]^0(\mathcal{V}') = 0 \vee \pi^\sharp[\phi]^0(\mathcal{V}) = 1$
$\Rightarrow$ [logical transformation]
$\forall\, \mathcal{V}' \in \mathcal{V}{\cdot}\mathcal{V}_X\ \forall\, \pi^\sharp \in \mathit{Runs}([\![\mathsf{D}, S]\!]^\sharp_{\mathsf{ran}(\mathcal{V})}) : \pi^\sharp[\theta]^0(\mathcal{V}') \leq 1/2 \vee \pi^\sharp[\phi]^0(\mathcal{V}) = 1$
$\Rightarrow$ [$X \subseteq \mathit{evovars}(\theta)$, Rem. 4.7 (Definiteness)]
$\forall\, \mathcal{V}' \in \mathcal{V}{\cdot}\mathcal{V}_X\ \forall\, \pi^\sharp \in \mathit{Runs}([\![\mathsf{D}, S]\!]^\sharp_{\mathsf{ran}(\mathcal{V})}) : \pi^\sharp[\theta]^0(\mathcal{V}') = 0 \vee \pi^\sharp[\phi]^0(\mathcal{V}) = 1$
$\Rightarrow$ [Thm. 5.12 (Spotlight Embedding)]
$\forall\, \mathcal{V} \in \mathit{Vals}_I(\mathit{vars}(\phi))\ \forall\, \pi \in \mathit{Runs}([\![\mathsf{D}, S]\!]_I) : \pi[\theta]^0(\mathcal{V}) = 0 \vee \pi[\phi]^0(\mathcal{V}) = 1$
$\Rightarrow$ [$\mathsf{D}^I_S[\![\theta \vee \phi]\!] = 1$ (as θ is evolution constraint for D and ϕ))]
$\forall\, \mathcal{V} \in \mathit{Vals}_I(\mathit{vars}(\phi))\ \forall\, \pi \in \mathit{Runs}([\![\mathsf{D}, S]\!]_I) : \pi[\phi]^0(\mathcal{V}) = 1$
$\Rightarrow$ [Def. 4.10]
$\mathsf{D}^I_S[\![\phi]\!] = 1$ ■

Proof of Lemma 6.12 (page 107)

$S_2\rangle_F = \alpha_{I_2}(S_1\rangle_F)$
$\iff$ [Def. 5.5 (Spotlight Abstraction), Def. 3.7 (Focus)]
There is a valuation $\mathcal{V} \in \mathit{Vals}_F(\mathcal{X})$ such that for each atom $a = p(x_1, \ldots, x_{k_p}) \in \mathit{Atoms}_\mathcal{X}(\mathcal{P}_{SL} \cup \{\circledcirc\})$, we have

$$S_2\rangle_F[\hat{a}](\mathcal{V}) = 1 \iff \alpha_{I_2}(S_1\rangle_F)[\hat{a}](\mathcal{V}) = 1$$

where $\hat{a} \in \{a, \neg a\}$.
$\iff$ [Def. 6.11 (Structure Formula)]
$\alpha_{I_2}(S_1\rangle_F)[\mathit{expr}(S_2\rangle_F, \mathcal{V})](\mathcal{V})$
$\iff$ [Def. 5.5 (Spotlight Abstraction) and Def. 3.7 (Focus) with $F = \mathsf{ran}(\mathcal{V}) \subseteq I_2$]
$S_1[\mathit{expr}(S_2\rangle_F, \mathcal{V})](\mathcal{V})$

■

Proof of Theorem 6.14 (page 108)

We show the contraposition.
Let $\delta = (((L_i, S_i))_{0\leq i\leq n}, \mathcal{V}) \in \mathit{Cex}(\mathsf{D}_S, \phi)$ be an abstract counterexample.

$\mathsf{D}_S[\![\neg(\varphi(\delta) \wedge \neg\phi)]\!] = 0$

$\iff$ [Def. 4.2 (Specification Satisfaction)]

There exists valuations $\mathcal{V} \in \mathit{Vals}_{Id}(\mathit{vars}(\phi))$ and $\mathcal{V}' \in \mathit{Vals}_{Id}(\mathit{vars}(\varphi(\delta)))$ with $\mathcal{V}'|_{\mathsf{dom}(\mathcal{V})} = \mathcal{V}$ and a run $\pi' = ((L'_i, S'_i))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!])$ with

$$\pi'[\varphi(\delta)]^0(\mathcal{V}') = 1 \ (\star) \ \mathit{and} \ \pi'[\phi]^0(\mathcal{V}) = 0 \ (\star\star).$$

$\iff$ [($\star$), Def. 6.13 (Counterexample Spec.), Def. 4.2 (Specification Satisfaction)]

There exists a monotone function $f : \bot(\delta) \to \mathbb{N}_0$ such that

$$\pi'[\mathit{evo}(L_i, \mathcal{V}, i) \wedge \mathit{expr}(S_i\rangle\!\rangle_{A(L_i)}, \mathcal{V})]^{f(i)}(\mathcal{V}') = 1$$

for all $i \in \bot(\delta)$.

$\iff$ [Lem. 6.12 (Spotlight Formula) with $F := A(L_i) \subseteq \mathsf{ran}(\mathcal{V})$]

For all $i \in \bot(\delta)$, $L_i = \alpha_{\mathsf{ran}(\mathcal{V})}(L'_{f(i)})$ and $S_i\rangle\!\rangle_{A(L_i)} = \alpha_{\mathsf{ran}(\mathcal{V})}(S'_{f(i)}\rangle\!\rangle_{A(L_i)})$.

$\iff$ [($\star\star$), Def. 6.8 (Counterexample Concretisation)]

$\exists\, \pi' \in \mathit{Runs}([\![\mathsf{D}, S]\!]) : \pi' \in \gamma(\delta)$

$\iff$ [Def. 6.9 (Spurious Counterexample)]

$\neg F(\delta)$ ■

Proof of Lemma 6.19 (page 114)

Let $\mathcal{S} = (\mathcal{X}, \mathcal{P}_S, \mathcal{P}_L, \mathcal{P}_E)$ be a signature and $\delta_i = (\pi, \mathcal{V}) \in \mathit{Cex}(\mathsf{D}_S, \theta_i \to \phi)$ with

$$\theta_i := (\neg\varphi(\delta_1) = 1) \wedge \ldots \wedge (\neg\varphi(\delta_{i-1}) = 1)$$

be the abstract counterexample obtained in the i-th iteration of algorithm 1b.

$\pi[\theta_i \to \phi]^0(\mathcal{V}) = 1/2$
$\implies$ [semantics of $\to$, $\pi[\theta = 1]^i(\mathcal{V}) \in \mathbb{B}$]
$\pi[\theta_i]^0(\mathcal{V}) = 1$ and $\pi[\phi]^0(\mathcal{V}) = 1/2$
$\implies$ [logical transformation]
$\pi[\neg\varphi(\delta_j) = 1]^0(\mathcal{V}) = 1$ for all $1 \leq j < i$.

By Def. 6.13 (Counterexample Specification) we have $\pi[\varphi(\delta_i)=1]^0(\mathcal{V}) = 1$, hence

$$\pi[\varphi(\delta_i)=1 \wedge \bigwedge_{1\leq j<i} \neg\varphi(\delta_j)=1]^0(\mathcal{V}) = 1 \qquad (\star)$$

Now let $nF\text{-}Specs_{\mathcal{S}}$ denote the set of nested-finally specifications over $\mathcal{S}$, generated by the grammar

$$\eta ::= \mathsf{F}(\psi \wedge \eta_1) \mid \mathrm{tt}$$

where $\psi \in Forms(\mathcal{P})$ is a formula.

For $\eta_1, \eta_2 \in nF\text{-}Specs_{\mathcal{S}}$, we write $\eta_1 \Subset \eta_2$ to denote that η_1 is a subformula of η_2. By Def. 4.2 have have that the satisfaction of η_2 implies the satisfaction of the subformula η_1, that is

$$\mathsf{D}^{\sharp}_{X,S}[\![(\eta_2=1) \rightarrow (\eta_1=1)]\!] = 1 \qquad (\star\star)$$

for variables $X \subseteq \mathcal{X}$.

Now assume that algorithm 1b diverges. As $\mathcal{P}_{SL}$ is a finite set of predicates, we eventually reach iteration k where $\varphi(\delta_j) \Subset \varphi(\delta_k)$ for some $j < k$, that is, we generate a counterexample specification for which a subformula has already been generated before. With $(\star)$ this entails

$$\pi[(\varphi(\delta_k)=1) \wedge \bigwedge_{1\leq i<k} (\neg\varphi(\delta_i)=1)]^0(\mathcal{V}) = 1$$

which is a contradiction to $(\star\star)$. Thus algorithm 1b terminates. ■

Proof of Lemma 6.21 (page 117)

As S' is reachable from S by L', there exists a run $\pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in Runs([\![\mathsf{D}, S]\!])$ with $S = S_i$ and $(L', S') = (L_j, S_j)$ for some $i, j \in \mathbb{N}_0$ with $i \leq j$. Let $K := \{i, k_1, \ldots, k_n, j\} \subset \mathbb{N}_0$ be the maximal set of values with

- $i < k_1 < \ldots < k_n < j$, and
- $(L_k, S_k)[\phi]^0(\mathcal{V}_k) = 0$ for some valuation $\mathcal{V}_k \in Vals_{Id}(vars(\phi))$ for all $k \in K$, and
- $(L_l, S_l)[\phi]^0(\mathcal{V}) = 1$ for all valuations $\mathcal{V} \in Vals_{Id}(vars(\phi))$ and $l \in \{i, \ldots, j\} \setminus K$.

Note that $|K| \geq 1$ as (L_j, S_j) is a violation of ϕ from S. Then $(L_{\min K}, S_{\min K})$ is a minimal violation of ϕ from S by Def. 6.20.

■

Proof of Lemma 6.23 (page 118)

Let $[\![\mathsf{D}, S]\!] = (\mathbf{S}, \mathbf{S}_0, \mathbf{L}, \rightarrow)$ and $\phi' := (\bigwedge_{\sigma\in\Sigma} \sigma(\neg\phi)\,)\,\mathsf{U}\,\phi$.

"$\Rightarrow$" (via contraposition):

$\mathsf{D} \not\models \neg(\phi')$
$\Longrightarrow$ [Def. 4.10 (DES Satisfaction)]
$\exists\, \mathcal{V} \in \mathit{Vals}_{\mathit{Id}}(\mathcal{X}), \pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!]) : \pi[\neg\phi']^0(\mathcal{V}) = 0$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\exists\, i \in \mathbb{N}_0 : \pi[\neg\phi]^i(\mathcal{V}) = 0$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\exists\, i \in \mathbb{N}_0 : \pi[\phi]^i(\mathcal{V}) = 1$
$\Longrightarrow$ [Def. 4.10 (DES Satisfaction)]
$\mathsf{D} \not\models \neg\,\mathsf{F}\,\phi$

"$\Leftarrow$" (via contraposition):

$\mathsf{D} \not\models \neg\,\mathsf{F}\,\phi$
$\Longrightarrow$ [Def. 4.10 (DES Satisfaction)]
$\exists\, \mathcal{V} \in \mathit{Vals}_{\mathit{Id}}(\mathcal{X}), \pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!]) : \pi[\mathsf{F}\,\phi]^0(\mathcal{V}) = 1$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\exists\, j \in \mathbb{N}_0 : \pi[\phi]^j(\mathcal{V}) = 1$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\exists\, j \in \mathbb{N}_0 : \pi[\neg\phi]^j(\mathcal{V}) = 0$
$\Longrightarrow$ [Def. 6.20 (Minimal Violation)]
(L_j, S_j) is a violation $\neg\phi$ from $\mathbf{S}_0$
$\Longrightarrow$ [Lem. 6.21 (Minimal Violation)]
$\exists\, (L_k, S_k) \in \mathbf{L} \times \mathbf{S} : \mathbf{S}_0 \overset{\neg\phi}{\dashrightarrow\!\bullet}\ (L_k, S_k)$
$\Longrightarrow$ [Def. 6.21 (Minimal Violation)]
$\exists\, \pi' = ((L_i', S_i'))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!])$ such that

1. $\pi'[\neg\phi]^k(\mathcal{V}') = 0$ for some valuation $\mathcal{V}' \in \mathit{Vals}_{\mathit{Id}}(\mathcal{X})$, and
2. $\pi'[\neg\phi]^l(\mathcal{V}'') = 1$ for all valuations $\mathcal{V}'' \in \mathit{Vals}_{\mathit{Id}}(\mathcal{X})$ and all $0 \leq l < k$.

$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\pi'[\phi']^0(\mathcal{V}') = 1$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$\pi'[\neg\phi']^0(\mathcal{V}') = 0$
$\Longrightarrow$ [Def. 4.10 (DES Satisfaction)]
$\mathsf{D} \not\models \neg\phi'$

■

Proof of Lemma 6.25 (page 119)

Let $\delta = (((L_i^\delta, S_i^\delta))_{i\in\mathbb{N}_0}, \mathcal{V}_\delta) \in \mathit{Cex}(\mathsf{C}_S, \phi)$.

"$\Rightarrow$":

$\mathit{F}(\delta)$
$\Longrightarrow$ [Theorem 6.14 (Counterexample Validation)]
$\mathsf{D}_S[\![\neg(\varphi(\delta) \wedge \neg\phi)]\!] = 1$
$\Longrightarrow$ [$\mathsf{D}_S[\![\neg\varphi(\delta)]\!] = 1 \implies \mathsf{D}_S[\![\neg\varphi_{\mathsf{U}}(\delta)]\!] = 1$ by Def. 4.2 (Specification Satisfaction)]
$\mathsf{D}_S[\![\neg(\varphi_{\mathsf{U}}(\delta) \wedge \neg\phi)]\!] = 1$

"$\Leftarrow$" (via contraposition):

$\neg\mathit{F}(\delta)$
$\Longrightarrow$ [Theorem 6.14 (Counterexample Validation)]
$\exists\, \pi' = ((L_i', S_i'))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!]), \mathcal{V} \in \mathit{Vals}_{Id}(\mathcal{X}) : \pi'[\varphi(\delta) \wedge \neg\phi]^0(\mathcal{V}) = 1$ $(\star)$

We show by induction over the nesting depth of $\varphi_{\mathsf{U}}(\delta)$ the existence of a run $\pi = ((L_i, S_i))_{i\in\mathbb{N}_0} \in \mathit{Runs}([\![\mathsf{D}, S]\!])$ with $\pi[\varphi_{\mathsf{U}}(\delta) \wedge \neg\phi]^0(\mathcal{V}) = 1$ and $(L_i, S_i) = (L_i', S_i')$ for all $i \in \bot(\delta)$.

Induction Base.

For a nesting depth of zero we have $\varphi_{\mathsf{U}}(\delta)^0 = \varphi(\delta)^0 = \mathbf{tt}$ and set $\pi = \pi'$.

Induction Step.

By induction hypothesis we have a run $\pi \in \mathit{Runs}([\![\mathsf{D}, S]\!])$ with

$$\pi[\tau(L_i^\delta, S_i^\delta, \mathcal{V}_\delta, i)]^{f(i)}(\mathcal{V}) = 1$$

By $(\star)$ we have that $S_{f(i)} \dashrightarrow (L_{f(i+1)}, S_{f(i+1)})$ with

$$\pi[\tau(L_{i+1}^\delta, S_{i+1}^\delta, \mathcal{V}_\delta, i+1)]^{f(i+1)}(\mathcal{V}) = 1$$

Assume there exists a pair (L_k, S_k) with $i < k < f(i)$ with $(L_k, S_\rangle = (\sigma(L_{f(i)}), S_k = \sigma(S_{f(i)})$ for a permutation $\sigma \in \Sigma_{X_{f(i)}}(\mathit{vars}(\tau(L_{i+1}^\delta, S_{i+1}^\delta, \mathcal{V}_\delta, i+1)))$. This implies $(L_i, S_i) \dashrightarrow (L_k), S_k)$ and $(L_k, S_k) \dashrightarrow (L_{f(i+1)}, S_{f(i+1)})$.

We can iteratively apply this relation until we obtain the identity permutation of $(L_{f(i+1)}, S_{f(i+1)})$. So we can construct a continuation of π/i with $(L_i, S_i) \dashrightarrow (L_k', S_k')$ where $(L_k', S_k') = (\sigma^n(L_k'), \sigma^n(S_k'))$ and $(L_l, S_l) \neq (\sigma'(L_k'), \sigma'(S_k'))$ for any permutation $\sigma' \in \Sigma_{X_{f(i)}}(\mathit{vars}(\tau(L_{i+1}^\delta, S_{i+1}^\delta, \mathcal{V}_\delta, i+1)))$. This implies

$$\pi[(\Sigma_{X_{i+1}}(\neg\tau(L_{i+1}, S_{i+1}, \mathcal{V}, n+1))) \,\mathsf{U}\, \tau(L_{i+1}, S_{i+1}, \mathcal{V}, i+1)]^{f(i+1)}(\mathcal{V}) = 1$$

by Def. 6.22 (Formula Permutation) and $\pi[\neg\phi]^{f(i+1)}(\mathcal{V})$ by the induction hypothesis.

■

Proof of Lemma 6.27 (page 121)

Let $a_i \in \{a_1, \ldots, a_n\} = \bot(\delta)$ and

$$\delta' = (((L_i^1, S_i^1))_{0 \leq i \leq n_1}, \mathcal{V}')$$
$$\delta = (((L_i^2, S_i^2))_{0 \leq i \leq n_2}, \mathcal{V}).$$

As $\delta' \in \mathit{Cex}(\mathsf{D}_S, \neg(\varphi_{\mathsf{U}}(\delta) \wedge \neg\phi))$ there exists $j, k \in \{0, \ldots, \mathsf{len}(\delta')\}$ such that $j = \max\{f(a_{i-1}), 0\}$ and $k = f(a_i)$ according to Def. 6.8 (Counterexample Concretisations).

As $\delta'[\varphi_{\mathsf{U}}(\delta)^i]^j(\mathcal{V}') = 1$ we have that there is no permutation $\sigma \in \Sigma_{A(S_k^1)}$ with

$$\delta'[\sigma(\tau(L_k^1, S_k^1, \mathcal{V}', k+1)]^l(\mathcal{V}') = 1$$

for $l \in \{j, \ldots, k\}$. Thus δ' does not cover δ according to Def. 6.26.

■

Proof of Lemma 6.35 (page 131)

Let $((S, \Pi), \mathsf{rcv}[m](u_1, u_2), (S', \Pi')) \in \mathit{succ}^K$. By Def. 3.31 (DES Semantics of DCS) this implies a corresponding send evolution

$$((S_m, \Pi_m), \mathsf{snd}[m](u_2, u_1), (S'_m, \Pi'_m)) \in \mathit{succ}^K$$

with $(S'_s, \Pi'_s)) \dashrightarrow (S, \Pi)$ induced by a send transition $tr_s = (q_s, c!m, q'_s) \in \mathit{succ}$. For $S_s = (U, \iota)$ we obtain $\iota(c)(u_2, u_1) = 1$ by Def. 3.31. We distinguish between the two possibilities how the interpretation of predicate 'c' for (u_2, u_1) becomes 1.

Case 1: There is a create transition $tr_\star = (q, *_t^c, q) \in \mathit{succ}$ with $tr_\star \overset{c}{\rightarrowtail} tr_s$ which denotes reachability on a transition path without modification of channel c, i.e.

$$tr \overset{c}{\rightarrowtail} tr' :\Longleftrightarrow \exists\, tr_0, tr_1, \ldots, tr_n \in \mathit{Paths}_{\mathsf{C}}(tr) : tr_n = tr' \wedge \forall\, i \in \{1, \ldots, n\} : \mathit{chan}(tr_i) = c \rightarrow \mathit{kind}(tr_i) = \mathsf{snd}$$

This entails by Def. 6.31 (Reply Messages) that $m \in \lhd^{\mathsf{M}}(\star)$ and $\lhd^{\mathsf{N}}(\star) > 1$. The corresponding create evolution

$$((S_\star, \Pi_\star), \mathsf{create}[t](u_2, u_1), (S'_\star, \Pi'_\star)) \in \mathit{succ}^K$$

with $(S'_\star, \Pi'_\star) \dashrightarrow (S_s, \Pi_s)$ ensures that $\Pi((u_1, u_2), E) > 1$ for some $E \subseteq \{m\}$ by the K-augmentation of $[\![\mathsf{D}(\mathsf{C})]\!]_I$ according to Def. 6.34.

Case 2: There is a receive transition $tr_r = (q_r, ?m'(c), q_r) \in succ$ with $tr_r \overset{c}{\rightarrowtail} tr_s$. This entails $m \in \lhd^{\mathsf{M}}(m')$ and $\lhd^{\mathsf{N}}(m') > 1$. The corresponding send evolution for m'

$$((S_{m'}, \Pi_{m'}), \mathsf{snd}[m'](u_1, u_2), (S'_{m'}, \Pi'_{m'})) \in succ^K$$

with $(S'_{m'}, \Pi'_{m'}) \dashrightarrow (S_s, \Pi_s)$ ensures that $\Pi((u_1, u_2), E) > 1$ for some $E \subseteq \{m\}$ by the K-augmentation of $[\![\mathsf{D}(\mathsf{C})]\!]_I$ according to Def. 6.34. Note that m' can in particular be an environment message.

■

Proof of Theorem 6.37 (page 132)

Let $\pi = (L_i, (S_i, \Pi_i))_{i \in \mathbb{N}_0} \in Runs([\![\mathsf{D}(\mathsf{C})]\!]_K)$ be a run and $\mathcal{V} \in Vals_I(\mathcal{X})$ a valuation.

$\pi[\mathsf{X}\,\mathsf{rcv}[m](x,y)]^i(\mathcal{V}) = 1$
$\Longrightarrow$ [Def. 4.2 (Specification Satisfaction)]
$L_{i+1} = \mathsf{rcv}[m](x,y)[\mathcal{V}]$
$\Longrightarrow$ [Lem. 6.35 (Positive Counter)]
$\exists\, E \in \mathsf{dom}(\Pi^c_K) : m \in E \wedge \Pi_i((\mathcal{V}(x), \mathcal{V}(y), E) > 1$
$\Longrightarrow$ [Def. 6.36 (Counter Formula)]
$\pi[m^+(x,y)]^i(\mathcal{V}) = 1$
$\Longrightarrow$ [logical implication]
$\pi[\mathsf{X}\,\mathsf{rcv}[m](x,y) \rightarrow m^+(x,y)]^i(\mathcal{V}) = 1$

This entails by Definition 4.10 (DES Specification Evaluation) that $\mathsf{D}(\mathsf{C}_k) \models \varphi(m)$ and by Definition 6.5 (Evolution Constraint) that $\varphi(m)$ is an evolution constraint for C_K.

■

Appendix B

Tool Samples

This appendix gives a first impression of the developed tool implementation SARMC (Spotlight Abstraction Refinement Model-Checker, cf. Sect. 7.1). To this end, we provide the textual representations of both the model and the specifications for the running adhoc networking DES example Ad, and we show the console output when invoking the SARMC tool on these examples.

The DES model of Ad is specified in an XML document called 'adhoc.xml' which we list at the end of this section. The two specifications (cf. Sect. 7.2.1)

$$\phi_2 = \mathsf{G}\left(\neg \mathsf{sl}(x)\right)$$
$$\phi_4 = \mathsf{G}\left(\mathsf{link}(x_1, x_2) \rightarrow \neg \ominus x_1\right)$$

are provided in simple text files as follows:

```
$ cat 2.speck
forall x . G ( ! sl(x) )

$ cat 4.speck
forall x,y . G ( link(x,y) -> !(-)x )
```

Calling the SARMC for verification produces the following outputs which correspond to the data given in the tables in Section 7.2.1:

```
$ sarmc -v adhoc.xml 2.speck
(stats) depth: 0 -- iteration: 0 -- spotlight: 1
(speck) forall x . G ( ! sl(x) )
(spot) 1
(return) 1/2
(stats) depth: 1 -- iteration: 0 -- spotlight: 2
(speck) forall x,b0 . !( (!(con(x,bot) * (o)x * sl(x) * !link(x,x)) *
!(con(b0,bot) * (o)b0 * sl(b0) * !link(b0,b0)) * TRUE) U (con(x,b0) *
(o)x * sl(x) * !link(x,x)) * (TRUE)) + (  G ( ! sl(x) ) )
```

```
(spot) 2
(return) 0
(result) Specification is violated (see 2.speck.wfv).
```

The output indicates the current level of iteration recursion and maximal spotlight for the current specification. The current size of the spotlight required for the actual valution is given in the '`(spot)`' lines. The '`(return)`' lines denote the three-valued result of the current verification task. The final '`(result)`' line gives the return value of the overall verification task for the given model and initial specification. In this example, the resulting counterexample is stored in the file '`2.speck.wfv`', which can be visualised by the trace-viewer tool as shown in Figure 7.2 on page 147.

```
$ sarmc -v adhoc.xml 4.speck
(stats) depth: 0 -- iteration: 0 -- spotlight: 2
(speck) forall x,y . G ( link(x,y) -> !(-)x )
(spot) 2
(return) 1/2
(stats) depth: 1 -- iteration: 0 -- spotlight: 3
(speck) forall x,y,b0 . !( (!(dis(bot,x) * (o)x * dev(x) * !link(x,x) *
link(x,y)) * !(dis(bot,b0) * (o)b0 * dev(b0) * !link(b0,b0) *
link(b0,y)) * !(dis(bot,x) * (o)x * dev(x) * !link(x,x) * link(x,b0)) *
!(dis(bot,y) * (o)y * dev(y) * !link(y,y) * link(y,x)) * !(dis(bot,b0) *
(o)b0 * dev(b0) * !link(b0,b0) * link(b0,x)) * !(dis(bot,y) * (o)y *
dev(y) * !link(y,y) * link(y,b0)) * TRUE) U (dis(b0,x) * (o)x * dev(x) *
!link(x,x) * link(x,y)) * (TRUE)) + ( G ( link(x,y) -> !(-)x ))
(spot) 1
(spot) 2
(spot) 2
(spot) 2
(spot) 3
(return) 1
(evocon) !((F(dis(bot,x) * (o)x * dev(x) * !link(x,x) * link(x,y)))
(stats) depth: 0 -- iteration: 1 -- spotlight: 2
(speck) forall x,y . ((!((F(dis(bot,x) * (o)x * dev(x) * !link(x,x) *
link(x,y)) * TRUE))) * TRUE) -> ( G ( link(x,y) -> !(-)x ))
(spot) 2
(return) 1
(result) Specification holds.
```

Finally, we present the XML description of the Ad model.

```
$ cat adhoc.xml
<des name="adhoc" author="tobe" xmlns="http://www.avacs.org/
    des" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.avacs.org/des deslib.xsd">

<signature>
  <states>
    <predicate name="dev" arity="1"/>
    <predicate name="sl" arity="1"/>
    <predicate name="ma" arity="1"/>
  </states>
  <links>
    <predicate name="link" arity="2"/>
  </links>
  <evolutions>
    <predicate name="new" arity="1"/>
    <predicate name="con" arity="2"/>
    <predicate name="dis" arity="2"/>
    <predicate name="free" arity="1"/>
    <predicate name="del" arity="1"/>
  </evolutions>
  <variables>
    <variable name="x"/>
    <variable name="y"/>
  </variables>
</signature>

<rules>
 <rule name="new">
   <params>
     <param name="x"/>
   </params>
   <guard>
     <dead><param name="x"/></dead>
   </guard>
   <actions>
     <create><param name="x"/></create>
     <action name="dev" ow="true"><param name="x"/></action>
   </actions>
 </rule>
```

```
<rule name="con">
  <params>
    <param name="x" />
    <param name="y" />
  </params>
  <guard>
    <term name="dev"><param name="x" /></term>
    <string name="and" />
    <string name="(" />
    <term name="dev"><param name="y" /></term>
    <string name="or" />
    <term name="ma"><param name="y" /></term>
    <string name=")" />
    <string name="and" />
    <neq>
      <param name="x" />
      <param name="y" />
    </neq>
  </guard>
  <actions>
    <action name="sl" ow="true"><param name="x" /></action>
    <action name="ma" ow="true"><param name="y" /></action>
    <action name="link">
      <param name="x" />
      <param name="y" />
    </action>
    <action name="link">
      <param name="y" />
      <param name="x" />
    </action>
  </actions>
</rule>

<rule name="dis">
  <params>
    <param name="x" />
    <param name="y" />
  </params>
  <guard>
    <term name="ma"><param name="x" /></term>
    <string name="and" />
    <term name="link">
      <param name="x" />
```

```
      <param name="y"/>
    </term>
  </guard>
  <actions>
    <action name="dev"><param name="y"/></action>
    <action name="link" negate="true">
      <param name="x"/>
      <param name="y"/>
    </action>
    <action name="link" negate="true">
      <param name="y"/>
      <param name="x"/>
    </action>
  </actions>
</rule>

<rule name="free">
  <params><param name="x"/></params>
  <guard>
    <term name="ma"><param name="x"/></term>
    <string name="and not"/>
    <term name="link"><param name="x"/></term>
  </guard>
  <actions>
    <action name="dev" ow="true"><param name="x"/></action>
  </actions>
</rule>

<rule name="del">
  <params><param name="x"/></params>
  <guard>
    <term name="dev"><param name="x"/></term>
  </guard>
  <actions>
    <action name="dev" negate="true"><param name="x"/></
        action>
    <destroy><param name="x"/></destroy>
  </actions>
</rule>

</rules>
</des>
```

Index

Curriculum Vitae

Tobe Toben, born 16.11.1975 in Wilhelmshaven, Germany

1982 - 1995	school in Wittmund, Germany
1995 - 1996	alternative civilian service in Wittmund, Germany
10/1996 - 02/2002	study of computer science at the Carl von Ossietzky Universität Oldenburg, Germany
02/2002 - 01/2004	research assistent at the OFFIS institute for computer science in Oldenburg, Germany
since 02/2004	research assistent at the Carl von Ossietzky Universität Oldenburg, Germany
February 2009	defense of the dissertation