



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

Development of Correct Graph Transformation Systems

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von
Dipl.-Inform. Karl-Heinz Pennemann

Oldenburg, 18. Mai 2009

Prüfungskommission

Vorsitzender: Prof. Dr. Martin-Fränzle
Gutachterin: Prof. Dr. Annegret Habel
Gutachter: Prof. Dr. Ernst-Rüdiger Olderog
Gutachter: Prof. Dr. Arend Rensink
Mitglied: Dr. Sybille Fröschle
Datum der Disputation: 11. September 2009

Abstract

Graph transformation has many application areas in computer science, such as software engineering or the design of concurrent and distributed systems. Being a visual modeling technique, graph transformation has the potential to play a decisive role in the development of increasingly larger and complex systems. However, the use of visual modeling techniques alone does not guarantee the correctness of a design. In context of rising standards for trustworthy systems, there is a growing need for the verification of graph transformation systems and programs. The research of appropriate methods for this purpose is the topic of this thesis.

The primary goal is to obtain the capability to decide graphical program specifications. These specifications consists of a graphical precondition, a graph program, and a graphical postcondition. As usual, such a specification is said to be correct, if all those system states satisfy the postcondition that are reachable by applying the program on a start state satisfying the precondition. In the considered programs, the selection, deletion, addition and deselection of a graph's nodes and edges are the elementary constructs that can be composed to more complex programs by non-deterministic choice, sequential composition and iteration. The resulting programming language is computationally complete and is able to model transactions that deal with an unbounded number of nodes and edges. As language for the specification of state properties, graph conditions are investigated and used. We show that graph conditions provide an intuitive formalism for first-order structural properties and are suited to infer knowledge about the behavior of graph transformation systems and programs.

According to Dijkstra, the correctness of program specifications can be shown by constructing a weakest precondition of the program relative to the postcondition and checking whether the specified precondition implies the weakest precondition. Hence the correctness problem of program specifications is reduced to an implication problem of conditions. In this thesis, it is shown how to construct weakest preconditions for graph programs and graph conditions. Following a dual approach, a sound and complete satisfiability

algorithm for graph conditions is investigated and a fragment of conditions is identified, for which the algorithm decides. On the other hand, a resolution-based calculus for graph conditions is presented and its soundness is proven. Implementations of the aforementioned deciders for conditions are compared with existing theorem provers and satisfiability solvers for first-order logic by verifying three case studies: a railroad control, an access control for computer systems, and, as an external example, a car platoon maneuver protocol.

The research is done within the framework of the so-called weak adhesive high-level replacement categories. Therefore, the results will be applicable to different kinds of graph replacement systems and Petri nets, providing theoretical fundamentals and general concepts for the development of correct transformation-based systems and programs.

Zusammenfassung

Graphtransformation hat viele Anwendungsgebiete in der Informatik, zum Beispiel im Softwareentwurf oder in der Modellierung von nebenläufigen oder verteilten Systemen. Als visuelle Modellierungstechnik hat Graphtransformation das Potenzial, eine entscheidende Rolle in der Entwicklung von immer größer und komplexer werdenden Systemen einzunehmen. Allerdings garantiert die Benutzung einer visuellen Modellierungstechnik noch nicht die Korrektheit eines Modells. Im Hinblick auf steigende Standards für vertrauenswürdige Systeme ergibt sich ein wachsendes Interesse an der Verifikation von Graphtransformationssystemen und -programmen. Die Entwicklung entsprechender Methoden zu diesem Zweck ist das Thema dieser Dissertation.

Primäres Ziel ist die Erlangung der Fähigkeit, grafische Programmspezifikationen zu entscheiden. Diese Spezifikationen bestehen aus einer grafischen Vorbedingung, einem Graphprogramm und einer grafischen Nachbedingung. Man nennt eine solche Spezifikation korrekt, wenn diejenigen Systemzustände der Nachbedingung genügen, die durch Ausführung des Graphprogramms von einem die Vorbedingung erfüllenden Startzustand aus erreichbar sind. In den betrachteten Programmen sind das Selektieren, Löschen, Hinzufügen und Deselektieren von Knoten und Kanten eines Graphen die elementaren Konstrukte, die durch nichtdeterministische Auswahl, sequentielle Komposition und Iteration zu komplexeren Programmen verknüpfbar sind. Die entstehende Programmiersprache ist Turing-vollständig und erlaubt beispielsweise die Modellierung von Transaktionen, die eine unbeschränkte Anzahl von Knoten und Kanten betreffen. Als Beschreibungssprache für Zustandseigenschaften werden Graphbedingungen untersucht und benutzt. Es wird gezeigt, dass Graphbedingungen einen intuitiven Formalismus für Struktureigenschaften erster Stufe bereit stellen und darüber hinaus geeignet sind, Informationen über das Verhalten von Systemen und Programmen abzuleiten.

Die Korrektheit von Programmspezifikationen kann, nach Dijkstra, untersucht werden durch Konstruktion einer schwächsten Vorbedingung aus dem Programm relativ zur Nachbedingung und durch Entscheiden, ob die spezifizierte Vorbedingung die schwächste Vorbedingung impliziert. In diesem

Sinne wird das Problem der Korrektheit von Programmspezifikationen auf das Implikationsproblem von Bedingungen reduziert. In dieser Arbeit wird gezeigt, wie schwächste Vorbedingungen für Graphprogramme und Graphbedingungen konstruiert werden. Einem dualen Ansatz folgend, wird einerseits ein korrekter und vollständiger Erfüllbarkeitsalgorithmus für Graphbedingungen untersucht und ein Fragment von Graphbedingungen identifiziert, für das der Algorithmus entscheidet. Andererseits wird ein resolutionsbasierter Kalkül für das Beweisen von Graphbedingungen präsentiert und seine Korrektheit bewiesen. Implementierungen der zuvor genannten Komponenten werden mit bestehenden Werkzeugen für Logik erster Stufe anhand dreier Fallstudien verglichen: einem Eisenbahnkontrollsystem, einer Zugangskontrolle für Computersysteme und, als externe Fallstudie, einem Protokoll für Manöver von Autokolonnen.

Die Untersuchungen werden innerhalb des Rahmenwerks der so genannten schwach adhesiven high-level Ersetzungskategorien durchgeführt. Die Ergebnisse sind damit auf verschiedene Arten von Graphersetzungssystemen und Petri-Netzen anwendbar und stellen ein generelles Konzept zur Entwicklung von korrekten transformationsbasierten Systemen und Programmen dar.

Danksagung

Diese Arbeit wurde durch die Deutsche Forschungsgemeinschaft (GRK 1076/1 Graduiertenkolleg Vertrauenswürdige Softwaresysteme) unterstützt. Ich danke den Leitern des Graduiertenkollegs für das in mich gesetzte Vertrauen. Weiter danke ich allen, die mich während meiner Promotion fachlich oder persönlich unterstützt und motiviert haben.

Ganz herzlich danke ich meiner Betreuerin Frau Prof. Annegret Habel, den Gutachtern Herrn Prof. Ernst-Rüdiger Olderog und Herrn Prof. Arend Rensink, den Mitgliedern des Prüfungsausschusses Herrn Prof. Martin Fränzle und Frau Dr. Sibylle Fröschle, den Mitgliedern der Arbeitsgruppe Formale Sprachen, Christian Zuckschwerdt, Stefan Moll und Karl Azab, und den Mitgliedern des Graduiertenkollegs. Mein ganz besonderer Dank gilt Birgitt und unseren Familien.

Diese Arbeit widme ich Birgitt und unserem Kind.

*“So little time,
so much to do.”*

Winston Churchill

Contents

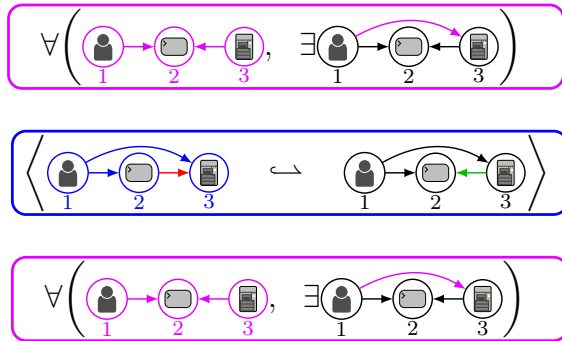
1	Introduction	1
2	Preliminaries	7
2.1	The category of graphs	7
2.2	Weak adhesive HLR categories	15
3	Conditions	19
3.1	Nested conditions	19
3.2	Comparison of \mathcal{M} - and \mathcal{A} -satisfiability	23
3.3	Comparison of graph conditions and graph formulas	31
3.4	Summary and discussion	42
4	Programs and transformation systems	43
4.1	Programs with interface	43
4.2	Elimination of Fix statements	51
4.3	Related concepts	53
4.4	Summary and discussion	55
5	Correctness of program specifications	57
5.1	Program specifications	57
5.2	Basic transformations on conditions	58
5.3	Weakest liberal preconditions	67
5.4	Weakest invariants	74
5.5	Related concepts	79
5.6	Summary and discussion	82
6	The implication problem of conditions	83
6.1	Implication, tautology and satisfiability	83
6.2	Satisfiability solver SeekSat	87
6.3	Theorem prover PROCON	98
6.4	Related concepts	110

6.5	Summary and discussion	112
7	Case studies	115
7.1	General set-up	115
7.2	Railroad control system	117
7.3	Car platoon maneuver protocol	124
7.4	Access control for computer systems	133
7.5	Summary and discussion	141
8	Conclusion	145
8.1	Results	145
8.2	Open problems and future work	147
A	Partial monomorphisms	149
B	Proof	153
	Bibliography	157
	Abbreviations	171
	Index	173
	Curriculum Vitae	177

1. Introduction

Graph transformation has many application areas in computer science, such as software engineering or the design of concurrent and distributed systems. Especially the operational behavior of structure changing systems such as “mobile” systems (in the sense of dynamically changing communication topologies) is suited to be modeled by graph transformation rules [EHKP91].

In the context of increasingly larger and more complex systems that hardware and software engineers have to construct, visual modeling techniques such as graph transformation can be expected to play a key role in the future. However, the use of visual modeling techniques alone does not guarantee the correctness of a design. The complexity of the problem to consider all possible outcomes of a given behavior specification remains the same whether such a specification is visual or not. In context of rising standards for trustworthy systems, there is a growing need for the verification of graph transformation systems and programs. The research of appropriate methods for this purpose is the topic of this thesis. More precisely, a major goal is the ability to determine the correctness of graph program specifications consisting of a graph precondition, a graph program and a graph postcondition such as the one presented in Figure 1.1. As usual, such a specification is *correct*, if all



precondition: Every user logged into a system has the appropriate access right.

program: If a user with the appropriate access right proposes a session, it is accepted.

postcondition: Every user logged into a system has the appropriate access right.

Figure 1.1: Example specification of an access control system

those system states satisfy the postcondition that are reachable by applying the program on a start state satisfying the precondition.

In our approach, *graphs* represent *system states*. Directed labeled graphs are ubiquitous in computer science and suited to represent discrete aspects of system states. A key feature of graphs is their well-known graphical representation, which is able to directly visualize relations between a set of elements.

As language for specifying *state properties*, we consider *graph conditions* [HHT96, HW95, KMP05, EEHP04, Pen04, Ren04a, HP05, EEHP06]. Graph conditions may be seen as a tree of graph mappings equipped with logical symbols and provide an intuitive, yet precise formalism, well-suited to describe structural properties.

We introduce so-called *graph programs with interface* to model *system transitions* by structural transformations. The selection, deletion, addition and deselection of a graph's nodes and edges are the basic program statements that can be composed to more complex programs by non-deterministic choice, sequential composition and iteration. The considered programming language subsumes double pushout transformation rules and programs based thereon [HP01, HPR06] and is able to model transactions that deal with an unbounded number of elements.

According to Dijkstra, the *correctness of program specifications* can be shown in a classical way by constructing a *weakest precondition* of the program relative to the postcondition and checking whether the specified precondition implies the weakest precondition. In this sense, the correctness problem of program specifications is reduced onto the *implication problem* of conditions.

To determine whether or not a precondition implies a weakest precondition, either a proof or a counterexample must be found. We follow a dual approach by investigating dedicated components for proving and solving conditions. Following the outlines of [AHPZ07], the components sketched in Figure 1.2 are implemented. The overall approach is evaluated by modeling and verifying three case studies. The prover and solver components are evaluated against existing tools such as VAMPIRE, DARWIN and PARADOX applied onto straightforward translations of graph conditions into first-order graph formulas [Ren04a, HP06, HP09], using the axiomatization of Courcelle [Cou97] for finite, directed graphs, which is extended by labeling axioms.

We are interested in programs and conditions over graph-like structures. To avoid similar investigations for comparable structures, we abstract from specific definitions and conduct our research in the framework of weak adhesive high-level replacement categories. Therefore, our results hold for replacement-capable structures such as Petri-nets, graphs, and hypergraphs.

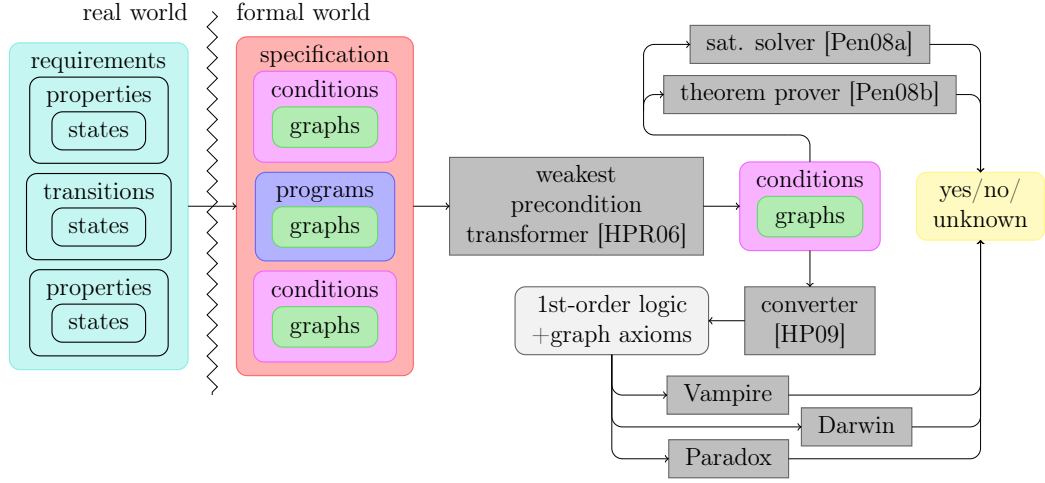


Figure 1.2: Overview of this thesis

Related work

More than 35 years ago [EPS73], graph transformation was proposed as a generalization of string replacement with the intent to yield an operational notation that captures and inherits the advantage of graphs – the ability to directly visualize relationships between elements. Since then, a number of programming languages based on graph transformation rules have been investigated, for instance, a minimal, but still computationally complete graph program language [HP01].

Yet, surprisingly little effort was made to transfer the idea of graph-based formalisms to logical languages, let alone program specifications. Traditionally, the analysis of graph transformation systems focused on properties concerning only the systems themselves, such as confluence [Plu93, Plu05] and termination [Plu95]. In the last six years, verification of graph transformation systems in terms of model checking became en-vogue. Ultimately, these approaches are based on the simple idea of a systematical exhaustive exploration of all reachable graphs and transitions of a graph transformation system with respect to a start graph. Consequently they are restricted to finite transformation systems [Var03, DFRdS03], finite transformation systems up to isomorphism [Ren04b], or they apply abstraction [BK02, RD06, Bau06] and necessarily cover only fragments of logics [BKK03] that are compatible with the considered abstraction. In the latter case, for instance, one may verify that “Two nodes are never connected (by a path)” but not “Two nodes are always connected (by a path)”.

At the starting point of our research, there existed no approach that was able to decide graph-based program specifications such as the one pre-

sented in Figure 1.1. However, parallel and independent to our work, Strecker [Str08] investigated a model of graph programs in the proof assistant Isabelle. His approach supports the manual verification of formulas of “a fragment of first-order logic enriched by transitive closure”, therefore is at least able to formalize specifications such as the one given in Figure 1.1.

Contributions

The main contribution of this thesis is the ability to automatically verify or refute graph program specifications. While the correctness problem of the considered graph program specifications is undecidable, almost every test specification of the case studies considered in the evaluation of our approach can be automatically decided.

To this end, we introduce and investigate programs with interface to describe structural transformations within a category of objects that are restrictable to a previously selected context handed over between elementary computation steps. We define weakest liberal preconditions of programs with interface and nested conditions. We present a construction for weakest liberal preconditions and consider algorithms for the construction of (weakest) invariants. We relate the decidability of the implication, the tautology and the satisfiability problem of conditions. We classify the expressivity of graph conditions by investigating translations between graph conditions and first-order formulas on graphs. We lift the undecidability of the satisfiability problem of first-order logic on graphs to the satisfiability problem of graph conditions. Following a dual approach, we investigate a sound satisfiability solver for conditions that is complete for a class of weak adhesive HLR categories. We investigate a fragment of conditions for which the solver terminates and decides. On the other hand, we present a calculus for proving conditions and show its soundness. We evaluate our approach and the implementations of the aforementioned transformations and algorithms by modeling and verifying three different case studies: a railroad control [Pen04, HP05], an access control for computer systems [HPR06], and, as an external example, a car platoon maneuver protocol [Bau06, HESV91]. We show that our implementation is able to decide 327 of 330 considered test specifications. We also show that the developed prover and solver for conditions are superior in terms of performance and coverage to existing first-order theorem provers and satisfiability solvers applied onto straightforward translations of graph conditions into first-order graph logic. We give a number of possible reasons for this observation.

Thesis structure

Chapter 2 recalls the basic notions such as categories, morphisms, and the framework of weak adhesive high-level replacement (HLR) categories. As an example structure, we consider finite, directed, labeled graphs which – together with the class of injective graph morphisms – constitutes a weak adhesive HLR category.

In Chapter 3 we consider nested conditions, investigate two notions of satisfiability for conditions, namely \mathcal{M} - and \mathcal{A} -satisfiability, and classify the expressiveness of nested graph conditions by showing that nested graph conditions and first-order graph formulas are expressively equivalent.

Chapter 4 introduces programs with interface as a means of structural transformation and considers a normal form. We classify the computational power of this programming language by relating it to the existing notion of graph programs.

In Chapter 5 we define correctness of program specifications, define and show how to construct weakest liberal preconditions of programs with interface and postconditions, and demonstrate the use of weakest liberal preconditions to reduce the correctness problem of program specifications onto the implication problem of nested conditions.

Chapter 6 investigates the connections between the implication, the tautology and the satisfiability problem of conditions, shows that all problems are undecidable for the category of graphs, and presents a satisfiability algorithm as well as a calculus for proving conditions over a class of weak adhesive HLR categories.

In Chapter 7 we model and verify selected aspects of three real-world systems to evaluate our approach to the verification of program specification.

Chapter 8 summarizes the results of this thesis and discusses topics for future work. Appendix A introduces partial monomorphism as spans of morphisms, while in Appendix B, some of the proofs are collected.

2. Preliminaries

In the following we recall the basic notions used in this thesis such as objects and morphisms, categories, and weak adhesive high-level replacement (HLR) categories. We review the standard definition of directed, labeled graphs which – together with the class of injective graph morphisms – constitutes a weak adhesive HLR category. We assume the reader to be familiar with basic mathematical concepts and termini such as functions and mappings.

2.1 The category of graphs

In the following we recall the notions of categories, in particular the category of graphs, pushouts and pushout complements. A category consists of a set of objects and morphisms between the objects. In this thesis, an object represents discrete aspects of a system state, while a morphism relates a pair of objects. Objects can be all kinds of structures which are of interest in computer science and mathematics such as Petri-nets, (hyper)graphs, and algebraic specifications. Morphisms, such as net, (hyper)graph, and specification morphisms, are typically structure-preserving mappings relating the elements contained within the objects. Readers interested in the category-theoretic background may consult [AL91, AHS90, AM75].

Definition 2.1 (category). A *category* is a tuple $\mathcal{C} = \langle \mathcal{O}, \mathcal{A} \rangle$ consisting of a class \mathcal{O} of objects and a class \mathcal{A} of so-called *morphisms* between the objects. Each morphism $a \in \mathcal{A}$ has a unique source object, called *domain* and denoted by $\text{dom}(a)$, and a unique target object, called *codomain* and denoted by $\text{codom}(a)$. We write $a: \text{dom}(a) \rightarrow \text{codom}(a)$ or just $\text{dom}(a) \rightarrow \text{codom}(a)$ to denote a morphism. Let $\mathcal{A}(A, B)$ denote the class of all morphisms with domain A and codomain B . There is a binary operation $\mathcal{A}(A, B) \times \mathcal{A}(B, C) \rightarrow \mathcal{A}(A, C)$ called *composition of morphisms*, denoted by $b \circ a$ for morphisms a, b with $\text{codom}(a) = \text{dom}(b)$, for every objects $A, B, C \in \mathcal{O}$. Moreover, the following axioms hold:

- *associativity*: for every morphisms a, b, c with $\text{codom}(a) = \text{dom}(b)$ and $\text{codom}(b) = \text{dom}(c)$, we have $c \circ (b \circ a) = (c \circ b) \circ a$.
- *identity*: for every object A , there exists a morphism $\text{id}_A: A \rightarrow A$, called the *identity morphism for A* , such that for every morphism a , we have $\text{id}_{\text{codom}(a)} \circ a = a = a \circ \text{id}_{\text{dom}(a)}$.

A morphism m is a *monomorphism*, if $m \circ a = m \circ b$ implies $a = b$ for all morphisms a, b with $\text{codom}(a) = \text{codom}(b) = \text{dom}(m)$. A morphism e is an *epimorphism*, if $a \circ e = b \circ e$ implies $a = b$ for all morphisms a, b with $\text{dom}(a) = \text{dom}(b) = \text{codom}(e)$. A morphism a is an *isomorphism*, if there is a morphism a^{-1} such that $a \circ a^{-1} = \text{id}_{\text{codom}(a)}$ and $a^{-1} \circ a = \text{id}_{\text{dom}(a)}$. For an isomorphism a , $\text{dom}(a)$ and $\text{codom}(a)$ are *isomorphic*, which is denoted by $\text{dom}(a) \cong \text{codom}(a)$. Let *Mon*, *Epi*, *Iso* denote the set of all monomorphisms, epimorphisms and isomorphisms, respectively. A category in which every monomorphic epimorphism is an isomorphism is called *balanced*.

Assumption 2.2. From now on, we assume that \mathcal{C} is a balanced category.

Notation. Objects are denoted by upper-case letters while morphisms are denoted by lower-case letters. We sometimes write $A \in \mathcal{C}$ as a synonym for $A \in \mathcal{O}$, and $B \leftarrow A$ as a synonym for $A \rightarrow B$. We write “ \hookrightarrow ” instead of “ \rightarrow ” to indicate that a morphism is in *Mon*, and we write “ \leftrightarrow ” instead of “ \rightarrow ” to indicate that a morphism is in *Iso*.

As a typical category, we consider the category of finite, directed, labeled graphs [CMR⁺97, Ehr79]: Graphs are a well-known, general-purpose structure and their graphical representation is able to directly visualize relationships between elements. Graphs consists of a set of nodes (or vertices), a set of edges, and every node and edge is assigned to a label from a finite label alphabet. Graph morphisms are total, structure-preserving pairs of mappings: every node in the domain is mapped onto some node in the codomain; every edge in the domain is mapped onto some edge in the codomain. Moreover, for all mapped edges, the image of the source node of an edge corresponds to the source node of the edge’s image. The same holds for target nodes. Finally, labels are preserved: nodes and edges may only be mapped onto elements carrying the same label.

Definition 2.3 (graphs and graph morphisms). Let $C = \langle C_V, C_E \rangle$ be a fixed, finite, disjoint label *alphabet*. A *graph* over C is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ consisting of two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, (total) *source* and *target functions* $s_G, t_G: E_G \rightarrow V_G$, and

two (total) *labeling functions* $l_G: V_G \rightarrow C_V$ and $m_G: E_G \rightarrow C_E$. A graph with an empty set of nodes is *empty* and denoted by \emptyset . A *graph morphism* $g: G \rightarrow H$ consists of two (total) functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources, targets, and labels, that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $l_H \circ g_V = l_G$, and $m_H \circ g_E = m_G$. A graph morphism g is *injective* (*surjective*) if both g_V and g_E are injective (surjective). The composition of graph morphisms is the componentwise composition, that is, $(h \circ g) = \langle h_V \circ g_V, h_E \circ g_E \rangle$.

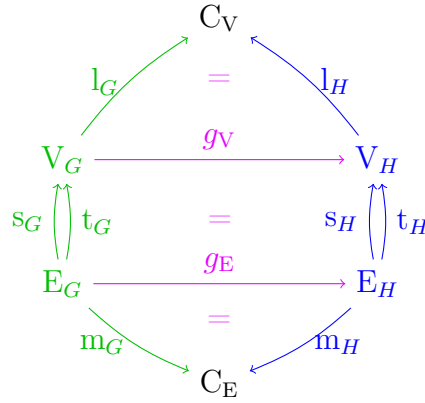


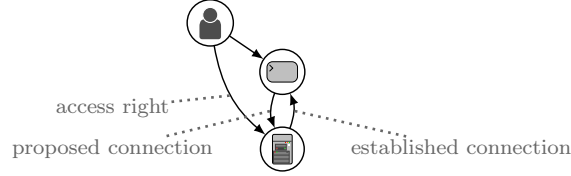
Figure 2.1: A graph morphism $g: G \rightarrow H$

Notation. In drawings of graphs, nodes are drawn by circles carrying the node label inside, for instance, “ \textcircled{V} ”; edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow, for instance, “ \xrightarrow{a} ”. In graphical depictions of graph morphisms, indices convey (only) the mapping of nodes and edges, if necessary. Indices do not state whether nodes and edges of the domain and codomain are identical or not. Additionally, colors may be used to highlight mapped elements and/or to highlight elements of the codomain that have no preimage.

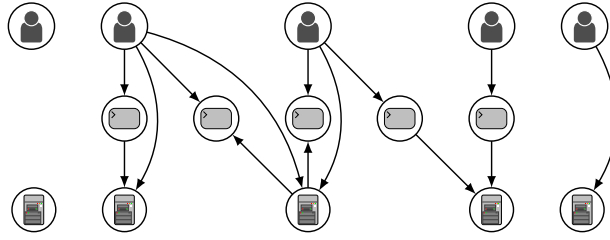
Example 2.4 (access control graphs). In the following we present state graphs of a simple access control for computer systems, which abstracts authentication and models user and session management in a simple way. We use this example solely for illustrative purposes. A more elaborated, role-based access control model is considered in [KMP05].

Let be $C = \langle \{\textcircled{U}, \textcircled{S}, \textcircled{C}\}, \{ _ \} \rangle$ the access control alphabet. The basic items of our control system are users \textcircled{U} , sessions \textcircled{S} , and computer systems \textcircled{C} , which we represent as labeled nodes (\textcircled{U} , \textcircled{S} , and \textcircled{C}) in our access control

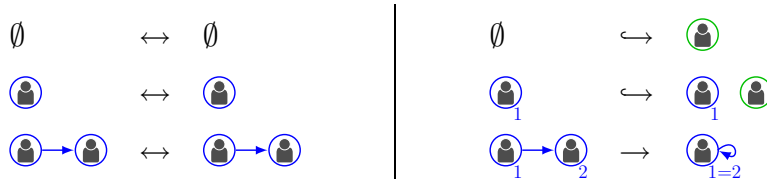
graphs. Directed edges between these nodes represent binary relations on these elements.



An edge between a user and a system ($\text{User} \rightarrow \text{System}$) represents that the user has the right to access the system, that is, the right to establish a session with the system. Every session is connected to a user ($\text{User} \rightarrow \text{Session}$) and a system ($\text{Session} \rightarrow \text{System}$ or $\text{System} \leftarrow \text{Session}$). The direction of the latter edge differentiates between sessions that have been proposed ($\text{Session} \rightarrow \text{System}$) and sessions that have been established ($\text{System} \leftarrow \text{Session}$). A typical access control graph, such as the one depicted below, consists of an arbitrary, but finite number of user, session and computer nodes connected by directed edges as described below.



Consider the following example access control morphisms. According to our convention, indices convey the mapping, not the identities of the nodes. For instance, the isomorphism $\text{User}_1 \leftrightarrow \text{User}_1$ maps a user node onto a user node, but it remains open whether the user nodes refer to the same user or not.



Graphs and graph morphisms form a category.

Fact 2.5 ([Ehr79]). The set of all (finite, directed, labeled) graphs and the set of all graph morphisms form the category **Graphs**. Graph monomorphisms are exactly injective graph morphisms and graph epimorphisms are exactly surjective graph morphisms, and graph isomorphisms are exactly morphisms that are both injective and surjective [EPT06, Example A.11, Fact A.14 and Fact 2.15].

category	category of graphs
object	graph
morphism	(graph) morphism
monomorphism	injective morphism
epimorphism	surjective morphism
isomorphism	morphisms both injective and surjective

Table 2.1: Correspondence of categorical and graph notions

Later on, we use the fact that isomorphisms are closed under decomposition.

Fact 2.6 (decomposition of isomorphisms). For all morphisms $a, b \in \mathcal{A}$, we have: $(b \circ a)$ is an isomorphism implies a and b are isomorphisms.

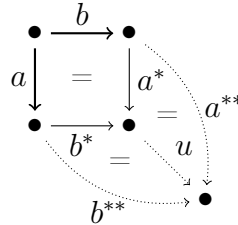
Proof. We have that a is an isomorphism: $a^{-1} = (b \circ a)^{-1} \circ b$ is the inverse of a with $a^{-1} \circ a = \text{id}_{\text{dom}(a)}$ and $a \circ a^{-1} = \text{id}_{\text{codom}(a)}$. We have that b is an isomorphism: $b^{-1} = a \circ (b \circ a)^{-1}$ is the inverse of b with $b^{-1} \circ b = \text{id}_{\text{dom}(b)}$ and $b \circ b^{-1} = \text{id}_{\text{codom}(b)}$. \square

Spans and cospans are pairs of morphisms with a common domain (codomain) and used in the definition of pushout and pullbacks.

Definition 2.7 (span and cospan). A *span* $\langle a, b \rangle$ is a pair of morphisms a, b with a common domain $\text{dom}(a) = \text{dom}(b)$. Analogously, a *cospan* $\langle a^*, b^* \rangle$ is a pair of morphisms a^*, b^* with a common codomain $\text{codom}(a^*) = \text{codom}(b^*)$.

The following categorical notion will be used to formally define the semantics of the considered programming language.

Definition 2.8 (pushout). Given a span of morphisms $\langle a, b \rangle$, a *pushout*, is a cospan of morphisms $\langle b^*, a^* \rangle$ such that the diagram $b^* \circ a = a^* \circ b$ commutes and the *universal property* holds: For every other cospan of morphisms $\langle a^{**}, b^{**} \rangle$ such that $b^{**} \circ a = a^{**} \circ b$, there exists a unique morphism $u: \text{codom}(a^*) \rightarrow \text{codom}(a^{**})$ such that $a^{**} = u \circ a^*$ and $b^{**} = u \circ b^*$.



Sometimes, we refer to the whole diagram $b^* \circ a = a^* \circ b$ as the pushout and to the common codomain $\text{codom}(a^*) = \text{codom}(b^*)$ as the *pushout object*.

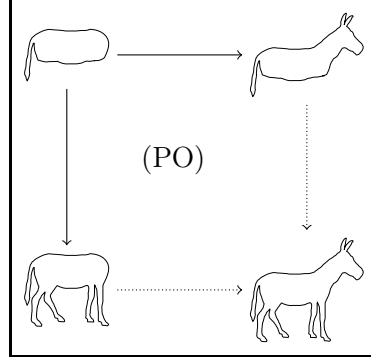
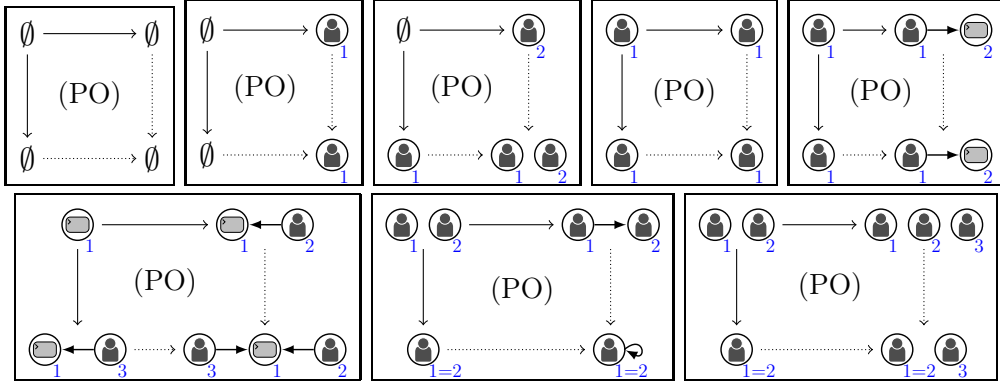


Figure 2.2: A set-theoretic depiction of a pushout

Remark 2.9. In the category of sets and similar categories, the pushout can be seen as a most general union, or alternatively, addition. Suppose a, b are inclusions in the category of sets and $\text{codom}(a) \cap \text{codom}(b) = \text{dom}(a) = \text{dom}(b)$. Then the pushout object is the union of $\text{codom}(a)$ and $\text{codom}(b)$. For a span $\langle a, b \rangle$ of injective graph morphisms, the pushout is intuitively constructed by “gluing” the graphs $\text{codom}(a)$ and $\text{codom}(b)$ together according to their common domain $\text{dom}(a) = \text{dom}(b)$.

Example 2.10 (graph pushouts). Consider the following pushouts.



Note that the indices represent the mappings, but not necessarily the identities of the elements. For instance, in the third pushout, whether user ①_1 of the bottom left corner and ②_2 of the upper right corner have the same identity or not, the pushout object consists of two distinct users. Furthermore, the last two pushouts involve non-injective morphisms, and therefore identify elements.

In the category **Graphs**, pushouts exist for every span $\langle a, b \rangle$ of morphisms and can be effectively constructed. The following formal description of the

pushout construction is taken from [HMP01, Ehr79].

Construction (graph pushouts). For a span of graph morphisms $\langle a, b \rangle$, the *pushout* $\langle b^*, a^* \rangle$ is constructed as follows: Let $\text{codom}(a^*) = D$ and construct D , separately for nodes and edges, by constructing the disjoint union $\text{codom}(a) + \text{codom}(b)$ and fusing elements according to the smallest equivalence relation \approx with $a(x) \approx b(x)$ for all x in $\text{dom}(a)$. Let $[x]$ denote the equivalence class of x according to \approx , for $x \in \text{codom}(a) + \text{codom}(b)$. Define: $s_D([e]) = [s_{\text{codom}(a)}(e)]$, if $e \in E_{\text{codom}(a)}$, otherwise $s_D([e]) = [s_{\text{codom}(b)}(e)]$. Define t_D analogously to s_D . Define: $l_D([v]) = l_{\text{codom}(a)}(v)$ if $v \in V_{\text{codom}(a)}$, otherwise $l_D([v]) = l_{\text{codom}(b)}(v)$. Define m_D analogously to l_D . Now $a^*: \text{codom}(b) \rightarrow D$ and $b^*: \text{codom}(a) \rightarrow D$ are the graph morphisms that send each element to its equivalence class, that is, $a^*(x) = [x]$ for $x \in \text{codom}(b)$ and $b^*(x) = [x]$ for $x \in \text{codom}(a)$, separately for nodes and edges.

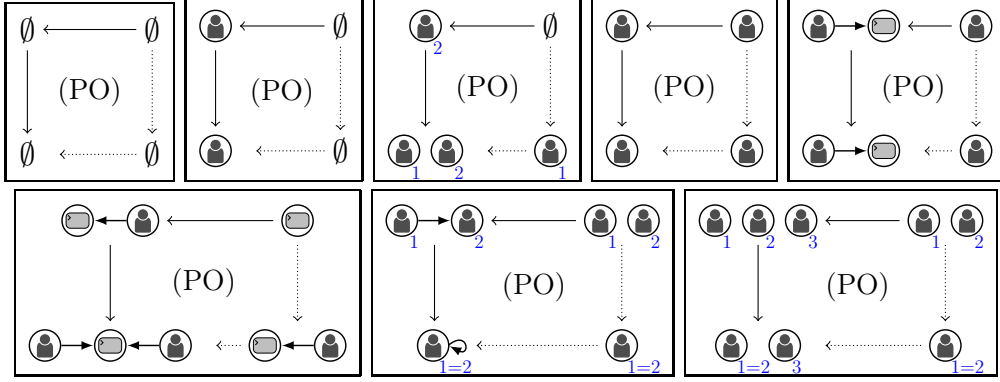
If a composition of morphisms $a \circ b$ is given, the question arises whether or not it can be extended to a pushout.

Definition 2.11 (pushout complement). Given a composition of morphisms $a \circ b$, a *pushout complement* is a composition of morphisms $b^* \circ a^*$ such that $a \circ b = b^* \circ a^*$ is a pushout.

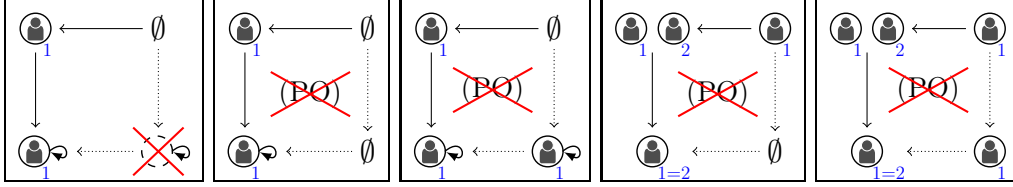
$$\begin{array}{ccc} \bullet & \xleftarrow{b} & \bullet \\ a \downarrow & \text{(PO)} & \downarrow a^* \\ \bullet & \xleftarrow{b^*} & \bullet \end{array}$$

Remark 2.12. In the category of sets and similar categories, the pushout complement construction of a composition of monomorphisms $a \circ b$ corresponds intuitively to the deletion of those elements in $\text{codom}(a)$ that have a preimage in $\text{dom}(a) = \text{codom}(b)$, but not a preimage in $\text{dom}(b)$. However, in the category of graphs, the pushout complement cannot always be constructed: source and target nodes of preserved edges may not be deleted and identified elements must be preserved. Otherwise, either the definition of graph or pushout is violated. The existence of a pushout complement for a composition of graph morphisms $a \circ b$ can be expressed as sufficient condition, called “gluing condition” [CMR⁺97, Ehr79].

Example 2.13 (graph pushout complements). Consider the following example pushout complements.



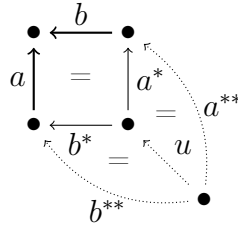
Example 2.14 (non-existing graph pushout complements).



In the first square, the lower right object depicts a “dangling edge”, a situation that would occur if the node $\textcircled{1}$ is deleted but the loop is preserved. For the remaining examples, the squares do not constitute pushouts. There are no objects for the lower right corner of these diagrams such that the result would be a pushout.

Finally, let us review the notion of the pullback, which is the categorical dual of the pushout. If the pushout is seen as a most general union, the pullback corresponds to the intersection. For instance, if a, b are inclusions in the category of sets, the pullback object would be $\{(x, y) \in \text{dom}(a) \times \text{dom}(b) \mid a(x) = b(y)\}$. For details, we refer to [Ehr79, EEPT06].

Definition 2.15 (pullback). Given a cospan of morphisms $\langle a, b \rangle$, a *pullback*, is a span of morphisms $\langle a^*, b^* \rangle$ such that the diagram $a \circ b^* = b \circ a^*$ commutes and the *universal property* holds: For every other cospan of morphisms $\langle a^{**}, b^{**} \rangle$ such that $a \circ b^{**} = b \circ a^{**}$, there exists a unique morphism $u: \text{dom}(a^{**}) \rightarrow \text{dom}(a^*)$ such that $a^{**} = a^* \circ u$ and $b^{**} = b^* \circ u$.



Sometimes, we refer to the whole diagram $a \circ b^* = b \circ a^*$ as the pullback and to the common domain $\text{dom}(a^*) = \text{dom}(b^*)$ as the *pullback object*.

2.2 Weak adhesive HLR categories

While graphs are a suitable structure for our research, our results should not be limited to a particular definition of graphs. In fact, our goal is to support similar structures such as Petri-nets and hypergraphs while avoiding separate investigations. Therefore, we abstract from a specific structure and use the framework of so-called weak adhesive high-level replacement (HLR) categories. These are categories with additional properties with respect to a selected class of monomorphisms. For details, we refer to [EEPT06, EHPP06, LS04, EHKP91].

Definition 2.16 (weak adhesive HLR category). A category \mathcal{C} with a morphism class $\mathcal{M} \subseteq \mathcal{A}$ is a *weak adhesive HLR category*, if the following properties hold:

- (1) \mathcal{M} is a class of monomorphisms closed under isomorphisms, composition, and decomposition, that is, for morphisms $g \circ f$: ($f \in \mathcal{M}$, $g \in \text{Iso}$ or $f \in \text{Iso}$, $g \in \mathcal{M}$) implies $g \circ f \in \mathcal{M}$; $f, g \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$; and $g \circ f \in \mathcal{M}$, $g \in \mathcal{M}$ implies $f \in \mathcal{M}$.
- (2) \mathcal{C} has pushouts and pullbacks along \mathcal{M} -morphisms, that are, pushouts and pullbacks where at least one of the given morphisms is in \mathcal{M} , and \mathcal{M} -morphisms are closed under pushouts and pullbacks, that is, given a pushout (1), $m \in \mathcal{M}$ implies $n \in \mathcal{M}$ and, given a pullback (1), $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.

$$\begin{array}{ccc} \bullet & \longrightarrow & \bullet \\ m \downarrow & (1) & \downarrow n \\ \bullet & \longrightarrow & \bullet \end{array}$$

- (3) Pushouts in \mathcal{C} along \mathcal{M} -morphisms are weak van Kampen-squares, that is, for any commutative cube in \mathcal{C} where we have the pushout in the bottom with $m \in \mathcal{M}$ and ($f \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$) and the back faces are pullbacks, it holds: the top is pushout iff the front faces are pullbacks.

$$\begin{array}{ccccc} & & \bullet & \longrightarrow & \bullet \\ & \swarrow & \vdots & \searrow & \downarrow c \\ \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet \\ \downarrow b & \swarrow m & \downarrow d & \searrow f & \downarrow \\ \bullet & \longrightarrow & \bullet & \longrightarrow & \bullet \end{array}$$

The framework of weak adhesive HLR categories is suitable to represent directed, labeled graphs.

Fact 2.17 (Ehrig et al. 2006c). The category $\langle \mathbf{Graphs}, Inj \rangle$ of graphs with class Inj of all injective graph morphisms is a weak adhesive HLR category. Further examples of weak adhesive HLR categories are the categories of hypergraphs with all injective hypergraph morphisms, place-transition nets with all injective net morphisms, and algebraic specifications with all strict injective specification morphisms.

Weak adhesive HLR categories have a number of desired properties, called HLR properties [EHKP91].

Fact 2.18 (HLR properties of weak adhesive HLR categories). Given a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$, the following HLR conditions are satisfied.

- (1) Pushouts along \mathcal{M} -morphisms are pullbacks.
- (2) Pushout-pullback decomposition. If the diagram (1)+(2) is a pushout, (2) a pullback, $w \in \mathcal{M}$ and ($l \in \mathcal{M}$ or $c \in \mathcal{M}$), then (1) and (2) are pushouts and also pullbacks.

$$\begin{array}{ccccc}
 \bullet & \xrightarrow{c} & \bullet & \longrightarrow & \bullet \\
 l \downarrow & (1) & \downarrow & (2) & \downarrow \\
 \bullet & \longrightarrow & \bullet & \xrightarrow{w} & \bullet
 \end{array}$$

- (3) Uniqueness of pushout complements for \mathcal{M} -morphisms. Given morphisms $m \in \mathcal{M}$ and $a \in \mathcal{A}$, then there is up to isomorphism at most one tuple $\langle a^*, m^* \rangle$ of morphisms such that diagram (3) is a pushout.

$$\begin{array}{ccc}
 \bullet & \xrightarrow{m} & \bullet \\
 a^* \downarrow & (3) & \downarrow a \\
 \bullet & \xrightarrow{m^*} & \bullet
 \end{array}$$

Proof. See [LS04, EEPT06]. □

Besides the properties of weak adhesive HLR categories, we require additional properties for some of our results, see Remark 2.20 below.

Assumption 2.19. Assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with

- (1) an \mathcal{M} -initial object I , that is, an object $I \in \mathcal{C}$ such that, for every object $G \in \mathcal{C}$, there exists a unique morphism $i_G: I \hookrightarrow G$, called the *initial morphism* to G , and i_G is in \mathcal{M} ,
- (2) *epi- \mathcal{M} -factorization*, that is, for every morphism there is an epi-mono-factorization with monomorphism in \mathcal{M} and this decomposition is unique up to isomorphism of the intermediate object, and
- (3) *pullback-pushout- \mathcal{M} property*, that is, for every pair of \mathcal{M} -morphisms $B \hookrightarrow D \hookleftarrow C$, the unique morphism $D' \rightarrow D$ of the pushout $\langle B \hookrightarrow D', D' \hookleftarrow C \rangle$ of the pullback $\langle B \hookleftarrow A \hookrightarrow C \rangle$ of $B \hookrightarrow D \hookleftarrow C$ is in \mathcal{M} .

$$\begin{array}{ccc}
 A & \hookrightarrow & C \\
 \downarrow & \text{(PO)} & \downarrow \\
 B & \hookrightarrow & D
 \end{array}
 \begin{array}{c}
 \nearrow \\
 \searrow \\
 \nearrow \\
 \searrow
 \end{array}
 \begin{array}{c}
 \\
 = \\
 \\
 =
 \end{array}$$

As the weak adhesive HLR category framework does not consider finiteness properties, we require additionally:

- (4) a *finite number of \mathcal{M} -matches*, that is, for every morphism l in \mathcal{M} and every object G , there exist only a finite number of morphisms $m: \text{codom}(l) \hookrightarrow G$ in \mathcal{M} such that $\langle l, m \rangle$ has a pushout complement. This property implies a *finite number of \mathcal{M} -morphisms*, that is, for every objects L, G , there exists only a finite number of morphisms $L \hookrightarrow G$ in \mathcal{M} (up to isomorphism),
- (5) a *finite number of epimorphisms* for any domain G , that is, for every object G , there is only a finite number of epimorphisms $e: G \rightarrow H$ (up to isomorphism),
- (6) a *finite number of \mathcal{M} -decompositions*, that is, the set of all decompositions $a_2 \circ a_1 = a$ of $a \in \mathcal{M}$ with a_1, a_2 in \mathcal{M} is finite (up to isomorphism), and
- (7) a *finite length of \mathcal{M} -decompositions*, that is, for every morphism m in \mathcal{M} , the length of every decomposition $m_n \circ \dots \circ m_1 = m$ consisting of non-epimorphisms m_j in \mathcal{M} ($1 \leq j \leq n$) is finite.

Remark 2.20. The first two assumptions are essential for our results: We assume the existence of an initial object in the definition of conditions, while epi- \mathcal{M} -factorization is used, for instance, in the proof of Lemma 5.4, therefore concerns the soundness of the construction of weakest liberal preconditions presented in Section 5.3. The pullback-pushout- \mathcal{M} property is required in

Section 6.2 to show the completeness of our satisfiability solver. The properties (4)-(6) ensure the effectiveness of the constructions in this thesis, while the last property is again needed for the completeness of our satisfiability solver, presented in Section 6.2.

Fact 2.21 (Properties of Graphs). The weak adhesive HLR category $\langle \mathbf{Graphs}, Inj \rangle$ with Inj the class of all injective graph morphisms satisfies Assumption 2.19.

Proof. The empty graph \emptyset is the Inj -initial object in $\langle \mathbf{Graphs}, Inj \rangle$. The epi- Inj -factorization is the epi-mono-factorization [EEPT06, Fact A.15], and the monomorphisms are exactly those morphisms that are injective [EEPT06, Fact A.14 and Fact 2.15]. Moreover, every graph is finite, the number of epimorphisms and matches is finite, and the number and length of injective decompositions is finite. \square

For the definition of programs with interface in Section 4.1, we require partial monomorphisms. Given a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$, a partial monomorphisms $p = \langle a, b \rangle$ can be seen as span of \mathcal{M} -morphisms $\langle a, b \rangle$ without the need of further axioms. There is a composition of partial monomorphisms, based on pullback construction, that is associative and for which identity morphisms are neutral elements. For details, we refer to Appendix A.

3. Conditions

In this chapter we consider nested conditions for weak adhesive HLR categories, investigate two notions of satisfiability for conditions, namely \mathcal{M} - and \mathcal{A} -satisfiability, and classify the expressiveness of nested graph conditions by showing that nested graph conditions and first-order graph formulas are expressively equivalent.

3.1 Nested conditions

In search for a graphical formalism to specify sets of objects as well as morphisms, we use the framework of weak adhesive HLR categories and consider nested conditions for high-level structures such as Petri nets, (hyper)graphs, and algebraic specifications. Syntactically, nested conditions may be seen as a tree of morphisms equipped with certain logical symbols such as first-order quantifiers and connectives. An overview on the different symbols of a condition is given in Table 3.1.

condition over P	is satisfied by a morphism with domain P , if ...
true	always
false	never
$\exists a$	there exists a commutative \mathcal{M} -morphism
$\exists(a, c)$	there exists a commutative \mathcal{M} -morphism satisfying c
$\forall(a, c)$	for all commutative \mathcal{M} -morphisms, c is satisfied
$\bigwedge_{j \in J} c_j$	all subconditions are satisfied
$\bigvee_{j \in J} c_j$	some subcondition is satisfied
$c \sqcup d$	exclusively c or d is satisfied
$c \Rightarrow d$	satisfaction of c implies satisfaction of d

Table 3.1: Conditions and their meaning

A condition true is *satisfied* by all morphisms and objects. A morphism p *satisfies* a condition $\exists(a, c)$, if there exists a morphism q in \mathcal{M} such that $q \circ a = p$ and q satisfies c .

$$\exists(\begin{array}{ccc} P & \xrightarrow{a} & C \\ & \searrow p & \swarrow q \\ & G & \end{array}, \begin{array}{c} \triangle \\ \text{c} \end{array})$$

A morphism p *satisfies* a condition $\neg c$ over C , if p does not satisfy c . A morphism p *satisfies* a condition $\bigwedge_{j \in J} c_j$ over C , if p satisfies c_j for each $j \in J$. An object G *satisfies* a condition $\exists(a, c)$, if the condition is over the initial object I and the initial morphism $\text{id}_G: I \hookrightarrow G$ satisfies the condition. The satisfaction of conditions by objects is extended onto Boolean formulas over conditions in the usual way. We write $G \models c$ to denote that the object G satisfies c and write $p \models c$ to denote morphism p satisfies c . For two conditions c, d over C , d is a *deduction* or *logical implication* of c , written $c \Rightarrow d$, if for all morphisms p in \mathcal{M} with domain C , $p \models c$ implies $p \models d$. Two conditions c and c' over C are *equivalent*, denoted by $c \equiv c'$, if, for all morphisms p with domain C , $p \models c$ iff $p \models c'$. Two conditions c and c' over C are \mathcal{M} -*equivalent*, denoted by $c \equiv_{\mathcal{M}} c'$, if, for all \mathcal{M} -morphisms p with domain C , $p \models c$ iff $p \models c'$. For a condition c over C , we write $\text{dom}(c)$ to denote its *domain* C .

Example 3.2. For graph morphisms with domain \bigcirc_1 , the graph condition $c = \exists(\bigcirc_1 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc) \vee \exists(\bigcirc_1 \hookrightarrow \bigcirc_1 \leftarrow \bigcirc)$ has the meaning “The image of the node has a proper outgoing or incoming edge”. For graphs, one may consider the universal closure $\forall(\emptyset \hookrightarrow \bigcirc_1, c)$ with the meaning “All nodes have an outgoing or incoming proper edge”.

In the context of objects, conditions (over the initial object I) are also called *constraints*.

Notation. For every morphism $a: P \rightarrow C$ in a condition, we just depict the codomain C , if the domain P can be unambiguously inferred. This is the case for constraints, which are by definition conditions over I . For instance, the constraint $\forall(\emptyset \hookrightarrow \bigcirc_1, \exists(\bigcirc_1 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc_2))$ with the meaning “Every node has an outgoing edge to another distinct node” can be represented by $\forall(\bigcirc_1, \exists(\bigcirc_1 \rightarrow \bigcirc_2))$. Note that many of the automatically generated presentations of conditions are in conjunctive normal form (which will be of importance in Section 6.3) and therefore may contain the connectives \bigvee or \bigwedge over singletons, for instance $\bigvee \forall(\bigcirc_1, \bigvee \exists(\bigcirc_1 \rightarrow \bigcirc_2))$.

Example 3.3 (access control conditions). Consider the access control graphs introduced in Example 2.4. Conditions allow to formulate statements on the graphs of the access control and can be combined to form more complex statements. The following conditions are over the empty graph:

$\exists(\text{user} \rightarrow \text{system})$	A user is logged into a system.
$\exists(\text{user} \rightarrow \text{system})$	A user has an access right to the system.
$\exists(\text{user} \rightarrow \text{system} \leftarrow \text{system})$	There is a user logged into a system with an access right.
$\exists(\text{session} \rightarrow \text{system})$	A session is proposed
$\exists(\text{session} \leftarrow \text{system})$	A session is established
$\forall(\text{session}, \exists(\text{session} \rightarrow \text{system}) \vee \exists(\text{session} \leftarrow \text{system}))$	Every session is either proposed or established
$\neg \exists(\text{user} \rightarrow \text{system} \leftarrow \text{user})$	No session is shared between two users
$\forall(\text{session}, \exists(\text{session} \leftarrow \text{user}))$	Every session is associated to a user
$\forall(\text{session}, \exists(\text{session} \leftarrow \text{user}) \wedge \neg \exists(\text{user} \rightarrow \text{session} \leftarrow \text{user}))$	Every session is associated to exactly one user
$\neg \exists(\text{user} \leftrightarrow \text{system})$	There exist at most one access right for every user and every computer.
$\forall(\text{user} \rightarrow \text{system} \leftarrow \text{system}, \exists(\text{user} \rightarrow \text{system} \leftarrow \text{system}))$	Every user that is logged into a system, has an access right.

In our examples, we will focus on the last condition, which we will call *secure*.

Remark 3.4 (history of conditions). Conditions for graph morphisms of the form $\neg \exists a$ were first introduced in [HHT96] as *negative application conditions* in the context of graph transformation rules. Shortly thereafter, conditions for graphs of the form $\forall(I \hookrightarrow P, \exists(P \rightarrow C))$ were introduced as *graph consistency constraints* in [HW95]. The concepts of graph constraints

and application conditions of [HW95] were lifted to weak adhesive HLR categories in [EEHP06, EEHP04], and, unified and generalized to nested conditions in [Ren04a] for edge-labeled graphs (without parallel edges) and in [Pen04, HP05] for weak adhesive HLR categories.

Since [HP09], we have a common notion of equivalence for morphisms and objects. We use the following fact in Section 6.2 by defining transformations for conditions that hold with respect to objects as well as morphisms.

Fact 3.5 (equivalence). For conditions c, c' over the initial object I ,

$$\begin{aligned} c \equiv c' & \quad \text{iff} \quad c \equiv_{\mathcal{M}} c' \\ c \equiv_{\mathcal{M}} c' & \quad \text{iff} \quad (\text{for all objects } G, G \models c \Leftrightarrow G \models c'). \end{aligned}$$

Proof. First statement.

Only if. Assume $c \equiv c'$. For all morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$. As every \mathcal{M} -morphism is a morphism, we have for all \mathcal{M} -morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$. Therefore, $c \equiv_{\mathcal{M}} c'$.

If. Assume $c \equiv_{\mathcal{M}} c'$. For all \mathcal{M} -morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$. As every morphism with domain I is in \mathcal{M} , we have for all morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$. Therefore, $c \equiv c'$.

Second statement.

Only if. Assume, $c \equiv_{\mathcal{M}} c'$. For all morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$. In particular, this means for objects G , $i_G \models c \Leftrightarrow i_G \models c'$ where i_G is the initial \mathcal{M} -morphisms to G . By Definition 3.1, we conclude for every object G , $G \models c \Leftrightarrow G \models c'$.

If. Conversely, if for every object G , $G \models c \Leftrightarrow G \models c'$, then $i_G \models c \Leftrightarrow i_G \models c'$ where i_G is the initial \mathcal{M} -morphism to G . Let $p: I \hookrightarrow G$ be any morphism. By the \mathcal{M} -initiality of I , we know that $p = i_G$ in \mathcal{M} , therefore we conclude for all \mathcal{M} -morphisms p with domain I , $p \models c \Leftrightarrow p \models c'$ and $c \equiv_{\mathcal{M}} c'$. \square

Conditions are allowed to consist of morphisms not in \mathcal{M} . In context of arbitrary morphisms, such conditions may be useful to specify the identification or non-identification of elements. However, for conditions over I , the use of morphisms not in \mathcal{M} does not increase the expressiveness. For certain constructions such as **SeekSat** in Section 6.2, we assume conditions to be in *\mathcal{M} -normal form*, that is, if for all subconditions $\exists(a, c)$ the morphism a is in \mathcal{M} . We show that every condition over I can be transformed into *\mathcal{M} -normal form*.

Definition 3.6 (\mathcal{M} -normal form). A condition is in *\mathcal{M} -normal form* (\mathcal{MNF}), if for all subconditions of the form $\exists(a, c)$, the morphism a is in \mathcal{M} .

Fact 3.7 (\mathcal{M} -normal form). For every condition c over \mathbf{I} , there is a condition $\mathcal{MNF}(c)$ in \mathcal{M} -normal form such that $c \equiv \mathcal{MNF}(c)$.

Construction. Define $\mathcal{MNF}(\text{true}) = \text{true}$ and

$$\mathcal{MNF}(\exists(a, c')) = \begin{cases} \text{false} & \text{if } a \notin \mathcal{M} \\ \exists(a, \mathcal{MNF}(c')) & \text{otherwise.} \end{cases}$$

For Boolean formulas over conditions, the transformation is extended in the usual way, that is, $\mathcal{MNF}(\neg c') = \neg \mathcal{MNF}(c')$ and $\mathcal{MNF}(\bigwedge_{j \in J} c_j) = \bigwedge_{j \in J} \mathcal{MNF}(c_j)$.

Proof. Obviously, for every condition c , $\mathcal{MNF}(c)$ is in \mathcal{M} -normal form. By induction over the structure of conditions, we show for every $p \in \mathcal{M}$, $p \models c$ iff $p \models \mathcal{MNF}(c)$.

Basis. For $c = \text{true}$, we have $c = \text{true} = \mathcal{MNF}(\text{true}) = \mathcal{MNF}(c)$.

Hypothesis. Assume, for every $p \in \mathcal{M}$, $p \models c'$ iff $p \models \mathcal{MNF}(c')$ and $p \models c_j$ iff $p \models \mathcal{MNF}(c_j)$ for every $j \in J$.

Step. Let $c = \exists(a, c')$. If $a \in \mathcal{M}$, then we have, for every $p \in \mathcal{M}$, $p \models \mathcal{MNF}(c) = \mathcal{MNF}(\exists(a, c')) = \exists(a, \mathcal{MNF}(c'))$ iff $p \models \exists(a, c') = c$ by the definition of \mathcal{MNF} and the induction hypothesis. Otherwise $a \notin \mathcal{M}$, and $\mathcal{MNF}(\exists(a, c')) = \text{false}$. We show, for every $p \in \mathcal{M}$, $p \models \exists(a, c')$ iff $p \models \text{false}$.

Only if. Assume morphism p in \mathcal{M} satisfies $\exists(a, c')$. Then there is a morphism q in \mathcal{M} with $p = q \circ a$. However, p, q in \mathcal{M} and \mathcal{M} closed under decomposition implies $a \in \mathcal{M}$, contradiction.

If. No morphism satisfies false, therefore for every morphism p , $p \models \text{false}$ implies $p \models \exists(a, c')$. For Boolean formulas over conditions, the statement follows directly from the definitions and the inductive hypothesis. For $c = \neg c'$, we have, for every $p \in \mathcal{M}$, $p \models \mathcal{MNF}(c) = \mathcal{MNF}(\neg c') = \neg \mathcal{MNF}(c')$ iff $p \models \neg c' = c$. For $c = \bigwedge_{j \in J} c_j$, we have, for every $p \in \mathcal{M}$, $p \models \mathcal{MNF}(c) = \mathcal{MNF}(\bigwedge_{j \in J} c_j) = \bigwedge_{j \in J} \mathcal{MNF}(c_j)$ iff $p \models \bigwedge_{j \in J} c_j = c$. \square

3.2 Comparison of \mathcal{M} - and \mathcal{A} -satisfiability

In this section, we investigate the different satisfiability notions for conditions. As it turns out in Section 3.3, the transformations Msat and Asat are an important step in the conversion of graph conditions into first-order graph formulas and vice versa. However, we require this conversion mainly to evaluate our algorithms on condition, therefore we suggest to skip this and the next section at first and to read it when needed.

Remark 3.8 (\mathcal{M} -satisfiability). The satisfaction of a condition is established by the presence and absence of certain morphisms from the objects within the condition to the tested object. The presented satisfiability notion restricts these morphisms to the class \mathcal{M} : for **Graphs**, no identification of nodes and edges is allowed. Hence, explicit counting such as the existence/non-existence of n nodes or n edges is easily expressible. We speak of \mathcal{M} -satisfiability and \mathcal{M} -satisfiable conditions.

Beside \mathcal{M} -satisfiability one may consider \mathcal{A} -satisfiability and \mathcal{A} -satisfiable conditions, where \mathcal{A} denotes the class of all morphisms. The definition of \mathcal{A} -satisfiability is obtained from the one of \mathcal{M} -satisfiability by replacing all occurrences of \mathcal{M} by \mathcal{A} or by deleting all occurrences of “in \mathcal{M} ”, respectively.

Definition 3.9 (\mathcal{A} -satisfiability). Every object and morphism \mathcal{A} -satisfies true. An object G \mathcal{A} -satisfies a condition $\exists(a, c)$, if the condition is over the initial object I and the initial morphism $i_G: I \hookrightarrow G$ \mathcal{A} -satisfies the condition. A morphism p \mathcal{A} -satisfies a condition $\exists(a, c)$, if there exists a morphism q such that $q \circ a = p$ and q \mathcal{A} -satisfies c . The \mathcal{A} -satisfaction of conditions by objects and morphisms is extended onto Boolean formulas over conditions in the usual way. We write $G \models_{\mathcal{A}} c$ resp. $p \models_{\mathcal{A}} c$ to denote that the object G resp. the morphism p \mathcal{A} -satisfies c . Two conditions c and c' are \mathcal{A} -equivalent, denoted by $c \equiv_{\mathcal{A}} c'$, if, for all morphisms p , $p \models_{\mathcal{A}} c$ iff $p \models_{\mathcal{A}} c'$.

Remark 3.10 (\mathcal{A} -satisfiability). \mathcal{A} -satisfiability allows nodes or edges of a condition to be identified and is closely related to the satisfiability of first-order formulas as indicated in Section 3.3 for the case of directed, labeled graphs. We will see that, under reasonable assumptions, \mathcal{A} - and \mathcal{M} -satisfiability are expressively equivalent. Unless explicitly stated, theorems concern \mathcal{M} -satisfiable conditions.

Example 3.11 (\mathcal{A} -satisfiability). The meaning of the graph condition $\exists(\bigcirc_1 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc)$ for graph morphisms with domain \bigcirc_1 with respect to \mathcal{A} -satisfiability is: “There exists an outgoing edge, that is, either a proper edge or a loop”.

In the following we prove that \mathcal{A} -satisfiability and \mathcal{M} -satisfiability are expressively equivalent. First, there is a transformation from \mathcal{A} - to \mathcal{M} -satisfiability. In case pushouts for arbitrary pairs of morphisms exist, e.g. for the category **Graphs** [EEPT06, Fact A.19], this transformation of conditions has some similarities to the transformation A in Section 5.2. In the case that the existence of pushouts cannot be guaranteed, one can resort to a modified transformation which requires an \mathcal{M} -initial object and makes use of the existence of pushouts along \mathcal{M} -morphisms.

Theorem 3.12 (from \mathcal{A} - to \mathcal{M} -satisfiability). *Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with \mathcal{M} -initial object and epi- \mathcal{M} -factorization. There is a transformation Msat such that, for every condition c and for every morphism p , $p \models \text{Msat}(c) \Leftrightarrow p \models_{\mathcal{A}} c$, and for every condition c over \mathbf{I} and for every object G , $G \models \text{Msat}(c) \Leftrightarrow G \models_{\mathcal{A}} c$.*

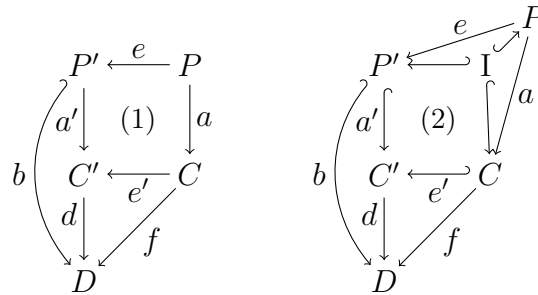
\mathcal{A} -satisfiable conditions allow elements to be identified. The idea of the following construction is to consider a disjunction of \mathcal{M} -satisfiable conditions, one for each possible level of identification. The construction of $\text{Msat}(c)$ with single parameter of type condition is defined by the auxiliary transformation $\text{Msat}(e, c)$ where the first parameter is of type epimorphism and the second parameter is of type condition.

Construction. For a condition c over P , $\text{Msat}(c) = \bigvee_{e \in E'} \exists(e, \text{Msat}(e, c))$ where the set E' ranges over all epimorphisms with domain P . For every epimorphism e , the transformation $\text{Msat}(e, c)$ is defined inductively by $\text{Msat}(e, \text{true}) = \text{true}$ and

$$\text{Msat}(e, \exists(a, c')) = \bigvee_{d \in E} \exists(b, \text{Msat}(f, c'))$$

where, in the case that $\langle \mathcal{C}, \mathcal{M} \rangle$ has pushouts, (1) is the pushout of the morphisms a and e leading to morphisms $a': P' \rightarrow C'$ and $e': C \rightarrow C'$ and the set E ranges over all epimorphisms d with domain C' such that $b = d \circ a'$ is in \mathcal{M} and $f = d \circ e'$.

In the case that $\langle \mathcal{C}, \mathcal{M} \rangle$ has pushouts only along \mathcal{M} -morphisms, construct the pushout (2) of the initial \mathcal{M} -morphisms $i_C: \mathbf{I} \hookrightarrow C$ and $i_{P'}: \mathbf{I} \hookrightarrow P'$ leading to morphisms $a': P' \rightarrow C'$ and $e': C \rightarrow C'$ and the set E ranges over all epimorphisms d with domain C' such that $b = d \circ a'$ is in \mathcal{M} , $f = d \circ e'$ and $b \circ e = f \circ a$.



For Boolean formulas over conditions, $\text{Msat}(_)$ and $\text{Msat}(_, _)$ are extended in the usual way.

Example 3.13. The meaning of condition $c = \exists(\bigcirc_1 \bigcirc_2 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc_2)$ for graph morphisms w.r.t. \mathcal{A} -satisfiability is “There exists an edge from the image of

node 1 to the image of node 2". $\text{Msat}(c)$ is constructed as follows:

$$\begin{aligned} \text{Msat}(c) &= \vee_{e \in E} \exists(e, \text{Msat}(e, c)) = \exists(\text{id}, \text{Msat}(\text{id}, c)) \vee \exists(f, \text{Msat}(f, c)) \\ &= \exists(\text{id}, \exists(\bigcirc_1 \bigcirc_2 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc_2)) \vee \exists(f, \exists(\bigcirc_1 \hookrightarrow \bigcirc_1 \rightarrow)) \\ &\equiv \exists(\bigcirc_1 \bigcirc_2 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc_2) \vee \exists(\bigcirc_1 \bigcirc_2 \rightarrow \bigcirc_{1=2}) \end{aligned}$$

where $f: \bigcirc_1 \bigcirc_2 \rightarrow \bigcirc_{1=2}$. The meaning of $\text{Msat}(c)$ in case of \mathcal{M} -satisfiability is “There is a proper edge from the image of node 1 to a distinct image of node 2 or both images are identical and there is a loop”.

Remark 3.14. The definition of a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$ requires the existence of pushouts along \mathcal{M} -morphisms, but does not guarantee the existence of pushouts for arbitrary morphisms. In the construction of Theorem 3.12, both e and a may not be \mathcal{M} -morphisms. Especially, one may not assume that a is in \mathcal{M} (i.e. part of a condition in \mathcal{M} -normal form), because in the context of \mathcal{A} -satisfiability, the requirement to use only \mathcal{M} -morphisms would restrict the expressiveness of conditions. For instance, a property like “Two nodes are distinct” can only be expressed by an \mathcal{A} -satisfiable condition by using a morphisms not in \mathcal{M} , e.g. $\neg \exists((1) \otimes (2) (=))$. We include the more general construction for the sake of a more general result, but prefer to use the simpler and more efficient construction for the category **Graphs**.

Before we prove Theorem 3.12, we prove a property for the auxiliary transformation $\text{Msat}(e, c)$.

Lemma 3.15 ($\text{Msat}(e, c)$). Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with \mathcal{M} -initial object and epi- \mathcal{M} -factorization. For every condition c over P , every epimorphism $e: P \rightarrow P'$, and every morphism $p': P' \hookrightarrow G$ in \mathcal{M} , $p' \models \text{Msat}(e, c) \Leftrightarrow p' \circ e \models_{\mathcal{A}} c$.

$$\begin{array}{ccccc} & & \text{Msat}(e, c) & & \\ & \triangleleft & & & \triangleleft \\ & P' & \xleftarrow{e} & P & \\ & \searrow p' & \searrow = & \swarrow p & \\ & & G & & \end{array}$$

Proof. By structural induction.

Basis. For $c = \text{true}$, we have $c = \text{true} = \text{Msat}(e, \text{true}) = \text{Msat}(e, c)$.

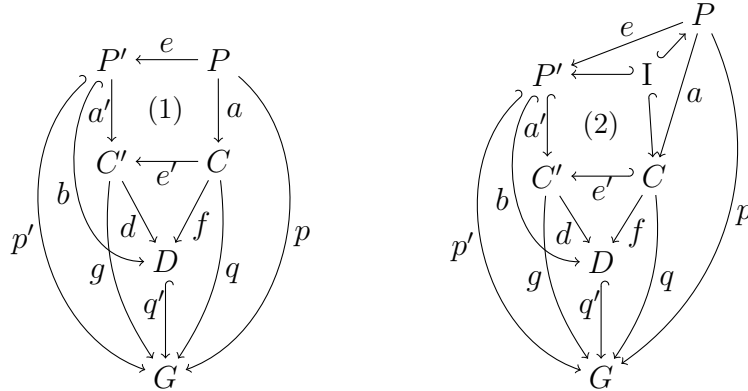
Hypothesis. Assume the statement holds for condition c' .

Step. For $c = \exists(a, c')$, we distinguish two cases:

Case 1. $\langle \mathcal{C}, \mathcal{M} \rangle$ has pushouts of arbitrary morphisms. **Only if.** Let $p' \models \text{Msat}(e, \exists(a, c')) = \vee_{d \in E} \exists(b, \text{Msat}(f, c'))$. There is some epimorphism

$d: C' \rightarrow D$ with $b = d \circ a'$ in \mathcal{M} and $f = d \circ e'$ such that $p' \models \exists(b, \text{Msat}(f, c'))$. By definition of \mathcal{M} -satisfiability, there is some $q': D \hookrightarrow G$ in \mathcal{M} such that $q' \circ b = p'$ and $q' \models \text{Msat}(f, c')$. Define now $q = q' \circ f$. Together with $q' \circ b = p'$, $b = d \circ a'$, $a' \circ e = e' \circ a$, $f = d \circ e'$, we observe $q \circ a = p$ ($p \models \exists a$). By inductive hypothesis, $q \models_{\mathcal{A}} c'$, therefore $p \models_{\mathcal{A}} \exists(a, c')$. **If.** Let $p \models_{\mathcal{A}} \exists(a, c')$. By definition of \mathcal{A} -satisfiability, there is some $q: C \rightarrow G$ such that $q \circ a = p$ and $q \models_{\mathcal{A}} c'$. Let (1) be the pushout of $a: P \rightarrow C$ and $e: P \rightarrow P'$. By the universal property of pushouts, there is some $g: C' \rightarrow G$ with $g \circ a' = p'$ and $g \circ e' = q$. Let $g = q' \circ d$ be an epi- \mathcal{M} -factorization of g with epimorphism d and monomorphism q' in \mathcal{M} , $b = d \circ a'$, and $f = d \circ e'$. Since p' and q' are in \mathcal{M} , $q' \circ b = p'$, and \mathcal{M} is closed under decomposition, the morphism b is in \mathcal{M} and d is in E . As $p' = g \circ a'$, $g = q' \circ d$ and $b = d \circ a'$, we observe $p' = q' \circ b$ ($p' \models \exists b$). By inductive hypothesis, $q' \models \text{Msat}(f, c')$, therefore $p' \models \bigvee_{d \in E} \exists(b, \text{Msat}(f, c')) = \text{Msat}(e, \exists(a, c'))$.

Case 2. $\langle \mathcal{C}, \mathcal{M} \rangle$ has only pushouts along \mathcal{M} -morphisms. **Only if.** As above, by using the morphism $f: C \rightarrow D$ and the inductive hypothesis. **If.** Let $p \models_{\mathcal{A}} \exists(a, c')$. By definition of \mathcal{A} -satisfiability, there is some $q: C \rightarrow G$ such that $q \circ a = p$ and $q \models_{\mathcal{A}} c'$. Since $\langle \mathcal{C}, \mathcal{M} \rangle$ has pushouts along \mathcal{M} -morphisms, we can construct the pushout (2) of the \mathcal{M} -morphisms $i_C: I \hookrightarrow C$ and $i_{P'}: I \hookrightarrow P'$ leading to morphisms $a': P' \rightarrow C'$ and $e': C \rightarrow C'$. By the universal property of pushouts, there is some $g: C' \rightarrow G$ with $g \circ a' = p'$ and $g \circ e' = q$. Let $g = q' \circ d$ be an epi- \mathcal{M} -factorization of g with epimorphism d and monomorphism q' in \mathcal{M} , $b = d \circ a'$, and $f = d \circ e'$. Since p' and q' are in \mathcal{M} , $q' \circ b = p'$, and \mathcal{M} is closed under decomposition, the morphism b is in \mathcal{M} and d is in E . It turns out that $q' \circ b \circ e = q' \circ f \circ a$ and, by the monomorphism property of q' , $b \circ e = f \circ a$. As b is in \mathcal{M} and $f \circ a = b \circ e$, the tuple $\langle D, b, f \rangle$ belongs to the construction. As $p' = g \circ a'$, $g = q' \circ d$ and $b = d \circ a'$, we observe $p' = q' \circ b$ ($p' \models \bigvee_{d \in E} \exists b$). By inductive hypothesis, $q' \models \text{Msat}(f, c')$, therefore $p' \models \bigvee_{d \in E} \exists(b, \text{Msat}(f, c')) = \text{Msat}(e, \exists(a, c'))$.



For Boolean formulas over conditions, the statement follows from the definitions and the inductive hypothesis. Consequently, the statement holds for all conditions. \square

Theorem 3.12 follows directly from Lemma 3.15.

Proof of Theorem 3.12. Let $p: P \rightarrow G$ be a morphism and $p = p' \circ e$ an epi- \mathcal{M} -factorization of p with epimorphism e and monomorphism p' in \mathcal{M} . By Lemma 3.15, the definitions of \mathcal{M} -satisfiability (\models) and $\text{Msat}(c)$, and the uniqueness of epi- \mathcal{M} -factorizations up to isomorphism, we have: $p \models_{\mathcal{A}} c \Leftrightarrow p' \models \text{Msat}(e, c) \Leftrightarrow p \models \exists(e, \text{Msat}(e, c)) \Leftrightarrow p \models \bigvee_{e \in E'} \exists(e, \text{Msat}(e, c)) = \text{Msat}(c)$. Fact 3.5 lifts the result to objects and conditions over \mathbf{I} . \square

If \mathcal{M} is *strictly* closed under decomposition, that is, $g \circ f \in \mathcal{M}$ implies $f \in \mathcal{M}$, there is also a transformation from \mathcal{M} - to \mathcal{A} -satisfiability. As the class Mon of all monomorphisms of a category is always strictly closed under decomposition, a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$ inherits this property, if \mathcal{M} coincides with Mon , as in the case of **Graphs**.

Theorem 3.16 (from \mathcal{M} - to \mathcal{A} -satisfiability). *Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with epi- \mathcal{M} -factorization and \mathcal{M} strictly closed under decomposition. There is a transformation Asat on conditions such that, for every condition c and for every morphism p , $p \models_{\mathcal{A}} \text{Asat}(c) \Leftrightarrow p \models c$, and for every condition c over \mathbf{I} and for every object G , $G \models_{\mathcal{A}} \text{Asat}(c) \Leftrightarrow G \models c$.*

\mathcal{A} -satisfiable conditions allow elements to be identified. The idea of the construction is to prevent identification by expressing the property “the morphism is in \mathcal{M} ” by subconditions.

Construction. For a morphism $a: P \rightarrow C$ and a condition c over C , $\text{Asat}(\text{true}) = \text{true}$ and $\text{Asat}(\exists(a, c')) = \exists(a, \text{inM}_C \wedge \text{Asat}(c'))$ where $\text{inM}_C = \bigwedge_{e \in E} \neg \exists e$ is a condition over C , the conjunction ranges over all epimorphisms $e: C \rightarrow C'$ not in \mathcal{M} . For Boolean formulas over conditions, the transformation is extended in the usual way.

Example 3.17. The condition $c = \exists(\textcircled{1} \textcircled{2} \hookrightarrow \textcircled{1} \rightarrow \textcircled{2})$ meaning for graph morphisms w.r.t. \mathcal{M} -satisfiability “There exists a proper edge from the image of 1 to the distinct image of 2” is transformed into the condition $\text{Asat}(c) = \exists(\textcircled{1} \textcircled{2} \hookrightarrow \textcircled{1} \rightarrow \textcircled{2}, \neg \exists(\textcircled{1} \rightarrow \textcircled{2} \rightarrow \textcircled{1=2}))$ meaning w.r.t. \mathcal{A} -satisfiability “There exists an edge from the image of 1 to the image of 2 and the images are distinct”.

Example 3.18 (access control condition). Consider the access control condition

$$secure = \forall \left(\begin{array}{c} \text{Diagram 1: } \text{Person}_1 \rightarrow \text{Phone}_2 \leftarrow \text{Phone}_3 \\ \text{Diagram 2: } \text{Person}_1 \rightarrow \text{Phone}_2 \leftarrow \text{Phone}_3 \end{array} \right).$$

Due to the fact that every node has a distinct label, no inequations are generated, that is, $\text{inM}(I \hookrightarrow \text{Diagram 1}) = \text{true}$ and $\text{inM}(\text{Diagram 1} \hookrightarrow \text{Diagram 2}) = \text{true}$. In this case, $\text{Asat}(secure) = secure$.

Proof of Theorem 3.16. First, we prove that for every morphism $q: C \rightarrow G$, $q \models_{\mathcal{A}} \text{inM}_C$ iff q is in \mathcal{M} . Proof by contraposition. **Only if.** Assume $q \models_{\mathcal{A}} \text{inM}_C$, but q not in \mathcal{M} . Consider an epi- \mathcal{M} -factorization $q = q' \circ e$ of q with epimorphism e and monomorphism q' in \mathcal{M} . Then e is not in \mathcal{M} , $q \models_{\mathcal{A}} \exists e$ and $q \not\models_{\mathcal{A}} \text{inM}_C$. Otherwise, by closure of \mathcal{M} under compositions, e and q' in \mathcal{M} would imply q in \mathcal{M} , contradiction. **If.** Assume $q \not\models_{\mathcal{A}} \text{inM}_C$ and q in \mathcal{M} . Then $q \models_{\mathcal{A}} \exists e$ for some epimorphism $e: C \rightarrow C'$ not in \mathcal{M} . Then there is some $q': C' \rightarrow G$ such that $q' \circ e = q$. Then q is not in \mathcal{M} . Otherwise, by the strict closure of \mathcal{M} under decomposition, q in \mathcal{M} would imply e in \mathcal{M} , contradiction.

$$\exists \left(\begin{array}{ccc} C & \xrightarrow{e} & C' \\ & \searrow q & \swarrow q' \\ & G & \end{array} \right)$$

The statement for Asat is shown by structural induction:

Basis. For $c = \text{true}$, we have $c = \text{true} = \text{Asat}(\text{true}) = \text{Asat}(c)$.

Hypothesis. Assume the statement holds for condition c' .

Step. For $c = \exists(a, c')$, we have the following. **If.** Let $p \models \exists(a, c')$. Then there is a morphism $q: C \hookrightarrow G$ in \mathcal{M} such that $q \circ a = p$ and $q \models c'$. By the inductive hypothesis and the application condition inM_C being equivalent to “morphism is in \mathcal{M} ”, $q \models_{\mathcal{A}} \text{inM}_C$ and $q \models_{\mathcal{A}} \text{Asat}(c')$. Consequently, $p \models_{\mathcal{A}} \exists(a, \text{inM}_C \wedge \text{Asat}(c')) = \text{Asat}(\exists(a, c'))$. **Only if.** Let $p \models_{\mathcal{A}} \text{Asat}(\exists(a, c')) = \exists(a, \text{inM}_C \wedge \text{Asat}(c'))$. Then there is some $q: C \rightarrow G$ such that $q \circ a = p$, $q \models_{\mathcal{A}} \text{inM}_C$, and $q \models_{\mathcal{A}} \text{Asat}(c')$. The property of inM_C yields $q \in \mathcal{M}$, and by the inductive hypothesis, $q \models c'$. Together, $p \models \exists(a, c')$. For Boolean formulas over conditions, the statement follows from the definitions and the inductive hypothesis. This completes the inductive proof. Fact 3.5 lifts the result to objects and conditions over I . \square

Fact 3.19. The construction above inserts the requirement “in \mathcal{M} ” behind every morphism of the condition. However, it suffices to express that newly

introduced elements are distinct. An optimized transformation Asat can be defined as follows:

$$\begin{aligned}\text{Asat}(\text{true}) &= \text{Asat}'(\text{true}) = \text{true} \\ \text{Asat}(\exists(a, c')) &= \exists(a, \text{inM}_C \wedge \text{Asat}'(c')) \\ \text{Asat}'(\exists(a, c')) &= \exists(a, \text{inM}_a \wedge \text{Asat}'(c'))\end{aligned}$$

where $\text{inM}_a = \bigwedge_{e \in E'} \neg \exists e$ is a condition over C and E' is constructed as follows: Let E be set of all epimorphisms $e: C \rightarrow C'$ not in \mathcal{M} such that $e \circ a$ in \mathcal{M} . The set E' consists of all epimorphisms $e \in E$ such that there is no (non-isomorphic) decomposition of e into epimorphisms $e'' \circ e' = e$ such that $e' \in E$ and e'' not in \mathcal{M} . For all morphisms p in \mathcal{M} and q with $p = q \circ a$, $q \models_{\mathcal{A}} \text{inM}_a$ iff q in \mathcal{M} . Note that for the initial \mathcal{M} -morphism $i_C: I \hookrightarrow C$, $\text{inM}_{i_C} = \text{inM}_C$.

Proof. We show for all morphisms p in \mathcal{M} and q with $p = q \circ a$, $q \models_{\mathcal{A}} \text{inM}_a$ iff q in \mathcal{M} . By contraposition. Let p be in \mathcal{M} and $p = q \circ a$.

$$\begin{array}{ccccc} & & e \circ a & & \\ & \curvearrowright & & \curvearrowleft & \\ P & \xrightarrow{a} & C & \xrightarrow{e} & C' \\ & \searrow & \swarrow q & \searrow & \\ & p & G & q' & \end{array}$$

Only if. Assume $q \models_{\mathcal{A}} \text{inM}_a$, but q not in \mathcal{M} . As in the proof of Theorem 3.16, consider an epi- \mathcal{M} -factorization $q = q' \circ e$ of q with epimorphism e and monomorphism q' in \mathcal{M} . Then e is not in \mathcal{M} . Otherwise, by closure of \mathcal{M} under compositions, e and q' in \mathcal{M} would imply q in \mathcal{M} . Moreover, $e \circ a$ in \mathcal{M} . Otherwise, by q' in \mathcal{M} and the closure of \mathcal{M} under decompositions, $e \circ a$ not in \mathcal{M} would imply p not in \mathcal{M} . Now e is an epimorphism not in \mathcal{M} and $e \circ a$ in \mathcal{M} , therefore $e \in E$. If there is now a decomposition $e'' \circ e'$ of e with $e' \in E$ and e'' not in \mathcal{M} , let $e = e'$ from now and repeat this argument. If there is no such decomposition, we have $e \in E'$, and we conclude $q \models_{\mathcal{A}} \exists e$ and $q \not\models_{\mathcal{A}} \text{inM}_a$, contradiction. **If.** Assume $q \not\models_{\mathcal{A}} \text{inM}_a$ and q in \mathcal{M} . As in the proof of Theorem 3.16, $q \not\models_{\mathcal{A}} \text{inM}_a$ implies q not in \mathcal{M} , contradiction. \square

By Theorems 3.12 and 3.16, we obtain the following corollary.

Corollary 3.20 (equivalence). For weak adhesive HLR categories with \mathcal{M} -initial object, epi- \mathcal{M} -factorization, and \mathcal{M} strictly closed under decomposition, \mathcal{A} -satisfiability and \mathcal{M} -satisfiability are expressively equivalent.

conditions	$\xrightarrow{\text{Msat}}$	conditions
\mathcal{A} -satisfiability	$\xleftarrow{\text{Asat}}$	\mathcal{M} -satisfiability

3.3 Comparison of graph conditions and graph formulas

We are interested in classifying nested conditions, that is, we want to classify the kind of graph properties that can be expressed by graph conditions, and we want to know if it is decidable whether or not a graph condition is satisfiable at all (satisfiability problem), or whether or not a graph condition is always valid (tautology problem). To this effect, we compare graph conditions and first-order formulas on graphs [Cou90, Cou97] and prove that the concepts are expressively equivalent. Similar to [Ren04a], we show that there are transformations from \mathcal{A} -satisfiable graph conditions into equivalent graph formulas and vice versa. However, we consider graphs with parallel edges, that is, multiple, distinguishable edges which may have the same label. Together with the transformations between \mathcal{M} - and \mathcal{A} -satisfiability, we yield the wanted result. As an additional result, the transformations enable the use of existing first-order proof tools to tackle the tautology and satisfiability problem of graph conditions.

For simplicity, we consider graphs with a common labeling function for nodes and edges, while maintaining the disjointness of the node and edge alphabet, that is a single labeling function $l_G: V_G + E_G \rightarrow C_V \cup C_E$ with $l_G(V_G) \subseteq C_V$ and $l_G(E_G) \subseteq C_E$, where $+$ denotes the disjoint union. Note that all considerations can be done for graphs with separate labeling functions as well.

The definition of first-order graph formulas is similar to [Cou90, Cou97]: We allow quantification over nodes and edges, consider a tertiary incidence relation and introduce a unary predicate for each label. For a fixed, finite label alphabet $C = \langle C_V, C_E \rangle$ with $C_V \cap C_E = \emptyset$, the induced signature $\Sigma = (\emptyset, \{\text{lab}_b \mid b \in C\} \cup \{\text{inc}, =\})$ contains a unary predicate symbol lab_b for every label b , a tertiary predicate symbol inc and a binary predicate symbol $=$.

Definition 3.21 (first-order graph formulas). Let Var be an infinite, countable set of variables. The set of all (*first-order graph*) *formulas* over Σ is inductively defined: For $b \in C$ and $x, y, z \in \text{Var}$, $\text{lab}_b(x)$, $\text{inc}(x, y, z)$ and $x = y$ are formulas over Σ . For formulas F, F_j ($j \in J$) over Σ and $x \in \text{Var}$, true , $\neg F$, $\bigwedge_{j \in J} F_j$, and $\exists x F$ are formulas over Σ . Additionally, false abbreviates $\neg \text{true}$, $\bigvee_{j \in J} F_j$ abbreviates $\neg \bigwedge_{j \in J} \neg F_j$, $F \Rightarrow G$ abbreviates $\neg F \vee G$, $\forall x F$ abbreviates $\neg \exists x \neg F$, $\text{edge}(x)$ abbreviates $\exists y \exists z \text{inc}(x, y, z)$, and $\text{node}(x)$ abbreviates $\neg \text{edge}(x)$. For a formula F , $\text{Free}(F)$ denotes the set of all *free* variables of F . A formula is *closed*, if $\text{Free}(F) = \emptyset$, that is, if F does not contain free variables.

The semantics of graph formulas are given in terms of a domain of values and an interpretation of the (non-logical) symbols of Σ .

Definition 3.22 (semantics of graph formulas). For a non-empty graph G , let (D_G, I_G) be the induced Σ -structure consisting of a non-empty domain $D_G = V_G + E_G$ and the interpretation I_G of the predicate symbols with $I_G(\text{lab}_b)(d) = \text{true}$ iff $l_G(d) = b$, $I_G(\text{inc})(e, u, v) = \text{true}$ iff $e \in E_G$, $s_G(e) = u$ and $t_G(e) = v$, and $I_G(=)(d, d') = \text{true}$ iff $d = d'$. The *semantics* $G[F](\sigma)$ of a formula F over Σ in the graph G under the assignment $\sigma: \text{Var} \rightarrow D_G$ is inductively defined by:

$$G[p(x_1, \dots, x_n)](\sigma) = I_G(p)(\sigma(x_1), \dots, \sigma(x_n)) \text{ for an } n\text{-ary predicate } p.$$

$$G[\exists x F](\sigma) = \text{true} \text{ iff there exists } d \in D_G \text{ such that } G[F](\sigma\{x/d\}) = \text{true}, \\ \text{where } \sigma\{x/d\} \text{ is the modified assignment with } \sigma\{x/d\}(x) = d \text{ and} \\ \sigma\{x/d\}(y) = \sigma(y) \text{ otherwise.}$$

The semantics is extended to the operators true , \neg and \wedge in the usual way.

A graph G *satisfies* a formula F , denoted by $G \models F$, iff for all assignments $\sigma: \text{Var} \rightarrow D_G$, $G[F](\sigma) = \text{true}$.

Example 3.23. The first-order graph formula

$$F = \text{node}(u) \wedge \text{lab}_0(u) \Rightarrow \exists v \exists e \text{ node}(v) \wedge \text{lab}_1(v) \wedge \text{inc}(e, u, v) \wedge \text{lab}_a(e)$$

has the meaning “For all nodes with label 0, there exists an edge with label a from this node to a node with label 1”.

For automated theorem proving, one requires an exact axiomatization of the above structures to restrict considerations to directed, totally labeled graphs. We use the axiomatization of [Cou97] for unlabeled graphs and extend it to labeled graphs.

Fact 3.24 (axiomatization). For an alphabet $C = \langle C_V, C_E \rangle$ with $C_V \cap C_E = \emptyset$, the class of structures (D_G, I_G) over Σ are exactly those which satisfy the following properties:

- (1) (Target and source) nodes cannot be edges:
 $\forall e \forall x \forall y (\text{inc}(e, x, y) \Rightarrow \neg \exists u \exists v (\text{inc}(x, u, v) \vee \text{inc}(y, u, v)))$
- (2) An edge has at most one source and one target:
 $\forall e \forall x \forall y \forall x' \forall y' ((\text{inc}(e, x, y) \wedge \text{inc}(e, x', y')) \Rightarrow (x = x' \wedge y = y'))$

- (3) An element has at most one label:
 $\forall x \wedge_{b,d \in C_V, b \neq d} \neg(\text{lab}_b(x) \wedge \text{lab}_d(x)) \quad \wedge \quad \wedge_{b,d \in C_E, b \neq d} \neg(\text{lab}_b(x) \wedge \text{lab}_d(x))$
- (4) Every node has a node label, every edge has an edge label:
 $\forall x \text{ node}(x) \Leftrightarrow \bigvee_{b \in C_V} \text{lab}_b(x) \text{ and } \forall x \text{ edge}(x) \Leftrightarrow \bigvee_{b \in C_E} \text{lab}_b(x)$

Every other statement is implicit, such as “An edge has source and target nodes” (otherwise it is not an edge).

There is a transformation from first-order graph formulas into graph conditions.

Theorem 3.25 (from formulas to conditions). *There is a transformation Cond from first-order graph formulas to graph conditions, such that, for all first-order formulas F over Σ and all graphs G ,*

$$G \models F \text{ if and only if } G \models_{\mathcal{A}} \text{Cond}(F).$$

Before we present a construction of Cond , let us make some preliminary considerations. We consider formulas on directed, labeled graphs with multiple parallel edges. Hence edges are handled as individuals. If F is a *rectified* formula, that is, distinct quantifiers bind occurrences of distinct variables, the variables of F can be represented by isolated nodes and edges in the graphs of a constructed condition. Let X be such a graph. Let $D_X = V_X + E_X$ be the domain and $D'_X \subseteq V_X + E_X$ be the set of all isolated nodes and all edges in X . If D'_X is a subset of the set Var of variables, then every graph morphism $m: X \rightarrow G$ into a non-empty graph G induces an assignment $\sigma: \text{Var} \rightarrow D_G$ such that $m = \sigma[D'_X]$, that is, $m(x) = \sigma(x)$ for each $x \in D'_X$. Vice versa, an assignment $\sigma: \text{Var} \rightarrow D_G$ induces a mapping $D'_X \rightarrow D_G$ that may be extended to a graph morphism $m: X \rightarrow G$ with $m = \sigma[D'_X]$.

$$\begin{array}{ccccccc}
 & & \text{Free}(F) & & & & \\
 & & \subseteq & & & & \\
 & \text{Var} & \xrightarrow{\subseteq} D'_X & \subseteq & D_X & \cdots & X \\
 & & \searrow & & \downarrow & = & \downarrow m \\
 & & \sigma & & D_G & \cdots & G
 \end{array}$$

The main problem of translating a formula into a condition is quantifiers. The flexibility of formulas allows to separate quantifiers and predicates. For instance, it is possible to express “There is an element” and leaving open whether that element is a node or an edge and how it is labeled. In contrast, the rigid structure of conditions does not allow to separate quantifiers and statements. Due to the fact that we consider total, totally labeled graphs, we

either have to depict a labeled node or a labeled edge (together with labeled source and target nodes).

The key idea of the transformation Cond is to represent existential quantification in a formula by disjunction over all possible choices, that is, nodes or edges with all their possible labels. Some of these branches may later become unsatisfiable, depending on occurring lab predicates.

Construction. Assume that F is a closed and rectified formula over Σ . Otherwise, consider the universal closure of F and rename the variables. The graph condition is given by $\text{Cond}(F) = \text{Cond}(\emptyset, F)$, where \emptyset denotes the empty graph. For a formula F over Σ and a graph X with $\text{Free}(F) \subseteq D'_X \subseteq \text{Var}$, the graph condition $\text{Cond}(X, F)$ is constructed as follows:

$\text{Cond}(X, \text{lab}_b(x)) = \text{true}$ if $l_X(x) = b$; false otherwise.

$\text{Cond}(X, \text{inc}(x, y, z)) = \exists(X \rightarrow X[s_X(x) = y][t_X(x) = z])$ if $x \in E_X$, $y, z \in V_X$, $l_X(s_X(x)) = l_X(y)$ and $l_X(t_X(x)) = l_X(z)$; false otherwise.

A graph $X[x=y]$ is obtained from X by identifying the elements x and y .

$\text{Cond}(X, x = y) = \exists(X \rightarrow X[x = y])$ iff x and y are identifiable, that is, $(x, y \in V_X \text{ or } x, y \in E_X, s_X(x) = s_X(y), t_X(x) = t_X(y))$ and $l_X(x) = l_X(y)$; false otherwise.

$\text{Cond}(X, \exists x F) =$
 $\bigvee_{b \in C_V} \exists(X \hookrightarrow Y, \text{Cond}(Y, F)) \vee \bigvee_{b \in C_E, d, d' \in C_V} \exists(X \hookrightarrow Z, \text{Cond}(Z, F))$
 where $Y = X + \textcircled{b}_x$ is obtained from X by adding a node x with label b and $Z = X + \textcircled{d}_x \xrightarrow{b}_{d'} \textcircled{d'}$ by adding an edge x with label b together with a d -labeled source and a d' -labeled target.

$\text{Cond}(X, F)$ is extended to Boolean formulas with the operators true , \neg , \wedge as usual.

Example 3.26. Let $C_V = \{0, 1\}$ and $C_E = \{a, b\}$. The first-order graph formula $F = \exists x \text{lab}_a(x)$ with the meaning “There exists an item with label a ” is transformed into the graph condition

$$\begin{aligned}
 & \text{Cond}(\exists x \text{lab}_a(x)) = \text{Cond}(\emptyset, \exists x \text{lab}_a(x)) \\
 &= \bigvee_{m \in C_V} \exists(\emptyset \hookrightarrow \textcircled{m}_x, \text{Cond}(\textcircled{m}_x, \text{lab}_a(x))) \\
 & \quad \vee \bigvee_{k, n \in C_V, m \in C_E} \exists(\emptyset \hookrightarrow \textcircled{k}_x \xrightarrow{m}_{\textcircled{n}}, \text{Cond}(\textcircled{k}_x \xrightarrow{m}_{\textcircled{n}}, \text{lab}_a(x))) \\
 &= \exists(\emptyset \hookrightarrow \textcircled{0}, \text{false}) \vee \exists(\emptyset \hookrightarrow \textcircled{1}, \text{false}) \\
 & \quad \vee \bigvee_{k, n \in C_V} (\exists(\emptyset \hookrightarrow \textcircled{k}_x \xrightarrow{a}_{\textcircled{n}}, \text{true}) \vee \exists(\emptyset \hookrightarrow \textcircled{k}_x \xrightarrow{b}_{\textcircled{n}}, \text{false})) \\
 &\equiv \bigvee_{k, n \in C_V} \exists(\emptyset \hookrightarrow \textcircled{k}_x \xrightarrow{a}_{\textcircled{n}}),
 \end{aligned}$$

with the meaning “There exists an edge with label a and arbitrarily labeled source and target”. This example shows that, in case of total, totally labeled graphs, unspecifiedness is represented by disjunction over all possibilities, which may, at least temporarily, lead to rather large conditions. A remedy could be the consideration of conditions over partial and/or partially labeled graphs.

The proof of Theorem 3.25 depends on the following lemma.

Lemma 3.27. For all rectified formulas F over Σ , all graphs G , and all graphs X with $\text{Free}(F) \subseteq D'_X \subseteq \text{Var}$ we have: For all morphisms $m: X \rightarrow G$ and all assignments $\sigma: \text{Var} \rightarrow D_G$ with $m = \sigma[D'_X]$, $G[F](\sigma) = \text{true}$ if and only if $m \models_{\mathcal{A}} \text{Cond}(X, F)$.

Proof. By structural induction.

Basis. For $F = \text{true}$, the statement is straightforward. For atomic formulas, the statement follows directly from the definitions:

$$\begin{aligned}
 & G[\text{lab}_b(x)](\sigma) = \text{true} \\
 \Leftrightarrow & l_G(\sigma(x)) = b && (\text{Def. } G[_](\sigma)) \\
 \Leftrightarrow & l_G(m(x)) = b && (m = \sigma[D'_X], x \in \text{Free}(\text{lab}_b(x)) \subseteq D'_X) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \text{true} \text{ and } l_X(x) = b && (m \text{ label-preserving}) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \text{Cond}(X, \text{lab}_b(x)) && (\text{Def. Cond})
 \end{aligned}$$

$$\begin{aligned}
 & G[\text{inc}(x, y, z)](\sigma) = \text{true} \\
 \Leftrightarrow & \sigma(x) \in E_G, s_G(\sigma(x)) = \sigma(y) \text{ and } t_G(\sigma(x)) = \sigma(z) && (\text{Def. } G[_](\sigma)) \\
 \Leftrightarrow & m(x) \in E_G, s_G(m(x)) = m(y) \text{ and } t_G(m(x)) = m(z) && (m = \sigma[D'_X], x, y, z \in \text{Free}(\text{inc}(x, y, z)) \subseteq D'_X) \\
 \Leftrightarrow & x \in E_X, y, z \in V_X, m(s_X(x)) = m(y) && \\
 & \text{and } m(t_X(x)) = m(z) && (m \text{ is a morphism}) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \exists(X \rightarrow X[s_X(x) = y][t_X(x) = y]) && \\
 & \text{and } x \in E_X, y, z \in V_X && \\
 & \text{and } l_X(s_X(x)) = l_X(y), l_X(t_X(x)) = l_X(z) && (\text{Def. } \models_{\mathcal{A}}) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \text{Cond}(X, \text{inc}(x, y, z)) && (\text{Def. Cond})
 \end{aligned}$$

$$\begin{aligned}
 & G[x = y](\sigma) = \text{true} \\
 \Leftrightarrow & (\sigma(x), \sigma(y) \in V_G \text{ or } \sigma(x), \sigma(y) \in E_G) \text{ and } \sigma(x) = \sigma(y) && (\text{Def. } G[_](\sigma)) \\
 \Leftrightarrow & m(x) = m(y) && (m = \sigma[D'_X], x, y \in \text{Free}(x = y) \subseteq D'_X, m \text{ is a morphism}) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \exists(X \rightarrow X[x = y]) \text{ and } x, y \text{ are identifiable} && (\text{Def. } \models_{\mathcal{A}}) \\
 \Leftrightarrow & m \models_{\mathcal{A}} \text{Cond}(X, x = y) && (\text{Def. Cond})
 \end{aligned}$$

Hypothesis. Assume, the statement holds for rectified formulas F and F_j ($j \in J$).

Step. For formulas of the form $\exists x F$, the proof uses the inductive hypothesis:

$$\begin{aligned}
& G[\![\exists x F]\!](\sigma) = \text{true} \\
\Leftrightarrow & \exists o \in D_G. G[\![F]\!](\sigma\{x/o\}) = \text{true} && (\text{Def. } [\![\cdot]\!]) \\
\Leftrightarrow & \exists o \in V_G. \exists b \in C_V. l_G(o) = b \text{ and } G[\![F]\!](\sigma\{x/o\}) = \text{true} \text{ or} \\
& \exists o \in E_G. \exists b \in C_E. l_G(o) = b \text{ and } G[\![F]\!](\sigma\{x/o\}) = \text{true} && (\text{Assignment}) \\
\Leftrightarrow & \exists b \in C_V. \exists m'. m = m' \circ X \rightarrow Y \text{ and } m' \models_{\mathcal{A}} \text{Cond}(Y, F) \text{ or} \\
& \exists b \in C_E. \exists d, d' \in C_V. \exists m'. m = m' \circ X \rightarrow Z \\
& \text{and } m' \models_{\mathcal{A}} \text{Cond}(Z, F) && (\text{Hypothesis, } m' = \sigma\{x/o\}[D'_X]) \\
\Leftrightarrow & \exists b \in C_V. m \models_{\mathcal{A}} \exists(X \rightarrow Y, \text{Cond}(Y, F)) \text{ or} \\
& \exists b \in C_E. \exists d, d' \in C_V. m \models_{\mathcal{A}} \exists(X \rightarrow Z, \text{Cond}(Z, F)) && (\text{Def. } \models_{\mathcal{A}}) \\
\Leftrightarrow & m \models_{\mathcal{A}} \text{Cond}(X, \exists x F) && (\text{Def. Cond})
\end{aligned}$$

where $Y = X + \textcircled{b}$ is obtained from X by adding a node x with label b and $Z = X + \textcircled{d} \xrightarrow{b} \textcircled{d'}$ by adding an edge x with label b together with d -labeled source and d' -labeled target nodes. The rectifiedness of F guarantees that x does not already exist. For formulas built with the operators \neg and \wedge , the proof of the statement is straightforward and uses the inductive hypothesis. This completes the inductive proof. \square

With Lemma 3.27 we prove Theorem 3.25.

Proof of Theorem 3.25. For all closed, rectified formulas F over Σ and all graphs G , we have: $G \models F$, iff for all assignments $\sigma: \text{Var} \rightarrow D_G$, $G[\![F]\!](\sigma) = \text{true}$ (Definition \models), iff for all morphisms $i_G: \emptyset \hookrightarrow G$, $i_G \models_{\mathcal{A}} \text{Cond}(\emptyset, F)$ (Lemma 3.27, for $\text{Free}(F) = \emptyset = D'_X$ and $X = \emptyset$), iff $G \models_{\mathcal{A}} \text{Cond}(\emptyset, F)$ (Definition $\models_{\mathcal{A}}$). \square

Vice versa, there is a transformation from graph conditions into first-order graph formulas.

Theorem 3.28 (from conditions to formulas). *There exists a transformation Form from graph conditions to first-order graph formulas, such that, for all graph conditions c over the empty graph \emptyset and all graphs G ,*

$$G \models_{\mathcal{A}} c \text{ if and only if } G \models \text{Form}(c).$$

For the construction of the transformation Form , we consider morphisms and conditions in a certain normal form, such that the identities of the nodes and edges in a condition can be associated with variables, that is, no unnecessary identity changes of elements take place within a condition. A morphism

$a: P \rightarrow C$ is *identity-preserving*, if $y \in C$ implies ($y = a(y)$ or ($y \notin a(P)$ and $y \notin P$)), elementwise for nodes and edges. This is no restriction, as every condition may be transformed into an equivalent condition, by subsequently replacing all morphisms not in this form by morphisms with this property.

Fact 3.29. Every identity-preserving graph morphism a with domain P is either the identity $\text{id}: P \leftrightarrow P$ or can be decomposed into morphisms that, starting from P , subsequently add nodes, add edges, identify nodes, and identify edges. There may be multiple decompositions for a given morphism.

The key idea of the transformation Form is to decompose the condition $\exists(a, c')$ into a nested condition $\exists(a_1, \dots, \exists(a_n, \exists(\text{id}, c')))$, such that in each morphism a_j , $1 \leq j \leq n$, either exactly one element is added or two elements are identified. To this end, we introduce the following notation.

Notation. Let $[vb]a$, $[euvb]a$, $[u=v]a$ and $[e=e']a$ denote morphisms that are decomposable into an atomic morphism (adding exactly either one node, one edge, or identifying two elements) and a remaining morphism $a: P \rightarrow C$. More precisely, a morphism

$[vb]a$ is decomposable into $[vb]P \hookrightarrow P \xrightarrow{a} C$ such that the morphism $[vb]P \hookrightarrow P$ only adds a b -labeled node v in P .

$[euvb]a$ is decomposable into $[euvb]P \hookrightarrow P \xrightarrow{a} C$ such that the morphism $[euvb]P \hookrightarrow P$ only adds a b -labeled edge e from a node u to a node v in P .

$[u=v]a$ is decomposable into $[u=v]P \rightarrow P \xrightarrow{a} C$ such that the morphism $[u=v]P \rightarrow P$ only identifies two nodes u, v in P .

$[e=e']a$ is decomposable into $[e=e']P \rightarrow P \xrightarrow{a} C$ such that the morphism $[e=e']P \rightarrow P$ only identifies two edges e, e' in P (that already have common source and target nodes).

Construction. For conditions based on identity-preserving graph morphisms, Form is defined as follows:

$$\begin{aligned}
\text{Form}(\text{true}) &= \text{true} \\
\text{Form}(\exists(\text{id}, c')) &= \text{Form}(c') \\
\text{Form}(\exists([vb]a, c')) &= \exists v (\text{node}(v) \wedge \text{lab}_b(v) \wedge \text{Form}(\exists(a, c'))) \\
\text{Form}(\exists([euvb]a, c')) &= \exists e (\text{inc}(e, u, v) \wedge \text{lab}_b(e) \wedge \text{Form}(\exists(a, c'))) \\
\text{Form}(\exists([u=v]a, c')) &= (u=v \wedge \text{Form}(\exists(a, c'))) \\
\text{Form}(\exists([e=e']a, c')) &= (e=e' \wedge \text{Form}(\exists(a, c')))
\end{aligned}$$

Form is extended for the operators \neg, \wedge as usual. Note that Form is not unique, but well defined.

Remark 3.30. For every graph condition c over P , all free variables of the constructed formula $\text{Form}(c)$ correspond to elements in P : $\text{Free}(\text{Form}(c)) \subseteq D_P \subseteq \text{Var}$.

Example 3.31. Let $C_V = \{0, 1\}$ and $C_E = \{b, d\}$. The graph condition

$$\forall(\emptyset \hookrightarrow \textcircled{0}_u, \exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v))$$

with the meaning “For every node with label 0, there exists an outgoing b -labeled edge to a node with label 1” is transformed into the following first-order formula:

$$\begin{aligned} & \text{Form}(\neg\exists(\emptyset \hookrightarrow \textcircled{0}_u, \neg\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v))) \\ &= \neg\text{Form}(\exists(\emptyset \hookrightarrow \textcircled{0}_u, \neg\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \wedge \text{Form}(\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u, \neg\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v)))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \wedge \text{Form}(\neg\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \wedge \neg\text{Form}(\exists(\textcircled{0}_u \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \wedge \neg\exists v (\text{node}(v) \wedge \text{lab}_1(v) \\ & \quad \wedge \text{Form}(\exists(\textcircled{0}_u \textcircled{1}_v \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v)))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \wedge \neg\exists v (\text{node}(v) \wedge \text{lab}_1(v) \wedge \exists e (\text{inc}(e, u, v) \wedge \text{lab}_b(e) \\ & \quad \wedge \text{Form}(\exists(\textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v \hookrightarrow \textcircled{0}_u \xrightarrow[b]{e} \textcircled{1}_v)))))) \\ &= \neg\exists u (\text{node}(u) \wedge \text{lab}_0(u) \\ & \quad \wedge \neg\exists v (\text{node}(v) \wedge \text{lab}_1(v) \wedge \exists e (\text{inc}(e, u, v) \wedge \text{lab}_b(e) \wedge \text{true}))) \\ &\equiv \forall u ((\text{node}(u) \wedge \text{lab}_0(u)) \Rightarrow \exists v \exists e (\text{node}(v) \wedge \text{lab}_1(v) \wedge \text{inc}(e, u, v) \wedge \text{lab}_b(e))). \end{aligned}$$

with the same meaning “For every node u with label 0, there is a node v with label 1 and an b -labeled edge from u to v .”

Example 3.32 (access control formula). Consider the access control condition

$$\text{secure} = \forall \left(\textcircled{\text{person}}_1 \xrightarrow{\text{pink}} \textcircled{\text{tablet}}_2 \xleftarrow{\text{pink}} \textcircled{\text{server}}_3, \exists \left(\textcircled{\text{person}}_1 \xrightarrow{\text{pink}} \textcircled{\text{tablet}}_2 \xleftarrow{\text{pink}} \textcircled{\text{server}}_3 \right) \right).$$

The application of Form on secure yields the following first-order graph formula

$$\begin{aligned} \text{Form}(\text{secure}) &= \forall n1 (\text{lab}_{\text{person}}(n1) \Rightarrow (\forall n2 (\text{lab}_{\text{tablet}}(n2) \Rightarrow (\forall n3 (\text{lab}_{\text{server}}(n3) \Rightarrow \\ & \quad (\forall e4 ((\text{inc}(e4, n1, n2) \wedge \text{lab}(e4)) \Rightarrow \\ & \quad (\forall e5 ((\text{inc}(e5, n3, n2) \wedge \text{lab}(e5)) \Rightarrow \\ & \quad \exists e6 (\text{inc}(e6, n1, n3) \wedge \text{lab}(e6)))))))))). \end{aligned}$$

The proof of Theorem 3.28 depends on the following lemma.

Lemma 3.33. For all graph conditions c over P and all graphs G we have:
For all graph morphisms $m: P \rightarrow G$ and all assignments $\sigma: \text{Var} \rightarrow D_G$ with $m = \sigma[D_P]$,

$$m \models_{\mathcal{A}} c \text{ iff } G[\llbracket \text{Form}(c) \rrbracket](\sigma) = \text{true}.$$

Proof. By structural induction.

Basis. For $c = \text{true}$, we have $m \models \text{true} \Leftrightarrow G[\llbracket \text{true} \rrbracket](\sigma) = \text{true}$.

Hypothesis. Assume the statement holds for condition c' .

Step. For conditions of the form $c = \exists(a, c')$, the statement is proved by induction over the decomposition of morphisms. If the morphism a is the identity $\text{id}: P \rightarrow P$, we observe for all morphisms $m: P \rightarrow G$, and all assignments $\sigma: \text{Var} \rightarrow D_G$ with $m = \sigma[D_P]$, $m \models_{\mathcal{A}} \exists(\text{id}, c') \Leftrightarrow m \models_{\mathcal{A}} c' \Leftrightarrow G[\llbracket \text{Form}(c') \rrbracket](\sigma) = \text{true}$ (hypothesis for c').

Assume, the statement holds for a condition $\exists(a, c')$. We prove that the statement holds for all conditions $\exists(a', c')$ with extended morphism $a' = a[vb]$, $a[eu vb]$, $a[u=v]$, and $a[e=e']$:

$$\begin{aligned} & m \models_{\mathcal{A}} \exists([vb]a, c') \\ \Leftrightarrow & m \models_{\mathcal{A}} \exists([vb]P \rightarrow P, \exists(a, c')) && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \exists m'. m = m' \circ [vb]P \rightarrow P \text{ and } m' \models_{\mathcal{A}} \exists(a, c') && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \exists o \in D_G. \sigma' = \sigma\{v/o\} \text{ and } G[\llbracket \text{node}(v) \rrbracket](\sigma') = \text{true} \\ & \text{and } G[\llbracket \text{lab}_b(v) \rrbracket](\sigma') = \text{true} \\ & \text{and } G[\llbracket \text{Form}(\exists(a, c')) \rrbracket](\sigma') = \text{true} && (\text{Hypothesis, } \sigma'[D_P] = m') \\ \Leftrightarrow & G[\llbracket \exists v (\text{node}(v) \wedge \text{lab}_b(v) \wedge \text{Form}(\exists(a, c'))) \rrbracket](\sigma) = \text{true} && (\text{Def. } G[\llbracket _ \rrbracket](\sigma)) \\ \Leftrightarrow & G[\llbracket \text{Form}(\exists([vb]a, c')) \rrbracket](\sigma) = \text{true} && (\text{Def. Form}) \\ \Leftrightarrow & G \models \text{Form}(\exists([vb]a, c')) && (\text{Def. } \models) \end{aligned}$$

$$\begin{aligned} & m \models_{\mathcal{A}} \exists([eu vb]a, c') \\ \Leftrightarrow & m \models_{\mathcal{A}} \exists([eu vb]P \rightarrow P, \exists(a, c')) && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \exists m'. m = m' \circ [eu vb]P \rightarrow P \text{ and } m' \models_{\mathcal{A}} \exists(a, c') && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \exists o \in D_G. \sigma' = \sigma\{e/o\} \text{ and } G[\llbracket \text{inc}(e, u, v) \rrbracket](\sigma') = \text{true} \\ & \text{and } G[\llbracket \text{lab}_b(e) \rrbracket](\sigma') = \text{true} \\ & \text{and } G[\llbracket \text{Form}(\exists(a, c')) \rrbracket](\sigma') = \text{true} && (\text{Hypothesis, } \sigma'[D_P] = m') \\ \Leftrightarrow & G[\llbracket \exists e (\text{inc}(e, u, v) \wedge \text{lab}_b(e) \wedge \text{Form}(\exists(a, c'))) \rrbracket](\sigma) = \text{true} && (\text{Def. } G[\llbracket _ \rrbracket](\sigma)) \\ \Leftrightarrow & G[\llbracket \text{Form}(\exists([eu vb]a, c')) \rrbracket](\sigma) = \text{true} && (\text{Def. Form}) \\ \Leftrightarrow & G \models \text{Form}(\exists([eu vb]a, c')) && (\text{Def. } \models) \end{aligned}$$

$$\begin{aligned} & m \models_{\mathcal{A}} \exists([u=v]a, c') \\ \Leftrightarrow & m \models_{\mathcal{A}} \exists([u=v]P \rightarrow P, \exists(a, c')) && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \exists m'. m = m' \circ [u=v]P \rightarrow P \text{ and } m' \models_{\mathcal{A}} \exists(a, c') && (\text{Def. } \models_{\mathcal{A}}) \\ \Leftrightarrow & \sigma(u) = \sigma(v) \text{ and } G[\llbracket u=v \rrbracket](\sigma) = \text{true} \\ & \text{and } G[\llbracket \text{Form}(\exists(a, c')) \rrbracket](\sigma) = \text{true} && (\text{Hypothesis, } \sigma[D_P] = m') \end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow G \llbracket (u = v \wedge \text{Form}(\exists(a, c')) \rrbracket(\sigma) = \text{true} && (\text{Def. } G \llbracket _ \rrbracket(\sigma)) \\
&\Leftrightarrow G \llbracket \text{Form}(\exists([u=v]a, c')) \rrbracket(\sigma) = \text{true} && (\text{Def. Form}) \\
&\Leftrightarrow G \models \text{Form}(\exists([u=v]a, c')) && (\text{Def. } \models)
\end{aligned}$$

$$m \models_{\mathcal{A}} \exists([e=e']a, c') \Leftrightarrow G \models \text{Form}(\exists([e=e']a, c')) \quad (\text{as above})$$

Since all identity-preserving graph morphisms may be decomposed into graph morphisms that subsequently add nodes, add edges, identify nodes and identify edges, the statement holds for all conditions of the form $\exists(a, c')$. For conditions built with the operators \neg , \wedge , the proof of the statement is straightforward. \square

With Lemma 3.33 we prove Theorem 3.28.

Proof of Theorem 3.28. For all graph conditions c over \emptyset and all graphs G , we have the following: $G \models_{\mathcal{A}} c$ iff for all morphisms $i_G: \emptyset \hookrightarrow G$, $i_G \models_{\mathcal{A}} c$ (Definition $\models_{\mathcal{A}}$) iff for all assignments $\sigma: \text{Var} \rightarrow D_G$, $G \llbracket \text{Form}(c) \rrbracket(\sigma) = \text{true}$ (Lemma 3.33, for $P = \emptyset$ and $\text{Free}(\text{Form}(c)) = \emptyset = D_P$) iff $G \models \text{Form}(c)$ (Definition \models). \square

By Theorems 3.25 and 3.28, we obtain the equivalence of graph conditions under \mathcal{A} -satisfiability and first-order graph formulas.

Corollary 3.34 (\mathcal{A} : equivalence of graph conditions and formulas).

\mathcal{A} -satisfiable graph conditions and first-order graph formulas are expressively equivalent.

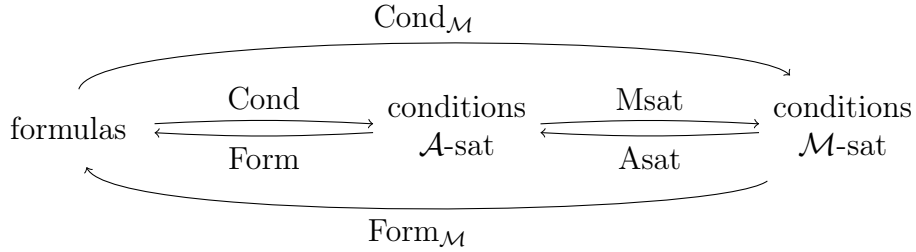
$ \begin{array}{ccc} \text{first-order} & \xrightarrow{\text{Cond}} & \text{conditions} \\ \text{graph formulas} & \xleftarrow{\text{Form}} & \mathcal{A}\text{-satisfiability} \end{array} $

The standard semantics for conditions is \mathcal{M} -satisfiability. By Fact 2.21 and Theorems 3.12 and 3.16, \mathcal{A} -satisfiable graph conditions can be transformed into \mathcal{M} -satisfiable graph conditions and vice versa (in the case of **Graphs**, the class \mathcal{M} of injective graph morphisms is strictly closed under decomposition). Therefore, \mathcal{M} -satisfiable graph conditions and first-order graph formulas are expressively equivalent, as well.

Corollary 3.35 (\mathcal{M} : equivalence of graph conditions and graph formulas). \mathcal{M} -satisfiable graph conditions and first-order graph formulas are expressively equivalent.

$ \begin{array}{ccc} \text{first-order} & \xrightarrow{\text{Cond}_{\mathcal{M}}} & \text{conditions} \\ \text{graph formulas} & \xleftarrow{\text{Form}_{\mathcal{M}}} & \mathcal{M}\text{-satisfiability} \end{array} $

Proof. Immediate consequence of Theorems 3.25, 3.28 together with Theorems 3.12, 3.16, which allow to switch between \mathcal{A} - and \mathcal{M} -satisfiability: Define $\text{Cond}_{\mathcal{M}} = \text{Msat} \circ \text{Cond}$ and $\text{Form}_{\mathcal{M}} = \text{Form} \circ \text{Asat}$. By Theorems 3.25 and 3.12, for all formulas F over Σ and all graphs G , $G \models F \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(F) \Leftrightarrow G \models \text{Msat}(\text{Cond}(F)) = \text{Cond}_{\mathcal{M}}(F)$. By Theorems 3.28 and 3.16, for all formulas F over Σ and all graphs G , $G \models c \Leftrightarrow G \models_{\mathcal{A}} \text{Asat}(c) \Leftrightarrow G \models \text{Form}(\text{Asat}(c)) = \text{Form}_{\mathcal{M}}(c)$.



□

Conditions and formulas are *finite*, if the index set J of every conjunction $\bigwedge_{j \in J}$ and every disjunction $\bigvee_{j \in J}$ is finite. In the following, we want to strengthen the statement of Corollary 3.35 by showing that every finite graph condition can be translated into a finite first-order graph formula and vice versa. Assumption 2.19 ensures the effectiveness of the constructions.

Asat and Msat yield finite results for finite inputs.

Fact 3.36 (transformation of finite conditions). Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with a finite number of epimorphisms for every given domain. For every finite condition c , the conditions $\text{Msat}(c)$ and $\text{Asat}(c)$ of Theorem 3.12 and 3.16 are finite, respectively.

Proof. For $\text{Msat}(c)$ we have: As \mathcal{C} has a finite number of epimorphisms for a given domain, the set E in the construction of $\text{Msat}(e, c)$ is finite for every epimorphism e , and the set E in the construction of $\text{Msat}(c)$ is finite. For $\text{Asat}(c)$ we have: As \mathcal{C} has a finite number of epimorphisms for a given domain, the set E in the construction of inM_C is finite for every object $C \in \mathcal{C}$, therefore $\text{Asat}(c)$ is finite. □

Every finite graph condition can be translated into a finite first-order graph formula and vice versa.

Fact 3.37 (equivalence of finite graph conditions and formulas). Finite \mathcal{M} -satisfiable graph conditions, finite \mathcal{A} -satisfiable graph conditions and finite first-order graph formulas are expressively equivalent.

Proof. By Corollaries 3.34 and 3.35, it suffices to show that all involved transformations preserve finiteness. For every finite graph condition c , the graph formula $\text{Form}(c)$ is finite: For every graph morphism, there exists a finite decomposition, such that each morphism either adds a node, adds an edge, identifies two nodes, or identifies two edges. For every finite first-order graph formula F , the graph condition $\text{Cond}(F)$ is finite: The disjunctions in the case $\text{Cond}(X, \exists x F)$ are finite, as the label alphabet $C = C_V \cup C_E$ is finite. As the category $(\text{Graphs}, \text{Inj})$ satisfies Assumption 2.19, we have by Fact 3.36: For every finite graph condition c , the graph conditions $\text{Msat}(c)$ and $\text{Asat}(c)$ are finite. \square

3.4 Summary and discussion

We use the framework of weak adhesive HLR categories and consider (*nested*) *conditions* as a graphical formalism to specify sets of objects as well as morphisms, similar to graph predicates as introduced independently in [Ren04a]. For a category associated with a graphical representation such as graphs, conditions are a graphical and intuitive, yet precise formalism, well-suited to describe structural properties.

We show that (nested) graph conditions are expressively equivalent to first-order graph formulas, that is, (nested) graph conditions exactly capture first-order graph properties. The first part of this proof includes transformations between two satisfiability notions of conditions, namely \mathcal{M} -satisfiability and \mathcal{A} -satisfiability. The second part is specific to the category **Graphs** and includes transformations between \mathcal{A} -satisfiable graph conditions and first-order formulas on graphs, similar to a proof presented earlier in [Ren04a] for edge-labeled graphs without parallel edges.

The composite transformation of graph formulas into \mathcal{M} -satisfiable graph conditions is used in Section 6.1 to show the undecidability of certain problems on the level of (graph) conditions. The composite transformation of \mathcal{M} -satisfiable graph conditions to graph formulas allows the use of existing first-order tools to solve problems for graph conditions. We will use this fact in Chapter 7 for a comparison of our satisfiability solver and theorem prover for conditions with off-the-shelf tools for first-order logic.

4. Programs and transformation systems

In the following we introduce our notion of structural transformations, namely programs with interface. Programs with interface are based on four elementary constructs, that is, the selection, addition, deletion and unselection of an object's elements. These primitives can be combined to more complex programs by the usual control constructs, such as non-deterministic choice, sequential composition, conditional execution and iteration. The resulting programming language is computationally complete.

The fact that programs with interface allow explicit control over the selection of elements and are capable of handing over selections between computations is our main justification for their consideration: This hand-over mechanism can be used to restrict the execution of programs to a previously selected context, which is, for instance, crucial for the definition of a satisfiability algorithm as it turns out later in Section 6.2.

As we will discover in Section 4.3, programs with interface are a generalization of programs over transformation rules, as considered in [HP01, PS04]. Graph programs over graph transformation rules, while computationally complete in the context of relabeling [HP01], do not have a comparable control mechanism. They always unselect all preserved and created elements after every computation step. To recover an element in a next computation step, elements are marked by fresh labels or reserved structure such as loops, which requires changes to the considered graph model.

4.1 Programs with interface

We define programs with interface that are capable of handing over a selection of elements between computation steps. An overview over the program constructs together with their intuitive meaning is given in Table 4.1.

Skip	// No computation
Abort	// Abort of computation
Assert (c)	// Assertion of a condition c
Sel ($x: X \hookrightarrow L$)	// Selection of additional elements ($L - x(X)$)
Sel ($x: X \hookrightarrow L, c$)	// As above, but any result must satisfy condition c
Del ($l: L \hookleftarrow K$)	// Selective deletion of elements ($L - l(K)$)
Add ($r: K \hookrightarrow R$)	// Addition of elements ($R - r(K)$)
Uns ($y: R \hookleftarrow Y$)	// Unselection of selected elements ($R - y(Y)$)
$\langle\langle L \hookleftarrow K \hookrightarrow R \rangle, ac_L\rangle$	// Transformation rule with left application condition
(P; Q)	// Sequential composition of program Q after P
{P, ..., Q}	// (Demonic) non-deterministic choice of programs
Fix (P)	// Computation path-specific unselection of elements
P^j	// j -times execution of program P
P^*	// Reflexive, transitive closure of program P
$\downarrow P \downarrow$	// As long as possible iteration of program P
if c then P fi	// Conditional execution of program P
if c then P else Q fi	// Conditional execution of programs P and Q
while c do P od	// Conditional iteration of program P

Table 4.1: Program constructs and their meaning

Definition 4.1 (syntax of programs with interface). Programs with interface are defined inductively. For \mathcal{M} -morphisms $m: C \hookrightarrow D$ and conditions d over D , the expressions $\text{Sel}(m, d)$, $\text{Del}(m)$, $\text{Add}(m)$, $\text{Uns}(m)$ are elementary *programs with interface* C . Given a program P with interface C and a program Q with arbitrary interface, the expression $(P; Q)$ is a program with interface C . Given programs P, \dots, Q with interface C , the expressions $\{P, \dots, Q\}$, $\text{Fix}(P)$, P^* , $\downarrow P \downarrow$ are programs with interface C . Programs P^* , $\downarrow P \downarrow$ are iterated programs. For every program $\downarrow P \downarrow$, we assume P is free of any form of iteration.

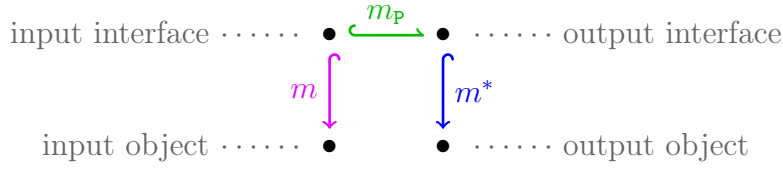
Additionally, we introduce the following abbreviations:

$$\begin{aligned}
\text{Assert}(c) &= \text{Sel}(\text{id}_C, c), \\
\text{Skip} &= \text{Assert}(\text{true}), \\
\text{Abort} &= \text{Assert}(\text{false}), \\
\text{Sel}(m) &= \text{Sel}(m, \text{true}), \\
\langle\langle L \hookleftarrow K \hookrightarrow R \rangle, ac_L \rangle &= \text{Sel}(I \hookrightarrow L, ac_L); \text{Del}(L \hookleftarrow K); \\
&\quad \text{Add}(K \hookrightarrow R); \text{Uns}(R \hookleftarrow I), \\
P^0 &= \text{Skip}, \\
P^j &= (\text{Fix}(P); P^{j-1}) \text{ with } j > 0, \\
\text{if } c \text{ then } P \text{ fi} &= \{(\text{Assert}(c); P), \text{Assert}(\neg c)\}, \\
\text{if } c \text{ then } P \text{ else } Q \text{ fi} &= \{(\text{Assert}(c); P), (\text{Assert}(\neg c); Q)\}, \text{ and} \\
\text{while } c \text{ do } P \text{ od} &= ((\text{Assert}(c); P)^*; \text{Assert}(\neg c)).
\end{aligned}$$

Definition 4.2 (transformation system). For programs P, \dots, Q with interface, programs of the form $\{P, \dots, Q\}^*$ are *transformation systems*.

Traditionally, the term transformation system refers to the reflexive, transitive closure of a non-deterministic choice of transformation rules. In this sense, programs with interface generalize transformation rules.

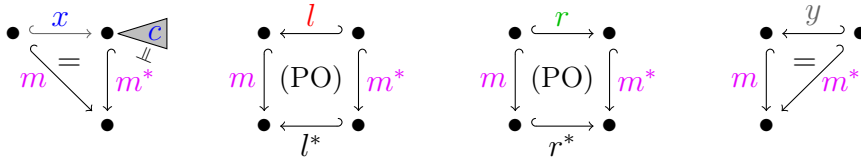
Let \mathcal{P} denote the set of all partial monomorphisms, see Appendix A. Programs with interface transform morphisms instead of objects. The semantics of a program P consists of triples $\langle m, m^*, m_P \rangle \in \mathcal{M} \times \mathcal{M} \times \mathcal{P}$, where the first two \mathcal{M} -morphisms represent *input* and *result*, while the last partial monomorphism is an “*interface relation*” from the domain of the input to the domain of the result morphism. The input interface represents a number of elements that are assumed to be present in the input object at the begin of a computation. In this sense the input interface of a program may be seen as a kind of input type.

Figure 4.1: Triple $\langle m, m^*, m_P \rangle$ of the semantics of a program P

Definition 4.3 (semantics of programs). Let m, m^*, x, l, r, y be \mathcal{M} -morphisms and m_P, m_Q partial monomorphisms. The semantics of a program P with interface C , denoted by $\llbracket P \rrbracket$, is a subset of $\mathcal{M} \times \mathcal{M} \times \mathcal{P}$ such that for all $\langle m, m^*, m_P \rangle \in \llbracket P \rrbracket$, $\text{dom}(m) = C = \text{dom}(m_P)$, and $\text{dom}(m^*) = \text{codom}(m_P)$ and is defined as follows:

$$\begin{aligned}
\llbracket \text{Sel}(x, c) \rrbracket &= \{ \langle m, m^*, x \rangle \mid m^* \circ x = m, \ m^* \in \mathcal{M} \text{ and } m^* \models c \} \\
\llbracket \text{Del}(l) \rrbracket &= \{ \langle m, m^*, l^{-1} \rangle \mid l^* \circ m^* \text{ is pushout complement of } m \circ l \} \\
\llbracket \text{Add}(r) \rrbracket &= \{ \langle m, m^*, r \rangle \mid \langle r^*, m^* \rangle \text{ is pushout of } \langle m, r \rangle \} \\
\llbracket \text{Uns}(y) \rrbracket &= \{ \langle m, m \circ y, y^{-1} \rangle \mid \text{true} \} \\
\llbracket \text{Fix}(P) \rrbracket &= \{ \langle m, m^* \circ m_P, \text{id} \rangle \mid \langle m, m^*, m_P \rangle \in \llbracket P \rrbracket \text{ and } m_P \in \mathcal{M} \} \\
\llbracket (P; Q) \rrbracket &= \{ \langle m, m^*, m_Q \circ m_P \rangle \mid \langle m, m', m_P \rangle \in \llbracket P \rrbracket \text{ and } \langle m', m^*, m_Q \rangle \in \llbracket Q \rrbracket \} \\
\llbracket \{P, \dots, Q\} \rrbracket &= \bigcup_{R \in \{P, \dots, Q\}} \llbracket R \rrbracket \\
\llbracket P^* \rrbracket &= \bigcup_{j=0}^{\infty} \llbracket P^j \rrbracket \\
\llbracket \downarrow P \downarrow \rrbracket &= \{ \langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \mid \nexists m'. \langle m^*, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \}
\end{aligned}$$

where id is the identity on the domain of m (and m^*). Two programs P, Q are *equivalent*, denoted by $P \equiv Q$, if $\llbracket P \rrbracket = \llbracket Q \rrbracket$.

Figure 4.2: Semantics of $\text{Sel}(x, c)$, $\text{Del}(l)$, $\text{Add}(r)$ and $\text{Uns}(y)$

Example 4.4 (access control rule). Before we formally describe the access control system as a graph program with interface, let us consider the graph transformation rule



which is a simplified part of the login procedure. The rule **Access** has the form $\langle L \hookrightarrow R \rangle$, which is an abbreviation for $\langle \langle L \leftrightarrow K \hookrightarrow R \rangle, \text{true} \rangle$ where we depict the span $\langle L \leftrightarrow K, K \hookrightarrow R \rangle$ as a partial monomorphism $L \hookrightarrow R$.

Concerning the semantics of such a rule: the left-hand side L expresses the prerequisites of a rule's application "If a user proposes a session to a system for which he has the appropriate access right" and the right-hand side R , together with the partial monomorphism, expresses the local effect of the rule's application "Then this proposed session is accepted and becomes established by first deleting the edge from the session to the computer node and by adding an edge in the reverse direction".

Example 4.5 (access control system). Consider the access control graphs in Example 2.4. The dynamic part of the access control system is the reflexive, transitive closure **AccessControl*** of the non-deterministic choice of graph programs **AccessControl** = {AddUser, Grant, Login, Logout, Process, Revoke, Delete}. The programs model the addition and deletion of users, the grant and removal of access rights and the login/logout procedure.

(AddUser) Adds a user to the system. A user node is created and unselected:

$$\left(\text{Add } \begin{array}{c} \text{Ⓜ} \\ \text{①} \end{array} ; \text{Uns } \begin{array}{c} \text{Ⓜ} \\ \text{①} \end{array} \right)$$

(Grant) Grants a user access to a system. Selects a user and a system (for which not already an access right exists), adds an access right, and unselects everything:

$$\left(\text{Sel } \left\langle \begin{array}{c} \text{Ⓜ} \\ \text{①} \end{array}, \begin{array}{c} \text{Ⓜ} \\ \text{②} \end{array}, \neg \exists \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \end{array} \right\rangle ; \text{Add } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \end{array} ; \text{Uns } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \end{array} \right)$$

(Login) A user requests to log into a system. The program selects a user and a system, adds a session node with its edges and unselects everything:

$$\left(\text{Sel } \begin{array}{c} \text{Ⓜ} \\ \text{①} \end{array}, \begin{array}{c} \text{Ⓜ} \\ \text{②} \end{array} ; \text{Add } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{③} \quad \text{②} \end{array} ; \text{Uns } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{③} \quad \text{②} \end{array} \right)$$

(Logout) A user is logged out. A session is selected and whether it is established or proposed, it is closed:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \end{array} ; \\ \left\{ \begin{array}{l} \left(\text{Sel } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \quad \text{④} \end{array} ; \text{Del } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \rightarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \quad \text{④} \end{array} \right), \\ \left(\text{Sel } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \leftarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \quad \text{④} \end{array} ; \text{Del } \begin{array}{c} \text{Ⓜ} \rightarrow \text{Ⓜ} \leftarrow \text{Ⓜ} \\ \text{①} \quad \text{②} \quad \text{④} \end{array} \right) \end{array} \right\} ; \\ \text{Uns } \begin{array}{c} \text{Ⓜ} \\ \text{①} \end{array}, \begin{array}{c} \text{Ⓜ} \\ \text{④} \end{array} \end{array} \right)$$

(Process) The system reacts to a log in. The program selects a proposed session. If the user has the appropriate access right, the session is established. Otherwise, the session is closed:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} ; \\ \text{if } \left(\exists \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} \right) \\ \text{then } \left(\text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} ; \text{Add } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \leftarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \leftarrow \text{System}_3 \end{array} ; \text{Uns } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \leftarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \leftarrow \text{System}_3 \end{array} \right) \\ \text{else } \left(\text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} ; \text{Uns } \begin{array}{c} \text{User}_1 \\ \text{System}_3 \end{array} \right) \text{fi} \end{array} \right)$$

(Revoke) The access right of a user to a system is revoked. Beforehand, the user's established sessions to that system are closed:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_2 \end{array} ; \\ \text{while } \left(\exists \begin{array}{c} \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \end{array} \right) \\ \text{do } \left(\text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \end{array} ; \text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_4 \rightarrow \text{System}_2 \end{array} \right) \text{od} ; \\ \text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_2 \end{array} ; \\ \text{Uns } \begin{array}{c} \text{User}_1 \\ \text{System}_2 \end{array} \end{array} \right)$$

(Delete) A user is deleted. Beforehand, the user's sessions are closed and the user's access rights are revoked:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User}_1 \\ \text{User}_1 \end{array} ; \\ \text{while } \left(\exists \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} \right) \\ \text{do } \left(\text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} ; \text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} \right) \text{od} ; \\ \text{while } \left(\exists \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} \right) \\ \text{do } \left(\text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} ; \text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \\ \text{User}_1 \rightarrow \text{System}_2 \rightarrow \text{System}_3 \end{array} \right) \text{od} ; \\ \text{while } \left(\exists \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_2 \end{array} \right) \\ \text{do } \left(\text{Sel } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_2 \end{array} ; \text{Del } \begin{array}{c} \text{User}_1 \rightarrow \text{System}_2 \\ \text{User}_1 \rightarrow \text{System}_2 \end{array} ; \text{Uns } \begin{array}{c} \text{User}_1 \\ \text{System}_2 \end{array} \right) \text{od} ; \\ \text{Del } \begin{array}{c} \text{User}_1 \\ \text{User}_1 \end{array} \end{array} \right)$$

We now make some observations over program in general, before we discuss properties of some constructs.

Remark 4.6 (interface and sequential composition). For a given input morphism, all possible executions of a program may be depicted as an

unfolded tree. Therefore, programs have a single input interface (by definition), but may have a set of possible *output interfaces* that correspond to the leafs of the aforementioned tree. Consider the semantics of the sequential composition $(P; Q)$. A prerequisite for the existence of results is that the interface of Q coincides with one of the output interfaces of P . Otherwise such a composition is doomed to abort.

Remark 4.7 (iteration and Fix). The semantics of the sequential composition implies that a program with interface C may only be iterated, if the output interface of the previous computation equals the (input) interface C . Consider the semantics of $\text{Fix}(P)$. The statement Fix is a generic way of making programs iterable that do not delete or unselect elements of their interface. Fix ensures that every possible computation ends with the output interface C by finally deselecting all elements additionally selected during a run of the program. Figure 4.3 illustrates the different semantics: while $\langle m, m^*, r \rangle \in \llbracket \text{Add}(\emptyset \hookrightarrow \bigcirc) \rrbracket$, we have $\langle m, m^* \circ r, \text{id} \rangle \in \llbracket \text{Fix}(\text{Add}(\emptyset \hookrightarrow \bigcirc)) \rrbracket$. In this sense, the semantics of Fix corresponds to a specific Uns statement at the end of each program branch, as we show later in Section 4.2.

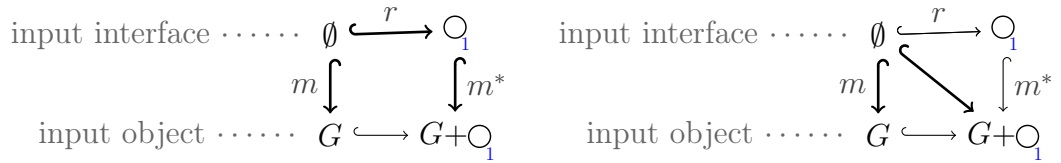


Figure 4.3: Semantics of $\text{Add}(\emptyset \hookrightarrow \bigcirc)$ and $\text{Fix}(\text{Add}(\emptyset \hookrightarrow \bigcirc))$

Remark 4.8 (\mathcal{M} -matching). Consider the semantics of $\text{Sel}(x, c)$. The result morphism m^* is required to be in \mathcal{M} . This corresponds to the notion of \mathcal{M} -matching in [HP09] and means intuitively that selected elements are always distinct. For example, the graph program $\text{Sel}(\emptyset \hookrightarrow \bigcirc_1 \bigcirc_2)$ with interface \emptyset will always attempt to select two distinct nodes of an input object.

Remark 4.9 (determinism). For a given morphism m , the outcome of the elementary programs Del , Add , Uns is deterministic, that is, unique up to isomorphism. In contrast, Sel is non-deterministic as there may be several morphisms m^* for a given input morphism m .

Remark 4.10 (existence of results). For a given input morphism m , the elementary programs Sel , Del may fail, whereas Add , Uns are guaranteed to yield a result morphism m^* .

For all objects C , the identity morphism id_C on C reduces the elementary constructs **Sel**, **Del**, **Add** and **Uns** to the semantics of **Skip**.

Fact 4.11 (identity as parameter). $\text{Sel}(\text{id}, \text{true}) = \text{Assert}(\text{true}) = \text{Skip} \equiv \text{Del}(\text{id}) \equiv \text{Add}(\text{id}) \equiv \text{Uns}(\text{id})$.

The statement **Skip** is the identity element (or neutral element) of sequential composition, while **Abort** is the annihilator of sequential composition.

Fact 4.12 (Skip). $(\text{Skip}; R) \equiv R \equiv (R; \text{Skip})$ and $(\text{Abort}; R) \equiv \text{Abort} \equiv (R; \text{Abort})$.

Sequential composition is associative.

Fact 4.13 (associativity). For all programs P, Q, R , we have $((P; Q); R) \equiv (P; (Q; R))$.

Proof. By the semantics of the sequential composition, we have

$$\begin{aligned}
& \llbracket (P; Q); R \rrbracket \\
= & \{ \langle m, m^*, m_R \circ m_{P;Q} \rangle \mid \langle m, m', m_{P;Q} \rangle \in \llbracket (P; Q) \rrbracket \\
& \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
= & \{ \langle m, m^*, m_R \circ m_Q \circ m_P \rangle \mid \langle m, m'', m_P \rangle \in \llbracket P \rrbracket \\
& \text{and } \langle m'', m', m_Q \rangle \in \llbracket Q \rrbracket \text{ and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
= & \{ \langle m, m^*, m_{Q;R} \circ m_P \rangle \mid \langle m, m'', m_P \rangle \in \llbracket P \rrbracket \\
& \text{and } \langle m'', m^*, m_{Q;R} \rangle \in \llbracket (Q; R) \rrbracket \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
= & \llbracket (P; (Q; R)) \rrbracket \quad (\text{Def. } \llbracket (_; _) \rrbracket)
\end{aligned}$$

□

As the order of sequential compositions is irrelevant for the semantics of programs, we allow to write $(P; Q; R)$.

Sequential composition is right-distributive over nondeterministic choice. Later on, we use this property to transform programs into equivalent programs that do not contain the statement **Fix**.

Fact 4.14 (distributivity). For all programs P, \dots, Q, R , we have

$$(\{P, \dots, Q\}; R) \equiv \{(P; R), \dots, (Q; R)\}.$$

Proof. Let \mathcal{S} denote $\{P, \dots, Q\}$. By the semantics of the sequential composition, nondeterministic choice, we have

$$\begin{aligned}
& \llbracket (\mathcal{S}; R) \rrbracket \\
= & \{ \langle m, m^*, m_{\mathcal{S}} \circ m_R \rangle \mid \langle m, m', m_{\mathcal{S}} \rangle \in \llbracket \mathcal{S} \rrbracket \\
& \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \} \quad (\text{Def. } \llbracket (_; _) \rrbracket)
\end{aligned}$$

$$\begin{aligned}
&= \{ \langle m, m^*, m_S \circ m_R \rangle \mid \langle m, m', m_S \rangle \in \bigcup_{S \in \mathcal{S}} \llbracket S \rrbracket \\
&\quad \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \} \quad (\text{Def. } \llbracket \{ _, \dots, _ \} \rrbracket) \\
&= \bigcup_{S \in \mathcal{S}} \{ \langle m, m^*, m_S \circ m_R \rangle \mid \langle m, m', m_S \rangle \in \llbracket S \rrbracket \\
&\quad \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \} \quad (\{ \dots \mid \dots \}) \\
&= \bigcup_{S \in \mathcal{S}} \llbracket (S; R) \rrbracket \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
&= \llbracket (\bigcup_{S \in \mathcal{S}} (S; R)) \rrbracket \quad (\text{Def. } \llbracket \{ _, \dots, _ \} \rrbracket)
\end{aligned}$$

□

4.2 Elimination of Fix statements

The statement **Fix** is a generic way of making programs iterable that do not delete or unselect elements of their interface. **Fix** ensures that every possible computation ends with the “output” interface C by finally deselecting all elements additionally selected during a run of the program. In this sense, the semantics of **Fix** corresponds to a specific **Uns** statement at the end of each program branch.

In the following we present a construction yielding for every program an equivalent program not containing the statement **Fix** by generating specific **Uns** statement at the end of every branch of the program. On the one hand, the construction demonstrates that **Fix** is not a strictly necessary part of our program language. On the other hand, as **Fix** is a core construct in our language, we depend on the following construction later on as an indirect way to construct weakest preconditions for **Fix**.

Definition 4.15 (Fix-free program). A program with interface is **Fix-free**, if it does not contain the statement **Fix**.

Theorem 4.16 (Fix does not increase the generative power). *For every program P , there exists a Fix-free program $\text{Ffree}(P)$ such that $\text{Ffree}(P) \equiv P$.*

Construction. We introduce an intermediate program construct $\text{Fix}(P, p)$, where P is a program with some interface C and p a partial monomorphism with codomain C . The semantics is

$$\llbracket \text{Fix}(P, p) \rrbracket = \{ \langle m, m^* \circ (m_P \circ p), p^{-1} \rangle \mid \langle m, m^*, m_P \rangle \in \llbracket P \rrbracket \text{ and } (m_P \circ p) \in \mathcal{M} \}.$$

Construct $\text{Ffree}(P)$ by applying the following equations, interpreted as substitutions strictly from left to right, as long as possible: For a program P with interface C , let be $\text{Fix}(P) = \text{Fix}(P, \text{id}_C)$.

For a program P and partial monomorphism p , $\text{Fix}(P, p)$ is defined as follows: Let be $\text{Fix}(\text{Skip}, p) = \text{Uns}(p)$ if $p \in \mathcal{M}$, and $\text{Fix}(\text{Skip}, p) = \text{Abort}$

otherwise. If P is neither **Skip**, nor of the form $(Q; R)$ for arbitrary programs Q, R , then let $\text{Fix}(P, p) = \text{Fix}((P; \text{Skip}), p)$. Furthermore, let be

$$\begin{aligned}
\text{Fix}((\text{Skip}; R), p) &= \text{Fix}(R, p) \\
\text{Fix}((\text{Abort}; R), p) &= \text{Abort} \\
\text{Fix}((\text{Assert}(c); R), p) &= (\text{Assert}(c); \text{Fix}(R, p)) \\
\text{Fix}((\text{Sel}(x, c); R), p) &= (\text{Sel}(x, c); \text{Fix}(R, x \circ p)) \\
\text{Fix}((\text{Del}(l); R), p) &= (\text{Del}(l); \text{Fix}(R, l^{-1} \circ p)) \\
\text{Fix}((\text{Add}(r); R), p) &= (\text{Add}(r); \text{Fix}(R, r \circ p)) \\
\text{Fix}((\text{Uns}(y); R), p) &= (\text{Uns}(y); \text{Fix}(R, y^{-1} \circ p)) \\
\text{Fix}((\{P, \dots, Q\}; R), p) &= \bigcup_{S \in \{P, \dots, Q\}} \text{Fix}(S; R, p) \\
\text{Fix}((\text{Fix}(P); R), p) &= \text{Fix}(P); \text{Fix}(R, p) \\
\text{Fix}((P; Q; R), p) &= \text{Fix}((P; (Q; R)), p) \\
\text{Fix}((\text{if } c \text{ then } P \text{ fi}; R), p) &= \text{if } c \text{ then } \text{Fix}((P; R), p) \text{ else } \text{Fix}(R, p) \text{ fi} \\
\text{Fix}((\text{if } c \text{ then } P \text{ else } Q \text{ fi}; R), p) &= \text{if } c \text{ then } \text{Fix}((P; R), p) \\
&\quad \text{else } \text{Fix}((Q; R), p) \text{ fi} \\
\text{Fix}((P^0; R), p) &= \text{Fix}(R, p) \\
\text{Fix}((P^j; R), p) &= (P^j; \text{Fix}(R, p)) \\
\text{Fix}((P^*; R), p) &= (P^*; \text{Fix}(R, p)) \\
\text{Fix}((\downarrow P \downarrow; R), p) &= (\downarrow P \downarrow; \text{Fix}(R, p)) \\
\text{Fix}((\text{while } c \text{ do } P \text{ od}; R), p) &= (\text{while } c \text{ do } P \text{ od}; \text{Fix}(R, p))
\end{aligned}$$

Proof. See Appendix B. □

Fact 4.17 (idempotency of Fix). $\text{Fix}(\text{Fix}(P)) \equiv \text{Fix}(P)$.

Proof. By the semantics of the **Fix** construct, we have:

$$\begin{aligned}
&\llbracket \text{Fix}(\text{Fix}(P)) \rrbracket \\
&= \{ \langle m, m_{\text{Fix}}^* \circ m_{\text{Fix}}, \text{id} \rangle \mid \langle m, m_{\text{Fix}}^*, m_{\text{Fix}} \rangle \in \llbracket \text{Fix}(P) \rrbracket \\
&\quad \text{and } m_{\text{Fix}} \in \mathcal{M} \} \quad (\text{Def. } \llbracket \text{Fix} \rrbracket) \\
&= \{ \langle m, m^* \circ m_P \circ \text{id}, \text{id} \rangle \mid \langle m, m^* \circ m_P, \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \} \quad (\text{Def. } \llbracket \text{Fix} \rrbracket) \\
&= \{ \langle m, m^* \circ m_P, \text{id} \rangle \mid \langle m, m^*, m_P \rangle \in \llbracket P \rrbracket \text{ and } m_P \in \mathcal{M} \} \quad (\text{Def. } \llbracket \text{Fix} \rrbracket) \\
&= \text{Fix}(P) \quad (\text{Def. } \llbracket \text{Fix} \rrbracket)
\end{aligned}$$

□

4.3 Related concepts

Programs with interface relate easily to existing transformation concepts such as double pushout (DPO) transformation rules [Ehr79, CMR⁺97, EEPT06] and programs based thereon [HP01].

First, every transformation rule $\langle L \leftarrow K \hookrightarrow R \rangle$ with matching morphisms restricted to the morphism class \mathcal{M} , called \mathcal{M} -*matching* and \mathcal{M} -*matched* transformation rule, can be simulated by a sequence of elementary programs with initial interface I , as suggested by the definition. Let $G \Rightarrow_{\rho, m, m^*} H$ be a derivation as defined in [EEPT06].

Fact 4.18. For every transformation rule ρ , $G \Rightarrow_{\rho, m, m^*} H$ with $m, m^* \in \mathcal{M}$ iff $\langle i_G, i_H, id_I \rangle \in \llbracket \rho \rrbracket$.

Moreover, every program P over transformation rules with the matching morphism restricted to \mathcal{M} [HP01, PS04, HP09] can be seen as a program with the initial object as interface. For a program P , let $\llbracket P \rrbracket_{\text{HP01}} \subseteq \mathcal{O} \times \mathcal{O}$ be the semantics of programs, as defined in [HP01], where \mathcal{O} is the set of objects of a given category \mathcal{C} .

Fact 4.19. For every program P over \mathcal{M} -matched transformation rules, $\langle G, H \rangle$ in $\llbracket P \rrbracket_{\text{HP01}}$ if and only if $\langle i_G, i_H, id_I \rangle$ in $\llbracket P \rrbracket$.

Proof. By induction on the structure of programs over \mathcal{M} -matched transformation rules, we show that every program is syntactically a program with interface: For every set \mathbf{R} of DPO transformation rules, \mathbf{R} and $\downarrow \mathbf{R} \downarrow$ are programs with interface I . Given programs P, Q with interface I , then $(P; Q)$ is a program with interface I . By induction on the structure of programs over \mathcal{M} -matched transformation rules, we show that the program semantics correspond: We have $\langle G, H \rangle \in \llbracket \mathbf{R} \rrbracket_{\text{HP01}}$ iff $\langle G, H \rangle \in \bigcup_{\rho \in \mathbf{R}} \llbracket \rho \rrbracket_{\text{HP01}}$ iff there a $\rho \in \mathbf{R}$ such that $\langle G, H \rangle \in \llbracket \rho \rrbracket_{\text{HP01}}$ iff there a $\rho \in \mathbf{R}$ such that $\langle i_G, i_H, id_I \rangle \in \llbracket \rho \rrbracket$ iff $\langle i_G, i_H, id_I \rangle \in \bigcup_{\rho \in \mathbf{R}} \llbracket \rho \rrbracket$ iff $\langle i_G, i_H, id_I \rangle \in \llbracket \mathbf{R} \rrbracket$. Moreover, $\langle G, H \rangle \in \llbracket \downarrow \mathbf{R} \downarrow \rrbracket_{\text{HP01}}$ iff $\langle G, H \rangle \in \llbracket \mathbf{R} \rrbracket_{\text{HP01}}^*$ and there is no $M \in \mathcal{O}$ such that $\langle H, M \rangle \in \llbracket \mathbf{R} \rrbracket_{\text{HP01}}$ iff $\langle i_G, i_H, id_I \rangle \in \llbracket \mathbf{R}^* \rrbracket$ and there is no $i_M \in \mathcal{M}$ such that $\langle i_H, i_M, id_I \rangle \in \llbracket \text{Fix}(\mathbf{R}) \rrbracket$ iff $\langle i_G, i_H, id_I \rangle \in \llbracket \downarrow \mathbf{R} \downarrow \rrbracket$. Finally, $\langle G, H \rangle \in \llbracket (P; Q) \rrbracket_{\text{HP01}}$ iff $\langle G, M \rangle \in \llbracket P \rrbracket_{\text{HP01}}$ and $\langle M, H \rangle \in \llbracket Q \rrbracket_{\text{HP01}}$ iff $\langle i_G, i_M, id_I \rangle \in \llbracket P \rrbracket$ and $\langle i_M, i_H, id_I \rangle \in \llbracket Q \rrbracket$ iff $\langle i_G, i_M, id_I \rangle \in \llbracket (P; Q) \rrbracket$. \square

Graph programs over injectively matched, relabeling DPO graph transformation rules, as shown in [HP01], are computationally complete. However, the proof makes use of special relabeling rules, for instance $\langle \textcircled{S} \leftarrow \textcircled{1} \hookrightarrow \textcircled{T} \rangle$. Strictly speaking, the graphs and the morphisms used in these rules do not

fit to our definition of totally labeled graphs and label-preserving morphisms. And in contrast to edges, it is not possible to use rules such as $\langle \textcircled{S} \leftrightarrow \emptyset \hookrightarrow \textcircled{T} \rangle$ to delete nodes (temporarily) and recreate them with the new label: a node \textcircled{S} may have incident edges which effectively prevents its deletion.

However, relabeling can be simulated by a graph program with interface by selecting the node \textcircled{S}_1 , creating the node \textcircled{T}_2 and shifting any edge incident to \textcircled{S}_1 to \textcircled{T}_2 .

Fact 4.20 (simulation of relabeling). For a fixed, finite label alphabet $C = \langle C_V, C_E \rangle$, every relabeling transformation rule $\langle \textcircled{S}_1 \leftrightarrow \textcircled{}_1 \hookrightarrow \textcircled{T}_1 \rangle$ can be simulated by the following program

$$\begin{array}{l} \text{Sel}(\textcircled{S}_1); \text{Add}(\textcircled{S}_1 \textcircled{T}_2); \\ \downarrow \bigcup_{A \in C_V, a \in C_E} \text{MoveEdge}_{A,a} \cup \bigcup_{a \in C_E} \text{MoveLoop}_a \downarrow; \\ \text{Del}(\textcircled{S}_1 \textcircled{T}_2); \text{Uns}(\textcircled{T}_2) \end{array}$$

where for a node label $A \in C_V$ and edge label $a \in C_E$, the subprogram $\text{MoveEdge}_{A,a}$ is

$$\left\{ \begin{array}{l} (\text{Sel}(\textcircled{S}_1 \xrightarrow{a} \textcircled{A}_3 \textcircled{T}_2); \text{Del}(\textcircled{S}_1 \xrightarrow{a} \textcircled{A}_3 \textcircled{T}_2); \text{Add}(\textcircled{S}_1 \textcircled{A}_3 \xrightarrow{a} \textcircled{T}_2); \text{Uns}(\textcircled{S}_1 \textcircled{A}_3 \xleftarrow{a} \textcircled{T}_2)), \\ (\text{Sel}(\textcircled{S}_1 \xleftarrow{a} \textcircled{A}_3 \textcircled{T}_2); \text{Del}(\textcircled{S}_1 \xleftarrow{a} \textcircled{A}_3 \textcircled{T}_2); \text{Add}(\textcircled{S}_1 \textcircled{A}_3 \xrightarrow{a} \textcircled{T}_2); \text{Uns}(\textcircled{S}_1 \textcircled{A}_3 \xrightarrow{a} \textcircled{T}_2)), \end{array} \right\}$$

and the subprogram MoveLoop_a is

$$\{ (\text{Sel}(\textcircled{S}_1 \xrightarrow{a} \textcircled{T}_2); \text{Del}(\textcircled{S}_1 \xrightarrow{a} \textcircled{T}_2); \text{Add}(\textcircled{S}_1 \xrightarrow{a} \textcircled{T}_2); \text{Uns}(\textcircled{S}_1 \xleftarrow{a} \textcircled{T}_2)) \}.$$

As we can simulate the computationally complete graph programs considered in [HP01], every computable function on graphs can be computed by a graph program with interface.

Corollary 4.21 (computational completeness). Graph programs with interface are computationally complete.

It is possible to consider an extension of DPO transformation rules that provides the concept of explicit selection and deselection on the level of transformation rules, as investigated in [Pen08a].

Definition 4.22 (rules with external interface). A rule with external interface $\rho = \langle \langle X \hookrightarrow L \hookrightarrow K \hookrightarrow R \rangle, ac_L \rangle$ consists of a partial monomorphism $x: X \hookrightarrow L$, the external interface, two \mathcal{M} -morphisms $l: K \hookrightarrow L$, $r: K \hookrightarrow R$,

and a (left) *application condition* ac_L over L .

$$\begin{array}{ccccccc}
 X & \xrightarrow{x} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 & \searrow m & \downarrow m' \text{ (PO)} & & \downarrow d \text{ (PO)} & & \downarrow m^* \\
 & & G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H
 \end{array}$$

For an \mathcal{M} -morphism m , a triple $\langle m, m^*, r \circ l^{-1} \circ x \rangle$ is in the *semantics* of ρ , denoted by $\llbracket \rho \rrbracket$, if there is an \mathcal{M} -morphism $d: K \hookrightarrow D$ and two pushouts $\langle m', l^* \rangle$ and $\langle r^*, m^* \rangle$ such that $m = m' \circ x$ and $m' \models ac_L$. As \mathcal{M} -morphisms are closed under pushouts, the *match* $m': L \rightarrow G$ is in \mathcal{M} .

Fact 4.23 (rules with external interface). For every \mathcal{M} -matched rule with external interface $\rho = \langle \langle X \hookrightarrow L \hookleftarrow K \hookrightarrow R \rangle, ac_L \rangle$ with partial monomorphism $X \hookrightarrow L = \langle X \hookrightarrow Y \hookrightarrow L \rangle$, there is a program with interface $P = \text{Uns}(X \hookrightarrow Y); \text{Sel}(Y \hookrightarrow L, ac_L); \text{Del}(L \hookrightarrow K); \text{Add}(K \hookrightarrow R)$ such that $P \equiv \rho$.

Programs over transformation rules with external interface, as considered in [Pen08a], are defined similarly to programs with interface. In fact, every program over \mathcal{M} -matched rules with external interface can be simulated by a program with interface.

4.4 Summary and discussion

We use the framework of weak adhesive HLR categories and introduce *programs with interface*. Programs with interface can be seen as a generalization of programs over graph transformation rules, as considered in [HP01, PS04]. The main motivation for considering programs with interface is their ability to explicitly hand-over matching informations between computation steps. Instead of objects G , \mathcal{M} -morphisms $m: X \hookrightarrow G$ of a given weak adhesive HLR category are input and output of computation steps, representing a selection of elements X in G .

Consequently, the selection, deletion, addition and deselection of an object's elements are the basic program statements that can be composed to more complex programs by non-deterministic choice, sequential composition and iteration. The resulting programming language is computationally complete and is able to model transactions that deal with an unbounded number of elements. We use the features of this programming language to model the

dynamic behavior of an access control for computer systems and to implement a satisfiability solver for conditions, see Section 6.2.

An important program construct is the statement **Fix**(P) which is used to make a program P iterable. The effect of **Fix** on program P corresponds to a computation path-specific deselection of elements selected during the execution of P. However, this dynamic behavior makes static analysis difficult, especially the construction of weakest preconditions. Therefore, we show that **Fix** statements can be replaced by a specific **Uns** statement in every branch of a program, that is, every program can be converted into a program that does not contain **Fix**.

5. Correctness of program specifications

In the following we consider program specifications consisting of a precondition, a program with interface and a postcondition and define their correctness. With the intention of reducing the correctness problem of program specifications onto the implication problem of conditions, we define and show how to construct weakest liberal preconditions of programs with interface and conditions, similar to the ones for Dijkstra's guarded commands in [Dij76, DS89]. For the construction, we require a number of transformations on conditions that we introduce beforehand.

5.1 Program specifications

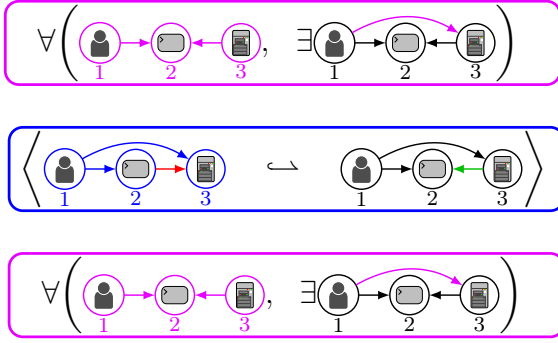
We are interested in formalizing the requirements of real-world systems in a graphical way. We have seen that graph conditions visually describe system state properties and that graph programs with interface provide a visual model of system transitions. As usual, a program specification consists of a (nested) precondition, a program with interface and a (nested) postcondition to specify the input/output behavior of systems and programs.

Definition 5.1 (program specification). A *program specification* is a triple $\{c\}P\{d\}$, where c, d are conditions and P is a program with interface $\text{dom}(c)$. The conditions c and d are called *precondition* and *postcondition*, respectively.

A program specification is correct, if all results of a program, starting with any input satisfying the precondition, satisfy the postcondition.

Definition 5.2 (correctness). A specification $\{c\}P\{d\}$ is *correct*, if for all \mathcal{M} -morphisms m with domain $\text{dom}(c)$ that satisfy c , $\langle m, m^*, m_p \rangle \in \llbracket P \rrbracket$ implies $m^* \models d$, for all \mathcal{M} -morphisms m^* and all partial monomorphisms m_p .

Example 5.3 (access control specification). The following access control specification expresses that if every user logged into a system has the appropriate access right (precondition) and the system grants a user access to a system to which he possesses an access right (program; more precisely, a transformation rule), then every user logged into a system has the appropriate access right (postcondition).



precondition: Every user logged into a system has the appropriate access right.

program: If a user with the appropriate access right proposes a session, it is accepted.

postcondition: Every user logged into a system has the appropriate access right.

While this specification seems to be correct, let us apply our formal methods to decide the correctness.

Correctness, as defined above, is more precisely *weak partial correctness*. The notion is called *weak*, as opposed to *strong* [Apt81], because it does not guarantee the existence of results (programs with interface may abort without results). The notion is called *partial*, as opposed to *total*, because the above definition does not guarantee the termination of P for every input satisfying c . In [HPR06], we consider weakest preconditions ensuring the termination of programs and the existence of results. In the following we will restrict ourselves to so-called weakest liberal preconditions ensuring (weak partial) correctness.

5.2 Basic transformations on conditions

In this section, we present transformations on conditions that we use later on for the construction of weakest preconditions, see Section 5.3, and for the definition of certain deduction rules of our calculus for proving conditions, see Section 6.3.

Let m be an \mathcal{M} -morphism, c be a condition and let $Cond$ denote the set of all conditions. We consider the following transformations that create or

modify a condition:

transformation	intuitive meaning
$A(m, c)$	shifts c along m ; conjunctively incorporates $\exists m$ into c
$\text{Del}(m, c)$	applies $\text{Del}(m)$ on every morphism of c
$\text{Deletable}(m)$	expresses the applicability of $\text{Del}(m)$ as a condition
$\text{Add}(m, c)$	applies $\text{Add}(m)$ on every morphism of c

First, we consider a transformation $A: \mathcal{M} \times \text{Cond} \rightarrow \text{Cond}$, that for an \mathcal{M} -morphism m , *shifts* a condition c over $\text{dom}(m)$ along m to yield a condition over $\text{codom}(m)$ representing a conjunctive combination of c and $\exists m$. This construction is used in Section 5.3 in the construction of weakest liberal preconditions for the program construct **Uns**, as well as in the definition of the deduction rules (Lift), (Partial lift), and (Supporting lift) in Section 6.3.

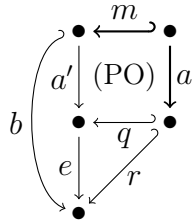
Lemma 5.4 (shifting of conditions along \mathcal{M} -morphisms). There is a transformation A , such that for all $m \in \mathcal{M}$ and every condition c over $\text{dom}(m)$ we have: For all $m'' \in \mathcal{M}$ with $\text{dom}(m'') = \text{codom}(m)$,

$$m'' \models A(m, c) \Leftrightarrow m'' \circ m \models c.$$

$$A(m, c) \triangleright \bullet \xleftarrow{m} \bullet \triangleleft c$$

The idea of the transformation A is a case differentiation of all possible overlappings of the additional elements in $\text{codom}(m)$ with the elements in the objects of the condition c . Transformation A generalizes the corresponding construction for “basic conditions” in [EEHP06], first described for graphs in [HW95]. At first view, it is similar to the construction of Theorem 3.12, but most importantly, as $m \in \mathcal{M}$, the pushout is guaranteed to exist.

Construction. For \mathcal{M} -morphisms m and conditions over $\text{dom}(m)$, let $A(m, \text{true}) = \text{true}$ and $A(m, \exists(a, c')) = \bigvee_{e \in \text{Epi}} \exists(b, A(r, c'))$, where $\langle a', q \rangle$ is the pushout of $\langle m, a \rangle$. The disjunction $\bigvee_{e \in \text{Epi}}$ ranges over all epimorphisms e with domain $\text{codom}(a')$ such that both $b = e \circ a'$ and $r = e \circ q$ are in \mathcal{M} . Furthermore, $A(m, \neg c') = \neg A(m, c')$ and $A(m, \bigwedge_{j \in J} c'_j) = \bigwedge_{j \in J} A(m, c'_j)$.



Example 5.5 (transformation A). Consider the access control condition

$$secure = \forall \left(\begin{array}{c} \text{Diagram 1} \\ \text{Diagram 2} \end{array} \right).$$

The transformation $A((\text{Diagram 1} \leftrightarrow I, secure))$ yields the following condition over Diagram 1 (superfluous subconditions are omitted):

$$ac_A = \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 1} \\ \text{Diagram 2} \end{array} \right) \wedge \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 3} \\ \text{Diagram 4} \end{array} \right) \wedge \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 5} \\ \text{Diagram 6} \end{array} \right) \wedge \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 7} \\ \text{Diagram 8} \end{array} \right) \wedge \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 9} \\ \text{Diagram 10} \end{array} \right) \wedge \bigvee \neg \exists \left(\begin{array}{c} \text{Diagram 11} \\ \text{Diagram 12} \end{array} \right)$$

Note that in the above condition, any occurrence of the connective \bigvee ranges over singletons.

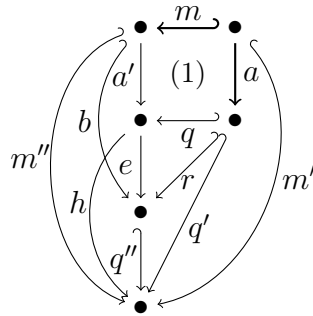
Proof of Lemma 5.4. By induction over the structure of conditions.

Basis. For $c = \text{true}$, we have $A(m, c) = A(m, \text{true}) = \text{true} = c$.

Hypothesis. Assume the statement holds for condition c' .

Step. Let $c = \exists(a, c')$.

Only if. Assume $m'' \models A(m, \exists(a, c')) = \bigvee_{e \in Epi} \exists(b, A(r, c'))$. There is an $e \in Epi$ such that $m'' \models \exists(b, A(r, c'))$. By definition of \models , there exists an \mathcal{M} -morphism q'' with $m'' = q'' \circ b$. Define $q' = q'' \circ r$. As $q'', r \in \mathcal{M}$ and \mathcal{M} closed under composition, we have $q' \in \mathcal{M}$. Let $m' = m'' \circ m$. As $m, m'' \in \mathcal{M}$ and \mathcal{M} closed under composition, $m' \in \mathcal{M}$. By construction, (1) is a pushout and we have $a' \circ m = q \circ a$, $r = e \circ q$ and $b = e \circ a'$. Together, $m'' \circ m = m' = q' \circ a$ and $m' = m'' \circ m \models \exists a$. Using the inductive hypothesis, $q'' \models A(m, \exists(a, c'))$ implies $q' = q'' \circ r \models c'$, we have $m' = m'' \circ m \models \exists(a, c')$.



If. Assume $m'' \circ m \models \exists(a, c')$. Let $m' = m'' \circ m$. As m'', m in \mathcal{M} and \mathcal{M} closed under composition, m' in \mathcal{M} . By definition of \models , there exists a morphism q' in \mathcal{M} with $m' = q' \circ a$. Following the construction, we yield the pushout (1) with object C' together with the morphisms a' and q . As $q' \circ a = m' = m'' \circ m$, the pushout guarantees the existence of a unique morphism h with $m'' = h \circ a'$ and $q' = h \circ q$. Consider $q'' \circ e = h$, an epi- \mathcal{M} -factorization of h with epimorphism e and \mathcal{M} -morphism q'' . Let $r = e \circ q$. As $q'' \circ r = q'$, q', q'' in \mathcal{M} and \mathcal{M} closed under decomposition, r in \mathcal{M} . Define $b = e \circ a'$. As m'', q'' are in \mathcal{M} and \mathcal{M} closed under decomposition, we have b is in \mathcal{M} . In every case, $m'' = h \circ a'$, $h = q'' \circ e$ and $b = e \circ a'$ yield $m'' = q'' \circ b$ ($m'' \models \bigvee_{e \in \text{Epi}} \exists b = A(m, \exists a)$). Using the inductive hypothesis, $q' = q'' \circ r \models c'$ implies $q'' \models A(m, \exists(a, c'))$, we have $m'' \models \bigvee_{e \in \text{Epi}} \exists(b, A(r, c')) = A(m, \exists(a, c'))$.

For Boolean formulas over conditions, the statement follows directly from the definitions and the inductive hypothesis. Thus, the statement holds for all conditions. \square

Next, we consider a transformation $\text{Del}: \mathcal{M} \times \text{Cond} \rightarrow \text{Cond}$ that takes an \mathcal{M} -morphism r and a condition c as parameters and yields a condition c^* representing the pushout complement of c with respect to r . In this sense, the pushout complement construction $\mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M} \times \mathcal{M}$, as defined in Def. 2.11, is lifted from morphisms to conditions and Del can be seen as a generalization of the program construct Del from morphisms to conditions. The following lemma states that for every transition $\langle m^*, m, r \rangle$ in the semantics of $\text{Add}(r)$, the input m^* satisfies c^* if and only if the output m satisfies condition c . This relation is used in Section 5.3 in the construction of weakest liberal preconditions for the program construct Add .

Lemma 5.6 (pushout complement of conditions). There is a transformation Del such that, for every \mathcal{M} -morphism r , every condition c , and every tuple $\langle m^*, m, r \rangle \in \llbracket \text{Add}(r) \rrbracket$,

$$m^* \models \text{Del}(r, c) \text{ iff } m \models c.$$

$$\text{Del}(r, c) \triangleright \bullet \xrightarrow{r} \bullet \triangleleft c$$

The effect of the transformation is the deletion of elements as specified by r . The logical structure of the input condition is either preserved or truncated, depending on whether or not all pushout complements exist.

Construction. Transformation $\text{Del}(r, c)$ is defined inductively as follows: $\text{Del}(r, \text{true}) = \text{true}$ and $\text{Del}(r, \exists(a, c')) = \exists(a^*, \text{Del}(r^*, c'))$ if the pushout

complement $r^* \circ a^*$ of $a \circ r$ exists, otherwise $\text{Del}(r, \exists(a, c')) = \text{false}$. Note that as r is in \mathcal{M} , the pushout complement is unique, if existent. Furthermore, $\text{Del}(r, \neg c') = \neg \text{Del}(r, c')$ and $\text{Del}(r, \bigwedge_{j \in J} c'_j) = \bigwedge_{j \in J} \text{Del}(r, c'_j)$.

$$\begin{array}{ccc} \bullet & \xrightarrow{r} & \bullet \\ a^* \downarrow & \text{(POC)} & \downarrow a \\ \bullet & \xrightarrow{r^*} & \bullet \end{array}$$

Example 5.7 (transformation Del). Consider the condition ac_A constructed in Example 5.5. The transformation $\text{Del}(\text{Diagram}_1 \leftrightarrow \text{Diagram}_2, ac_A)$ yields the following condition over Diagram_1 :

$$ac_{\text{Del}} = \bigvee \neg \exists \left(\text{Diagram}_1, \text{Diagram}_2, \text{Diagram}_3 \right) \wedge \bigvee \neg \exists \left(\text{Diagram}_1, \text{Diagram}_2, \text{Diagram}_3 \right) \wedge \bigvee \neg \exists \left(\text{Diagram}_1, \text{Diagram}_2, \text{Diagram}_3 \right) \wedge \bigvee \neg \exists \left(\text{Diagram}_1, \text{Diagram}_2, \text{Diagram}_3 \right) \wedge \bigvee \neg \exists \left(\text{Diagram}_1, \text{Diagram}_2, \text{Diagram}_3 \right)$$

Proof. By structural induction. Let $\langle m^*, m, r \rangle \in \llbracket \text{Add}(r) \rrbracket$ be arbitrary and let $\langle r^*, m \rangle$ be the pushout of $\langle m^*, r \rangle$.

Basis. For $c = \text{true}$, we have $\text{Del}(r, c) = \text{Del}(r, \text{true}) = \text{true} = c$.

Hypothesis. Assume the statement holds for condition c' .

Step. Case $c = \exists(a, c')$ and the pair $a \circ r$ has a pushout complement.

Only if. Assume $m^* \models \text{Del}(r, \exists(a, c')) = \exists(a^*, \text{Del}(r^*, c'))$. There exists an \mathcal{M} -morphism q^* such that $q^* \circ a^* = m^*$ and $q^* \models \text{Del}(r^*, c')$. According to the construction, $\langle r^*, a \rangle$ is the pushout of $\langle a^*, r \rangle$. As \mathcal{M} is closed under pushouts, $r \in \mathcal{M}$ implies $r^* \in \mathcal{M}$. By the universal property of pushouts, there exists a unique morphism q such that the arising diagrams commute. As $q \circ a = m$ and $q \models c'$ (Hypothesis), we conclude $m \models \exists(a, c')$.

If. Assume $m \models \exists(a, c')$. There exists an \mathcal{M} -morphism q such that $q \circ a = m$ and $q \models c'$. Construct $\langle q^*, r^*, \rangle$ as the pullback of $\langle r^*, q \rangle$. By the universal property of pullbacks, there exists a unique morphism $a': \text{dom}(a^*) \rightarrow \text{codom}(a^*)$ such that the arising diagrams commute. Let $(1')$ denote $a \circ r = r^* \circ a'$. By the pushout-pullback decomposition, $(1')$ and (2) are pushouts. As pushouts are unique up to isomorphisms, $a' = a^*$ and $(1')$ equals (1) up

to isomorphism. As \mathcal{M} is closed under pushouts, $r \in \mathcal{M}$ implies $r^* \in \mathcal{M}$. As $q \circ a = m$ and $q \models c'$ (Hypothesis), we conclude $m \models \exists(a, c')$.

$$\begin{array}{ccc}
 & \bullet & \xrightarrow{r} \bullet \\
 m^* \swarrow & \downarrow a^* & (1) \downarrow a \searrow m \\
 & \bullet & \xrightarrow{r^*} \bullet \\
 & \downarrow q^* & (2) \downarrow q \\
 & \bullet & \xrightarrow{r^{**}} \bullet
 \end{array}$$

Case $c = \exists(a, c')$ and the pair $a \circ r$ has no pushout complement. In this case, $\text{Del}(r, \exists(a, c')) = \text{false}$. We have to show that $m^* \models \text{false} \Leftrightarrow m \models \exists(a, c')$: As no morphism satisfies false, it suffices to show $m \not\models \exists(a, c')$. Assume $m \models \exists(a, c')$. Then there exists some \mathcal{M} -morphism q with $q \circ a = m$. Thus there is a decomposition of the existing pushout into two pushouts (1) and (2) as above. Hence the pair $a \circ r$ has a pushout complement. Contradiction. For Boolean formulas over conditions, the statement follows directly from the definition and the inductive hypothesis. This concludes the proof. \square

As pushouts exists along \mathcal{M} -morphisms, that is, $\forall m^* \exists m. \langle m^*, m, r \rangle \in \llbracket \text{Add}(r) \rrbracket$, we can strengthen the statement of Lemma 5.6.

Corollary 5.8 (pushout complement of conditions). There is a transformation Del such that, for every \mathcal{M} -morphisms r, m^* , and every condition c ,

$$m^* \models \text{Del}(r, c) \text{ iff } (\forall m \in \mathcal{M}. \langle m^*, m, r \rangle \in \llbracket \text{Add}(r) \rrbracket \text{ implies } m \models c).$$

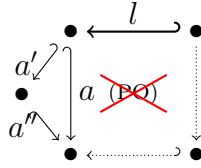
Next, we consider a transformation $\text{Deletable}: \mathcal{M} \rightarrow \text{Cond}$ that creates a condition expressing the applicability of the $\text{Del}(l)$ program construct. More precisely, the existence of a pushout complement for a pair of morphisms $m \circ l$ is expressed by a condition over $\text{codom}(l)$.

Lemma 5.9 (existence of pushout complements). There is a transformation Deletable such that, for every pair of \mathcal{M} -morphisms $m \circ l$,

$$m \models \text{Deletable}(l) \text{ iff } \exists m^*. \langle m, m^*, l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket.$$

Using conditions, one cannot directly express that a pushout complement exists. However, it is possible to consider the negation of a minimal covering of negative conditions for which pushout complements are guaranteed not to exist.

Construction. For an \mathcal{M} -morphism l , let $\text{Deletable}(l) = \bigwedge_{a \in A} \neg \exists a$ and the set A ranges over all \mathcal{M} -morphisms a with domain $\text{codom}(l)$ such that the pair $a \circ l$ has no pushout complement and whenever there is a decomposition $a = a'' \circ a'$ of morphism a with $a' \in \mathcal{M}$ (which implies $a'' \in \mathcal{M}$), \mathcal{M} -morphism a'' is an isomorphism or $a' \circ l$ has a pushout complement. The non-existence of such a decomposition ensures that we only consider minimal morphisms a . The morphism a'' is required to be non-isomorphic, otherwise there always is such a decomposition and A would be empty. The requirement a'' in \mathcal{M} is necessary in context of an \mathcal{M} -satisfiable condition. We can restrict A to a set of \mathcal{M} -morphisms as the lemma assumes m to be in \mathcal{M} .



Example 5.10 (transformation Deletable). For a morphism m with domain $\textcircled{\text{u}}$, an application of the program $\text{Del}(\textcircled{\text{u}} \leftrightarrow \emptyset)$ requires the absence of edges adjacent to the deleted user node. $\text{Deletable}(\emptyset \hookrightarrow \textcircled{\text{u}})$ yields the following condition over $\textcircled{\text{u}}$:

$$\begin{aligned} \text{Deletable}(\emptyset \hookrightarrow \textcircled{\text{u}}) = & \neg \exists (\textcircled{\text{u}} \rightarrow \textcircled{\text{u}}) \wedge \neg \exists (\textcircled{\text{u}} \leftarrow \textcircled{\text{u}}) \wedge \neg \exists (\textcircled{\text{u}} \rightarrow \textcircled{\text{u}}) \\ & \wedge \neg \exists (\textcircled{\text{u}} \leftarrow \textcircled{\text{u}}) \wedge \neg \exists (\textcircled{\text{u}} \rightarrow \textcircled{\text{u}}) \wedge \neg \exists (\textcircled{\text{u}} \leftarrow \textcircled{\text{u}}) \wedge \neg \exists (\textcircled{\text{u}} \rightarrow \textcircled{\text{u}}) \end{aligned}$$

Remark 5.11. For the category $\langle \text{Graphs}, \text{Inj} \rangle$, a morphism m satisfies the condition $\text{Deletable}(l)$ iff m satisfies the so-called *contact and identification condition* [Ehr79]. The graph condition $\text{Deletable}(l)$ is finite for every graph morphism l , that is, a finite conjunction of conditions (up to isomorphism).

Proof. Only if. Assume the pair $m \circ l$ has no pushout complement. Let $a = m$ and $m' = \text{id}_{\text{codom}(a)}$. Now, we have some \mathcal{M} -morphism a with $m = m' \circ a$ for some \mathcal{M} -morphism m' such that $a \circ l$ has no pushout complement. If there is a decomposition $a = a'' \circ a'$ of a such that $a' \in \mathcal{M}$, $a' \circ l$ has no pushout complement and a'' in \mathcal{M} is not an isomorphism, let be $a = a'$, $m' = m' \circ a''$, m' in \mathcal{M} and repeat the argument. As there are only finitely many \mathcal{M} -decompositions, eventually there is no such decomposition and a belongs to the construction. As $m = m' \circ a$ and m' in \mathcal{M} , $m \models \exists a$ and $m \not\models \text{Deletable}(l)$.

If. Assume the pair $m \circ l$ has a pushout complement $l^{**} \circ m^*$, but $m \not\models \text{Deletable}(l)$. Then there exists an $a \in A$ such that $m \models \exists a$, that is, there is an \mathcal{M} -morphism m' such that $m = m' \circ a$. Construct as pullback of $\langle m', l^{**} \rangle$. By the universal property of pullbacks, there is a morphism a^* such that the

resulting diagrams commute. By the pushout-pullback decomposition, the pushout (1)+(2) has a decomposition into two pushouts (1) and (2) and, in particular, $a \circ l$ has a pushout complement, contradiction. Consequently, for every morphism $a \in A$, $m \models \neg \exists a$ implying $m \models \text{Deletable}(l)$.

$$\begin{array}{ccc}
 \bullet & \xleftarrow{l} & \bullet \\
 \downarrow a & (1) & \downarrow a^* \\
 \bullet & \xleftarrow{\quad} & \bullet \\
 \downarrow m & (2) & \downarrow \\
 \bullet & \xleftarrow{l^*} & \bullet
 \end{array}
 \begin{array}{l}
 m \\
 \\
 m^*
 \end{array}$$

□

Next, we consider a transformation $\text{Add}: \mathcal{M} \times \text{Cond} \rightarrow \text{Cond}$ that takes an \mathcal{M} -morphism l and a condition c as parameters and yields a condition c^* representing the pushout of c with respect to r . In this sense, the notion of pushout, as defined in Def. 2.8, is lifted from morphisms to conditions and Add can be seen as a generalization of the program construct **Add** from morphisms to conditions. The following lemma states that for every transition $\langle m, m^*, l^{-1} \rangle$ in the semantics of $\text{Del}(l)$, the input m satisfies c^* if and only if the output m^* satisfies condition c . Together with Lemma 5.9, it is used in Section 5.3 in the construction of weakest liberal preconditions for the program construct **Del**.

Lemma 5.12 (pushout of conditions). There is a transformation Add such that, for every \mathcal{M} -morphism l , every condition c , and every tuple $\langle m^*, m, l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket$,

$$m^* \models \text{Add}(l, c) \text{ iff } m \models c.$$

$$\text{Add}(l, c) \triangleright \bullet \xleftarrow{l} \bullet \triangleleft c$$

The effect of the transformation is the addition of elements as specified by l . The logical structure of the input condition is preserved.

Construction. Transformation $\text{Add}(l, c)$ is defined inductively as follows: $\text{Add}(l, \text{true}) = \text{true}$ and $\text{Add}(l, \exists(a, c')) = \exists(a^*, \text{Add}(l^*, c'))$ where $\langle a^*, l^* \rangle$ is the pushout of $\langle l, a \rangle$. For Boolean conditions, we have $\text{Add}(l, \neg c') = \neg \text{Add}(l, c')$ and $\text{Add}(l, \bigwedge_{j \in J} c'_j) = \bigwedge_{j \in J} \text{Add}(l, c'_j)$.

$$\begin{array}{ccc}
 \bullet & \xleftarrow{l} & \bullet \\
 a^* \downarrow & (\text{PO}) & \downarrow a \\
 \bullet & \xleftarrow{l^*} & \bullet
 \end{array}$$

Example 5.13 (transformation Add). Consider the condition ac_{Del} constructed in Example 5.7. The transformation $\text{Add}((\text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3) \leftarrow (\text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3), ac_{\text{Del}})$ yields the following condition over $(\text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3)$:

$$ac_{\text{Add}} = \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

$$\wedge \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

$$\wedge \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

$$\wedge \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

$$\wedge \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

$$\wedge \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} , \bigvee \neg \exists \left(\begin{array}{c} \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \\ \text{person}_1 \rightarrow \text{car}_2 \rightarrow \text{car}_3 \end{array} \right) \right)$$

Proof. By structural induction. Let $\langle m^*, m, l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket$ be arbitrary and let $l^{**} \circ m$ be the pushout complement of $m^* \circ l$.

Basis. For $c = \text{true}$, we have $\text{Add}(l, c) = \text{Add}(l, \text{true}) = \text{true} = c$.

Hypothesis. Assume the statement holds for condition c' .

Step. Case $c = \exists(a, c')$.

Only if. Assume $m^* \models \text{Add}(l, \exists(a, c')) = \exists(a^*, \text{Add}(l^*, c'))$. There exists an \mathcal{M} -morphism q^* such that $q^* \circ a^* = m^*$ and $q^* \models \text{Add}(l^*, c')$. Construct $\langle l^*, q \rangle$ as the pullback of $\langle q^*, l^{**} \rangle$. By the universal property of pullbacks, there exists a unique morphism $a': \text{dom}(a) \rightarrow \text{codom}(a)$ such that the arising diagrams commute. Let (1') denote $a^* \circ l = l^* \circ a'$. By the pushout-pullback decomposition, (1') and (2) are pushouts. As pushouts are unique up to isomorphisms, $a' = a$ and (1') equals (1) up to isomorphism. As \mathcal{M} is closed under pushouts, $l \in \mathcal{M}$ implies $l^* \in \mathcal{M}$. As $q \circ a = m$ and $q \models c'$ (Hypothesis), we conclude $m \models \exists(a, c')$.

$$\begin{array}{ccc} & l & \\ m^* \left(\begin{array}{ccc} \bullet & \xleftarrow{\quad} & \bullet \\ a^* \downarrow & (1) & \downarrow a \\ \bullet & \xleftarrow{\quad} & \bullet \\ q^* \downarrow & (2) & \downarrow q \\ \bullet & \xleftarrow{\quad} & \bullet \\ & l^{**} & \end{array} \right) m \end{array}$$

If. Assume $m \models \exists(a, c')$. There exists an \mathcal{M} -morphism q such that $q \circ a = m$ and $q \models c'$. According to the construction, $\langle a^*, l^* \rangle$ is the pushout of $\langle l, a \rangle$. As \mathcal{M} is closed under pushouts, $l \in \mathcal{M}$ implies $l^* \in \mathcal{M}$. By the universal

property of pushouts, there exists a unique morphism q^* such that the arising diagrams commute. As $q^* \circ a^* = m^*$ and $q^* \models \text{Add}(l^*, c')$ (Hypothesis), we conclude $m^* \models \exists(a^*, \text{Add}(l^*, c'))$.

Case $c = \neg c'$. We have $m^* \models \text{Add}(l, c)$ iff $m^* \models \text{Add}(l, \neg c') = \neg \text{Add}(l, c')$ iff $m \models \neg c'$.

Case $c = \bigwedge_{j \in J} c'_j$. We have $m^* \models \text{Add}(l, c)$ iff $m^* \models \text{Add}(l, \bigwedge_{j \in J} c'_j) = \bigwedge_{j \in J} \text{Add}(l, c'_j)$ iff $m \models \bigwedge_{j \in J} c'_j$. This concludes the proof. \square

As we can express the applicability of **Del** construct as a condition, see Lemma 5.9, we can strengthen the statement of Lemma 5.12.

Corollary 5.14 (pushout of conditions). There is a transformation **Add** such that, for every \mathcal{M} -morphisms l, m^* , and every condition c ,

$$\begin{aligned} m^* &\models (\text{Deletable}(l) \Rightarrow \text{Add}(l, c)) \\ \text{iff } (\forall m \in \mathcal{M}. \langle m^*, m, r \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ implies } m &\models c). \end{aligned}$$

5.3 Weakest liberal preconditions

In the following we consider weakest liberal preconditions to reduce the correctness problem of program specifications into the implication problem of conditions. A weakest liberal precondition of a program P and postcondition d is a least restrictive precondition still ensuring correctness of P with respect to d .

Definition 5.15 (weakest liberal preconditions). A *liberal precondition* for a program P and a postcondition d is a condition c such that a program specification $\{c\}P\{d\}$ is correct. A *weakest liberal precondition* of a program P and a postcondition d , denoted by $\text{wlp}(P, d)$, is a liberal precondition such that any other liberal precondition of P relative to d implies $\text{wlp}(P, d)$.

One way to prove the correctness of a program specification $\{c\}P\{d\}$ is to prove that c really is a (liberal) precondition of P and d . A weakest liberal precondition covers all necessary premises for a program with respect to a postcondition, thus it suffices to prove that c is stronger than, or equivalently, implies $\text{wlp}(P, d)$. In this sense, the correctness problem of program specifications can be reduced onto the implication problem of conditions by constructing weakest preconditions, as depicted in Figure 5.1.

A weakest precondition is a precondition that is not more restrictive than necessary. In this sense, the following characterization points out a simple

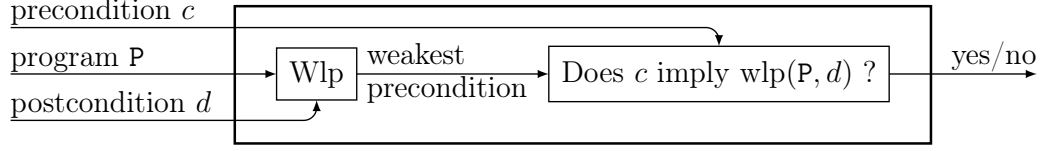


Figure 5.1: Decider for the correctness problem

proof scheme for weakest liberal preconditions. When comparing the characterization with Definition 5.2, notice that the attribute “weakest” is reflected by the “only if” of the following iff-statement.

Fact 5.16 (characterization wlp). A condition c over C is a weakest liberal precondition of program P with interface C relative to condition d if, for all morphisms m in \mathcal{M} with domain C , $(m \models c)$ iff $(\langle m, m^*, m_P \rangle \in \llbracket P \rrbracket$ implies $m^* \models d$, for all m^*, m_P (where $m^* \in \mathcal{M}$, $m_P \in \mathcal{P}$)).

Weakest liberal preconditions can be constructed for programs with interface. The following theorem is a major result of this thesis.

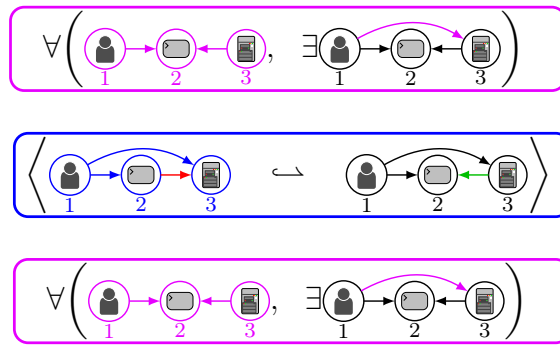
Theorem 5.17 (weakest liberal preconditions). *For every program P with interface C and condition d , a condition $\text{Wlp}(P, d)$ over C can be constructed that is a weakest liberal precondition of P and d , that is, $\text{Wlp}(P, d) \equiv \text{wlp}(P, d)$.*

Construction. The Wlp is defined by induction over the structure of programs:

$$\begin{aligned}
\text{Wlp}(\text{Skip}, d) &= d \\
\text{Wlp}(\text{Abort}, d) &= \text{true} \\
\text{Wlp}(\text{Assert}(c), d) &= (c \Rightarrow d) \\
\text{Wlp}(\text{Sel}(x, c), d) &= \forall(x, (c \Rightarrow d)) \\
\text{Wlp}(\text{Del}(l), d) &= \text{Deletable}(l) \Rightarrow \text{Add}(l, d) \\
\text{Wlp}(\text{Add}(r), d) &= \text{Del}(r, d) \\
\text{Wlp}(\text{Uns}(y), d) &= A(y, d) \\
\text{Wlp}(\text{Fix}(P), d) &= \text{Wlp}(\text{Ffree}(\text{Fix}(P)), d) \\
\text{Wlp}(\{P, \dots, Q\}, d) &= \bigwedge_{R \in \{P, \dots, Q\}} \text{Wlp}(R, d) \\
\text{Wlp}(P; Q, d) &= \text{Wlp}(P, \text{Wlp}(Q, d)) \\
\text{Wlp}(\text{if } c \text{ then } P \text{ fi}, d) &= (c \Rightarrow \text{Wlp}(P, d)) \wedge (\neg c \Rightarrow d) \\
\text{Wlp}(\text{if } c \text{ then } P \\
&\quad \text{else } Q \text{ fi}, d) &= (c \Rightarrow \text{Wlp}(P, d)) \wedge (\neg c \Rightarrow \text{Wlp}(Q, d)) \\
\text{Wlp}(P^0, d) &= d \\
\text{Wlp}(P^j, d) &= \text{Wlp}(\text{Fix}(P), \text{Wlp}(P^{j-1}, d)) \\
\text{Wlp}(P^*, d) &= \bigwedge_{j=0}^{\infty} \text{Wlp}(P^j, d) \\
\text{Wlp}(\downarrow P \downarrow, d) &= \text{Wlp}(P^*, \text{Wlp}(\text{Fix}(P), \text{false})) \Rightarrow d \\
\text{Wlp}(\text{while } c \text{ do } P \text{ od}, d) &= \text{Wlp}((\text{Assert}(c); P)^*, \neg c \Rightarrow d)
\end{aligned}$$

Furthermore, $\text{Wlp}(\langle \langle L \leftarrow K \hookrightarrow R \rangle, ac_L \rangle, d) = \forall(i_L, (\text{Deletable}(L \leftarrow K) \wedge ac_L) \Rightarrow \text{Add}(L \leftarrow K, \text{Del}(K \hookrightarrow R, A(i_R, d))))$.

Example 5.18 (access control wlp). Consider the access control specification of Example 5.3:



precondition: Every user logged into a system has the appropriate access right.

program: If a user with the appropriate access right proposes a session, it is accepted.

postcondition: Every user logged into a system has the appropriate access right.

To decide its correctness, we now want to construct a weakest liberal precondition of the above rule and postcondition, to which we refer to as **Access** and *secure*, respectively. As the rule's (implicit) application condition is true

and $\text{Deletable}((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \leftrightarrow (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3)) = \text{true}$ (deletion of edges is always possible), the weakest liberal precondition reduces to

$$\text{Wlp}(\text{Access}, \text{secure}) = \forall(\emptyset \leftrightarrow (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3), \\ \text{Add}((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \leftrightarrow (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3), \\ \text{Del}((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \leftrightarrow (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3), \\ \text{A}((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \leftrightarrow \emptyset, \text{secure}))))).$$

Taking the results of Example 5.5-5.13 into consideration, we yield the following condition over \emptyset :

$$\begin{aligned} ac_{\text{wlp}} = & \quad \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \quad (\text{person}_1 \rightarrow \text{phone}_2 \leftarrow \text{phone}_3), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \quad (\text{person}_1 \rightarrow \text{phone}_2 \leftarrow \text{phone}_3) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \leftarrow \text{phone}_5), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \leftarrow \text{phone}_5) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \leftarrow \text{phone}_6), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \leftarrow \text{phone}_6) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6 \leftarrow \text{phone}_7), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6 \leftarrow \text{phone}_7) \right) \end{aligned}$$

Consider furthermore the condition $\text{noSharing} = \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \leftarrow \text{person}_3)$. The weakest liberal precondition $\text{Wlp}(\text{Access}, \text{secure} \wedge \text{noSharing})$ is

$$\begin{aligned} ac2_{\text{wlp}} = & \quad \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \quad (\text{person}_1 \rightarrow \text{phone}_2 \leftarrow \text{phone}_3), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3) \quad (\text{person}_1 \rightarrow \text{phone}_2 \leftarrow \text{phone}_3) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \leftarrow \text{phone}_5), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \leftarrow \text{phone}_5) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5) \right) \\ & \wedge \forall \neg \exists \left((\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \leftarrow \text{phone}_6), \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \leftarrow \text{phone}_6) \right) \\ & \wedge \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6 \leftarrow \text{person}_7) \\ & \wedge \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6 \leftarrow \text{person}_7) \\ & \wedge \forall \neg \exists (\text{person}_1 \rightarrow \text{phone}_2 \rightarrow \text{phone}_3 \rightarrow \text{phone}_4 \rightarrow \text{phone}_5 \rightarrow \text{phone}_6 \leftarrow \text{person}_7) \end{aligned}$$

Proof of Theorem 5.17. By induction over the structure of programs.
Basis.

Case **Se1**. For every morphism m in \mathcal{M} :

$$m \models \text{wlp}(\text{Se1}(x, c), d)$$

$$\begin{aligned}
& \text{iff } \forall m^*, m_{\text{Sel}}. (\langle m, m^*, m_{\text{Sel}} \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } \forall m^*. ((m^* \circ x = m \text{ and } m^* \in \mathcal{M} \text{ and } m^* \models c) \\
& \quad \text{implies } m^* \models d) & (\text{Def. } \llbracket \text{Sel} \rrbracket) \\
& \text{iff } \forall m^*. ((m^* \circ x = m \text{ and } m^* \in \mathcal{M}) \\
& \quad \text{implies } (m^* \models c \text{ implies } m^* \models d)) & \left(\begin{smallmatrix} ((F \wedge G) \Rightarrow H) \\ \equiv (F \Rightarrow (G \Rightarrow H)) \end{smallmatrix} \right) \\
& \text{iff } \forall m^*. ((m^* \circ x = m \text{ and } m^* \in \mathcal{M}) \text{ implies } m^* \models (c \Rightarrow d)) & (\text{Def. } \models) \\
& \text{iff } m \models \forall(x, (c \Rightarrow d)) & (\text{Fact 3.1}) \\
& \text{iff } m \models \text{Wlp}(\text{Sel}(x, c), d) & (\text{Def. Wlp})
\end{aligned}$$

Case **Del**. For every morphism m in \mathcal{M} :

$$\begin{aligned}
& m \models \text{wlp}(\text{Del}(l), d) \\
& \text{iff } \forall m^*, m_{\text{Del}}. (\langle m, m^*, m_{\text{Del}} \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } \forall m^*. (\langle m, m^*, l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } m \models (\text{Deletable}(l) \Rightarrow \text{Add}(l, d)) & (\text{Cor. 5.14}) \\
& \text{iff } m \models \text{Wlp}(\text{Del}(l), d) & (\text{Def. Wlp})
\end{aligned}$$

Case **Add**. For every morphism m in \mathcal{M} :

$$\begin{aligned}
& m \models \text{wlp}(\text{Add}(r), d) \\
& \text{iff } \forall m^*, m_{\text{Add}}. (\langle m, m^*, m_{\text{Add}} \rangle \in \llbracket \text{Add}(r) \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } \forall m^*. (\langle m, m^*, r \rangle \in \llbracket \text{Add}(r) \rrbracket \text{ implies } m^* \models d) & (\text{Def. } \llbracket \text{Add} \rrbracket) \\
& \text{iff } m \models \text{Del}(r, d) & (\text{Cor. 5.8}) \\
& \text{iff } m \models \text{Wlp}(\text{Add}(r), d) & (\text{Def. Wlp})
\end{aligned}$$

Case **Uns**. For every morphism m in \mathcal{M} :

$$\begin{aligned}
& m \models \text{wlp}(\text{Uns}(y), d) \\
& \text{iff } \forall m^*, m_{\text{Uns}}. (\langle m, m^*, m_{\text{Uns}} \rangle \in \llbracket \text{Uns}(y) \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } \forall m^*. ((m^* = m \circ y) \text{ implies } m^* \models d) & (\text{Def. } \llbracket \text{Uns} \rrbracket) \\
& \text{iff } m \circ y \models d & (m^* = m \circ y) \\
& \text{iff } m \models \text{A}(y, d) & (\text{Lem. 5.4}) \\
& \text{iff } m \models \text{Wlp}(\text{Uns}(y), d) & (\text{Def. Wlp})
\end{aligned}$$

Case **Assert**(c).

$$\begin{aligned}
& \text{wlp}(\text{Assert}(c), d) \\
& = \text{wlp}(\text{Sel}(\text{id}_C, c), d) & (\text{Def. Assert}) \\
& = \text{Wlp}(\text{Sel}(\text{id}_C, c), d) & (\text{Hypothesis}) \\
& = \forall(\text{id}_C, (c \Rightarrow d)) & (\text{Def. Wlp}) \\
& \equiv (c \Rightarrow d) & (\text{Table 6.3})
\end{aligned}$$

Case **Skip**.

$$\begin{aligned}
& \text{wlp}(\text{Skip}, d) \\
& = \text{wlp}(\text{Assert}(\text{true}), d) & (\text{Def. Skip}) \\
& \equiv \text{Wlp}(\text{Assert}(\text{true}), d) & (\text{Hypothesis}) \\
& = (\text{true} \Rightarrow d) & (\text{Def. Wlp}) \\
& \equiv d
\end{aligned}$$

Case **Abort**.

$$\begin{aligned}
& \text{wlp}(\text{Abort}, d) \\
= & \text{wlp}(\text{Assert}(\text{false}), d) && (\text{Def. Abort}) \\
\equiv & \text{Wlp}(\text{Assert}(\text{false}), d) && (\text{Hypothesis}) \\
= & (\text{false} \Rightarrow d) && (\text{Def. Wlp}) \\
\equiv & \text{true}
\end{aligned}$$

Case $\rho = \langle \langle L \leftarrow K \hookrightarrow R \rangle, ac_L \rangle$.

$$\begin{aligned}
& \text{wlp}(\rho, d) \\
= & \text{wlp}(\text{Sel}(i_L, ac_L); \text{Del}(L \leftarrow K); \text{Add}(K \hookrightarrow R); \text{Uns}(i_R), d) && (\text{Def. } \rho) \\
\equiv & \text{Wlp}(\text{Sel}(i_L, ac_L); \text{Del}(L \leftarrow K); \text{Add}(K \hookrightarrow R); \text{Uns}(i_R), d) && (\text{Hypothesis}) \\
= & \forall(i_L, (\text{Deletable}(L \leftarrow K) \wedge ac_L) \\
& \quad \Rightarrow \text{Add}(L \leftarrow K, \text{Del}(K \hookrightarrow R, A(i_R, d)))) && (\text{Def. Wlp})
\end{aligned}$$

Hypothesis. Assume $\text{Wlp}(\mathbf{R}, d) = \text{wlp}(\mathbf{R}, d)$ for $\mathbf{R} \in \{\mathbf{P}, \dots, \mathbf{Q}\}$.

Step.

Case **Fix**(\mathbf{P}). As we have shown in Section 4.2, for every program **Fix**(\mathbf{P}), there is an equivalent, **Fix**-free program **Ffree**(**Fix**(\mathbf{P})).

$$\begin{aligned}
& \text{wlp}(\text{Fix}(\mathbf{P}), d) \\
\equiv & \text{wlp}(\text{Ffree}(\text{Fix}(\mathbf{P})), d) && (\text{Theorem 4.16}) \\
\equiv & \text{Wlp}(\text{Ffree}(\text{Fix}(\mathbf{P})), d) && (\text{Hypothesis})
\end{aligned}$$

Case $\mathcal{S} = \{\mathbf{P}, \dots, \mathbf{Q}\}$.

$$\begin{aligned}
& m \models \text{wlp}(\mathcal{S}, d) \\
\text{iff } & \forall m^*, m_{\mathcal{S}}. (\langle m, m^*, m_{\mathcal{S}} \rangle \in \llbracket \mathcal{S} \rrbracket \text{ implies } m^* \models d) && (\text{Def. wlp}) \\
\text{iff } & \forall m^*, m_{\mathcal{S}}. (\langle m, m^*, m_{\mathcal{S}} \rangle \in \bigcup_{\mathbf{R} \in \mathcal{S}} \llbracket \mathbf{R} \rrbracket \text{ implies } m^* \models d) && (\text{Def. } \llbracket \mathcal{S} \rrbracket) \\
\text{iff } & \forall m^*, m_{\mathcal{S}}. \bigwedge_{\mathbf{R} \in \mathcal{S}} (\langle m, m^*, m_{\mathcal{S}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ implies } m^* \models d) && \begin{aligned} & (\frac{(F \vee G) \Rightarrow H \equiv}{(F \Rightarrow H) \wedge (G \Rightarrow H)}) \\ & (\frac{\forall x F \wedge G \equiv}{\forall x F \wedge \forall x G}) \end{aligned} \\
\text{iff } & \bigwedge_{\mathbf{R} \in \mathcal{S}} \forall m^*, m_{\mathcal{S}}. (\langle m, m^*, m_{\mathcal{S}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ implies } m^* \models d) \\
\text{iff } & \bigwedge_{\mathbf{R} \in \mathcal{S}} m \models \text{wlp}(\mathbf{R}, d) && (\text{Def. wlp}) \\
\text{iff } & \bigwedge_{\mathbf{R} \in \mathcal{S}} m \models \text{Wlp}(\mathbf{R}, d) && (\text{Hypothesis}) \\
\text{iff } & m \models \bigwedge_{\mathbf{R} \in \mathcal{S}} \text{Wlp}(\mathbf{R}, d) && (\text{Def. } \models)
\end{aligned}$$

Case $(\mathbf{P}; \mathbf{Q})$.

$$\begin{aligned}
& m \models \text{wlp}((\mathbf{P}; \mathbf{Q}), d) \\
\text{iff } & \forall m^*, m_{\mathbf{P}; \mathbf{Q}}. (\langle m, m^*, m_{\mathbf{P}; \mathbf{Q}} \rangle \in \llbracket (\mathbf{P}; \mathbf{Q}) \rrbracket \text{ implies } m^* \models d) && (\text{Def. wlp}) \\
\text{iff } & \forall m^*, m_{\mathbf{Q}}. \forall m', m_{\mathbf{P}}. ((\langle m, m', m_{\mathbf{P}} \rangle \in \llbracket \mathbf{P} \rrbracket \\
& \quad \text{and } \langle m', m^*, m_{\mathbf{Q}} \rangle \in \llbracket \mathbf{Q} \rrbracket) \text{ implies } m^* \models d) && (\text{Def. } \llbracket (\mathbf{P}; \mathbf{Q}) \rrbracket) \\
\text{iff } & \forall m', m_{\mathbf{P}}. \forall m^*, m_{\mathbf{Q}}. (\langle m, m', m_{\mathbf{P}} \rangle \in \llbracket \mathbf{P} \rrbracket \\
& \quad \text{implies } (\langle m', m^*, m_{\mathbf{Q}} \rangle \in \llbracket \mathbf{Q} \rrbracket \text{ implies } m^* \models d)) && \begin{aligned} & (\frac{(F \wedge G) \Rightarrow H \equiv}{(F \Rightarrow (G \Rightarrow H))}) \end{aligned} \\
\text{iff } & \forall m', m_{\mathbf{P}}. (\langle m, m', m_{\mathbf{P}} \rangle \in \llbracket \mathbf{P} \rrbracket \\
& \quad \text{implies } \forall m^*, m_{\mathbf{Q}}. (\langle m', m^*, m_{\mathbf{Q}} \rangle \in \llbracket \mathbf{Q} \rrbracket \text{ implies } m^* \models d)) && (\frac{x \notin \text{Free}(F):}{\forall x (F \vee G) \equiv F \vee \forall x G})
\end{aligned}$$

$$\begin{aligned}
& \text{iff } \forall m', m_P. (\langle m, m', m_P \rangle \in \llbracket P \rrbracket \text{ implies } m' \models \text{wlp}(Q, d)) & (\text{Def. wlp}) \\
& \text{iff } \forall m', m_P. (\langle m, m', m_P \rangle \in \llbracket P \rrbracket \text{ implies } m' \models \text{Wlp}(Q, d)) & (\text{Hypothesis}) \\
& \text{iff } m \models \text{wlp}(P, \text{Wlp}(Q, d)) & (\text{Def. wlp}) \\
& \text{iff } m \models \text{Wlp}(P, \text{Wlp}(Q, d)) & (\text{Hypothesis})
\end{aligned}$$

Case `if c then P fi`.

$$\begin{aligned}
& \text{wlp}(\text{if } c \text{ then } P \text{ else } Q \text{ fi}, d) \\
&= \text{wlp}(\{(\text{Assert}(c); P), \text{Assert}(\neg c)\}, d) & (\text{Def. if-then}) \\
&\equiv \text{Wlp}(\{(\text{Assert}(c); P), \text{Assert}(\neg c)\}, d) & (\text{Hypothesis}) \\
&= \text{Wlp}((\text{Assert}(c); P), d) \wedge \text{Wlp}(\text{Assert}(\neg c), d) & (\text{Def. Wlp}) \\
&= \text{Wlp}(\text{Assert}(c), \text{Wlp}(P, d)) \wedge \text{Wlp}(\text{Assert}(\neg c), d) & (\text{Def. Wlp}) \\
&= (c \Rightarrow \text{Wlp}(P, d)) \wedge (\neg c \Rightarrow d) & (\text{Def. Wlp})
\end{aligned}$$

Case `if c then P else Q fi`.

$$\begin{aligned}
& \text{wlp}(\text{if } c \text{ then } P \text{ else } Q \text{ fi}, d) \\
&= \text{wlp}(\{(\text{Assert}(c); P), (\text{Assert}(\neg c); Q)\}, d) & (\text{Def. if-then-else}) \\
&\equiv \text{Wlp}(\{(\text{Assert}(c); P), (\text{Assert}(\neg c); Q)\}, d) & (\text{Hypothesis}) \\
&= \text{Wlp}((\text{Assert}(c); P), d) \wedge \text{Wlp}((\text{Assert}(\neg c); Q), d) & (\text{Def. Wlp}) \\
&= \text{Wlp}(\text{Assert}(c), \text{Wlp}(P, d)) \wedge \text{Wlp}(\text{Assert}(\neg c), \text{Wlp}(Q, d)) & (\text{Def. Wlp}) \\
&= (c \Rightarrow \text{Wlp}(P, d)) \wedge (\neg c \Rightarrow \text{Wlp}(Q, d)) & (\text{Def. Wlp})
\end{aligned}$$

Case P^0 .

$$\begin{aligned}
& \text{wlp}(P^0, d) \\
&= \text{wlp}(\text{Skip}, d) & (\text{Def. } P^0) \\
&\equiv \text{Wlp}(\text{Skip}, d) & (\text{Hypothesis}) \\
&= d & (\text{Def. Wlp})
\end{aligned}$$

Case $P^j, j > 0$.

$$\begin{aligned}
& \text{wlp}(P^j, d) \\
&= \text{wlp}((\text{Fix}(P); P^{j-1}), d) & (\text{Def. } P^j) \\
&\equiv \text{Wlp}((\text{Fix}(P); P^{j-1}), d) & (\text{Hypothesis}) \\
&= \text{Wlp}(\text{Fix}(P), \text{Wlp}(P^{j-1}, d)) & (\text{Def. Wlp})
\end{aligned}$$

Case P^* .

$$\begin{aligned}
& \text{wlp}(P^*, d) \\
&= \text{wlp}(\bigcup_{j=0}^{\infty} P^j, d) & (\text{Def. } P^*) \\
&\equiv \text{Wlp}(\bigcup_{j=0}^{\infty} P^j, d) & (\text{Hypothesis}) \\
&= \bigwedge_{j=0}^{\infty} \text{Wlp}(P^j, d) & (\text{Def. Wlp})
\end{aligned}$$

Case $\downarrow P \downarrow$.

$$\begin{aligned}
& m \models \text{wlp}(\downarrow P \downarrow, d) \\
& \text{iff } \forall m^*, m_{\downarrow P \downarrow}. (\langle m, m^*, m_{\downarrow P \downarrow} \rangle \in \llbracket \downarrow P \downarrow \rrbracket \text{ implies } m^* \models d) & (\text{Def. wlp}) \\
& \text{iff } \forall m^*. ((\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \\
& \quad \text{and } \nexists m'. (\langle m^*, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket)) \text{ implies } m^* \models d) & (\text{Def. } \llbracket \downarrow P \downarrow \rrbracket)
\end{aligned}$$

$$\begin{aligned}
& \text{iff } \forall m^*. ((\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \\
& \quad \text{and } \forall m'. (\langle m^*, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket) \text{ implies false}) \\
& \quad \text{implies } m^* \models d) \quad (\neg \exists x F) \\
& \quad \quad \quad (\equiv \forall x \neg F) \\
& \text{iff } \forall m^*. ((\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \text{ and } m^* \models \text{wlp}(\text{Fix}(P), \text{false})) \\
& \quad \text{implies } m^* \models d) \quad (\text{Def. wp}) \\
& \text{iff } \forall m^*. ((\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \text{ and } m^* \models \text{Wlp}(\text{Fix}(P), \text{false})) \\
& \quad \text{implies } m^* \models d) \quad (\text{Hypothesis}) \\
& \text{iff } \forall m^*. (\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \\
& \quad \text{implies } (m^* \models \text{Wlp}(\text{Fix}(P), \text{false}) \text{ implies } m^* \models d)) \quad (\begin{smallmatrix} (F \wedge G) \Rightarrow m^* \equiv \\ F \Rightarrow (G \Rightarrow H) \end{smallmatrix}) \\
& \text{iff } \forall m^*. (\langle m, m^*, \text{id} \rangle \in \llbracket P^* \rrbracket \\
& \quad \text{implies } m^* \models (\text{Wlp}(\text{Fix}(P), \text{false}) \Rightarrow d)) \quad (\text{Def. } \models) \\
& \text{iff } m \models \text{Wlp}(P^*, \text{Wlp}(\text{Fix}(P), \text{false}) \Rightarrow d) \quad (\text{Def. wlp, IH. } P^*)
\end{aligned}$$

Case `while c do P od`.

$$\begin{aligned}
& \text{wlp}(\text{while } c \text{ do } P \text{ od}, d) \\
& = \text{wlp}((\text{Assert}(c); P)^*, \text{Assert}(\neg c)), d) \quad (\text{Def. while } c \text{ do } P \text{ od}) \\
& \equiv \text{Wlp}((\text{Assert}(c); P)^*, \text{Assert}(\neg c)), d) \quad (\text{Hypothesis}) \\
& = \text{Wlp}((\text{Assert}(c); P)^*, \text{Wlp}(\text{Assert}(\neg c), d)) \quad (\text{Def. Wlp}) \\
& = \text{Wlp}((\text{Assert}(c); P)^*, \neg c \Rightarrow d) \quad (\text{Def. Wlp})
\end{aligned}$$

This completes the proof. \square

5.4 Weakest invariants

We have presented a construction of weakest liberal preconditions. For any program not containing a possibly unbounded form of iteration, the construction is effective. For as-long-as-possible and while-a-condition-is-satisfied iterations, the construction reduces to the construction of weakest preconditions for reflexive, transitive closures. However, the problem to construct a weakest (liberal) precondition for a reflexive, transitive closure contains the difficult problem to find a weakest invariant that implies the given postcondition.

Definition 5.19 (invariant). A condition c is an *invariant* of a program P , if $\{c\}P\{c\}$ is a correct specification.

First, we show that every weakest (liberal) precondition for a reflexive, transitive closure is necessarily an invariant.

Fact 5.20. For every program P and every condition d , $\text{Wlp}(P^*, d)$ is an invariant of P .

Proof. According to Definition 5.2, we have to show $\text{Wlp}(P^*, d) \Rightarrow \text{Wlp}(P, \text{Wlp}(P^*, d))$. We have $\text{Wlp}(P^*, d) = \bigwedge_{j=0}^{\infty} \text{Wlp}(P^j, d) \Rightarrow \bigwedge_{j=1}^{\infty} \text{Wlp}(P^j, d) = \bigwedge_{j=0}^{\infty} \text{Wlp}(P^{j+1}, d) = \bigwedge_{j=0}^{\infty} \text{Wlp}(P, \text{Wlp}(P^j, d)) \equiv \text{Wlp}(P, \bigwedge_{j=0}^{\infty} \text{Wlp}(P^j, d)) = \text{Wlp}(P, \text{Wlp}(P^*, d))$, where we use the fact that wlp is universal conjunctive, that is, $\text{wlp}(P, d \wedge d') \equiv \text{wlp}(P, d) \wedge \text{wlp}(P, d')$. \square

A direct consequence is the following.

Corollary 5.21. $\{c\}P^*\{c\}$ is correct if and only if $\{c\}P\{c\}$ is correct.

Concerning a specification $\{c\}P^*\{c\}$, either c is an invariant of P , or the specification is not correct. If P itself does not contain iteration, for instance, P is a choice of transformation rules, the correctness of such specifications can be investigated without the need for weakest invariants.

The main problem of the construction $\text{Wlp}(P^*, d)$, as presented in Section 5.3, is its ineffectiveness: the construction yields an infinite condition. Even worse, there may not always be a finite equivalent.

Example 5.22 (no finite weakest invariant). For discrete graphs, consider the program $P = \text{Sel}(\emptyset \hookrightarrow \bigcirc_1 \bigcirc_2); \text{Del}(\bigcirc_1 \bigcirc_2 \hookrightarrow \emptyset)$ that selects and deletes two nodes, and consider the postcondition $d = \neg \exists \bigcirc$ with the meaning “There is no node”. A weakest precondition $\text{Wlp}(\downarrow P \downarrow, d) = \text{Wlp}(P^*, (\neg \exists \bigcirc_1 \bigcirc_2 \Rightarrow d))$ is a condition expressing that the number of nodes is even. However, such a condition is not expressible by a finite first-order graph formula and thus is not expressible by a finite graph condition.

In some cases, it may be possible to approximate a finite representation of $\bigwedge_{j=0}^{\infty} \text{Wlp}(P^j, d)$ by replacing ∞ with some $k \in \mathbb{N}$.

Algorithm 5.23 (invariant approximation).

Parameters: program P and postcondition d .

Condition $inv = d$;

Condition $wlp = \text{Wlp}(P, inv)$;

while $(inv \not\equiv wlp)$ **do**

$inv = wlp$;

$wlp = (d \wedge \text{Wlp}(P, inv))$;

od;

return inv ;

For ascending $k \geq 0$, the algorithm tries to prove the invariance of the condition $\bigwedge_{j=0}^k \text{Wlp}(P^j, d)$ with respect to P . However, as the following example shows, the approximation is not guaranteed to terminate with a result, even if a finite weakest invariant exists.

Example 5.24 (non-termination). Consider the program $P = \text{Sel}(\emptyset \hookrightarrow \bigcirc_1); \text{Del}(\bigcirc_1 \hookrightarrow \emptyset)$ that selects and deletes a node, and consider the postcondition $d = \exists \bigcirc_1$ with the meaning “There is a node”. On the one hand, we have $\bigwedge_{j=0}^k (\text{Wlp}(P^j, d)) \not\equiv \bigwedge_{j=0}^{k+1} (\text{Wlp}(P^j, d))$ for arbitrary k , while on the other hand, $\text{Wlp}(P^*, d) \equiv \text{false}$.

The above algorithm does not terminate for a large class of inputs and seems therefore of little practical relevance. A different approach to approximate invariants uses counterexamples. The following algorithm tries to strengthen non-invariant conditions by forbidding any situations violating the invariance.

Algorithm 5.25 (counterexample-based invariant approximation).

Parameters: program P and postcondition d .

```

Condition  $inv = d$ ;
Condition  $wlp = \text{Wlp}(P, inv)$ ;
while( $inv \not\equiv wlp$ ) do
  while( $inv \not\equiv wlp$ ) do
     $cex = \text{counterexample}(inv \Rightarrow wlp)$ ;
     $inv = (inv \wedge \neg \exists cex)$ ;
  od;
   $wlp = \text{Wlp}(P, inv)$ ;
od;
return  $inv$ ;

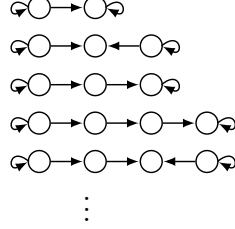
```

It is assumed that the method $\text{counterexample}(inv \Rightarrow wlp)$ returns a smallest counterexample. Moreover, the inner while-loop delays the recomputation of a weakest liberal precondition to make the procedure more robust with respect to the order of the counterexamples.

Algorithm 5.25 is sound, as after exiting the outer while loop, inv is invariant and implies d . However, Algorithm 5.25 is not guaranteed to terminate, especially if the sought invariant concerns properties other than first-order, for instance, paths of arbitrary length.

Example 5.26 (non-termination of Algorithm 5.25). Consider the specification $\{c\}\rho\{c\}$, consisting of the condition $c = \neg \exists \bigcirc_1 \bigcirc_2$ and the rule $\rho = \langle \bigcirc_1 \rightarrow \bigcirc_2 \hookrightarrow \bigcirc_1 \rightarrow \bigcirc_2 \rangle$ that moves a loop along a proper edge. The subsequent refinements involve paths of arbitrary length of which the start and end nodes

have each a loop:



Moreover, in case of termination, the approximated invariant may not be a weakest precondition and may be not weak enough to carry out a proof.

Example 5.27 (over-approximation). Consider the correct specification $\{c\}P\{c\}$, consisting of the condition $c = \forall(Q_1, \exists Q_1 \text{ loop})$ expressing “Every node has a loop” and the program $P = (\text{Sel } Q_1; \downarrow(\text{Sel } Q_1; \text{Del } Q_1) \downarrow; \text{Del } Q_1)$ that selects a node, selects and deletes all loops of the selected node and deletes the selected node. The weakest liberal precondition of program $\text{Del } Q_1$ and condition c is a condition over Q_1

$$wlp_{\text{Del}} = (\exists Q_1 \text{ loop} \vee \exists Q_1 \rightarrow Q_1 \vee \exists Q_1 \leftarrow Q_1 \vee \forall(Q_1 Q_2, \exists Q_1 Q_2 \text{ loop}))$$

but wlp_{Del} is not invariant with respect to $(\text{Sel } Q_1; \text{Del } Q_1)$ as the graph morphism $cex: Q_1 \rightarrow Q_1 Q_2$ is a counterexample of $(wlp_{\text{Del}} \Rightarrow wlp_{\text{SelDelDel}})$, where $wlp_{\text{SelDelDel}}$ is the weakest liberal precondition of $(\text{Sel } Q_1; \text{Del } Q_1)$ and wlp_{Del} , with

$$(wlp_{\text{Del}} \Rightarrow wlp_{\text{SelDelDel}}) \equiv wlp_{\text{SelDelDel}} = \forall \left(Q_1, \begin{array}{l} \vee \exists Q_1 \text{ loop} \\ \vee \exists Q_1 \rightarrow Q_1 \\ \vee \exists Q_1 \leftarrow Q_1 \\ \vee \forall(Q_1 Q_2, \exists Q_1 Q_2 \text{ loop}) \end{array} \right).$$

Algorithm 5.25 strengthens the input condition wlp_{Del} by conjunctively combining it with $\neg \exists Q_1 Q_2$. Although the refined condition $wlp_{\text{Del}} \wedge \neg \exists Q_1 Q_2$ is invariant, it turns out that it is not weak enough to prove the correctness of $\{c\}P\{c\}$.

In practice, counterexample-based refinement is able to approximate useful invariants, if the postcondition is \forall -free. In contrast, the approximation is usually too strong in the presence of universal conditions. Taking the above example as a guide, we see that Algorithm 5.25 draws the wrong conclusion from the counterexample: it should have strengthened the condition by appending $\forall(Q_1 Q_2, \exists Q_1 Q_2 \text{ loop})$ instead of $\neg \exists Q_1 Q_2$. The following improved algorithm tries to select a subcondition of a condition responsible for a counterexample and reasserts it.

Algorithm 5.28 (counterexample-guided invariant approximation).

Parameters: program P and postcondition d .

Condition $inv = d$;

Condition $wlp = Wlp(P, inv)$;

while($inv \not\Rightarrow wlp$) **do**

 Morphism $cex = \text{counterexample}(inv \Rightarrow wlp)$;

 Condition $theorem = (inv \Rightarrow wlp)$;

$\text{unmark}(theorem)$;

while($cex \neq \text{null}$) **do**

if ($\neg \text{markCondition}(theorem, cex)$) **throw new Exception()**;

$cex = \text{counterexample}((inv \wedge \text{getSubtree}(theorem)) \Rightarrow wlp)$;

od;

$inv = inv \wedge \text{getSubtree}(theorem)$;

$wlp = Wlp(P, inv)$;

od;

return inv ;

where, for a condition c over C , the subroutine $\text{markCondition}(c, cex)$ marks the smallest subcondition s of c over C such that the marking of s ensures $cex \models (\text{getSubtree}(c) \Rightarrow c)$ and returns true if and only if at least some subcondition of s was previously unmarked. For a condition c over C , the subroutine $\text{getSubtree}(c)$ returns the smallest subcondition of c over C that contains every marked subconditions. The subroutine $\text{unmark}(c)$ recursively removes all markings of a condition c . In this context, a marking is a binary meta-information temporarily added to a condition: marked parts will be reasserted in the next refinement as they can exclude a given counterexample, while non-marked parts do not (yet) play a role in the refinement. In case of an exception (signaling unsuccessful termination), one may resort to Algorithm 5.25.

Example 5.29 (counterexample-guided invariant approximation).

Let us resume Example 5.27, that is, consider the correct specification $\{c\}P\{c\}$, consisting of the condition $c = \forall(\bigcirc_1, \exists\bigcirc_1 \rightarrow \bigcirc_1)$ expressing “Every node has a loop” and the program $P = (\text{Sel } \bigcirc_1; \downarrow(\text{Sel } \bigcirc_1 \rightarrow \bigcirc_1; \text{Del } \bigcirc_1 \rightarrow \bigcirc_1); \text{Del } \bigcirc_1)$ that selects a node, selects and deletes all loops of the selected node and deletes the selected node. The weakest liberal precondition of program $\text{Del } \bigcirc_1$ and condition c is a condition over \bigcirc_1

$$wlp_{\text{Del}} = (\exists\bigcirc_1 \rightarrow \bigcirc_1 \vee \exists\bigcirc_1 \rightarrow \bigcirc_1 \vee \exists\bigcirc_1 \leftarrow \bigcirc_1 \vee \forall(\bigcirc_1 \bigcirc_2, \exists\bigcirc_1 \bigcirc_2))$$

but wlp_{Del} is not invariant with respect to $(\text{Sel } \bigcirc_1 \rightarrow \bigcirc_1; \text{Del } \bigcirc_1 \rightarrow \bigcirc_1)$ as the graph morphism $cex: \bigcirc_1 \rightarrow \bigcirc_1 \rightarrow \bigcirc_2$ is a counterexample of $(wlp_{\text{Del}} \Rightarrow wlp_{\text{SelDel}})$,

where $wlp_{\text{SelDelDel}}$ is the weakest liberal precondition of $(\text{Sel } \textcircled{1} \rightarrow; \text{Del } \textcircled{1} \rightarrow)$ and wlp_{Del} , with

$$(wlp_{\text{Del}} \Rightarrow wlp_{\text{SelDelDel}}) \equiv wlp_{\text{SelDelDel}} = \forall \left(\textcircled{1} \rightarrow, \begin{array}{l} \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \forall (\textcircled{1} \rightarrow \textcircled{2}, \exists \textcircled{1} \rightarrow \textcircled{2}) \end{array} \right).$$

The method `markCondition`($wlp_{\text{SelDelDel}}, \text{cex}$) marks the subcondition

$$\forall \left(\textcircled{1} \rightarrow, \begin{array}{l} \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \exists \textcircled{1} \rightarrow \textcircled{1} \rightarrow \\ \vee \forall (\textcircled{1} \rightarrow \textcircled{2}, \exists \textcircled{1} \rightarrow \textcircled{2}) \end{array} \right)$$

as it is the smallest subcondition able to neutralize the above counterexample. As there are no further markings, the method `getSubtree`($wlp_{\text{SelDelDel}}$) returns the optimized condition $\forall (\textcircled{1} \rightarrow \textcircled{2}, \exists \textcircled{1} \rightarrow \textcircled{2})$. The refined condition $(wlp_{\text{Del}} \wedge \forall (\textcircled{1} \rightarrow \textcircled{2}, \exists \textcircled{1} \rightarrow \textcircled{2}))$ is invariant and weak enough to prove $\{c\}P\{c\}$.

The counterexample-guided Algorithm 5.28 is used for the case studies in Chapter 7. It has the highest coverage of test cases compared to Algorithm 5.25 (good coverage of the car platooning case study, but insufficient coverage of access control specifications) and Algorithm 5.23 (completely useless).

All presented algorithms for finding invariants rely on deciding the implication problem. In this sense, finding invariants *really is* the hardest part of program verification as it may be necessary to decide a number of implication problems. Even if the computation of an invariant is successful, it may be not weak enough to prove the input specification and we (internally) yield a counterexample that may be spurious. Therefore, in cases where the construction of weakest liberal preconditions requires the computation of invariants, we conduct a state space exploration to verify the authenticity of counterexamples.

5.5 Related concepts

The Wp and Wlp predicate transformers, first introduced by Edsger W. Dijkstra in [Dij75], can be seen as a constructive interpretation of the well-known Floyd-Hoare calculus. The Floyd-Hoare calculus consists of a set of logical deduction rules enabling mathematical reasoning about the correctness of assignment-based while-programs with respect to pre- and postcondition. It

was first published in [Hoa69], acknowledging earlier contributions in form of a similar system for flowcharts [Flo67]. Program verification using the Floyd-Hoare calculus is traditionally understood as a manual and tedious task, performed by experts using mind and hand [AO97]. Our results show that predicate transformers, as intended by Dijkstra, are a suitable approach to automatize such proofs as far as possible.

A closely related approach to logically infer the correctness of graph transformation specifications is a translation of graph transformation rules into logical formulas [Cou90], similar to the translation of graph conditions into graph formulas. Following this idea, Strecker [Str08, SG06] models typed graph transformations rules and programs in the proof assistant ISABELLE. His approach supports the manual verification of “a fragment of first-order logic enriched by transitive closure”. While the proof assistant offers some guidance, verification remains a manual task while the advantages of a graphical notation are lost due to the translation.

A completely different approach to program verification is model checking, which usually refers to a systematical exhaustive exploration of all reachable states and transitions of a model with respect to a start state. In this sense, model checking proves the correctness of specifications of the form $\{S\}P\{d\}$, where S represents a single object instead of an (usually) infinite set of objects, as represented by a precondition. Model checking is possible for finite models, that is, programs or transformation systems that are guaranteed to terminate for S . It is also applicable to infinite models, if there exists a finite representation of the infinite state space that is compatible with the postcondition d .

Among the first papers considering model checking of graph transformations are [Var03, Var04], in which typed, attributed graph transformation rules with negative application conditions are translated to PROMELA and checked by SPIN against safety and reachability of “property graphs” which correspond to the $\exists\neg\exists$ -fragment/ \forall -free fragment of conditions. Due to the use of SPIN, which only checks models with a finite number of transitions, upper bounds of element types have to be fixed in advance. Effectively, only a prefix of the transformation system is checked. Furthermore, any counterexamples found by SPIN are not translated back into graphs. Independently, a similar approach is followed by [DFRdS03, dSDR04] in which “object-based graph grammars” are translated to PROMELA.

In [Ren04b, KR06], the transformation-based tool GROOVE is presented. It conducts a state space exploration of edge-labeled graph transformation rules with negative application conditions and verifies properties in CTL over first-order graph logic, enriched with transitive closure. While GROOVE performs graph isomorphism checks to detect cycles in a model, it is still re-

stricted to transformation systems with a finite number of states (up to isomorphism). An abstraction of infinite transformation systems is investigated in [RD06], but has not yet been implemented. In [RSV04], the approaches of Rensink and Varró are compared.

A Petri net-based approach to model checking is investigated in [BK02, BKK03, KK06, KK08]. Attributed hypergraphs are abstracted to so-called “Petri graphs” and node-preserving transformation rules without application conditions are over-approximated by McMillan unfoldings using counterexample guided abstraction refinement. The associated tool AUGUR checks properties in ACTL* over the \forall -fragment of MSOGL, for instance, the property “Never there exists a link/path between two elements” can be checked. However, properties such as “Always exists a link between two elements” are not expressible in the considered logic. In [BKR05], the approaches of Rensink and König are compared.

Another abstraction based model checking approach is investigated in [Bau06, BW07], in which a fixpoint approximation of directed, labeled graph relabeling rules with “local” negative application conditions, called “partner constraints”, is considered. The considered logic GL is based on CTL without the Next operator over first-order graph logic. In [BBKR08], an abstraction method is presented that generalizes previous approaches by Rensink and Bauer.

feature\name	CHECKVML D. Varró	GROOVE A. Rensink	AUGUR B. König	HIRALYSIS J. Bauer	M. Strecker	ENFORCE Pennemann
approach	model checking				theorem proving	
based on tool	SPIN	–	–	–	ISABELLE	–
automated	yes	yes	yes	yes	no	yes
graph. counterex.	no	yes	yes	yes	no	yes
infinite systems	no	no	yes	yes	yes	yes
AccessControl	no	yes	no	no	yes	yes
<i>secure</i>	no	yes	no	no ¹	yes	yes
<i>always, paths exists</i>	no	yes	no	no	yes	no
publications	2003/04	2003-08	2004-08	2006-07	2006/08	2005-08

Table 5.1: Summary of dedicated GT verification approaches

A summary of the various verification approaches dedicated to graph transformation (GT) is given in Table 5.1. At this point, each approach has

¹expressible in GL, but not preserved under abstraction

its own strengths and weaknesses. On the one hand, the model checking approaches are either restricted to finite models, or due to abstraction, consider only a certain fragment of a logic. On the other hand, these fragments are usually based on monadic second order logic or do at least include the transitive closure, thus a certain fragment of path properties is covered.

5.6 Summary and discussion

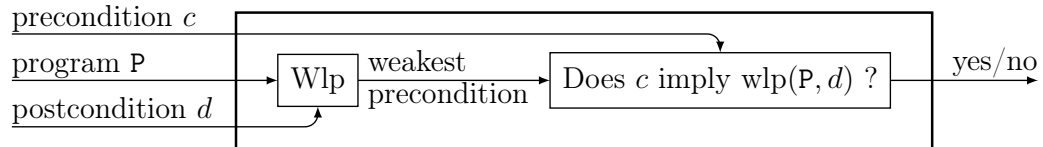
We are interested in deciding the *correctness* of *program specifications* consisting of a nested precondition, a program with interface and a nested postcondition. By considering *weakest liberal preconditions* we reduce the correctness problem of these specifications onto the problem whether or not the precondition implies a weakest precondition. We presented a construction Wlp of programs with interface and conditions into weakest liberal preconditions and proved its soundness.

Aside weakest liberal preconditions ensuring (partial) correctness, there exists weakest preconditions ensuring the existence of results and termination, as investigated for programs over double pushout transformation rules in [HPR06]. Complementary to weakest preconditions is the concept of strongest postconditions, as investigated for graph transformation rules and nested preconditions in [HP09].

For programs that do not contain a possibly unbounded form of iteration, the construction Wlp is effective. For as-long-as-possible and while-a-condition-is-satisfied iterations, the construction reduces to the construction of weakest preconditions for reflexive, transitive closure. In this case, the problem to construct a weakest (liberal) precondition for a reflexive, transitive closure contains the difficult problem to find a weakest invariant for a program that implies the given postcondition. Two algorithms to find (weakest) invariants were presented and we discussed their properties. We also pointed out that for some specifications, the construction of weakest invariants can be avoided, namely for specifications of $\{c\}P^*\{c\}$, if P itself does not contain iteration.

6. The implication problem of conditions

In this chapter we consider the implication problem of conditions, that is, the problem to decide for any conditions whether or not a condition implies another condition. After the computation of a weakest precondition, the implication problem is the second step in deciding whether or not a program specification is correct.



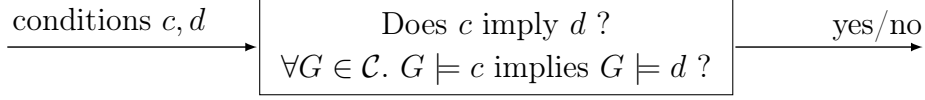
We investigate the connections between the implication, the tautology and the satisfiability problem of conditions, show that all problems are undecidable for the category **Graphs**, and present a satisfiability algorithm as well as a calculus for proving conditions over a class of weak adhesive HLR categories.

6.1 Implication, tautology and satisfiability

The implication problem may be seen as a special instance of the tautology problem, which again is complementary to the satisfiability problem of conditions. We show that, for recursively enumerable categories, the satisfiability problem is not decidable, but semi-decidable and present a sound and complete satisfiability algorithm. Consequently, the tautology problem is undecidable and we present a sound tautology algorithm. For the definitions of decidability, semi-decidability, and completeness, we refer to [LP98, Sch08].

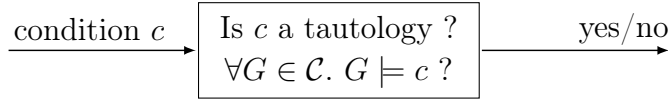
Definition 6.1 (implication problem). For a given category \mathcal{C} , the *implication problem* is the problem to decide for any given conditions c, d

over I whether or not $\forall G \in \mathcal{C}. G \models c$ implies $G \models d$.



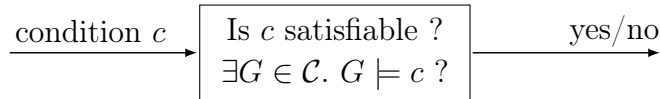
As “ $G \models c$ implies $G \models d$ ” iff “ $G \models (c \Rightarrow d)$ ”, the implication problem can be seen as a tautology problem, the fundamental problem to decide for any claimed statement whether or not it is true for all possible system states.

Definition 6.2 (tautology problem). For a given category \mathcal{C} , the *tautology problem* (or *validity problem*) is the problem to decide for any given condition c over I whether or not $\forall G \in \mathcal{C}. G \models c$. If so, c is a *tautology*, and a *contradiction* otherwise.



We write “ $\models c$ ” if c is a tautology and “ $\not\models c$ ” if c is not a tautology. As “ $\forall G \in \mathcal{C}. G \models c$ ” iff “ $\neg \exists G \in \mathcal{C}. G \models \neg c$ ”, the tautology problem is complementary to the satisfiability problem, the problem to decide for any statement whether or not there is a system state that satisfies it.

Definition 6.3 (satisfiability problem). For a given category \mathcal{C} , the *satisfiability problem* is the problem to decide for any given condition c over I whether or not $\exists G \in \mathcal{C}. G \models c$. If so, c is *satisfiable*, and *unsatisfiable* otherwise.



The satisfiability problem can be used to show that a system specification is conflict free or to prove that a statement is invalid, that is, if the negated statement is satisfiable. If some object G is provided along with a positive answer, one yields a counterexample for the latter case, illustrating an invalid system state. In this sense, a satisfiability algorithm complements a theorem prover, with the prover searching for proof and the satisfiability algorithm looking for a counterexample.

The following fact summarizes that any instance of an implication, tautology or satisfiability problem can be translated into an equivalent instance of the other two problem types.

Fact 6.4 (connection). The implication, tautology or satisfiability problem are connected as follows:

problem instance	equivalent instance
c is satisfiable	$\neg c$ is not a tautology
c is not satisfiable	$\neg c$ is a tautology
c is a tautology	$\neg c$ is not satisfiable
c is not a tautology	$\neg c$ is satisfiable
c is a tautology	true implies c
c is not a tautology	true does not imply c
c implies d	$(c \Rightarrow d)$ is a tautology
c does not imply d	$(c \Rightarrow d)$ is not a tautology

A direct consequence is the connection between the decidability of the three problems.

Corollary 6.5 (problem decidability). For a given category \mathcal{C} , the satisfiability problem is decidable if and only if the tautology problem is decidable if and only if the implication problem is decidable.

For the category **Graphs** of finite, directed, labeled graphs, conditions are expressively equivalent to first order graph formulas, as proved in Section 3.3. By the undecidability of first-order graph formulas [Tra50, Cou90], we get that there are no effective procedures for deciding if a given graph condition is satisfiable at all, satisfied by every graph, nor if a given graph condition implies another given graph condition.

Corollary 6.6 (Undecidability of graph conditions). The satisfiability, the tautology, and the implication problem of (finite) graph conditions are undecidable.

Proof. Assume, any of the problems were decidable for (finite) graph conditions. Then by Corollary 6.5, all of the problems would be decidable for (finite) graph conditions. By Corollary 3.35, we could construct for every (finite) first-order graph formula F_j ($j = 1, 2$) a (finite) graph condition $\text{Cond}_{\mathcal{M}}(F_j)$ such that $G \models F_j$ iff $G \models \text{Cond}_{\mathcal{M}}(F_j)$. Then the satisfiability problem of (finite) first-order graph formulas would be decidable, contradiction [Tra50, Cou90]. \square

The undecidability of the implication problem alone implies the undecidability of the correctness problem of the considered program specifications.

Corollary 6.7 (Undecidability of correctness). The correctness of specifications consisting of programs with interface and nested conditions is undecidable.

Proof. We have: $(\{c\}\mathbf{Skip}\{d\} \text{ is correct})$, iff $(c \text{ implies } \mathbf{Wlp}(\mathbf{Skip}, d))$, iff $(c \text{ implies } d)$. If the correctness problem were decidable, the implication problem would be decidable, contradiction. \square

As the aforementioned problems of conditions are undecidable for **Graphs**, there does not exist an algorithm that decides the satisfiability of arbitrary conditions over arbitrary categories. Any category-independent algorithm for the satisfiability problem of conditions is necessarily either unsound, incomplete or not guaranteed to terminate. As the satisfiability problem is semi-decidable for recursively enumerable categories such as **Graphs**, we seek a sound and complete satisfiability algorithm that may not terminate for some unsatisfiable conditions.

Definition 6.8 (recursive enumerability). A set $S = \{s_1, s_2, \dots\}$ is *recursively enumerable*, if there is an algorithm that enumerates the members s_1, s_2, \dots of S . A category $\mathcal{C} = \langle \mathcal{O}, \mathcal{A} \rangle$ is *recursively enumerable*, if \mathcal{O} is recursively enumerable.

Example 6.9 (recursive enumerability of graphs). Let $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ be an arbitrary, finite alphabet of node and edge labels. If applied onto the empty graph, the reflexive transitive closure of the following graph program with interface enumerates all graphs over \mathcal{C} :

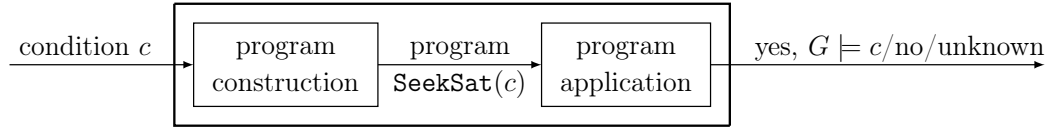
$$\begin{aligned} & \bigcup_{m \in \mathcal{C}_V} \{ \langle \emptyset \hookrightarrow \textcircled{m} \rangle \} \\ \cup & \bigcup_{l \in \mathcal{C}_E, m \in \mathcal{C}_V} \{ \langle \textcircled{m}_1 \hookrightarrow \textcircled{m}_1^l \rangle \} \\ \cup & \bigcup_{l \in \mathcal{C}_E, m, n \in \mathcal{C}_V} \{ \langle \textcircled{m}_1 \textcircled{n}_2 \hookrightarrow \textcircled{m}_1^l \textcircled{n}_2 \rangle \} \end{aligned}$$

Fact 6.10 (semi-decidability). If the category \mathcal{C} is recursively enumerable, then the satisfiability problem is semi-decidable.

Proof. Let \mathcal{C} be a recursively enumerable category. As programs with interface are computationally complete, there exists a program **Enumerate** that, if applied onto the initial object, enumerates \mathcal{C} . For any condition c , the program **Enumerate; Assert**(c) enumerates all objects satisfying c , hence the satisfiability problem is semi-decidable. \square

6.2 Satisfiability solver SeekSat

In this section we present a sound and complete algorithm for conditions over weak adhesive HLR categories that is not always guaranteed to terminate for unsatisfiable conditions, but will find a satisfiable object for every satisfiable condition in finite time. The algorithm answers yes, as soon a result is found, answers no, if it terminates without results, and does not answer in case of non-termination. Instead of enumerating all possible objects of a category to approach the problem, the presented algorithm uses the input condition in a constructive way. Starting from the initial object, for instance, the empty graph, elements of positive statements are added if necessary, while the absence of forbidden patterns is checked. The result is a monotone (non-deleting) algorithm which non-deterministically progresses towards a satisfying object. Technically, we generate for each condition c a program $\text{SeekSat}(c)$.



Satisfaction of conditions by objects is defined by the presence (or absence) of morphisms. For each condition c , we define a program $\text{Sat}(c)$ that for a given input m in \mathcal{M} is supposed to deliver some results m^* in \mathcal{M} such that $m^* \models c$.

Construction (SeekSat). For a condition c over the initial object I in MNF , define $\text{SeekSat}(c) = \text{Sat}(c)$ and define Sat as follows:

$$\begin{aligned}
 \text{Sat}(\text{true}) &= \text{Skip} \\
 \text{Sat}(\neg \text{true}) &= \text{Abort} \\
 \text{Sat}(\exists(a, c)) &= \text{if } \neg \exists(a, c) \text{ then Fix}(\bigcup_{a_2 \circ a_1 = a, a_1, a_2 \in \mathcal{M}} \{ \text{Sel}(a_1); \text{Add}(a_2) \}; \text{Sat}(c)) \text{ fi} \\
 \text{Sat}(\neg \exists(a, c)) &= \text{while } \exists(a, c) \text{ do Fix}(\text{Sel}(a, c); \text{Sat}(\neg c)) \text{ od} \\
 \text{Sat}(\neg \bigwedge_{j \in J} c_j) &= \text{if } \bigwedge_{j \in J} c_j \text{ then } \bigcup_{j \in J} \{ \text{Sat}(\neg c_j) \} \text{ fi} \\
 \text{Sat}(\bigwedge_{j \in J} c_j) &= \text{while } (\neg \bigwedge_{j \in J} c_j) \text{ do } \left(\text{;}_{j \in J} \text{Sat}(c_j) \right) \text{ od} \\
 \text{Sat}(\neg \neg c) &= \text{Sat}(c)
 \end{aligned}$$

where $a_1, a_2 \in \mathcal{M}$ and $\text{;}_{j \in \{1, \dots, n\}} P_j = ((P_1; P_2); \dots; P_n)$ an arbitrary sequentialization.

No computation is necessary in the case of true, and the search for a satisfying morphism fails in case of false. Positive existential statements correspond to

an expansion of existing substructures (if necessary): Let be $a: P \hookrightarrow C$ and $a_2 \circ a_1: P \hookrightarrow C' \hookrightarrow C$ be a decomposition of a into two \mathcal{M} -morphisms a_1, a_2 . Given an \mathcal{M} -morphism $m: P \hookrightarrow G$ and provided that $\exists(a, c)$ is not already satisfied, the program $\text{Sat}(\exists(a, c))$ non-deterministically extends any partial occurrence C' of C in G to C and subsequently applies $\text{Sat}(c)$ on that occurrence.

In case of negative existential statements $\neg\exists(a, c)$, an occurrence of C in G satisfying c is selected in the hope that a subsequent application of $\text{Sat}(\neg c)$ yields a result in which C satisfies $\neg c$, or equivalently does not satisfy c (this iteration may not terminate). For the special case $c = \text{true}$, if $\exists a$ is satisfied, then $\neg\exists a$ is and will remain unsatisfiable for this branch of the search tree. Hence the computation can be aborted and a depth-first interpreter would backtrack, see $\text{Sat}(\neg\exists a)$ in Remark 6.11 below.

Moreover, conjunction corresponds to an iterated random sequentialization until a solution is found (this iteration may not terminate). The completeness of $\text{Sat}(c)$ implies that the execution order of the subprograms c_j is irrelevant for the overall problem, and it suffices to consider any sequentialization. Disjunction corresponds to nondeterministic choice between alternatives: only one subcondition has to be satisfied such that the disjunction becomes satisfied.

Remark 6.11. For abbreviated conditions, the construction of Sat is extended according to Definition 3.1 and the previous construction:

```

Sat(false)      = Sat( $\neg\text{true}$ ) = Abort
Sat( $\neg\text{false}$ )    = Sat(true) = Skip
Sat( $\exists a$ )         = if  $\neg\exists a$  then Fix(
                     $\bigcup_{a_2 \circ a_1 = a, a_1, a_2 \in \mathcal{M}} \{ \text{Sel}(a_1); \text{Add}(a_2) \}; \text{Sat}(\text{true})$ ) fi
                    = if  $\neg\exists a$  then Fix(
                     $\bigcup_{a_2 \circ a_1 = a, a_1, a_2 \in \mathcal{M}, a_2 \notin \text{Iso}} \{ \text{Sel}(a_1); \text{Add}(a_2) \}$ ) fi
Sat( $\neg\exists a$ )      = while  $\exists(a, \text{true})$  do Fix( $\text{Sel}(a, \text{true}); \text{Sat}(\neg\text{true})$ ) od
                    = while  $\exists a$  do Abort od
                    = if  $\exists a$  then Abort fi
                    = Assert( $\neg\exists a$ )
Sat( $\forall(a, c)$ )    = Sat( $\neg\exists(a, \neg c)$ )
Sat( $c \Rightarrow d$ )   = Sat( $\neg c \vee d$ ) = {Sat( $\neg c$ ), Sat( $d$ )}
Sat( $\vee_{j \in J} c_j$ ) = Sat( $\neg \wedge_{j \in J} \neg c_j$ )
Sat( $\neg \vee_{j \in J} c_j$ ) = Sat( $\wedge_{j \in J} \neg c_j$ )

```

Example 6.12 (satisfiable graph condition). Consider the following graph condition $c = \forall(\textcircled{O}_1, \exists(\textcircled{O}_1 \rightarrow \textcircled{O})) \wedge \neg\exists(\textcircled{O} \leftrightarrow \textcircled{O}) \wedge \exists(\textcircled{O})$ expressing “All

nodes have an outgoing edge, there exists no cycle of length two and there is a node". The program **SeekSat**(c) is:

```

while  $\neg c$  do
  while  $\exists(Q_1, \neg \exists(Q_1 \rightarrow O))$  do Fix( //  $P_1$ 
    Sel( $\emptyset \hookrightarrow Q_1, \neg \exists(Q_1 \rightarrow O)$ ); // select a node
    if  $\neg \exists(Q_1 \rightarrow O)$  then Fix(
      {Sel( $Q_1 \hookrightarrow Q_1 Q_2$ ); Add( $Q_1 Q_2 \hookrightarrow Q_1 \rightarrow Q_2$ )}, // choose
      Sel( $Q_1 \hookrightarrow Q_1$ ); Add( $Q_1 \hookrightarrow Q_1 \rightarrow O$ )}
    ) fi
  ) od ;
  Assert( $\neg \exists(O \rightleftarrows O)$ ) ; //  $P_2$ 
  if  $\neg \exists(O)$  then Fix({Sel( $\emptyset \hookrightarrow \emptyset$ ); Add( $\emptyset \hookrightarrow O$ )}) fi //  $P_3$ 
od

```

A fragment of the semantics of **SeekSat**(c) is depicted in Figure 6.1 by representing each input/output morphism with its codomain (all those morphisms have domain \emptyset).

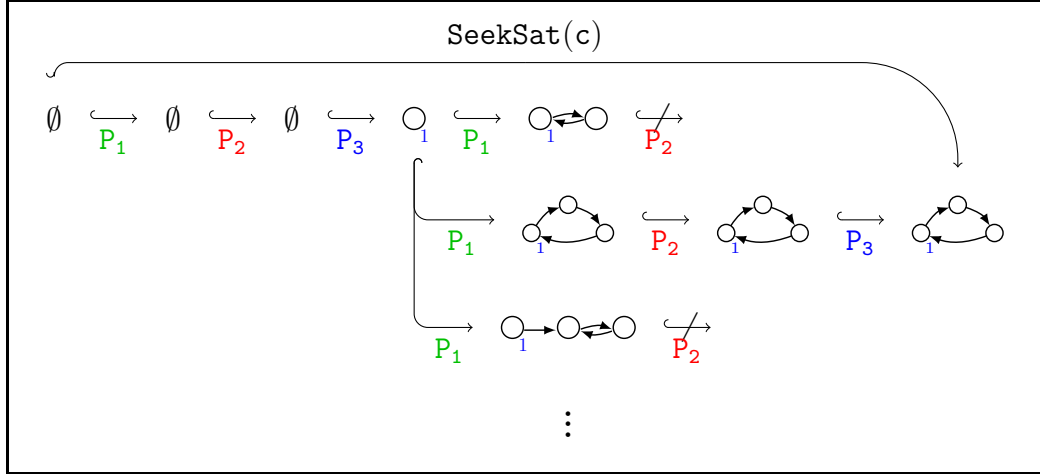
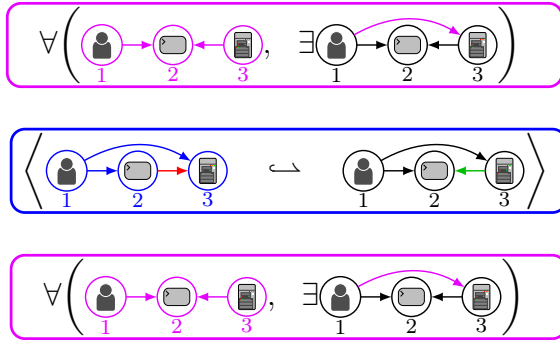


Figure 6.1: A fragment of the semantics of **SeekSat**(c)

There exists some $G \in \mathbf{Graphs}$ such that $\langle \text{id}_\emptyset, i_G, \text{id}_\emptyset \rangle \in \llbracket \mathbf{SeekSat}(c) \rrbracket$, hence c is satisfiable.

Example 6.13 (access control refutation). Consider the access control specification

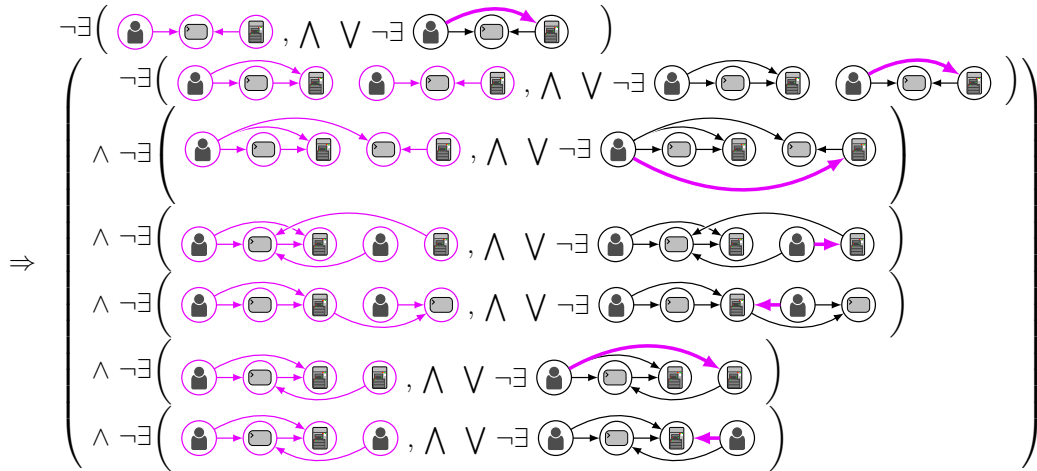


precondition: Every user logged into a system has the appropriate access right.

program: If a user with the appropriate access right proposes a session, it is accepted.

postcondition: Every user logged into a system has the appropriate access right.

To refute its correctness, we have constructed a weakest liberal precondition of the rule and the condition in Example 5.18. We now want to disprove that the precondition c implies the weakest precondition ac_{wlp} , therefore we search for a counterexample for the condition



We negate the above condition and bring it into normal form (to get a better overview).

$$c = \neg \exists \left(\text{Diagram 1}, \wedge \forall \neg \exists \left(\text{Diagram 2} \right) \right)$$

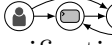
$$\wedge \left(\begin{array}{l} \exists \left(\text{Diagram 3}, \wedge \forall \neg \exists \left(\text{Diagram 4} \right) \right) \\ \vee \exists \left(\text{Diagram 5}, \wedge \forall \neg \exists \left(\text{Diagram 6} \right) \right) \\ \vee \exists \left(\text{Diagram 7}, \wedge \forall \neg \exists \left(\text{Diagram 8} \right) \right) \\ \vee \exists \left(\text{Diagram 9}, \wedge \forall \neg \exists \left(\text{Diagram 10} \right) \right) \\ \vee \exists \left(\text{Diagram 11}, \wedge \forall \neg \exists \left(\text{Diagram 12} \right) \right) \\ \vee \exists \left(\text{Diagram 13}, \wedge \forall \neg \exists \left(\text{Diagram 14} \right) \right) \end{array} \right)$$

The program **SeekSat**(c) is of the following form:

```

while  $\neg c$  do
  while  $\exists(\text{Diagram 1}, \neg \exists(\text{Diagram 2}))$  do Fix(
    Sel( $\text{Diagram 1}$ ,  $\neg \exists(\text{Diagram 2})$ );
    Add( $\text{Diagram 1} \leftrightarrow \text{Diagram 2}$ )
  ) od ; { // choice
    :
    if  $\neg \exists(\text{Diagram 3}, \neg \exists(\text{Diagram 4}))$  then Fix( // choice
      :
      Add( $\emptyset \leftrightarrow \text{Diagram 3}$ ); Assert( $\neg \exists(\text{Diagram 4})$ )
    }) fi
  }
od

```

Application of **SeekSat**(c) on the empty graph eventually yield the satisfying graph , which is a counterexample for the specification above. The specification is incorrect, because there is a *secure* situation, that is, the counterexample itself, in which granting a user access to a system, even though he has the appropriate access right, would lead to an insecure situation, as another user without an access right gets access to the system,

too.

Remark 6.14 (effectiveness of the construction). The construction is effective for all categories satisfying Assumption 2.19 in Chapter 2. For every finite condition c in \mathcal{MNF} , $\text{Sat}(c)$ is a program of finite size: As we consider only finite conjunctions and disjunctions of conditions and the number of all decompositions $a_2 \circ a_1 = a$ is finite for $a \in \mathcal{M}$, all program sets are finite, and the sequentialization in case of $\text{Sat}(\bigwedge_{j \in J} c_j)$ is of finite length.

For every condition c , $\text{SeekSat}(c)$ unselects any elements selected during an execution, which corresponds to an implicit enclosing **Fix** statement.

Fact 6.15 (implicit unselection). For every condition c ,

$$\llbracket \text{SeekSat}(c) \rrbracket = \llbracket \text{Fix}(\text{SeekSat}(c)) \rrbracket.$$

Proof. By induction over the definition of $\text{Sat}(c)$.

Basis. **Skip** and **Abort** do not select.

Hypothesis. Assume $\llbracket \text{SeekSat}(c) \rrbracket = \llbracket \text{Fix}(\text{SeekSat}(c)) \rrbracket$ for given condition c .

Step. Every **Sel** statement is enclosed by **Fix** statements. As $\llbracket \bigcup_{j \in J} \text{Fix}(P_j) \rrbracket = \llbracket \text{Fix}(\bigcup_{j \in J} P_j) \rrbracket$, we can prove $\llbracket \text{if } \neg \exists(a, c) \text{ then } \text{Fix}(\dots) \text{ fi} \rrbracket = \llbracket \text{Fix}(\text{if } \neg \exists(a, c) \text{ then } \dots \text{ fi}) \rrbracket$. Analogously, with $\llbracket \text{Fix}(P)^* \rrbracket = \llbracket \text{Fix}(P^*) \rrbracket$ we can show $\llbracket \text{while } \exists(a, c) \text{ do } \text{Fix}(\dots) \text{ od} \rrbracket = \llbracket \text{Fix}(\text{while } \exists(a, c) \text{ do } \dots \text{ od}) \rrbracket$. The remaining cases are proven in a similar way using the induction hypothesis. \square

As a consequence, every interface relation m_P of a program $\text{Sat}(c)$ is an identity.

Corollary 6.16 (identity). For every condition c over C in \mathcal{MNF} , $\langle m, m^*, m_P \rangle$ in $\llbracket \text{Sat}(c) \rrbracket$ implies m_P is the identity morphism for C .

For every condition c , $\text{SeekSat}(c)$ is a program that does not contain the statements **Del** and **Uns**.

Fact 6.17 (every morphism in \mathcal{M}). For every non-deleting, non-unselecting program with interface, that is, every program without the statements **Del** and **Uns**, every interface relation m_P is in \mathcal{M} . Moreover, for all tuples $\langle m, m^*, m_P \rangle$ in $\llbracket P \rrbracket$, there exists an \mathcal{M} -morphism m_P^* from the input object to the output object such that the resulting square commutes.

$$\begin{array}{ccccc}
 \text{input interface} & \cdots & \bullet & \xrightarrow{m_P} & \bullet & \cdots & \text{output interface} \\
 & & \downarrow m & & \downarrow m^* & & \\
 & & = & & & & \\
 & & \downarrow m_P^* & & & & \\
 \text{input object} & \cdots & \bullet & \xrightarrow{m_P} & \bullet & \cdots & \text{output object}
 \end{array}$$

A consequence of Fact 6.17 is the monotonicity of **SeekSat**.

Corollary 6.18 (monotonicity). For every condition c in \mathcal{MNF} , for every $\langle m, m^*, \text{id} \rangle \in \text{Sat}(c)$ implies $m = m^*$ or there is a morphism $m_p^*: \text{codom}(m) \hookrightarrow \text{codom}(m^*)$ in \mathcal{M} but not in Iso , such that $m_p^* \circ m = m^*$.

SeekSat preserves satisfiability: If **SeekSat**(c) is applied onto a morphism m satisfying c , it terminates without changing the input.

Fact 6.19 (preservation). For every condition c in \mathcal{MNF} , for every \mathcal{M} -morphism m with $m \models c$, $\{\langle m, m^*, \text{id} \rangle\} \in \text{Sat}(c)$ implies $m = m^*$.

Proof. By induction over the definition of $\text{Sat}(c)$. **Basis.** The statement is true for $\text{Sat}(\text{true}) = \text{Skip}$, as well as for $\text{Sat}(\text{false})$, as no morphism satisfies false. **Hypothesis.** Assume the statement is true for given condition c . **Step.** In each case, we see that the program $\text{Sat}(c)$ is either prefixed by **if** $\neg c$ **then** ... **fi** or **while** $\neg c$ **do** ... **od**. In the case of $m \models c$, the programs have the semantics of **Skip**, therefore $m = m^*$. \square

The soundness and completeness of **SeekSat** is a main result of this section.

Theorem 6.20 (SeekSat). For each condition c over \mathbf{I} in \mathcal{MNF} , there is a program **SeekSat**(c) that is sound and complete, that is,

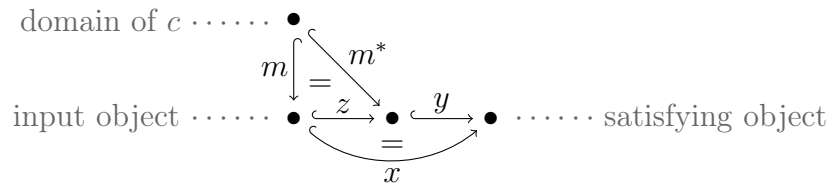
$$\begin{aligned} \langle \text{id}_I, i_M, \text{id}_I \rangle \in \llbracket \text{SeekSat}(c) \rrbracket & \text{ implies } M \models c, \\ (\exists H \in \mathcal{C}. H \models c) & \text{ implies } \exists M \in \mathcal{C}. \langle \text{id}_I, i_M, \text{id}_I \rangle \in \llbracket \text{SeekSat}(c) \rrbracket. \end{aligned}$$

The proof of Theorem 6.20 is based on the following lemma.

Lemma 6.21 (Sat). For each condition c in \mathcal{MNF} , $\text{Sat}(c)$ is a program that, with respect to the satisfiability problem, is

(sound) $\forall m, m^* \in \mathcal{M}. \langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket \text{ implies } m^* \models c,$

(complete) $\forall m, x \in \mathcal{M}. x \circ m \models c \text{ implies } \exists m^*, y, z \in \mathcal{M}. \langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket, m^* = z \circ m \text{ and } x = y \circ z.$



Completeness means, if a given \mathcal{M} -morphism m can be extended by an \mathcal{M} -morphism x to $x \circ m$ such that it satisfies c , there is a transition from m to an \mathcal{M} -morphism m^* ($m^* = z \circ m$ for some \mathcal{M} -morphism z) and m^* lies exactly along the way towards $x \circ m$ ($x = y \circ z$), preserving the possibility to eventually yield $x \circ m$. While this does not imply a guarantee to reach $x \circ m$ at all, we can reach a possibly smaller morphism, namely m^* that also satisfies the subcondition c (see soundness).

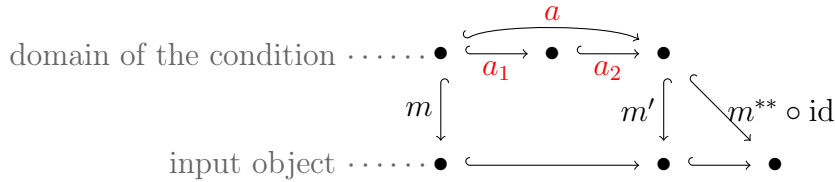
Proof of Lemma 6.21. We prove the soundness of $\text{Sat}(c)$ by induction over the definition of Sat .

Basis. $\text{Sat}(\text{true})$: all \mathcal{M} -morphisms $m \in \mathcal{M}$ satisfy true .

$\text{Sat}(\text{false})$: there is no triple $\langle m, m^*, \text{id} \rangle$ in $\llbracket \text{Sat}(\text{false}) \rrbracket = \llbracket \text{Abort} \rrbracket = \emptyset$.

Hypothesis. For a given condition c , $\text{Sat}(c)$ is sound.

Step. $\text{Sat}(\exists(a, c))$: $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\exists(a, c)) \rrbracket = \llbracket \text{if } \neg \exists(a, c) \text{ then } \dots \text{fi} \rrbracket \subseteq \llbracket \text{Fix}(\bigcup_{a_2 \circ a_1 = a} \{ \text{Sel}(a_1); \text{Add}(a_2) \}; \text{Sat}(c)) \rrbracket$ implies $\langle m, m^*, m_{\text{Sat}(\exists(a, c))} \rangle \in \llbracket \bigcup_{a_2 \circ a_1 = a} \{ \text{Sel}(a_1); \text{Add}(a_2) \}; \text{Sat}(c) \rrbracket$ for some $m^{**} \in \mathcal{M}$ with $m^* = m^{**} \circ m_{\text{Sat}(\exists(a, c))}$ (Def. Fix, Fact 6.17), which implies there is a $m' \in \mathcal{M}$ and some decomposition $a_2 \circ a_1 = a$ of a such that $\langle m, m', a \rangle \in \llbracket \text{Sel}(a_1); \text{Add}(a_2) \rrbracket$ and $\langle m', m^{**}, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$ with $m_{\text{Sat}(\exists(a, c))} = \text{id} \circ a$ (a_1, a_2 may be isomorphisms, semantic of seq. comp., Fact 6.15), which implies $m^{**} \circ a \models \exists a$ and $m^{**} \circ \text{id} \models c$ (inductive hypothesis) implies $m^* \models \exists(a, c)$.



$\text{Sat}(\neg \exists(a, c))$: $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\neg \exists(a, c)) \rrbracket = \llbracket \text{while } \exists(a, c) \text{ do } \dots \text{od} \rrbracket$ implies $m^* \models \neg \exists(a, c)$.

$\text{Sat}(\neg \bigwedge_{j \in J} c_j)$: $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\neg \bigwedge_{j \in J} c_j) \rrbracket = \llbracket \text{if } \bigwedge_{j \in J} c_j \text{ then } \bigcup_{j \in J} \{ \text{Sat}(\neg c_j) \} \text{fi} \rrbracket \subseteq \llbracket \bigcup_{j \in J} \{ \text{Sat}(\neg c_j) \} \rrbracket = \bigcup_{j \in J} \llbracket \text{Sat}(\neg c_j) \rrbracket$ implies there is some $j \in J$ with $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\neg c_j) \rrbracket$, which implies there is some $j \in J$ with $m^* \models \neg c_j$ (induction hypothesis), which implies $m^* \models \neg \bigwedge_{j \in J} c_j$.

$\text{Sat}(\bigwedge_{j \in J} c_j)$: $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\bigwedge_{j \in J} c_j) \rrbracket = \llbracket \text{while } (\neg \bigwedge_{j \in J} c_j) \text{ do } \dots \text{od} \rrbracket$ implies $m^* \models \bigwedge_{j \in J} c_j$.

$\text{Sat}(\neg\neg c)$: sound, as $\llbracket \text{Sat}(\neg\neg c) \rrbracket = \llbracket \text{Sat}(c) \rrbracket$ ($\neg\neg c \equiv c$, induction hypothesis).

We prove the completeness of $\text{Sat}(c)$ by induction over its definition. For the proof, we require (for the first time) the so-called pullback-pushout- \mathcal{M} property and certain assumptions on the finiteness of \mathcal{C} such as a finite length of \mathcal{M} -decompositions. For details, we refer to Assumption 2.19 in Chapter 2.

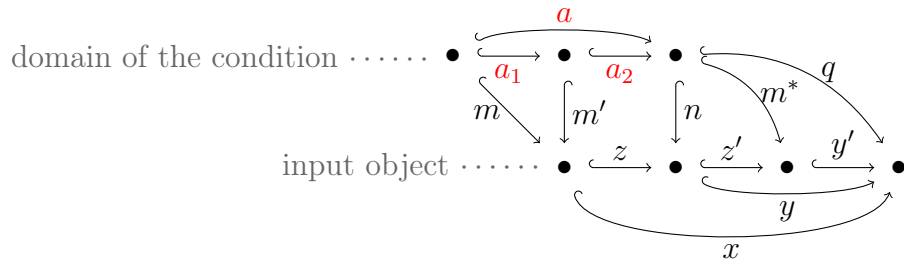
Basis. $\text{Sat}(\text{true})$: for every $m \in \mathcal{M}$, $\langle m, m, \text{id} \rangle \in \llbracket \text{Skip} \rrbracket = \llbracket \text{Sat}(\text{true}) \rrbracket$ with $m = \text{id} \circ m$ and $x = y \circ \text{id}$ ($m^* = m$, $y = x$ and $z = \text{id}$).

$\text{Sat}(\neg \text{true})$: no morphism satisfies not true, hence there are no $m, x \in \mathcal{M}$ such that $x \circ m \models \neg \text{true}$.

Hypothesis. For a given condition c , $\text{Sat}(c)$ is complete.

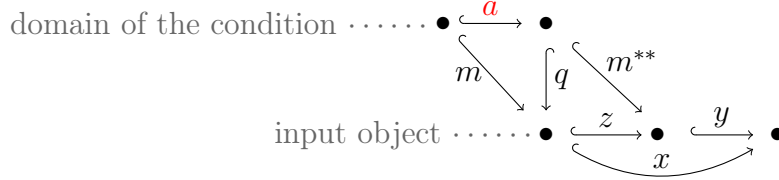
Step. $\text{Sat}(\exists(a, c))$: $x \circ m \models \exists(a, c)$ implies $\exists q \in \mathcal{M}$. $x \circ m = q \circ a$ and $q \models c$. As $x \circ m \models \exists a$, construct $\langle m', a_2 \rangle$ as the pullback of $\langle x, q \rangle$. As $x \circ m = q \circ a$, there is a unique morphism a_1 with $m = m' \circ a_1$ and $a = a_2 \circ a_1$. As $x, q \in \mathcal{M}$, $m', a_2 \in \mathcal{M}$. As $m, m' \in \mathcal{M}$, $a_1 \in \mathcal{M}$. According to the construction, the program $\text{Sel}(a_1); \text{Add}(a_2)$ is in $\bigcup_{a_2 \circ a_1 = a} \{\text{Sel}(a_1); \text{Add}(a_2)\}$. Apply the program (construct the pushout of $\langle m', a_2 \rangle$) to yield $\langle z, n \rangle$. As $x \circ m' = q \circ a_2$, there is a unique morphism y in \mathcal{M} (pullback-pushout- \mathcal{M} property) with $y \circ n = q$ and $x = y \circ z$. Consequently, $\langle m, n, a \rangle \in \llbracket \bigcup_{a_2 \circ a_1 = a} \{\text{Sel}(a_1); \text{Add}(a_2)\} \rrbracket$.

Now $y \circ n = q \models c$ implies $\exists m^*, y', z' \in \mathcal{M}$. $\langle n, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$, $m^* = z' \circ m'$ and $y = y' \circ z'$ (inductive hypothesis for c) implies $\exists m^* \circ a, y', z' \circ z \in \mathcal{M}$. $\langle m, m^* \circ a, \text{id} \rangle \in \llbracket \text{Fix}(\bigcup_{a_2 \circ a_1 = a} \{\text{Sel}(a_1); \text{Add}(a_2)\}; \text{Sat}(c)) \rrbracket = \llbracket \text{Sat}(\exists(a, c)) \rrbracket$, $m^* \circ a = z' \circ z \circ m$ and $x = y' \circ z' \circ z$.



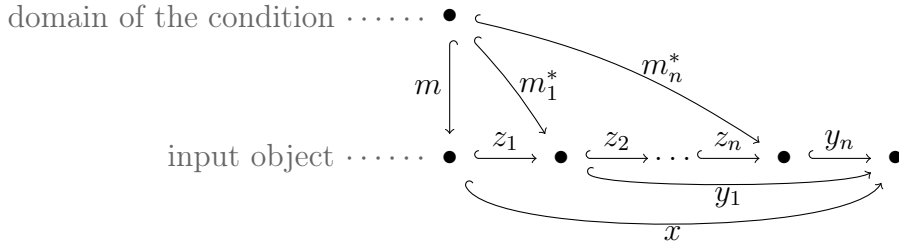
$\text{Sat}(\neg \exists(a, c))$: Assume $x \circ m \models \neg \exists(a, c)$. If $m \models \neg \exists(a, c)$, $\langle m, m, \text{id} \rangle \in \llbracket \text{while } \exists(a, c) \text{ do } \dots \text{od} \rrbracket = \llbracket \text{Sat}(\neg \exists(a, c)) \rrbracket$ with $m = \text{id} \circ m$ and $x = y \circ \text{id}$ ($m^* = m$, $y = x$ and $z = \text{id}$). If $m \models \exists(a, c)$, there is a morphism q in \mathcal{M} with $m = q \circ a$, $q \models c$ and $\langle m, q, a \rangle \in \llbracket \text{Sel}(a, c) \rrbracket$. As $x \circ m \models \neg \exists(a, c)$, $x \circ q \models \neg c$. By induction hypothesis, there $\exists m^*, y, z \in \mathcal{M}$. $\langle q, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$, $m^* = z \circ q$ and $x = y \circ z$.

Consequently, $\langle m, m^{**} \circ a, \text{id} \rangle \in \llbracket \text{Sat}(\neg \exists(a, c)) \rrbracket$, $m^{**} \circ a = z \circ m$ and $x = y \circ z$.



$\text{Sat}(\neg \wedge_{j \in J} c_j)$: $x \circ m \models \neg \wedge_{j \in J} c_j$ implies $\exists j \in J. x \circ m \models \neg c_j$ implies $\exists j \in J \exists m^*, y, z \in \mathcal{M}. \langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\neg c_j) \rrbracket \subseteq \bigcup_{j \in J} \llbracket \text{Sat}(\neg c_j) \rrbracket = \llbracket \bigcup_{j \in J} \{\text{Sat}(\neg c_j)\} \rrbracket = \llbracket \bigcup_{j \in J} \{\text{Sat}(\neg c_j)\} \rrbracket = \llbracket \text{Sat}(\neg \wedge_{j \in J} c_j) \rrbracket$, $m^* = z \circ m$ and $x = y \circ z$ (inductive hypothesis).

$\text{Sat}(\wedge_{j \in J} c_j)$: $x \circ m \models \wedge_{j \in J} c_j$ implies $\forall j \in J. x \circ m \models c_j$ implies $\forall j \in J \exists m_j^*, y_j, z_j \in \mathcal{M}. \langle m, m_j^*, \text{id} \rangle \in \llbracket \text{Sat}(c_j) \rrbracket$, $m_j^* = z_j \circ m$ and $x = y_j \circ z_j$ (inductive hypothesis). By induction over the length of the sequential composition j , one can show $\exists m^*, y, z \in \mathcal{M}. \langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\wedge_{j \in J} c_j) \rrbracket$, $m^* = z \circ m$ and $x = y \circ z$. By the monotonicity of Sat (Corollary 6.18) and the finite length of every \mathcal{M} -decomposition, $\exists m^*, y, z \in \mathcal{M}. \langle m, m^*, \text{id} \rangle \in \llbracket \text{while}(\neg \wedge_{j \in J} c_j \text{ do } (\text{;}_{j \in J} \text{Sat}(c_j)) \text{ od} \rrbracket = \llbracket \text{Sat}(\wedge_{j \in J} c_j) \rrbracket = \llbracket \text{Sat}(\wedge_{j \in J} c_j) \rrbracket$, $m^* = z \circ m$ and $x = y \circ z$.

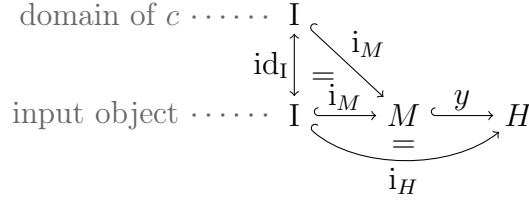


$\text{Sat}(\neg \neg c)$: complete, as $\llbracket \text{Sat}(\neg \neg c) \rrbracket = \llbracket \text{Sat}(c) \rrbracket$ ($\neg \neg c \equiv c$, induction hypothesis).

□

Proof of Theorem 6.20. As $\text{SeekSat}(c) = \text{Sat}(c)$, the soundness and completeness of $\text{SeekSat}(c)$ is a special case of the soundness and completeness of $\text{Sat}(c)$. More precisely, as $m = \text{id}_I: I \leftrightarrow I$ and any morphism from the initial object is in \mathcal{M} and c is a condition over I in \mathcal{MNF} , the soundness of $\text{Sat}(c)$ reduces to $\langle \text{id}_I, i_M, \text{id}_I \rangle \in \llbracket \text{Sat}(c) \rrbracket$ implies $i_M \models c$, which implies

$M \models c$, and the completeness of $\text{Sat}(c)$ reduces to $H \models c$ implies $i_H \models c$, which implies $\exists i_M$. $\langle \text{id}_I, i_M, \text{id}_I \rangle \in \llbracket \text{Sat}(c) \rrbracket$.



□

SeekSat is guaranteed to terminate for a certain fragment of conditions. Hence it is able to decide the satisfiability problem of this subclass.

Definition 6.22 (non-nested fragment). Let NonNested be the set of conditions such that for every subcondition $\exists(a, c)$, $c = \text{true}$.

The non-nested fragment of conditions includes the well-known fragment of negative application conditions (NAC) as introduced in [HHT96].

Theorem 6.23. *For the NonNested fragment of conditions, **SeekSat** is guaranteed to terminate.*

For the termination of $\text{SeekSat}(c)$ with $c \in \text{NonNested}$, it suffices to show that all conjunctive while-loops terminate. For the proof, we will use the fact that for each $c \in \text{NonNested}$, $\text{Sat}(c)$ either does not change the input, or reduces the number of subconditions for which the satisfiability of c is not guaranteed from m^* on for every possible extension x .

Proof. There is a termination function $\text{val}: \mathcal{M} \times \text{NonNested} \rightarrow \mathbb{N}$ such that $\text{val}(m, c) > \text{val}(m^*, c)$ or $m = m^*$, for all conditions $c \in \text{NonNested}$, for all $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$.

Define val as follows: $\text{val}(m, \text{true}) = 0$, $\text{val}(m, \neg c) = \text{val}(m, c)$, $\text{val}(m, \bigwedge_{j \in J} c_j) = \sum_{j \in J} \text{val}(m, c_j)$ and $\text{val}(m, \exists a) = 0$ if $m \models \exists a$, 1 otherwise.

First, $\text{val}(m, c)$ never increases for any condition $c \in \text{NonNested}$ and any extension $x \in \mathcal{M}$, that is, for every $x \circ m \in \mathcal{M}$, $\text{val}(m, c) \geq \text{val}(x \circ m, c)$, as $m \models \exists a$ implies $x \circ m \models \exists a$. Therefore $\text{val}(m, c) = 0$ implies $\text{val}(x \circ m, c) = 0$, for every $x \circ m \in \mathcal{M}$.

Second, for all conditions $c \in \text{NonNested}$, for all $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$, $\text{val}(m, c) > \text{val}(m^*, c)$ or $m = m^*$. By induction over NonNested , we prove:

Basis. For conditions $c = \text{true}$, $c = \text{false}$ or $c = \neg \exists a$, $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c) \rrbracket$ implies $m = m^*$, therefore $\text{val}(m, c) > \text{val}(m^*, c)$ or $m = m^*$.

For $\langle m, m^*, \text{id} \rangle \in \text{Sat}(\exists a)$, either we have $m \not\models \exists a$ and $1 = \text{val}(m, \exists a) > \text{val}(m^*, \exists a) = 0$, or we have $m \models \exists a$ and $m = m^*$.

Hypothesis. Assume, the statement holds for given NonNested conditions. **Step.** For $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\bigvee_{j \in J} c_j) \rrbracket$, there is a $j \in J$ such that $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(c_j) \rrbracket$. By inductive hypothesis, $\text{val}(m, c_j) > \text{val}(m^*, c_j)$ and $\sum_{j \in J} \text{val}(m, c_j) > \sum_{j \in J} \text{val}(m^*, c_j)$ (val never increases), or we have $m = m^*$. Similarly, for $\langle m, m^*, \text{id} \rangle \in \llbracket \text{Sat}(\bigwedge_{j \in J} c_j) \rrbracket$, each step $\langle m_j, m_j^*, \text{id} \rangle \in \llbracket \text{Sat}(c_j) \rrbracket$ in the sequence has the property $\text{val}(m_j, c_j) > \text{val}(m_j^*, c_j)$ or $m_j = m_j^*$. As val never increases, we have $\sum_{j \in J} \text{val}(m, c_j) > \sum_{j \in J} \text{val}(m^*, c_j)$ or $m = m^*$. \square

As already stated, each instance of the tautology problem may be viewed as an instance of the satisfiability problem, by negating both the input condition, as well as the answer. However, in contrast to positives answers, negative answers of a satisfiability algorithm may only be lifted to the tautology problem in case of termination and completeness. Otherwise, the incompleteness of the algorithm would correspond to unsoundness in the case of the tautology problem. A consequence of Theorem 6.20, Theorem 6.23 and Corollary 6.5 is the following:

Corollary 6.24. For the NonNested fragment of conditions, **SeekSat** decides the tautology problem.

Despite being a complete satisfiability solver, **SeekSat** does not cover the tautology problem for all tautologies. In fact, there are provable tautologies outside the decidable fragment of conditions for which **SeekSat** never terminates, therefore **SeekSat** cannot substitute a dedicated theorem prover.

Example 6.25 (non-termination). Consider the example graph condition $\forall(\bigcirc_1, \exists \bigcirc_2) \Rightarrow \forall(\bigcirc_1 \bigcirc_2, \exists \bigcirc_1 \bigcirc_2)$ with the meaning “Every node has a loop implies for every pair of nodes, each node has a loop”, which is clearly a valid statement. **SeekSat** cannot show the unsatisfiability of $c = \neg \exists(\bigcirc_1, \neg \exists \bigcirc_1) \wedge \exists(\bigcirc_1 \bigcirc_2, \neg \exists \bigcirc_1 \bigcirc_2)$. In each iteration, **SeekSat**(c) will first add a loop to every loop-free node, then add at least two nodes without a loop.

This clearly motivates the need for an algorithm dedicated to proving conditions.

We compare our satisfiability solver with related work in Section 6.4.

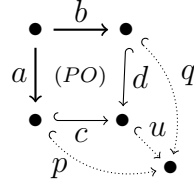
6.3 Theorem prover PROCON

In the previous section we investigated a sound and complete satisfiability algorithm that, for a certain fragment of conditions, decides the tautology

problem of conditions. However, we also showed that there are provable tautologies for which the algorithm does not terminate. To cover the full class of conditions, we now present a resolution-based calculus for proving conditions, show its soundness, and investigate the necessity of each deduction rule.

Beforehand, we introduce the notion of an \mathcal{M} -pushout, which we require for a certain rule in our calculus. An \mathcal{M} -pushout is a special pushout for which it can be guaranteed that the unique morphism u is in \mathcal{M} , if the commutative morphisms p, q are in \mathcal{M} .

Definition 6.26 (\mathcal{M} -pushout). A pushout $c \circ a = d \circ b$ with $c, d \in \mathcal{M}$ is called \mathcal{M} -pushout, if for all \mathcal{M} -morphisms p, q with $p \circ a = q \circ b$, the unique existing morphism u with $p = c \circ u$ and $q = d \circ u$ is in \mathcal{M} .



We use the following characterization of \mathcal{M} -pushouts.

Fact 6.27 (\mathcal{M} -pushout). A pushout $c \circ a = d \circ b$ with $c, d \in \mathcal{M}$ is an \mathcal{M} -pushout, if and only if for all epimorphisms e with $\text{dom}(e) = \text{codom}(d)$ we have $e \notin \mathcal{M}$ implies $e \circ c \notin \mathcal{M}$ or $e \circ d \notin \mathcal{M}$.

Proof. Via epi- \mathcal{M} -factorization using the converse statement.

$$\begin{aligned}
& e \notin \mathcal{M} \text{ implies } (e \circ c \notin \mathcal{M} \text{ or } e \circ d \notin \mathcal{M}) && \text{(characterization)} \\
\Leftrightarrow & \text{not } (e \circ c \notin \mathcal{M} \text{ or } e \circ d \notin \mathcal{M}) \text{ implies not } e \notin \mathcal{M} && \text{(converse)} \\
\Leftrightarrow & (e \circ c \in \mathcal{M} \text{ and } e \circ d \in \mathcal{M}) \text{ implies } e \in \mathcal{M} && \text{(deMorgan)} \\
\Leftrightarrow & (m \circ e \circ c \in \mathcal{M} \text{ and } m \circ e \circ d \in \mathcal{M}) \text{ implies } m \circ e \in \mathcal{M} && \left(\begin{array}{l} m \in \mathcal{M}, \mathcal{M} \text{ closed} \\ \text{under comp./decomp.} \end{array} \right) \\
\Leftrightarrow & (u \circ c \in \mathcal{M} \text{ and } u \circ d \in \mathcal{M}) \text{ implies } u \in \mathcal{M} && \text{(epi-}\mathcal{M}\text{-factorization)} \\
\Leftrightarrow & (p \in \mathcal{M} \text{ and } q \in \mathcal{M}) \text{ implies } u \in \mathcal{M} && \text{(commutativity)}
\end{aligned}$$

□

A straightforward approach to answer the tautology problem for a condition c is to deduce “true $\Rightarrow c$ ”, that is, to show that c is a logical consequence of true. This can be done by constructing a proof chain “true $\Rightarrow \dots \Rightarrow c$ ”, starting without any assumptions (true), yielding in logical deductions the given condition c . Instead of constructing such a proof top-down, resolution follows a more target-oriented view and considers the complementary problem of refuting the negated condition “ $\neg c$ ”. In this case, the goal is to find a refutation “ $\neg c \Rightarrow \dots \Rightarrow \text{false}$ ”.

After negation of an input F , a resolution-based algorithm on formulas [Rob65] would transform the negated statement $\neg F$ into prenex normal form and skolemize to yield clauses. However neither does there exist a comparable normal form for conditions, nor is skolemization possible for a given category such as **Graphs**: Skolemization requires the introduction of fresh function symbols of unbounded arity, for which there seems to be no equivalent operation on a fixed structure. Nevertheless, it is possible to transform the negated input $\neg c$ into conjunctive normal form.

Definition 6.28 (conjunctive normal form). The empty conjunction $\bigwedge_{j \in \emptyset} c_j \equiv \text{true}$ is in *conjunctive normal form (CNF)*. Every condition $\bigwedge_{j \in J} \bigvee_{k \in K_j} c_k$ is in CNF, if for every $j \in J$ and every $k \in K_j$, $c_k = \exists(a_k, d_k)$ or $c_k = \neg \exists(a_k, d_k)$, where $a_k \in \mathcal{M}$ is not an isomorphism and d_k is a condition in CNF.

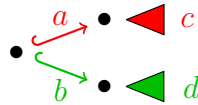
In general, there is no way to get rid of the nesting of a condition. For every existential subcondition $\exists(a, c)$ of a condition in CNF, c is again a (possibly non-empty) conjunction of disjunctions.

If a true resolution calculus were possible for conditions, derived facts (disjunctions) could now be added to the outermost conjunction. The goal would be the addition of false as a conjunct. Each refutation step “ \Rightarrow ” would be of the form:

- *Select* two disjunctions $(\neg \exists(a, c) \vee c_1)$ and $(\exists(b, d) \vee c_2)$ from the conjunction such that $\exists(b, d) \Rightarrow \exists(a, c)$.
- *Add the resolvent* $(c_1 \vee c_2)$ to that conjunction.

Special case: if c_1 and c_2 do not exist, or equivalently, are false, the resolvent is false and the negated input condition is refuted, which is the goal of this procedure.

However, we cannot restrict deductions to the outermost conjunction: We can decide $\exists b \Rightarrow \exists a$ by checking, if there is an \mathcal{M} -morphism m such that $m \circ a = b$. However, to decide $\exists(b, d) \Rightarrow \exists(a, c)$ is as hard as the original problem $\text{true} \Rightarrow c_{\text{input}}$ as it contains the subproblem $d \Rightarrow c$. Even more problematic, d and c are over different domains, and we cannot move d and c towards the root of the condition without losing information. Only a condition false may descend against a morphism, that is, $\exists(a, \text{false}) \equiv \text{false}$.



However, recall that we can shift conditions along \mathcal{M} -morphisms. If necessary, we can “lift” the whole condition $\exists(a, c)$ along b to bring c and d closer together. We reuse transformation $A(b, \exists(a, c))$ of Section 5.2 which, for an \mathcal{M} -morphism b , shifts $\exists(a, c)$ over b to yield a condition over $\text{codom}(b)$ of same depth, representing the incorporation of $\exists b$ into $\exists(a, c)$.

$$\exists(a, c) \blacktriangleright \bullet \xrightarrow{b} \bullet \blacktriangleleft A(b, \exists(a, c))$$

Besides a core resolution rule as described above, we will consider additional rules that create, manage and (hopefully) solve nested subproblems of this form to decide $\exists(b, d) \Rightarrow \exists(a, c)$.

First, we define the notion of a non-negated subcondition, which we use thereafter to restrict the applicability of deduction rules.

Definition 6.29 (non-negated subcondition). A condition c is a *non-negated subcondition* of a condition d , if $c = d$ or if d is of the form $\exists(a, e)$ or $\wedge_{j \in J} e_j$ or $\vee_{j \in J} e_j$ and c is a non-negated subcondition of e or e_j for some $j \in J$.

Formally, these deduction rules are defined as follows.

Definition 6.30 (deduction rules). Let c_1, \dots, c_n, e be conditions. A (*deduction*) rule R has the form

$$\boxed{\begin{array}{c} c_1 \\ \vdots \\ c_n \\ \hline e \end{array}} \quad \text{if } \alpha$$

and is shortly denoted by $R = [c_1, \dots, c_n / e] \alpha$. The conditions c_1, \dots, c_n are called *premises*, e is the *resolvent* and α is an (informal) *side condition*. A rule may be applied to a condition c in CNF if there exists a non-negated subcondition c' in c such that $c' = \wedge_{j \in J} d_j$ is a conjunction of disjunctions $(d_j)_{j \in J}$ that contains all premises of R , that is, for all $1 \leq k \leq n$, there is a $j \in J$ with $c_k = d_j$, and the side condition α is satisfied. Application of R yields a new condition d that is derived from c by adding the resolvent e (brought into CNF if necessary) to the conjunction c' . We write $c \vdash_R d$ to denote such a derivation step, whereas we write $c \vdash_{\mathcal{K}} d$ to denote a derivation sequence $c \vdash_R \dots \vdash_Q d$ with rules R, \dots, Q in some rule set \mathcal{K} .

The deduction rules of our calculus are possibly applicable on any non-negated subconditions within a condition. The rules themselves contain

variables for morphisms and conditions. Prior to a rule application, these variables must be matched in an unification process, as usual, to yield an applicable instance of the rule.

We now present our calculus \mathcal{K} for proving high-level conditions, representing the possible actions our theorem prover PROCON performs.

Definition 6.31 (calculus \mathcal{K}). The calculus \mathcal{K} for high-level conditions consists of the following four core rules: (Descend), (Partial resolve), (Lift) and (Supporting lift). Let a, b, m be morphisms and let c, d, c_1, c_2 be conditions.

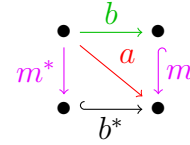
(Descend)

$$\frac{\exists(a, \text{false} \wedge c) \vee c_1}{c_1}$$

(Partial resolve)

$$\frac{\neg\exists(a, \text{true}) \vee c_1 \quad \exists(b, d) \vee c_2}{\neg\exists(m^*, \text{true}) \vee c_1 \vee c_2}$$

if $\exists m \in \mathcal{M}. m \circ b = a$ and $\langle m^*, b^* \rangle$ is the \mathcal{M} -pushout complement of $\langle b, m \rangle$ and $d \neq \text{false}$



(Lift)

$$\frac{\neg\exists(a, c) \vee c_1 \quad \exists(b, d) \vee c_2}{\exists(b, d \wedge \mathbf{A}(b, \neg\exists(a, c))) \vee c_1 \vee c_2}$$

if $c \neq \text{false}$
and $d \neq \text{false}$

(Supporting lift)

$$\frac{\exists(a, c) \vee c_1 \quad \exists(b, d) \vee c_2}{\exists(b, d \wedge \mathbf{A}(b, \exists(a, c))) \vee c_1 \vee c_2}$$

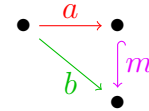
if $c \neq \text{false}$
and $d \neq \text{false}$

To simplify manual proofs, we introduce the deduction rules (Resolve) and (Partial lift). They can be derived using the above rules, as we show later:

(Resolve)

$$\frac{\neg\exists(a, \text{true}) \vee c_1 \quad \exists(b, d) \vee c_2}{c_1 \vee c_2}$$

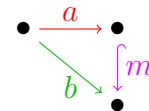
if $\exists m \in \mathcal{M}. m \circ a = b$
and $d \neq \text{false}$



(Partial lift)

$$\frac{\neg\exists(a, c) \vee c_1 \quad \exists(b, d) \vee c_2}{\exists(b, d \wedge \mathbf{A}(m, \neg c)) \vee c_1 \vee c_2}$$

if $\exists m \in \mathcal{M}. m \circ a = b$
and $c \neq \text{false}$
and $d \neq \text{false}$



The rule (Descend) is used to carry over a successful nested refutation into an outer refutation and is the only core rule that may reduce the number of disjuncts in a disjunction. The rule (Partial resolve) is necessary for proving the validity of conditions outside the decidable fragment of conditions. The rules (Lift) and (Supporting lift) are similar in the sense that they combine informations from different conditions creating additional facts for nested refutations. Note that (Supporting lift) is the only rule for which repeated applications on its own resolvent may be necessary. For the category **Graphs**, unbounded applications of (Supporting lift) are the only reason a theorem prover based on \mathcal{K} does not terminate, assuming that the deduction of structurally equivalent conditions is suppressed, as discussed later on.

The rule (Resolve) represents a straightforward case for which the problem $\exists(b, d) \Rightarrow \exists(a, c)$ is decidable. (Resolve) will be shown to be a special case of (Descend) and (Lift). The rule (Partial lift) is also a special case of (Lift) which only considers a preselected combination of the two facts $\neg\exists(a, c)$ and $\exists(b, d)$, given by the morphism m .

Example 6.32. Consider the tautology $\forall(\bigcirc_1, \exists\bigcirc_1\bigcirc_1) \Rightarrow \forall(\bigcirc_2\bigcirc_3, \exists\bigcirc_2\bigcirc_3)$ expressing “Every node has a loop implies for every pair of nodes, each node has a loop”. A transformation of the negated statement into CNF yields

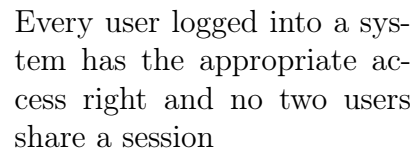
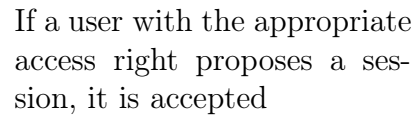
$$\begin{aligned} (1) \quad & \neg\exists(\bigcirc_1, \neg\exists\bigcirc_1\bigcirc_1) \\ (2) \quad & \wedge \exists(\bigcirc_2\bigcirc_3, \neg\exists\bigcirc_2\bigcirc_3) \end{aligned}$$

We showed in Example 6.25 that **SeekSat** does not terminate for this input, therefore is unable too prove the unsatisfiability. However, to prove the statement’s validity using \mathcal{K} is as follows:

$\begin{array}{l} (1) \quad \neg\exists(\bigcirc_1, \neg\exists\bigcirc_1\bigcirc_1) \\ (2) \quad \exists(\bigcirc_2\bigcirc_3, \neg\exists\bigcirc_2\bigcirc_3) \end{array}$	
$\begin{array}{l} (3) \quad \exists(\bigcirc_2\bigcirc_3, (3.1) \quad \neg\exists\bigcirc_2\bigcirc_3) \\ \quad \quad \quad (3.2) \quad \wedge \quad \exists\bigcirc_2\bigcirc_3 \\ \quad \quad \quad (3.3) \quad \wedge \quad \exists\bigcirc_2\bigcirc_3 \\ \quad \quad \quad (3.4) \quad \wedge \quad \neg\exists(\bigcirc_2\bigcirc_3, \neg\exists\bigcirc_2\bigcirc_3) \end{array}$	(Lift)

$\begin{array}{l} (3.1) \quad \neg\exists\bigcirc_2\bigcirc_3 \\ (3.2) \quad \exists\bigcirc_2\bigcirc_3 \\ \hline (3.5) \quad \neg\exists\bigcirc_2\bigcirc_3 \end{array}$	(Partial resolve)
---	-------------------

Every user logged into a system has the appropriate access right and no two users share a session


$$\Rightarrow \left(\begin{array}{l} \forall \left(\text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox}, \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \right) \\ \wedge \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Person} \\ \vee \neg \exists \left(\text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox}, \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \right) \\ \wedge \vee \neg \exists \left(\text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox}, \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \right) \\ \wedge \vee \neg \exists \left(\text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Mailbox}, \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Mailbox} \right) \\ \wedge \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \\ \wedge \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \\ \wedge \vee \neg \exists \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \rightarrow \text{Person} \rightarrow \text{Phone} \rightarrow \text{Mailbox} \end{array} \right)$$

is a tautology. After negation and transformation into CNF, we yield the condition below and it remains to show its unsatisfiability by deriving false.

$$\begin{aligned}
 & (1) \quad \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \leftarrow \text{person)} \\
 \wedge & (2) \quad \forall \neg \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \leftarrow \text{phone)} , \wedge \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \leftarrow \text{phone)} \right) \\
 \wedge & (3.1) \quad \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{person)} \\
 \vee & (3.2) \quad \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{person)} \\
 \vee & (3.3) \quad \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{person)} \\
 \vee & (3.4) \quad \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} , \wedge \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \leftarrow \text{phone)} \right) \\
 \vee & (3.5) \quad \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} , \wedge \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} \right) \\
 \vee & (3.6) \quad \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{person)} , \wedge \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} \right) \\
 \vee & (3.7) \quad \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} , \wedge \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} \right)
 \end{aligned}$$

Given the rules of \mathcal{K} , our goal is to refute the disjunction (3) with the help of the facts (1) and (2). The rule (Resolve) can be applied with argument (1) to resolve (3.1)-(3.3), for instance,

$$\boxed{
 \begin{array}{l}
 (1) \\
 (3.1) \vee ((3.2) \vee \dots \vee (3.7)) \\
 \hline
 (3.2) \vee \dots \vee (3.7)
 \end{array}
 } \quad (\text{Resolve})$$

Subconditions (3.4)-(3.7) are resolved by applying rule (Partial lift) with argument (2) and subsequent application of (Resolve) on the nested subconditions, and (Descend), for instance,

$$\boxed{
 \begin{array}{l}
 (2) \\
 (3.7) \vee ((3.4) \vee \dots \vee (3.6)) \\
 \hline
 (3.7') \vee (3.4) \vee \dots \vee (3.6)
 \end{array}
 } \quad (\text{Partial lift})$$

$$\text{with } (3.7') \quad \exists \left(\text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} , \forall \neg \exists \text{ (diagram: person} \rightarrow \text{phone} \rightarrow \text{phone} \leftarrow \text{phone)} \right)$$

Eventually, we yield an empty disjunction, or equivalently, false as an element of the outer conjunction, thus the negated input condition is refuted and the original condition proved.

The main result of this section is the soundness of the calculus \mathcal{K} . We show that every application of a rule R in \mathcal{K} corresponds to a logical deduction.

Theorem 6.34 (soundness of \mathcal{K}). *The calculus \mathcal{K} for high-level conditions is sound, that is, for every conditions c, d over C in CNF the following holds:*

$$c \vdash_{\mathcal{K}} d \text{ implies } c \Rightarrow d.$$

The proof is done in three steps: first, we establish that we can investigate the soundness of deduction rules independently of disjunctive context. In the following, let r, p_j, q_j be conditions for $1 \leq j \leq n$.

Fact 6.35. For rules $[(p_1 \vee q_1), \dots, (p_n \vee q_n) / (r \vee q_1 \vee \dots \vee q_n)]\alpha$ we have $(p_1 \wedge \dots \wedge p_n) \Rightarrow r$ implies $((p_1 \vee q_1) \wedge \dots \wedge (p_n \vee q_n)) \Rightarrow (r \vee q_1 \vee \dots \vee q_n)$.

Second, we can prove the soundness of each individual rule R in \mathcal{K} .

Lemma 6.36. For every rule $R = [c_1, \dots, c_n / d]\alpha$ in \mathcal{K} , if α holds then $(c_1 \wedge \dots \wedge c_n) \Rightarrow d$.

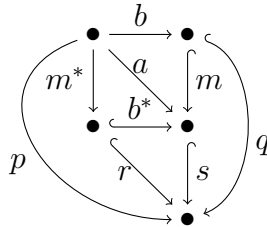
Proof. For the rule (Descend), we have $\exists(a, \text{false} \wedge c) \equiv \exists(a, \text{false}) \equiv \text{false}$. For every rule of the form $R = [(p_1 \vee q_1), \dots, (p_n \vee q_n) / (r \vee q_1 \vee \dots \vee q_n)]\alpha$ with $c_j = (p_j \vee q_j)$ for $1 \leq j \leq n$, we first show $(p_1 \wedge \dots \wedge p_n) \Rightarrow r$:

(Partial resolve). First, we transform the proof obligation:

$$\begin{aligned} & (\neg \exists(a, \text{true}) \wedge \exists(b, d)) \Rightarrow \neg \exists(m^*, \text{true}) \\ \equiv & \neg(\neg \exists(a, \text{true}) \wedge \exists(b, d)) \vee \neg \exists(m^*, \text{true}) & (\text{Def. } \Rightarrow) \\ \equiv & \exists(a, \text{true}) \vee \neg \exists(b, d) \vee \neg \exists(m^*, \text{true}) & (\text{De Morgan}) \\ \equiv & \exists(a, \text{true}) \Leftarrow \neg(\neg \exists(b, d) \vee \neg \exists(m^*, \text{true})) & (\text{Def. } \Rightarrow) \\ \equiv & \exists(a, \text{true}) \Leftarrow (\exists(b, d) \wedge \exists(m^*, \text{true})) & (\text{De Morgan}) \end{aligned}$$

Finally, we show $(\exists(b, d) \wedge \exists(m^*, \text{true})) \Rightarrow \exists(a, \text{true})$:

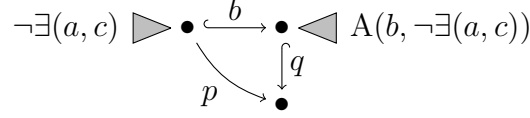
$$\begin{aligned} & p \models (\exists(b, d) \wedge \exists(m^*, \text{true})) \\ \Leftrightarrow & \exists q \in \mathcal{M}. q \circ b = p \text{ and } q \models d \\ & \text{and } \exists r \in \mathcal{M}. r \circ m^* = p \text{ and } r \models \text{true} & (\text{Def. 3.1}) \\ \Rightarrow & \exists s \in \mathcal{M}. r \circ m^* = s \circ b^* \circ m^* = s \circ a = p \text{ and } s \models \text{true} & (\mathcal{M}\text{-Pushout}) \end{aligned}$$



(Lift). We show $\exists(b, d) \wedge \neg \exists(a, c) \Rightarrow \exists(b, d \wedge \neg \exists(a, c))$:

$$\begin{aligned} & p \models \exists(b, d) \wedge \neg \exists(a, c) \\ \Leftrightarrow & \exists q \in \mathcal{M}. q \circ b = p \text{ and } q \models d \text{ and } q \circ b \models \neg \exists(a, c) & (\text{Def. 3.1}) \end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow \exists q \in \mathcal{M}. q \circ b = p \text{ and } q \models d \text{ and } q \models \mathbf{A}(b, \neg\exists(a, c)) && (\text{Lem. 5.4}) \\
&\Leftrightarrow p \models \exists(b, d \wedge \mathbf{A}(b, \neg\exists(a, c))) && (\text{Def. 3.1})
\end{aligned}$$



(Supporting lift). The proof is analogous to (Lift) except $\neg\exists(a, c)$ is replaced with $\exists(a, c)$.

By Fact 6.35, we can lift any statement $(p_1 \wedge \dots \wedge p_n \Rightarrow r)$ for any disjunctive context q_1, \dots, q_n and yield $(p_1 \vee q_1) \wedge \dots \wedge (p_n \vee q_n) \Rightarrow (r \vee q_1 \vee \dots \vee q_n)$. This concludes the soundness proof for the deduction rules in \mathcal{K} . \square

Third, we show that deductions concerning non-negated subconditions within a condition in CNF can be lifted to the whole condition.

Fact 6.37. For any non-negated condition c' within a condition c over C in CNF, with d derived from c by replacing c' with d' , we have $c' \Rightarrow d'$ implies $c \Rightarrow d$.

Proof. By induction over the structure of conditions.

Basis: $c = c' \Rightarrow d' = d$.

Step: We show, for all morphisms m in \mathcal{M} with domain C :

Case $\exists(a, c)$: $m \models \exists(a, c)$ iff $(\exists q \in \mathcal{M}. m = q \circ a \text{ and } q \models c)$ implies $(\exists q \in \mathcal{M}. m = q \circ a \text{ and } q \models d)$ iff $m \models \exists(a, d)$.

Case $(c \wedge e)$: $m \models (c \wedge e)$ iff $(m \models c \text{ and } m \models e)$ implies $(m \models d \text{ and } m \models e)$ iff $m \models (d \wedge e)$.

Case $(c \vee e)$: analogous to $(c \wedge e)$.

The case $\neg c$ is excluded by the assumption that c' is a non-negated subcondition. \square

Finally, we can prove the soundness of \mathcal{K} .

Proof of Theorem 6.34. Let c, d be arbitrary conditions over C in CNF. A deduction $c \vdash_{\mathcal{K}} d$ is a sequence of deductions $c \vdash_R \dots \vdash_Q d$ for rules R, \dots, Q in \mathcal{K} . Using induction over the length of the deduction, we can reduce the proof obligation to “ $c \vdash_R d$ implies $c \Rightarrow d'$ ”, where c, d are arbitrary conditions over C in CNF and $R = [c_1, \dots, c_n / e] \alpha$ is an arbitrary deduction rule in \mathcal{K} . Assume, $c \vdash_R d$. By Definition 6.30, there is a non-negated subcondition c' which is a conjunction $(c_1 \wedge \dots \wedge c_n \wedge q)$ and d is derived from c by adding e to the conjunction, that is, $(c_1 \wedge \dots \wedge c_n \wedge q) \vdash (e \wedge c_1 \wedge \dots \wedge c_n \wedge q)$. By Lemma 6.36, we have $(c_1 \wedge \dots \wedge c_n) \Rightarrow e$. Consequently, $(c_1 \wedge \dots \wedge c_n \wedge q) \Rightarrow (e \wedge c_1 \wedge \dots \wedge c_n \wedge q)$. By Fact 6.37, we conclude $c \Rightarrow d$. \square

In the following we prove that (Resolve) and (Partial lift) are derivatives of the four core rules (Descend), (Partial resolve), (Lift) and (Supporting lift). To this end, we require the notion $c \subseteq d$, expressing that c is a logical consequence of d and every result c' of a deduction sequence beginning with c is a logical consequence of the same deduction sequence beginning with d .

Definition 6.38. For two conditions c, d we write $c \subseteq d$, if $d \Rightarrow c$ and for all deductions $c \vdash_R c'$, there is a deduction $d \vdash_R d'$ such that $c' \subseteq d'$.

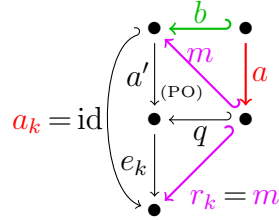
First, we prove (Partial lift) is a special case of (Lift).

Fact 6.39. For every deduction $c' \vdash_{(\text{Partial lift})} d'$, there is a deduction $c' \vdash_{(\text{Lift})} f'$ with $d' \subseteq f'$.

Proof. According to the premises and the side conditions, (Lift) is applicable whenever (Partial lift) is. Concerning the resolvents, it remains to prove $A(m, \neg c) \subseteq A(b, \neg \exists(a, c))$ if $m \circ a = b$.

According to the construction, $A(b, \neg \exists(a, c))$ is of the form $\bigwedge_{k \in K} \neg \exists(a_k, A(r_k, c))$: After constructing the pushout $\langle a', q \rangle$ of $\langle b, a \rangle$, the set K consists of indices for all epimorphisms e_k (up to isomorphism) with domain $\text{codom}(a') = \text{codom}(q)$ such that both $a_k = e_k \circ a'$ and $r_k = e_k \circ q$ are in \mathcal{M} .

As $m \circ a = b$, there is $k \in K$ and an e_k such that a_k is the identity (or an isomorphism) and $r_k = m$ (up to isomorphism), hence in that case $\neg \exists(a_k, A(r_k, c)) \equiv \neg \exists(\text{id}, A(m, c))$:



For the pair $\langle \text{id}, m \rangle$ we have $\text{id} \circ b = b = m \circ a$. Due to the universal property of pushouts, a unique morphism $e: \text{codom}(a') \rightarrow \text{codom}(b)$ with $a' \circ e = \text{id}$ and $q \circ e = m$ exists. As $m \circ a = b$ and $m, b \in \mathcal{M}$, a is in \mathcal{M} (\mathcal{M} closed under decomposition) and therefore $a' \in \mathcal{M}$ (\mathcal{M} closed under pushouts). As $a' \in \mathcal{M}$ and $e \circ a' = \text{id}$, e is an epimorphism. As both m and id are in \mathcal{M} , $e_k = e$ (up to isomorphism) and therefore $a_k = a' \circ e_k = a' \circ e = \text{id}$ and $r_k = q \circ e_k = q \circ e = m$.

Transformation of $\neg \exists(\text{id}, A(m, c))$ into CNF will yield $\neg A(m, c)$, hence $A(m, \neg c) = \neg A(m, c) = \neg \exists(\text{id}, A(m, c)) \subseteq \bigwedge_{k \in K} \neg \exists(a_k, A(r_k, c)) = A(b, \neg \exists(a, c))$, using the fact $d \subseteq d \wedge f$ for all conditions d, f . Using Fact 6.35 and Lemma 6.36, our considerations can be lifted to arbitrary c', d' with $c' \vdash_{(\text{Partial lift})} d'$. \square

Second, every deduction made by (Resolve) may be replaced by a sequence of deductions using (Partial lift) and (Descend).

Fact 6.40. For every deduction $c' \vdash_{(\text{Resolve})} d'$, there is a sequence of deductions $c' \vdash_{(\text{Partial lift})} e' \vdash_{(\text{Descend})} f'$ with $d' \subseteq f'$.

Proof. Let $c' = (\neg\exists(a, \text{true}) \wedge \exists(b, d))$. Then $c' \vdash_{(\text{Partial lift})} e'$ with $e' = (c' \wedge \exists(b, d \wedge \text{false}))$ and $e' \vdash_{(\text{Descend})} f'$ with $f' = (e' \wedge \text{false})$. We have $d' = \text{false} \subseteq (\text{false} \wedge e') = f'$. Using Fact 6.35 and Lemma 6.36, our considerations can be lifted to arbitrary c', d' with $c' \vdash_{(\text{Resolve})} d'$. \square

The deduction rules of \mathcal{K} represent the main computation steps our theorem prover PROCON performs. Besides an implementation of those rules, we require a method that transforms any input condition into conjunctive normal form.

Construction (conjunctive normal form). The equivalences listed in Table 6.3, strictly read from left to right, can be applied as long as possible to transform any condition into an (optimized) condition in conjunctive normal form.

In [Pen04], the equivalences are proven and it is shown that an as long as possible application yields the desired normal form.

Another implementational aspect is the prevention of redundancy of rule applications with the intention of containing non-termination as far as possible. For example, any of the rules (Partial lift), (Lift) or (Supporting lift) may add subconditions to a conjunction that are already present anyway. Repeated application of such a rule on its own resolvent would lead into an infinite redundant branch of the search space. In these cases, a notion of structural equivalence can help to filter out double subconditions to prevent unnecessary deductions.

Definition 6.41 (structural equivalence). Two conditions c over C and d over D are said to be *structurally equivalent* with respect to an isomorphism $m: C \leftrightarrow D$, denoted by $c \hat{=} d$, if $c = \text{true} = d$, or if $c = \neg c'$, $d = \neg d'$ and c', d' are structurally equivalent with respect to m , or if $c = (c_1 \wedge c_2)$, $d = (d_1 \wedge d_2)$ and at least (c_1, d_1) and (c_2, d_2) or (c_1, d_2) and (c_2, d_1) are structurally equivalent (case \vee analogous) with respect to m , or if $c = \exists(a, c')$, $d = \exists(b, d')$ and there exists an isomorphism $m' \circ a = b \circ m$ such that c', d' are structurally equivalent with respect to m' .

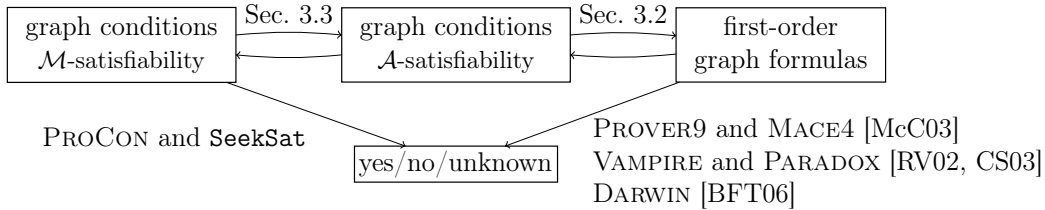
The applicability of deduction rules may then be restricted to those cases for which the resolvent is not structurally equivalent to already existing conditions. Except for the rule (Supporting lift), this effectively prevents recursive application of rules to derived conditions.

$\exists a$	\equiv	$\exists(a, \text{true})$	
$\forall(a, c)$	\equiv	$\neg \exists(a, \neg c)$	
$\exists(a, c)$	\equiv	false	if $a \notin \mathcal{M}$
$\neg \neg c$	\equiv	c	
$\neg \text{true}$	\equiv	false	
$\neg \text{false}$	\equiv	true	
$\exists(a, \bigvee_{j \in J} c_j)$	\equiv	$\bigvee_{j \in J} \exists(a, c_j)$	
$\neg(\bigvee_{j \in J} c_j)$	\equiv	$(\bigwedge_{j \in J} \neg c_j)$	
$\neg(\bigwedge_{j \in J} c_j)$	\equiv	$(\bigvee_{j \in J} \neg c_j)$	
$((\bigwedge_{j \in J} c_j) \vee c)$	\equiv	$(\bigwedge_{j \in J} (c_j \vee c))$	
$\exists(\text{id}, c)$	\equiv	c	
$\exists(a, \text{false})$	\equiv	false	
$\exists(a, \exists(b, c))$	\equiv	$\exists(b \circ a, c)$	
$\bigvee_{j \in J} c_j$	\equiv	true	if $\exists k \in J. c_k = \text{true}$
$\bigvee_{j \in J} c_j$	\equiv	$\bigvee_{j \in J \setminus \{k\}} c_j$	if $\exists k \in J. c_k = \text{false}$
$\bigwedge_{j \in J} c_j$	\equiv	$\bigwedge_{j \in J \setminus \{k\}} c_j$	if $\exists k \in J. c_k = \text{true}$
$\bigwedge_{j \in J} c_j$	\equiv	false	if $\exists k \in J. c_k = \text{false}$
$\bigvee_{j \in \emptyset} c_j$	\equiv	false	
$\bigwedge_{j \in \emptyset} c_j$	\equiv	true	

Table 6.3: Equivalences for conjunctive normal form

6.4 Related concepts

In Section 3.3 we have shown that graph conditions and first-order logic on graphs are expressively equivalent. The proof was based onto two steps: First, there are high-level transformations between \mathcal{M} -satisfiable conditions and \mathcal{A} -satisfiable conditions. Second, there are graph-specific transformations between \mathcal{A} -satisfiable graph conditions and graph formulas, relating the semantics of formulas and conditions: on the one hand assignments of variables to a structure representing the tested graph, on the other hand arbitrary morphisms from the graphs of the condition to the tested graph.



An obvious approach to solve the tautology and satisfiability problem of graph conditions would be a transformation of the input condition into a first-order graph formula and the application of existing provers and solvers. In fact we will use this strategy in Chapter 7 to compare **PROCON** and **SeekSat** with existing tools for first-order logic. However, such a translation does not represent a general solution to the problem, as the second part of the transformation is structure-specific and would have to be customized for every considered category. In contrast, **PROCON** and **SeekSat** make a translation of the problem unnecessary by providing a theorem prover and satisfiability solver for any category satisfying the assumptions of Chapter 2.

Many first-order satisfiability solvers, including the most successful ones [Pfe07] such as **DARWIN** [BFT06] and **PARADOX** [CS03] are based on *finite model building*, like their predecessors **MACE2**, **MACE4** [McC03], **FALCON** [Zha96] and **SEM** [ZZ95]. Most of these tools (except **DARWIN**) approach the problem by translating it, for a given domain size, into a decidable satisfiability problem of either propositional logic or at least ground clauses with equality. This has the advantage of using existing implementations of well-known (propositional) satisfiability algorithms, such as the dominating Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] and its derivatives, thus benefiting from years of experience and know-how. However, the translation phase is usually associated with a significant blow-up: Generating all ground instances over a domain of size n for a clause with v variables will yield n^v instances alone [McC03]. Also, the problem has to be solved again and again for increasing domain sizes, while only few tools are capable of reusing earlier results.

In contrast, **SeekSat** contains no such translation. Nevertheless, **SeekSat** seems, to some degree, related to the family of *enumeration* algorithms that are based on tree search and splitting, like the DPLL algorithm. **SeekSat** is based on a tree search where internal nodes correspond to partial solutions (morphisms), branches are choices (partitioning the search space), and leaf nodes are complete results or dead ends. Instead of *splitting*, that is, the process of branching by selecting a propositional variable x from a formula and assigning *true* and *false*, respectively, **SeekSat** will either skip, modify the morphism by adding elements to its codomain (positive statement) or back-track (negative statement), depending on the satisfaction of the considered subcondition by the current morphism.

In [BT03, BFT06], the Model Evolution (ME) Calculus is presented, which lifts the propositional DPLL procedure to first-order logic. Similar to **SeekSat**, the split rule of the ME calculus is restricted to positive literals (the model evolves only in case of positive statements). Like **SeekSat**, the ME calculus is shown to be sound and complete. The authors claim that the

ME calculus can decide the Bernays-Schönfinkel ($\exists\forall$) fragment of first-order logic. The tool DARWIN is an implementation of the ME calculus [BFT06] and was among the best satisfiability solvers at the CADE 2007 [Pfe07].

The earliest attempts to find deduction rules for proving graph conditions are made by Koch et. al. [KMP05], but remain incomplete. They investigate the notion of conflicting conditions of the form $\forall(I \hookrightarrow P, \exists(P \rightarrow C))$ and state prerequisites under which a conjunction of two graph conditions of this form is unsatisfiable.

Independently of our work, Orejas et. al. [OEP08] investigate sound and complete calculi for three fragments of graph conditions: the fragment of Boolean conditions over basic existential conditions $\exists(I \hookrightarrow C)$, the fragment of Boolean conditions over basic existential conditions $\exists(I \hookrightarrow C)$ and non-negated “atomic” conditions of the form $\forall(I \hookrightarrow P, \exists(P \rightarrow C))$, and the fragment of Boolean conditions over “atomic” conditions of the form $\forall(I \hookrightarrow P, \exists(P \rightarrow C))$. Their deduction rules relate to our own as follows: (R1) is a special case of (Resolve), (R2) is comparable to the rule (Supporting Lift), and (R3) is comparable to the rule (Partial lift), although (R2), (R3) do not lift (parts of) the resolvent (as this is neither necessary nor possible for the considered fragments of conditions). The operator \oplus used by Orejas et al. is a special instance of \mathbf{A} restricted to basic existential conditions.

In [Ore08], a sound and complete calculus for the fragment of “basic” and “positive atomic” attributed graph constraints is presented. Attributed graph constraints are conditions over attributed graphs combined with a formula expressing conditions on the attributes such as “ $(x > y)$ ”.

6.5 Summary and discussion

In Section 6.1 we have investigated the connections between the implication, the tautology and the satisfiability problem of conditions. These are the problems to decide for any condition whether or not a condition implies another condition, whether or not a condition is satisfied by every object, and whether or not a condition is satisfied by some object, respectively. It was shown that any instance of these problems can be translated into instances of the remaining two problems. These problems are undecidable for some categories, in particular for the category **Graphs**, but the satisfiability problem is semi-decidable for recursively enumerable categories such as **Graphs**.

In Section 6.2 we presented a satisfiability algorithm for conditions of weak adhesive high-level replacement categories. Instead of enumerating all possible objects of a category to approach the problem, the algorithm uses the input condition in a constructive way. Starting from the initial object,

for instance, the empty graph, elements of positive statements are added if necessary, while the absence of forbidden patterns is asserted. The result is a monotone (non-deleting) algorithm which non-deterministically progresses towards satisfying objects by searching through a tree shaped space, where internal nodes correspond to partially satisfying solutions and leafs are either satisfying results or dead ends.

We showed that for every condition c in \mathcal{MNF} , there exists a program with interface $\text{SeekSat}(c)$ that, with respect to the satisfiability problem, is correct and complete. Due to the semi-decidability of the satisfiability problem, the program $\text{SeekSat}(c)$ terminates, if c is satisfiable, but may or may not terminate, if c is unsatisfiable. A fragment of conditions was identified, namely the *NonNested* fragment of conditions, for which the termination of SeekSat was proven. Consequently, the algorithm decides the satisfiability problem as well as the complementary tautology problem for this subclass of conditions. We also showed that there exist provable tautologies for which SeekSat does not terminate, hence SeekSat cannot substitute a dedicated theorem prover.

In Section 6.3 we presented a calculus for proving conditions over weak adhesive high-level replacement categories. We took resolution [Rob65], the most successful approach to first-order theorem proving, as an ideal and presented deduction rules able to refute conditions in conjunctive normal form. We proved that every rule application corresponds to a logical deduction, and investigated whether or not omission of any rule leads to an incomplete calculus. We discussed practical aspects concerning our implementation *PROCON* such as filtering out structurally equivalent conditions. We also showed that *PROCON*'s calculus is able to prove tautologies for which SeekSat does not terminate. This is due to *PROCON*'s ability of considering infinite sets of morphisms, which enables it to prove the validity of statements involving universal quantifications: While every state of *PROCON* is a condition in CNF representing usually an infinite set of morphisms, every state of SeekSat is a state in the execution of a program with interface, representing at most a finite set of morphisms.

We propose to let run *PROCON* and SeekSat in parallel. For a given implication problem, *PROCON* will search for a proof, while SeekSat will search for a counterexample. This dual approach is motivated by the results of the system competition of the conference on automated deduction CADE [Pfe07]: well-respected theorem provers such as *VAMPIRE* [RV02] dominate in problem classes with provable statements, while dedicated satisfiability solver such as *PARADOX* [CS03] lead in problem classes with refutable statements.

7. Case studies

In this chapter we conduct case studies to evaluate our approach to the verification of graph transformation systems and programs. We model and verify selected aspects of three real-world systems: a railroad control [Pen04, HP05], a platoon maneuver protocol [Bau06, HESV91], and an access control for computer systems [HPR06]. For a start, we describe our general set-up and the content of our tests.

7.1 General set-up

In the following we describe how the case studies are conducted using our implementation *ENFORCE*. *ENFORCE* is a framework for verifying graphical program specifications. It is written in JAVA 6 and provides suitable data structures, for instance, graphs, categories, morphisms, conditions, rules, programs, as well as operations on these structures, such as, enumeration of all epimorphisms for a given domain, a construction of weakest liberal preconditions as presented in Section 5.3, the theorem prover *PROCON*, the satisfiability solver *SeekSat*, and so forth. The basic design of *ENFORCE* is presented in [AHPZ07, Zuc06]. Some graph-specific operations were adapted from [Bus04, Möl03] such as the enumeration of all morphisms between two graphs, also known as matching. The source code of *ENFORCE* is available in the repository (login required) `svn://homer.informatik.uni-oldenburg.de/svn/graphtrans/trunk`.

Each case study is represented by a JUNIT test, as shown in Table 7.1. The free and platform independent development environment *ECLIPSE* [Ecl] is recommended to view and execute the tests, as indicated in Figure 7.1.

The system states of our example systems are represented by directed labeled graphs. Consequently, we use graph conditions to model state properties and use graph programs or graph transformation rules to model state transitions. For each case study, a number of conditions and programs (or transformation rules) are considered. The necessary nodes, edges, graphs and graph morphisms are manually defined using the API of the classes *Node*,

case study	JUNIT test
railroad control	tests/enforce/dlengine/dlgraph/RailroadTest.java
car platoons	tests/enforce/dlengine/dlgraph/CarPlatooningTest.java
access control	tests/enforce/dlengine/dlgraph/AccessControlTest.java

Table 7.1: Correspondence of case studies and JUNIT tests

Edge, Graph, and Morphism of the package `enforce.dlengine.dlgraphs` and the operations of `enforce.dlengine.operations.DlgOperations` such as `initial(Graph)` and `inclusion(Graph, Graph)`.

The main goal of each case study is the automatic verification or refutation of a number of selected specifications. The majority of our specifications has the form $\{c\}P\{c\}$, where c is a condition and P is a program or transformation rule. We consider “elementary” specifications concerning a single program and single property as well as “composite” specifications that consists of a choice of programs and rules and/or a conjunction of conditions.

According to our approach, all specifications are tested by first computing a weakest precondition and then deciding the implication problem. If P is a program without iteration or a set of transformation rules, a weakest precondition can be effectively computed for specifications of the form $\{c\}P^*\{c\}$, using the fact that $\{c\}P^*\{c\}$ is correct iff $\{c\}P\{c\}$ is correct. In all other cases, an invariant is approximated using the algorithm of Section 5.4. However, if the approximation fails or yields a precondition that is not weak enough, even a correct specification may not be automatically verified.

Concerning the implication problem, we will compare the following pairs of theorem provers and satisfiability solvers, listed in Table 7.2. META refers to the hypothetical tool that arises by taking the minimum response time of the off-the-shelf provers and solvers for each test specification.

no.	prover and solver	reference
1	PROCON and SeekSat	Chapter 6
2	VAMPIRE 9.0 and PARADOX 3.0	[RV02, CS03]
3	DARWIN 1.3	[BT03, BFT06]
4	PROVER9 and MACE4 (v2008-09A)	[McC, McC06, McC03]
5	META	min time of 2, 3 and 4

Table 7.2: Considered pairs of theorem provers and satisfiability solvers

The external tools are applied onto translations of graph conditions into first-order graph formulas, as described in Section 3.3. For this task, ENFORCE

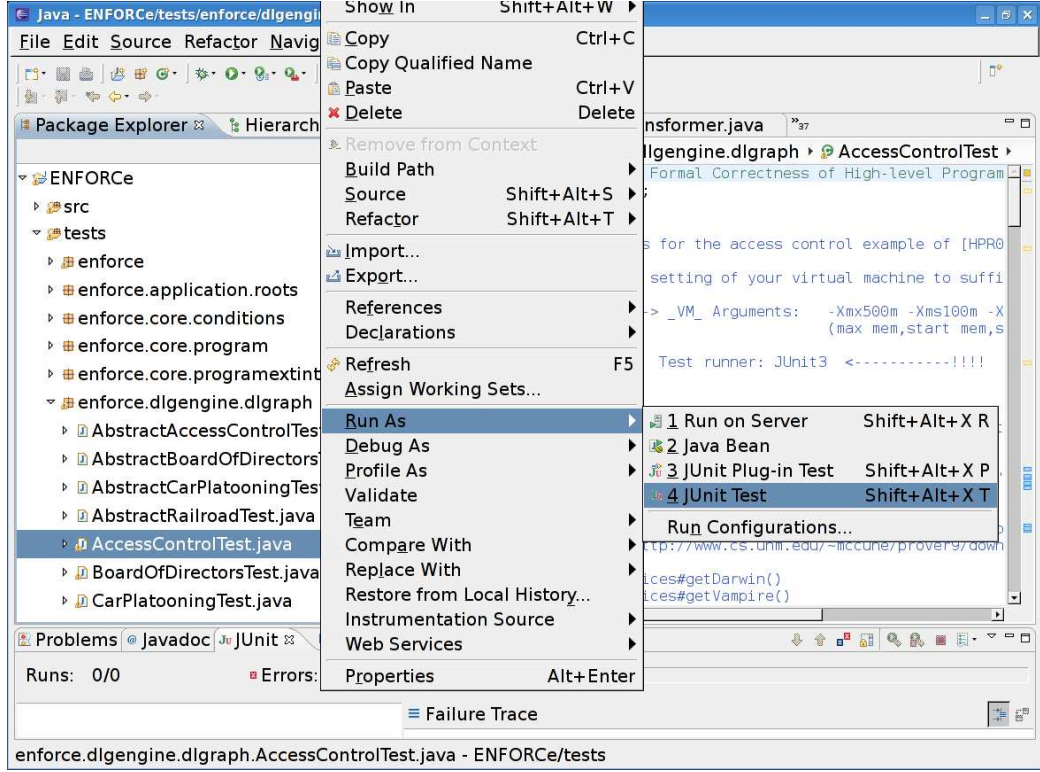


Figure 7.1: Execution of the access control JUNIT test in ECLIPSE

supports the file formats TPTP [SS98] and LADR [McC]. For each test case, we measure three values: the time it takes to construct a weakest precondition, denoted by t_{wlp} , the time to decide the implication problem, denoted by t_{\models} and the time to decide the specification, denoted by t_{Σ} , which is roughly the sum of the previous values. To enable a fair comparison between our deciders and the existing ones for first-order logic, the time to translate a graph condition into a first-order graph formula is excluded from t_{\models} . In the same spirit, the time it takes to transform a graph condition into conjunctive normal form is included in t_{\models} , in the case of PROCON.

7.2 Railroad control system

In the following we introduce state graphs of a simple control system for railroad networks. Our goal is to model the extension of a net of railroad tracks and the addition and movement of trains thereon. The basic items of our models are waypoints “○”, tracks “○—○”, switches “○ \rightarrow ○” and trains

“”.

Example 7.1 (railroad graphs). Let C be an alphabet that consists of suitable labels for waypoints, railroad tracks, switch edges and trains, for instance, $C = \langle \{\text{waypoint}\}, \{\text{track}, \text{switchhead}, \text{switchend}, \text{train}\} \rangle$. An example of a (concrete) graph and its graphical representation, called abstract graph, is given in Figure 7.2. Note that every proper undirected edge in the abstract graph represents a pair of directed edges in the concrete graph.

$G = \langle \{v_1, v_2, v_3\}, \{e_1, \dots, e_6\}, s_G, t_G, l_G, m_G \rangle$ with

i	1	2	3	4	5	6
$l_G(v_i)$	waypoint					
$s_G(e_i)$	v_1	v_2	v_1	v_1	v_2	v_3
$t_G(e_i)$	v_2	v_1	v_2	v_1	v_3	v_2
$m_G(e_i)$	track		train	switchhead		switchend

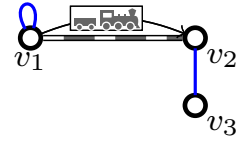


Figure 7.2: Concrete graph and its graphical representation

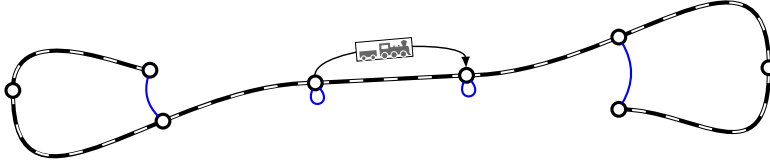


Figure 7.3: Railroad network with a train and two switches

With the intent to reason about the security of the railroad control, we exemplarily formalize some constraints on the railroad graphs. For instance, the control system should ensure that trains do not derail (every train is on a track), that trains do not crash (two trains do not occupy the same piece of track) and that trains keep a safety distance of one track (two trains do not occupy neighboring pieces of track, except if they drive apart).

Example 7.2 (railroad conditions). The following conditions are over the empty graph:

(*onTrack*) Every train occupies a piece of track:

$$\forall \left(\text{train icon on track segment } 1 \rightarrow 2, \exists \text{ train icon on track segment } 1 \rightarrow 2 \right)$$

(*noTwo*) Different trains occupy different waypoints (pieces of track):

$$\neg \exists \text{ train icon on track segment } 1 \rightarrow 2 \wedge \neg \exists \text{ train icon on track segment } 2 \rightarrow 1$$

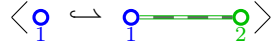
(*direction*) Two adjacent trains drive apart:



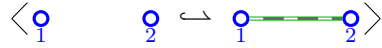
Let $decidable = noTwo \wedge direction$ and $consistent = onTrack \wedge decidable$.

Example 7.3 (railroad control system). The following graph transformation rules constitute the dynamic part of our railroad control system and model the extension of a net of railroad tracks and the addition and movement of trains thereon. The notation $\langle \langle L \hookrightarrow R \rangle, ac_L \rangle$ corresponds to a transformation rule $\langle \langle L \hookleftarrow K \hookrightarrow R \rangle, ac_L \rangle$, where $K \subseteq (L \cap R)$ is the graph that consists of all elements common to L and R .

(Build1) Extension of the net:



(Build2) Connection of two waypoints:



(Build3) Build of a switch:



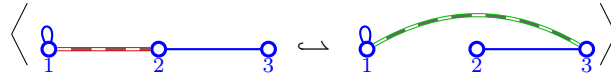
(Add) Addition of a train to the railroad net:



(Move) Movement of a train along the railroad net:

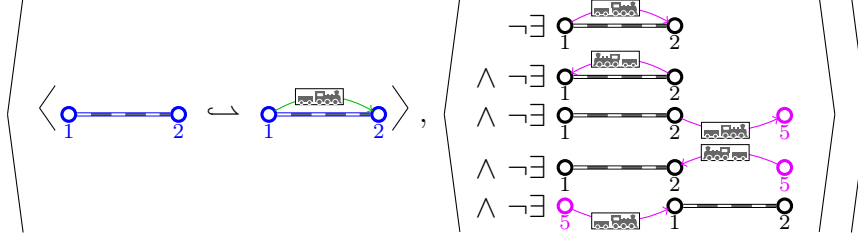


(Switch) Switch of points:

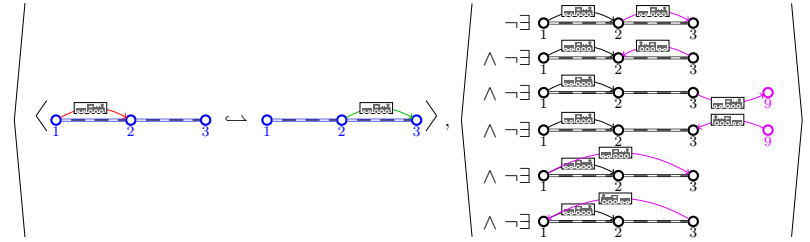


While already functional, not every transformation rule is safe with respect to the railroad graph conditions of Example 7.2. For instance, trains may crash if a train moves onto a piece of track that is already occupied by another train, or trains may derail, if a switch is toggled while a train is on it. Thus we need to formalize which behavior we exclude. One possible way is to describe the set of forbidden system states and to ensure that these states or graphs are never reached by a transition. In [Pen04], we have claimed that the following transformation rules restricted by application conditions will preserve the satisfaction of the presented railroad conditions.

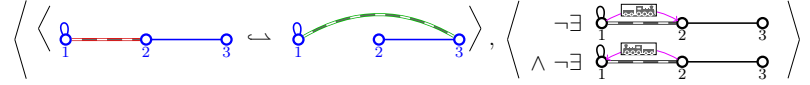
(AddAC) (Restricted) addition of a train:



(MoveAC) (Restricted) movement of a train:



(SwitchAC) (Restricted) switch of points:



Let $\text{Railroad} = \{\text{Build1}, \text{Build2}, \text{Build3}, \text{AddAC}, \text{MoveAC}, \text{SwitchAC}\}$ be the non-deterministic choice of the aforementioned graph transformation rules.

We now want to verify whether or not the railroad control system will preserve the railroad graph conditions of Example 7.2. We will use **ENFORCE** to automatically verify or refute test specifications of the form $\{c\}R\{c\}$, where c is a railroad condition and R is either a railroad transformation rule or the whole railroad control.

Example 7.4 (railroad verification). Table 7.3 on page 122 shows a log of the railroad case study. The second column states the specification of each test case. The column “ $|wlp|$ ” states the complexity of the constructed weakest preconditions, that is, the number of logical symbols. The column “result” compares the expected result with the computed result in the sense of an automated unit test. Here, **True** stands for tautology, **False** stands for contradiction, and an **Exception** would indicate no decision. The following columns state the number of seconds it took to decide each specification “ t_Σ ”, which is roughly the sum of the time it took to construct the weakest precondition “ t_{wlp} ” and the time to decide the implication problem “ t_\models ”. The

last column “decider” states which algorithm contributed the decision of the implication problem.

Aside the railroad conditions defined in Example 7.2, we consider two similar conditions, which turn out to be not invariant with respect to **Railroad**.

(*altNoTwo*) Different trains occupy different pieces of track (alternative):



(*altDirection*) Two adjacent trains face in opposite directions (alternative):



The results can be interpreted as follows: Most importantly, all test cases can be automatically decided by ENFORCE. Concerning performance, all but 5 test cases can be decided in less than 2 seconds, the majority in less than 0.2 seconds. Concerning correctness, we have proven that the railroad system preserves “consistency”, that is, every transition from a state satisfying the condition *consistent* will end up in a *consistent* state. As predicted, the unrestricted transformation rules **Add**, **Move** and **Switch** are not correct with respect to the condition *consistent*. Interestingly, one may not substitute the conditions *noTwo* and *direction* with their alternative versions, for instance, compare the cases #8 versus #9. Moreover, note that the condition *direction* itself is not invariant with respect to the rule **MoveAC** (#31), still **MoveAC** is consistent (#35). Concerning the implication problem, Figure 7.4 shows a comparison between the decision times t_{\models} of PROCON||SeekSat (see Table 7.3) and existing first-order theorem provers and satisfiability solvers such as VAMPIRE, PARADOX, DARWIN, PROVER9 and MACE4 (see Table 7.4). Even all first-order tools combined (META) can only decide 37 of 70 test specifications given 5 minutes of time per specification (INTEL T5600, 1.83GHz). Notably, VAMPIRE and DARWIN are able to solve different test cases, which one may take as an indication that each of these tools has its own difficulties with the test instances.

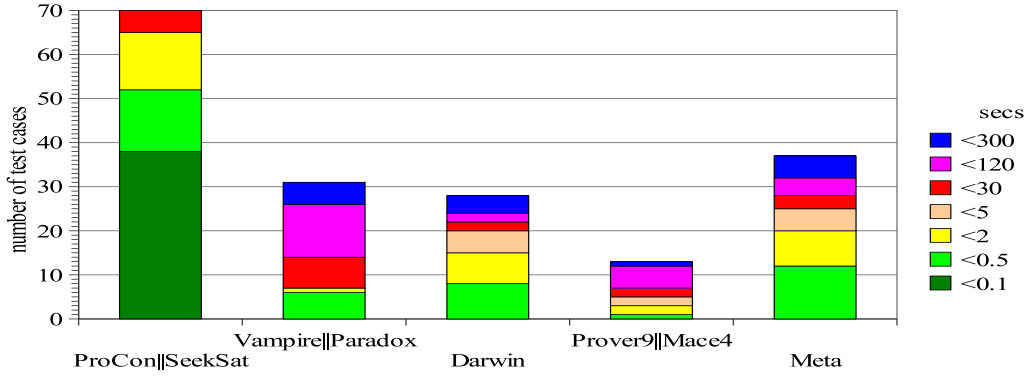
An interesting characteristic of the railroad control system is that its model involves many identically labeled elements. For instance, there is only one node label “waypoint” and most edge labels can be expected to be either tracks or trains. Remember also, that any undirected edge in the graphical representation really corresponds to a pair of directed edges. This leads to a relatively high number of elements that may or may not be identical – for once during the construction of weakest preconditions, resulting in rather complex conditions, but also during the translation of graph conditions into first-order graph formulas, resulting in a rather high number of inequations that are necessarily generated.

#	specification	$ wlp $	result	t_{Σ}	t_{wlp}	t_{\models}	decider
1	$\{noTwo\}$ Build1 $\{noTwo\}$	18	T=T	0.0	0.0	0.0	SeekSat
2	$\{altNoTwo\}$ Build1 $\{altNoTwo\}$	18	T=T	0.1	0.0	0.1	SeekSat
3	$\{direction\}$ Build1 $\{direction\}$	18	T=T	0.1	0.1	0.0	SeekSat
4	$\{altDirection\}$ Build1 $\{altDirection\}$	18	T=T	0.1	0.1	0.0	SeekSat
5	$\{onTrack\}$ Build1 $\{onTrack\}$	20	T=T	0.1	0.0	0.1	ProCon
6	$\{decidable\}$ Build1 $\{decidable\}$	34	T=T	0.1	0.1	0.0	SeekSat
7	$\{consistent\}$ Build1 $\{consistent\}$	51	T=T	0.7	0.1	0.6	ProCon
8	$\{noTwo\}$ Build2 $\{noTwo\}$	18	T=T	0.0	0.0	0.0	SeekSat
9	$\{altNoTwo\}$ Build2 $\{altNoTwo\}$	18	F=F	0.0	0.0	0.0	SeekSat
10	$\{direction\}$ Build2 $\{direction\}$	18	T=T	0.0	0.0	0.0	SeekSat
11	$\{altDirection\}$ Build2 $\{altDirection\}$	22	F=F	0.0	0.0	0.0	SeekSat
12	$\{onTrack\}$ Build2 $\{onTrack\}$	20	T=T	0.1	0.0	0.1	ProCon
13	$\{decidable\}$ Build2 $\{decidable\}$	34	T=T	0.1	0.1	0.0	SeekSat
14	$\{consistent\}$ Build2 $\{consistent\}$	51	T=T	0.9	0.0	0.9	ProCon
15	$\{noTwo\}$ Build3 $\{noTwo\}$	18	T=T	0.0	0.0	0.0	SeekSat
16	$\{altNoTwo\}$ Build3 $\{altNoTwo\}$	18	T=T	0.0	0.0	0.0	SeekSat
17	$\{direction\}$ Build3 $\{direction\}$	18	T=T	0.0	0.0	0.0	SeekSat
18	$\{altDirection\}$ Build3 $\{altDirection\}$	18	T=T	0.0	0.0	0.0	SeekSat
19	$\{onTrack\}$ Build3 $\{onTrack\}$	20	T=T	0.0	0.0	0.0	ProCon
20	$\{decidable\}$ Build3 $\{decidable\}$	34	T=T	0.1	0.1	0.0	SeekSat
21	$\{consistent\}$ Build3 $\{consistent\}$	51	T=T	0.3	0.1	0.2	ProCon
22	$\{noTwo\}$ AddAC $\{noTwo\}$	54	T=T	0.0	0.0	0.0	SeekSat
23	$\{altNoTwo\}$ AddAC $\{altNoTwo\}$	54	T=T	0.0	0.0	0.0	SeekSat
24	$\{direction\}$ AddAC $\{direction\}$	50	T=T	0.1	0.0	0.1	SeekSat
25	$\{altDirection\}$ AddAC $\{altDirection\}$	50	T=T	0.0	0.0	0.0	SeekSat
26	$\{onTrack\}$ AddAC $\{onTrack\}$	55	T=T	0.1	0.0	0.1	ProCon
27	$\{decidable\}$ AddAC $\{decidable\}$	66	T=T	0.2	0.1	0.1	SeekSat
28	$\{consistent\}$ AddAC $\{consistent\}$	79	T=T	0.1	0.1	0.9	ProCon
29	$\{noTwo\}$ MoveAC $\{noTwo\}$	76	T=T	0.3	0.2	0.0	SeekSat
30	$\{altNoTwo\}$ MoveAC $\{altNoTwo\}$	76	T=T	0.2	0.1	0.1	SeekSat
31	$\{direction\}$ MoveAC $\{direction\}$	80	F=F	0.5	0.0	0.5	SeekSat
32	$\{altDirection\}$ MoveAC $\{altDirection\}$	80	F=F	0.3	0.1	0.2	SeekSat
33	$\{onTrack\}$ MoveAC $\{onTrack\}$	79	T=T	0.4	0.0	0.4	ProCon
34	$\{decidable\}$ MoveAC $\{decidable\}$	102	T=T	0.2	0.0	0.2	SeekSat
35	$\{consistent\}$ MoveAC $\{consistent\}$	125	T=T	0.1	0.1	0.9	ProCon
36	$\{noTwo\}$ SwitchAC $\{noTwo\}$	62	T=T	0.0	0.0	0.0	SeekSat
37	$\{altNoTwo\}$ SwitchAC $\{altNoTwo\}$	62	F=F	0.1	0.0	0.1	SeekSat
38	$\{direction\}$ SwitchAC $\{direction\}$	128	T=T	0.3	0.1	0.2	SeekSat
39	$\{altDirection\}$ SwitchAC $\{altDirection\}$	128	F=F	1.1	0.4	0.7	SeekSat
40	$\{onTrack\}$ SwitchAC $\{onTrack\}$	73	T=T	1.0	0.0	1.0	ProCon
41	$\{decidable\}$ SwitchAC $\{decidable\}$	176	T=T	1.4	0.7	0.6	SeekSat
42	$\{consistent\}$ SwitchAC $\{consistent\}$	234	T=T	29.8	2.1	27.6	ProCon
43	$\{noTwo\}$ Add $\{noTwo\}$	34	F=F	0.0	0.0	0.0	SeekSat
44	$\{altNoTwo\}$ Add $\{altNoTwo\}$	13	F=F	0.0	0.0	0.0	SeekSat
45	$\{direction\}$ Add $\{direction\}$	26	F=F	0.0	0.0	0.0	SeekSat
46	$\{altDirection\}$ Add $\{altDirection\}$	18	F=F	0.0	0.0	0.0	SeekSat
47	$\{onTrack\}$ Add $\{onTrack\}$	27	T=T	0.1	0.0	0.1	ProCon
48	$\{decidable\}$ Add $\{decidable\}$	46	F=F	0.0	0.0	0.0	SeekSat
49	$\{consistent\}$ Add $\{consistent\}$	56	F=F	0.1	0.0	0.1	SeekSat
50	$\{noTwo\}$ Move $\{noTwo\}$	86	F=F	0.3	0.0	0.2	ProCon
51	$\{altNoTwo\}$ Move $\{altNoTwo\}$	58	F=F	0.2	0.0	0.2	SeekSat
52	$\{direction\}$ Move $\{direction\}$	94	F=F	0.2	0.0	0.2	SeekSat
53	$\{altDirection\}$ Move $\{altDirection\}$	58	F=F	1.4	0.1	1.3	SeekSat
54	$\{onTrack\}$ Move $\{onTrack\}$	69	T=T	0.4	0.0	0.4	ProCon
55	$\{decidable\}$ Move $\{decidable\}$	130	F=F	0.4	0.0	0.4	SeekSat
56	$\{consistent\}$ Move $\{consistent\}$	161	F=F	10.2	0.1	10.1	ProCon
57	$\{noTwo\}$ Switch $\{noTwo\}$	90	T=T	0.1	0.0	0.1	SeekSat
58	$\{altNoTwo\}$ Switch $\{altNoTwo\}$	90	F=F	0.4	0.0	0.4	ProCon
59	$\{direction\}$ Switch $\{direction\}$	222	T=T	0.5	0.1	0.4	SeekSat
60	$\{altDirection\}$ Switch $\{altDirection\}$	222	F=F	6.9	0.2	6.7	SeekSat
61	$\{onTrack\}$ Switch $\{onTrack\}$	83	F=F	0.4	0.0	0.4	ProCon
62	$\{decidable\}$ Switch $\{decidable\}$	310	T=T	1.2	0.3	0.9	SeekSat
63	$\{consistent\}$ Switch $\{consistent\}$	390	F=F	7.7	0.4	7.3	ProCon
64	$\{noTwo\}$ Railroad $\{noTwo\}$	183	T=T	0.1	0.0	0.1	SeekSat
65	$\{altNoTwo\}$ Railroad $\{altNoTwo\}$	183	F=F	0.2	0.1	0.1	SeekSat
66	$\{direction\}$ Railroad $\{direction\}$	249	F=F	0.4	0.2	0.2	SeekSat
67	$\{altDirection\}$ Railroad $\{altDirection\}$	253	F=F	1.2	0.5	0.6	SeekSat
68	$\{onTrack\}$ Railroad $\{onTrack\}$	219	T=T	0.9	0.0	0.9	ProCon
69	$\{decidable\}$ Railroad $\{decidable\}$	327	T=T	1.1	0.5	0.6	SeekSat
70	$\{consistent\}$ Railroad $\{consistent\}$	436	T=T	28.2	0.6	27.5	ProCon
μ	arithmetic mean	93.7		1.5	0.1	1.4	
$\mu_{1/2}$	median	60		0.2	0.0	0.1	

Table 7.3: Railroad control case study: results

#	result	t_{\models}	decider	result	t_{\models}	decider	result	t_{\models}	decider
1	T=T	22.3	Vampire	T=T	0.2	DarwinProver	T=T	2.5	Prover9
2	T=T	42.8	Vampire	T=T	7.2	DarwinProver	T=T	134.0	Prover9
3	T=T	21.3	Vampire	T=T	1.2	DarwinProver	T=T	30.5	Prover9
4	T=T	124.8	Vampire	T≠E	-	-	T≠E	-	-
5	T=T	21.3	Vampire	T=T	0.4	DarwinProver	T=T	34.6	Prover9
6	T=T	85.8	Vampire	T=T	1.2	DarwinProver	T≠E	-	-
7	T=T	168.9	Vampire	T=T	2.0	DarwinProver	T≠E	-	-
8	T=T	21.3	Vampire	T=T	0.2	DarwinProver	T=T	1.2	Prover9
9	F=F	0.2	Paradox	F=F	0.2	DarwinSolver	F=F	0.4	Mace4
10	T=T	41.7	Vampire	T=T	1.2	DarwinProver	T=T	30.5	Prover9
11	F=F	0.4	Paradox	F≠E	-	-	F≠E	-	-
12	T=T	42.8	Vampire	T=T	0.4	DarwinProver	T=T	16.1	Prover9
13	T=T	86.9	Vampire	T=T	1.2	DarwinProver	T≠E	-	-
14	T=T	86.9	Vampire	T=T	2.0	DarwinProver	T≠E	-	-
15	T=T	41.8	Vampire	T=T	0.2	DarwinProver	T=T	2.5	Prover9
16	T=T	42.8	Vampire	T=T	7.2	DarwinProver	T=T	71.5	Prover9
17	T=T	22.3	Vampire	T=T	1.2	DarwinProver	T=T	15.1	Prover9
18	T=T	65.3	Vampire	T≠E	-	-	T≠E	-	-
19	T=T	21.2	Vampire	T=T	0.4	DarwinProver	T=T	34.6	Prover9
20	T=T	168.0	Vampire	T=T	1.2	DarwinProver	T≠E	-	-
21	T=T	168.9	Vampire	T=T	2.0	DarwinProver	T≠E	-	-
22	T=T	97.1	Vampire	T=T	4.2	DarwinProver	T≠E	-	-
23	T≠E	-	-	T≠E	-	-	T≠E	-	-
24	T≠E	-	-	T≠E	-	-	T≠E	-	-
25	T≠E	-	-	T≠E	-	-	T≠E	-	-
26	T=T	67.4	Vampire	T≠E	-	-	T≠E	-	-
27	T≠E	-	-	T≠E	-	-	T≠E	-	-
28	T≠E	-	-	T≠E	-	-	T≠E	-	-
29	T≠E	-	-	T=T	79.7	DarwinProver	T≠E	-	-
30	T≠E	-	-	T≠E	-	-	T≠E	-	-
31	F≠E	-	-	F≠E	-	-	F≠E	-	-
32	F≠E	-	-	F≠E	-	-	F≠E	-	-
33	T≠E	-	-	T≠E	-	-	T≠E	-	-
34	T≠E	-	-	T≠E	-	-	T≠E	-	-
35	T≠E	-	-	T≠E	-	-	T≠E	-	-
36	T≠E	-	-	T=T	148.4	DarwinProver	T≠E	-	-
37	F≠E	-	-	F≠E	-	-	F≠E	-	-
38	T≠E	-	-	T≠E	-	-	T≠E	-	-
39	F≠E	-	-	F≠E	-	-	F≠E	-	-
40	T=T	181.4	Vampire	T≠E	-	-	T≠E	-	-
41	T≠E	-	-	T≠E	-	-	T≠E	-	-
42	T≠E	-	-	T≠E	-	-	T≠E	-	-
43	F=F	0.4	Paradox	F≠E	-	-	F≠E	-	-
44	F=F	0.3	Paradox	F=F	0.2	DarwinSolver	F=F	1.6	Mace4
45	F=F	0.4	Paradox	F≠E	-	-	F≠E	-	-
46	F=F	6.4	Paradox	F≠E	-	-	F≠E	-	-
47	T=T	37.7	Vampire	T=T	1.6	DarwinProver	T≠E	-	-
48	F=F	0.6	Paradox	F=F	202.7	DarwinSolver	F≠E	-	-
49	F=F	0.4	Paradox	F≠E	-	-	F≠E	-	-
50	F≠E	-	-	F≠E	-	-	F≠E	-	-
51	F≠E	-	-	F≠E	-	-	F≠E	-	-
52	F≠E	-	-	F≠E	-	-	F≠E	-	-
53	F≠E	-	-	F≠E	-	-	F≠E	-	-
54	T≠E	-	-	T=T	215.0	DarwinProver	T≠E	-	-
55	F≠E	-	-	F≠E	-	-	F≠E	-	-
56	F≠E	-	-	F≠E	-	-	F≠E	-	-
57	T≠E	-	-	T=T	51.0	DarwinProver	T≠E	-	-
58	F≠E	-	-	F≠E	-	-	F≠E	-	-
59	T≠E	-	-	T≠E	-	-	T≠E	-	-
60	F≠E	-	-	F≠E	-	-	F≠E	-	-
61	F≠E	-	-	F≠E	-	-	F≠E	-	-
62	T≠E	-	-	T≠E	-	-	T≠E	-	-
63	F≠E	-	-	F≠E	-	-	F≠E	-	-
64	T≠E	-	-	T=T	251.9	DarwinProver	T≠E	-	-
65	F≠E	-	-	F=F	2.5	DarwinSolver	F≠E	-	-
66	F≠E	-	-	F≠E	-	-	F≠E	-	-
67	F≠E	-	-	F≠E	-	-	F≠E	-	-
68	T≠E	-	-	T≠E	-	-	T≠E	-	-
69	T≠E	-	-	T≠E	-	-	T≠E	-	-
70	T≠E	-	-	T≠E	-	-	T≠E	-	-
μ		54.5			35.2			28.9	
$\mu_{1/2}$		41.7			1.4			16.1	

Table 7.4: Railroad: decision times of first-order provers and solvers

Figure 7.4: Railroad control: comparison of deciders (“ t_{\models} ”)

7.3 Car platoon maneuver protocol

According to [Bau06], the following car platooning case study is “a prototypical instance of a dynamic communication system”. Originally taken from the California PATH project [HESV91], it represents a protocol for cars on a highway that can organize themselves into platoons, by driving close together, with the aim to conserve space and fuel. A typical platoon, such as

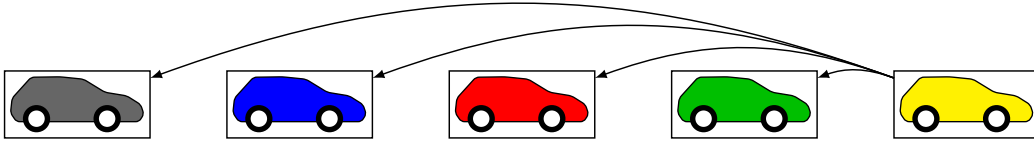


Figure 7.5: Car platoon

the one depicted in Figure 7.5, consists of a unique leader and a number of followers. Such a platoon can be modeled as a tree of height one, with the leader being the root and the followers constituting the leafs. Platoons may temporarily assume a different form during maneuvers such as the merge of two platoons or the splitting of a platoon.

Example 7.5 (car platooning graphs). Let $C = \langle \{fa, ldr, flw, fl, rl, spl, acc\}, \{_ \} \rangle$ be an alphabet. An overview of the labels and their meaning is given in Table 7.5. A car platooning state graph consists of zero or more platoons. As we will see later on, platoons are inductively defined: Every single car, called “free agent”, constitutes a platoon of size one. Two platoons may merge: the leader of the rear platoon, temporarily called “rear leader”,

hands over all followers to the leader of the platoon in front, temporarily called the “front leader”. If the “rear leader” has no followers left, it degrades into a follower of the “front leader” and the “front leader” becomes the new leader of the merged platoon. A number of car platoons are depicted in Figure 7.6. Any combination of them constitutes a car platooning state graph.

fa	free agent – a single car not associated with a platoon; may disappear
ldr	leader – the leader of a platoon, not engaged in a merge or split maneuver
flw	follower – a member of a platoon, not the leader
fl	front leader – the leader of a platoon that is about to merge with another platoon; will remain the leader
rl	rear leader – the leader of a platoon that is about to merge with another platoon; will become a follower
spl	splitting follower – the follower that becomes the leader after the split
acc	accelerating leader – the leader of a platoon that needs to split

Table 7.5: Car platoons case study: alphabet

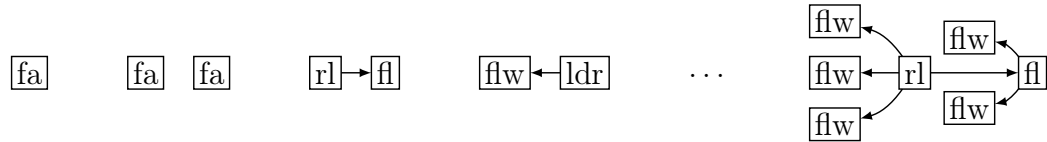


Figure 7.6: Car platoons state graphs

Example 7.6 (car platooning conditions). We now present a number of graph conditions over the empty graph that concern the structure of platoons, for instance, “Two leaders are never connected”. We cover 3 of 4 example properties considered in [Bau06]. The fourth property concerns a liveness aspect of an extension of the car platooning protocol with respect to asynchronous communication. To our best knowledge, it remains open whether the analysis techniques presented in [Bau06] are able to decide it. In the following a unlabeled node represents a labeled node with arbitrary label.

(*ldrNeverHasLdr*) A leader is never connected to another leader. More formally, for all nodes that have an incoming edge from a leader, it is not true that the label is leader, that is, they are not leaders themselves. This condition corresponds to property “(3.1)” of [Bau06, p. 45]:

$$\forall \left(\boxed{\text{ldr}}_1 \rightarrow \boxed{\phantom{\text{ldr}}}_2, \quad \neg \exists \boxed{\text{ldr}}_1 \rightarrow \boxed{\text{ldr}}_2 \right)$$

(*ldrHasNeighbor*) A leader must always be connected to someone else. More formally, for all leaders, there is a node connected with an outgoing edge or there is a node connected with an incoming edge. This condition corresponds to property “(3.2)” of [Bau06, p. 45]:

$$\forall \left(\boxed{\text{ldr}}_1, \quad \exists \boxed{\text{ldr}}_1 \rightarrow \boxed{\phantom{\text{ldr}}}_2 \vee \exists \boxed{\phantom{\text{ldr}}}_1 \leftarrow \boxed{\text{ldr}}_2 \right)$$

(*uniqueLdr*) Each car that is not a leader or a free agent has a unique leader. More formally, for every node, the label is leader or the label is free agent or the node is associated to one and not two leaders. This condition is comparable to property “(3.4)” of [Bau06, p. 56]:

$$\forall \left(\boxed{\phantom{\text{ldr}}}_1, \quad \exists \boxed{\text{ldr}}_1 \vee \exists \boxed{\text{fa}}_1 \vee \left(\exists \boxed{\phantom{\text{ldr}}}_1 \leftarrow \boxed{\text{ldr}}_2 \wedge \neg \exists \boxed{\phantom{\text{ldr}}}_1 \leftarrow \boxed{\text{ldr}}_2 \boxed{\text{ldr}}_3 \right) \right)$$

(*ldrHasFlw*) A leader must always be connected to a follower or splitting follower. More formally, for all leader nodes, there exists an outgoing edge to a follower node or there exists an outgoing edge to a splitting follower:

$$\forall \left(\boxed{\text{ldr}}_1, \quad \exists \boxed{\text{ldr}}_1 \rightarrow \boxed{\text{flw}}_2 \vee \exists \boxed{\text{ldr}}_1 \rightarrow \boxed{\text{spl}}_2 \right)$$

(*flwHasLdr*) Every follower has one and only one leader. More formally, for every follower node, there is an incoming edge from a leader node and there are not two incoming edges from two leader nodes:

$$\forall \left(\boxed{\text{flw}}_1, \quad \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{ldr}}_2 \wedge \neg \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{ldr}}_2 \boxed{\text{ldr}}_3 \right)$$

(*flwHasSomeLdr*) Every follower has some sort of leader:

$$\forall \left(\boxed{\text{flw}}_1, \quad \begin{array}{l} \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{ldr}}_2 \vee \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{fl}}_2 \\ \vee \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{rl}}_2 \vee \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{spl}}_2 \\ \vee \exists \boxed{\text{flw}}_1 \leftarrow \boxed{\text{acc}}_2 \end{array} \right)$$

(*freeAgents*) A free agent is isolated. More formally, there does not exist an outgoing edge to another node nor does there exist an incoming edge from another node:

$$\neg \exists \boxed{\text{fa}} \rightarrow \boxed{\phantom{\text{fa}}} \wedge \neg \exists \boxed{\phantom{\text{fa}}} \leftarrow \boxed{\text{fa}}$$

Example 7.7 (car platooning system). The following graph transformation rules, taken from [Bau06, p. 41ff], constitute the dynamic part of the car platooning protocol and model the appearance and disappearance of free agents, the merge and splitting of platoons. The “partner constraints” of the rules in [Bau06] have been translated into negative application conditions. Note that while we could, we do not use label schemata, but present each application condition explicitly. The notation $\langle\langle L \hookrightarrow R \rangle, ac_L\rangle$ corresponds to a transformation rule $\langle\langle L \hookleftarrow K \hookrightarrow R \rangle, ac_L\rangle$, where $K \subseteq (L \cap R)$ is the graph that consists of all elements common to L and R . Some of the rules apply relabeling, for instance **InitSplit1**. Strictly speaking, the graphs and the morphisms used in these rules do not fit to our definition of totally labeled graphs and label-preserving morphisms. However, ENFORCE also handles partially labeled graphs and label-preserving morphisms.

(Create) A free agent appears:

$$\langle \emptyset \hookrightarrow \boxed{\text{fa}} \rangle$$

(Destroy) A free agent disappears:

$$\langle \boxed{\text{fa}} \hookrightarrow \emptyset \rangle$$

(InitMerge1 – 4) Initiates a merge between two suitable actors, that is, any pair of leaders and free agents:

$$\begin{aligned} &\langle \boxed{\text{ldr}}_1 \quad \boxed{\text{ldr}}_2 \hookrightarrow \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rangle \\ &\langle \boxed{\text{ldr}}_1 \quad \boxed{\text{fa}}_2 \hookrightarrow \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rangle \\ &\langle \boxed{\text{fa}}_1 \quad \boxed{\text{ldr}}_2 \hookrightarrow \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rangle \\ &\langle \boxed{\text{fa}}_1 \quad \boxed{\text{fa}}_2 \hookrightarrow \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rangle \end{aligned}$$

(Pass1) Passes a follower of a rear leader to the front leader:

$$\langle \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\text{flw}}_3 \hookrightarrow \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\text{flw}}_3 \rangle$$

(Ldr2Flw) Converts a rear leader (with no followers) into a follower:

$$\left\langle \langle \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \hookrightarrow \boxed{\text{flw}}_1 \leftarrow \boxed{\text{ldr}}_2 \rangle, \left\langle \begin{array}{l} \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \quad \wedge \quad \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\phantom{\text{fl}}}_4 \\ \wedge \quad \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\phantom{\text{fl}}}_4 \quad \wedge \quad \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\phantom{\text{fl}}}_4 \\ \wedge \quad \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\phantom{\text{fl}}}_4 \quad \wedge \quad \neg \exists \boxed{\text{rl}}_1 \rightarrow \boxed{\text{fl}}_2 \rightarrow \boxed{\phantom{\text{fl}}}_4 \end{array} \right\rangle \right\rangle$$

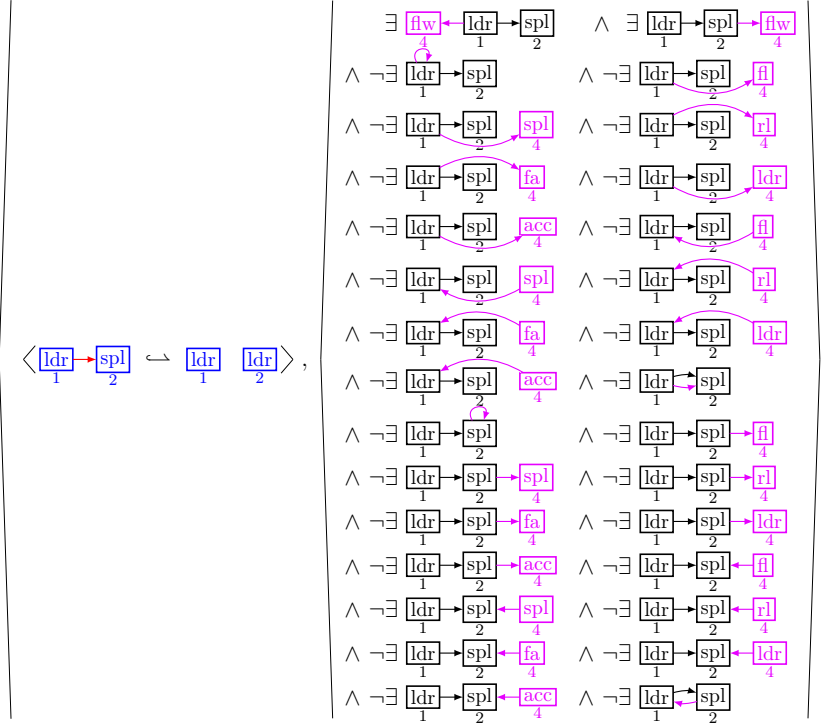
(InitSplit1) Initiates a split of a platoon by a follower:

$$\langle \boxed{\text{flw}}_1 \hookrightarrow \boxed{\text{spl}}_1 \rangle$$

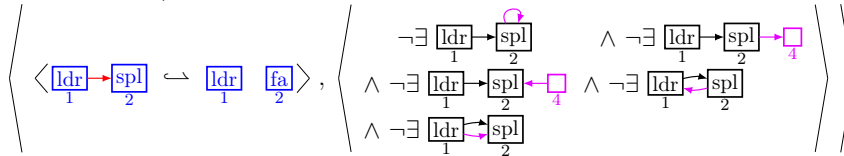
(Pass2) Passes a follower of a leader to the split initiating car, dividing the platoon into two:

$$\langle \boxed{\text{flw}}_1 \leftarrow \boxed{\text{ldr}}_2 \rightarrow \boxed{\text{spl}}_3 \hookrightarrow \boxed{\text{ldr}}_2 \rightarrow \boxed{\text{spl}}_3 \rightarrow \boxed{\text{flw}}_1 \rangle$$

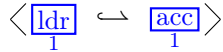
(SplitMiddle) Splits the platoon, converting the split initiating car (that has followers) into a leader:



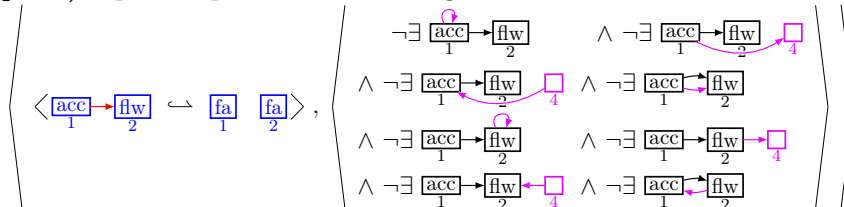
(SplitLast) Splits the platoon, converting the split initiating car (that has no followers) into a free agent:



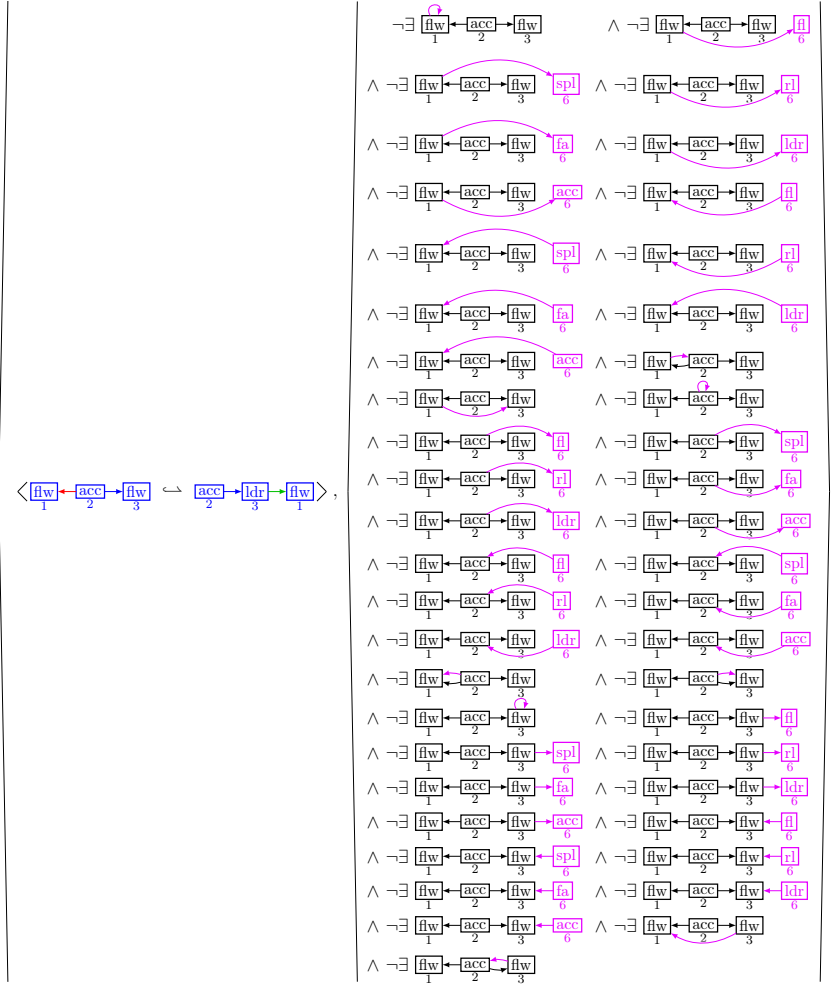
(InitSplit2) Initiates a split of a platoon by the leader:



(TwoSplit) Splits a platoon consisting of two cars:



(Pass3) Selects a new leader, handing over a follower:



(Pass4) Handing over another follower to the new platoon leader:

$$\langle \text{flw}_1 \leftarrow \text{acc}_2 \rightarrow \text{ldr}_3 \rangle \hookrightarrow \langle \text{acc}_2 \rightarrow \text{ldr}_3 \rightarrow \text{flw}_1 \rangle$$

(AccLeader) Splits the platoon, the leader becomes a free agent:

$$\left\langle \langle \text{acc}_1 \rightarrow \text{ldr}_2 \rangle \hookrightarrow \langle \text{fa}_1 \text{ ldr}_2 \rangle, \left\langle \begin{array}{l} \neg \exists \text{acc}_1 \rightarrow \text{ldr}_2 \quad \wedge \quad \neg \exists \text{acc}_1 \rightarrow \text{ldr}_2 \rightarrow \square_4 \\ \wedge \neg \exists \text{acc}_1 \rightarrow \text{ldr}_2 \rightarrow \square_4 \quad \wedge \quad \neg \exists \text{acc}_1 \rightarrow \text{ldr}_2 \\ \wedge \neg \exists \text{acc}_1 \rightarrow \text{ldr}_2 \end{array} \right\rangle \right\rangle$$

Let $\text{BasicCarPlatooning} = \{\text{Create}, \text{Destroy}, \text{InitMerge1}, \text{InitMerge2}, \text{InitMerge3}, \text{InitMerge4}, \text{Pass1}, \text{Ldr2Flw}\}$ be a non-deterministic choice of graph transformation rules, and let CarPlatooning be the non-deterministic choice consisting of every above mentioned graph transformation rule.

We now want to verify whether or not the car platooning rules are correct with respect to the car platooning conditions of Example 7.6. We will use

ENFORCE to automatically verify or refute test specifications of the form $\{c\}R\{c\}$ and $\{\neg\exists\bigcirc\}P^*\{c\}$, where c is a condition, R a transformation rule and P a set of transformation rules. The condition $\neg\exists\bigcirc$ represents the empty graph, the start graph of the car platooning protocol.

Example 7.8 (car platooning verification). Table 7.6 and Table 7.7 on page 131f show a log of the car platooning case study. The second column states the specification of each test case. The column “ $|wlp|$ ” states the complexity of the constructed weakest preconditions, that is, the number of logical symbols. The column “result” compares the expected result with the computed result in the sense of an automated unit test. Here, **True** stands for tautology, **False** stands for contradiction, and an **Exception** would indicate no decision. The following columns state the number of seconds it took to decide each specification “ t_Σ ”, which is roughly the sum of the time it took to construct the weakest precondition “ t_{wlp} ” and the time to decide the implication problem “ t_\models ”. The last column “decider” states which algorithm contributed the decision of the implication problem.

The results can be interpreted as follows: Most importantly, all test specifications can be automatically decided by ENFORCE. Concerning performance, all but 7 test cases can be decided in less than 2 seconds, the majority in less than 0.1 seconds. Concerning the correctness of **CarPlatooning** with respect to the start graph \emptyset , we have proven that a leader “ldr” is never connected to another leader node (*ldrNeverHasLdr*, see case #134), every follower is associated to some kind of leader, that is, “ldr”, “rl”, “fl”, “spl” or “acc” (*flwHasSomeLdr*, see case #139), and every free agent is isolated (*freeAgents*, see case #140). In accordance with [Bau06], we have shown that *ldrHasNeighbor* is satisfied in every reachable state from \emptyset by **BasicCarPlatooning***, but not by **CarPlatooning*** (compare #128 vs. #135). Furthermore, we have discovered that during a split maneuver, not every leader may have a follower (*ldrHasFlw*, see case #136).

Concerning the implication problem, Figure 7.7 shows a comparison between the decision times t_\models of PROCON||SeekSat (see Table 7.6 and 7.7) and selected first-order theorem provers and satisfiability solvers (see Table 7.8). In contrast to the railroad case study, this time the off-the-shelf tools perform almost competitive, both in terms of coverage and performance. All tools combined (META) are able to solve all but one specification (that is #121), given 5 minutes of time per specification (INTEL T5600, 1.83GHz). Again, VAMPIRE and DARWIN are able to solve different test cases.

The differences to the railroad control results may be explained by the fact that for the graphs used in the railroad conditions and rules, the number of elements is higher. At the same time, the railroad graphs feature fewer labels,

#	specification	wlp	result	t_{Σ}	t_{wlp}	t_{\models}	decider
1	{ldrNeverHasLdr} Create {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
2	{ldrHasNeighbor} Create {ldrHasNeighbor}	20	T=T	0.0	0.0	0.0	ProCon
3	{ldrHasFlw} Create {ldrHasFlw}	20	T=T	0.0	0.0	0.0	ProCon
4	{uniqueLdr} Create {uniqueLdr}	32	T=T	0.1	0.0	0.1	ProCon
5	{flwHasLdr} Create {flwHasLdr}	21	T=T	0.0	0.0	0.0	ProCon
6	{flwHasSomeLdr} Create {flwHasSomeLdr}	38	T=T	0.1	0.0	0.1	ProCon
7	{freeAgents} Create {freeAgents}	18	T=T	0.0	0.0	0.0	SeekSat
8	{ldrNeverHasLdr} Destroy {ldrNeverHasLdr}	25	T=T	0.0	0.0	0.0	SeekSat
9	{ldrHasNeighbor} Destroy {ldrHasNeighbor}	35	T=T	0.2	0.0	0.2	ProCon
10	{ldrHasFlw} Destroy {ldrHasFlw}	35	T=T	0.1	0.0	0.1	ProCon
11	{uniqueLdr} Destroy {uniqueLdr}	47	T=T	0.1	0.0	0.1	ProCon
12	{flwHasLdr} Destroy {flwHasLdr}	33	T=T	0.0	0.0	0.0	ProCon
13	{flwHasSomeLdr} Destroy {flwHasSomeLdr}	53	T=T	0.1	0.0	0.1	ProCon
14	{freeAgents} Destroy {freeAgents}	30	T=T	0.1	0.0	0.1	SeekSat
15	{ldrNeverHasLdr} InitMerge1 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
16	{ldrHasNeighbor} InitMerge1 {ldrHasNeighbor}	36	T=T	0.1	0.0	0.1	ProCon
17	{ldrHasFlw} InitMerge1 {ldrHasFlw}	20	T=T	0.1	0.0	0.1	ProCon
18	{uniqueLdr} InitMerge1 {uniqueLdr}	51	F=F	0.0	0.0	0.0	SeekSat
19	{flwHasLdr} InitMerge1 {flwHasLdr}	21	F=F	0.0	0.0	0.0	SeekSat
20	{flwHasSomeLdr} InitMerge1 {flwHasSomeLdr}	46	T=T	0.0	0.0	0.0	ProCon
21	{freeAgents} InitMerge1 {freeAgents}	26	T=T	0.0	0.0	0.0	SeekSat
22	{ldrNeverHasLdr} InitMerge2 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
23	{ldrHasNeighbor} InitMerge2 {ldrHasNeighbor}	36	T=T	0.0	0.0	0.0	ProCon
24	{ldrHasFlw} InitMerge2 {ldrHasFlw}	20	T=T	0.1	0.0	0.1	ProCon
25	{uniqueLdr} InitMerge2 {uniqueLdr}	55	F=F	0.0	0.0	0.0	SeekSat
26	{flwHasLdr} InitMerge2 {flwHasLdr}	21	F=F	0.0	0.0	0.0	SeekSat
27	{flwHasSomeLdr} InitMerge2 {flwHasSomeLdr}	46	T=T	0.1	0.0	0.1	ProCon
28	{freeAgents} InitMerge2 {freeAgents}	30	T=T	0.0	0.0	0.0	SeekSat
29	{ldrNeverHasLdr} InitMerge3 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
30	{ldrHasNeighbor} InitMerge3 {ldrHasNeighbor}	36	T=T	0.0	0.0	0.0	ProCon
31	{ldrHasFlw} InitMerge3 {ldrHasFlw}	20	T=T	0.1	0.0	0.1	ProCon
32	{uniqueLdr} InitMerge3 {uniqueLdr}	55	F=F	0.0	0.0	0.0	SeekSat
33	{flwHasLdr} InitMerge3 {flwHasLdr}	21	F=F	0.0	0.0	0.0	SeekSat
34	{flwHasSomeLdr} InitMerge3 {flwHasSomeLdr}	46	T=T	0.1	0.0	0.1	ProCon
35	{freeAgents} InitMerge3 {freeAgents}	30	T=T	0.0	0.0	0.0	SeekSat
36	{ldrNeverHasLdr} InitMerge4 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
37	{ldrHasNeighbor} InitMerge4 {ldrHasNeighbor}	36	T=T	0.0	0.0	0.0	ProCon
38	{ldrHasFlw} InitMerge4 {ldrHasFlw}	20	T=T	0.0	0.0	0.0	ProCon
39	{uniqueLdr} InitMerge4 {uniqueLdr}	51	F=F	0.0	0.0	0.0	SeekSat
40	{flwHasLdr} InitMerge4 {flwHasLdr}	21	T=T	0.0	0.0	0.0	ProCon
41	{flwHasSomeLdr} InitMerge4 {flwHasSomeLdr}	46	T=T	0.0	0.0	0.0	ProCon
42	{freeAgents} InitMerge4 {freeAgents}	22	T=T	0.0	0.0	0.0	SeekSat
43	{ldrNeverHasLdr} Pass1 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
44	{ldrHasNeighbor} Pass1 {ldrHasNeighbor}	44	T=T	0.1	0.0	0.1	ProCon
45	{ldrHasFlw} Pass1 {ldrHasFlw}	24	T=T	0.1	0.0	0.1	ProCon
46	{uniqueLdr} Pass1 {uniqueLdr}	66	T=T	0.4	0.0	0.4	ProCon
47	{flwHasLdr} Pass1 {flwHasLdr}	32	T=T	0.0	0.0	0.0	ProCon
48	{flwHasSomeLdr} Pass1 {flwHasSomeLdr}	46	T=T	0.0	0.0	0.0	ProCon
49	{freeAgents} Pass1 {freeAgents}	42	T=T	0.0	0.0	0.0	SeekSat
50	{ldrNeverHasLdr} Ldr2Flw {ldrNeverHasLdr}	41	F=F	0.0	0.0	0.0	SeekSat
51	{ldrHasNeighbor} Ldr2Flw {ldrHasNeighbor}	63	T=T	0.0	0.0	0.0	ProCon
52	{ldrHasFlw} Ldr2Flw {ldrHasFlw}	51	T=T	0.0	0.0	0.0	ProCon
53	{uniqueLdr} Ldr2Flw {uniqueLdr}	69	T=T	0.0	0.0	0.0	ProCon
54	{flwHasLdr} Ldr2Flw {flwHasLdr}	54	F=F	0.1	0.0	0.1	ProCon
55	{flwHasSomeLdr} Ldr2Flw {flwHasSomeLdr}	69	T=T	0.1	0.0	0.1	ProCon
56	{freeAgents} Ldr2Flw {freeAgents}	50	T=T	0.0	0.0	0.0	SeekSat
57	{ldrNeverHasLdr} InitSplit1 {ldrNeverHasLdr}	10	T=T	0.0	0.0	0.0	SeekSat
58	{ldrHasNeighbor} InitSplit1 {ldrHasNeighbor}	28	T=T	0.0	0.0	0.0	ProCon
59	{ldrHasFlw} InitSplit1 {ldrHasFlw}	24	T=T	0.0	0.0	0.0	ProCon
60	{uniqueLdr} InitSplit1 {uniqueLdr}	44	T=T	0.1	0.0	0.1	ProCon
61	{flwHasLdr} InitSplit1 {flwHasLdr}	21	T=T	0.1	0.0	0.1	ProCon
62	{flwHasSomeLdr} InitSplit1 {flwHasSomeLdr}	42	T=T	0.0	0.0	0.0	ProCon
63	{freeAgents} InitSplit1 {freeAgents}	26	T=T	0.0	0.0	0.0	SeekSat
64	{ldrNeverHasLdr} Pass2 {ldrNeverHasLdr}	19	T=T	0.0	0.0	0.0	SeekSat
65	{ldrHasNeighbor} Pass2 {ldrHasNeighbor}	44	T=T	0.0	0.0	0.0	ProCon
66	{ldrHasFlw} Pass2 {ldrHasFlw}	28	T=T	0.0	0.0	0.0	ProCon
67	{uniqueLdr} Pass2 {uniqueLdr}	65	F=F	0.0	0.0	0.0	SeekSat
68	{flwHasLdr} Pass2 {flwHasLdr}	50	F=F	0.0	0.0	0.0	ProCon
69	{flwHasSomeLdr} Pass2 {flwHasSomeLdr}	46	T=T	0.0	0.0	0.0	ProCon
70	{freeAgents} Pass2 {freeAgents}	42	T=T	0.0	0.0	0.0	SeekSat

Table 7.6: Car platoons case study: results 1-70

#	specification	$ wlp $	result	t_{Σ}	t_{wlp}	t_{\models}	decider
71	$\{ldrNeverHasLdr\}$ SplitMiddle $\{ldrNeverHasLdr\}$	143	T=T	0.0	0.0	0.0	SeekSat
72	$\{ldrHasNeighbor\}$ SplitMiddle $\{ldrHasNeighbor\}$	191	T=T	1.0	0.0	0.9	ProCon
73	$\{ldrHasFlw\}$ SplitMiddle $\{ldrHasFlw\}$	159	T=T	0.4	0.0	0.4	ProCon
74	$\{uniqueLdr\}$ SplitMiddle $\{uniqueLdr\}$	169	F=F	0.0	0.0	0.0	SeekSat
75	$\{flwHasLdr\}$ SplitMiddle $\{flwHasLdr\}$	154	F=F	0.1	0.0	0.1	SeekSat
76	$\{flwHasSomeLdr\}$ SplitMiddle $\{flwHasSomeLdr\}$	167	T=T	0.4	0.0	0.4	ProCon
77	$\{freeAgents\}$ SplitMiddle $\{freeAgents\}$	144	T=T	0.0	0.0	0.0	SeekSat
78	$\{ldrNeverHasLdr\}$ SplitLast $\{ldrNeverHasLdr\}$	37	T=T	0.0	0.0	0.0	SeekSat
79	$\{ldrHasNeighbor\}$ SplitLast $\{ldrHasNeighbor\}$	76	F=F	0.0	0.0	0.0	SeekSat
80	$\{ldrHasFlw\}$ SplitLast $\{ldrHasFlw\}$	52	F=F	0.0	0.0	0.0	SeekSat
81	$\{uniqueLdr\}$ SplitLast $\{uniqueLdr\}$	63	T=T	0.1	0.0	0.1	ProCon
82	$\{flwHasLdr\}$ SplitLast $\{flwHasLdr\}$	48	T=T	0.0	0.0	0.0	ProCon
83	$\{flwHasSomeLdr\}$ SplitLast $\{flwHasSomeLdr\}$	65	T=T	0.0	0.0	0.0	ProCon
84	$\{freeAgents\}$ SplitLast $\{freeAgents\}$	50	T=T	0.0	0.0	0.0	SeekSat
85	$\{ldrNeverHasLdr\}$ InitSplit2 $\{ldrNeverHasLdr\}$	10	T=T	0.0	0.0	0.0	SeekSat
86	$\{ldrHasNeighbor\}$ InitSplit2 $\{ldrHasNeighbor\}$	28	T=T	0.0	0.0	0.0	ProCon
87	$\{ldrHasFlw\}$ InitSplit2 $\{ldrHasFlw\}$	20	T=T	0.1	0.0	0.1	ProCon
88	$\{uniqueLdr\}$ InitSplit2 $\{uniqueLdr\}$	44	F=F	0.0	0.0	0.0	SeekSat
89	$\{flwHasLdr\}$ InitSplit2 $\{flwHasLdr\}$	21	F=F	0.0	0.0	0.0	SeekSat
90	$\{flwHasSomeLdr\}$ InitSplit2 $\{flwHasSomeLdr\}$	42	T=T	0.1	0.0	0.1	ProCon
91	$\{freeAgents\}$ InitSplit2 $\{freeAgents\}$	26	T=T	0.0	0.0	0.0	SeekSat
92	$\{ldrNeverHasLdr\}$ TwoSplit $\{ldrNeverHasLdr\}$	45	T=T	0.0	0.0	0.0	SeekSat
93	$\{ldrHasNeighbor\}$ TwoSplit $\{ldrHasNeighbor\}$	71	T=T	0.1	0.0	0.1	ProCon
94	$\{ldrHasFlw\}$ TwoSplit $\{ldrHasFlw\}$	55	T=T	0.1	0.0	0.1	ProCon
95	$\{uniqueLdr\}$ TwoSplit $\{uniqueLdr\}$	67	T=T	0.0	0.0	0.0	ProCon
96	$\{flwHasLdr\}$ TwoSplit $\{flwHasLdr\}$	53	T=T	0.1	0.0	0.1	ProCon
97	$\{flwHasSomeLdr\}$ TwoSplit $\{flwHasSomeLdr\}$	73	T=T	0.1	0.0	0.1	ProCon
98	$\{freeAgents\}$ TwoSplit $\{freeAgents\}$	62	T=T	0.0	0.0	0.0	SeekSat
99	$\{ldrNeverHasLdr\}$ Pass3 $\{ldrNeverHasLdr\}$	197	T=T	0.1	0.0	0.1	SeekSat
100	$\{ldrHasNeighbor\}$ Pass3 $\{ldrHasNeighbor\}$	227	T=T	1.2	0.0	1.2	ProCon
101	$\{ldrHasFlw\}$ Pass3 $\{ldrHasFlw\}$	207	T=T	0.6	0.0	0.6	ProCon
102	$\{uniqueLdr\}$ Pass3 $\{uniqueLdr\}$	238	T=T	0.2	0.0	0.2	ProCon
103	$\{flwHasLdr\}$ Pass3 $\{flwHasLdr\}$	210	T=T	0.2	0.0	0.2	ProCon
104	$\{flwHasSomeLdr\}$ Pass3 $\{flwHasSomeLdr\}$	229	T=T	1.2	0.0	1.2	ProCon
105	$\{freeAgents\}$ Pass3 $\{freeAgents\}$	210	T=T	0.1	0.0	0.1	SeekSat
106	$\{ldrNeverHasLdr\}$ Pass4 $\{ldrNeverHasLdr\}$	19	T=T	0.0	0.0	0.0	SeekSat
107	$\{ldrHasNeighbor\}$ Pass4 $\{ldrHasNeighbor\}$	44	T=T	0.1	0.0	0.1	ProCon
108	$\{ldrHasFlw\}$ Pass4 $\{ldrHasFlw\}$	24	T=T	0.0	0.0	0.0	ProCon
109	$\{uniqueLdr\}$ Pass4 $\{uniqueLdr\}$	65	F=F	0.0	0.0	0.0	SeekSat
110	$\{flwHasLdr\}$ Pass4 $\{flwHasLdr\}$	34	F=F	0.0	0.0	0.0	SeekSat
111	$\{flwHasSomeLdr\}$ Pass4 $\{flwHasSomeLdr\}$	46	T=T	0.0	0.0	0.0	ProCon
112	$\{freeAgents\}$ Pass4 $\{freeAgents\}$	42	T=T	0.0	0.0	0.0	SeekSat
113	$\{ldrNeverHasLdr\}$ AccLeader $\{ldrNeverHasLdr\}$	37	T=T	0.0	0.0	0.0	SeekSat
114	$\{ldrHasNeighbor\}$ AccLeader $\{ldrHasNeighbor\}$	76	F=F	0.0	0.0	0.0	SeekSat
115	$\{ldrHasFlw\}$ AccLeader $\{ldrHasFlw\}$	52	T=T	0.0	0.0	0.0	ProCon
116	$\{uniqueLdr\}$ AccLeader $\{uniqueLdr\}$	63	T=T	0.0	0.0	0.0	ProCon
117	$\{flwHasLdr\}$ AccLeader $\{flwHasLdr\}$	48	T=T	0.1	0.0	0.1	ProCon
118	$\{flwHasSomeLdr\}$ AccLeader $\{flwHasSomeLdr\}$	65	T=T	0.1	0.0	0.1	ProCon
119	$\{freeAgents\}$ AccLeader $\{freeAgents\}$	50	T=T	0.0	0.0	0.0	SeekSat
120	$\{ldrNeverHasLdr\}$ BasicCarPlatooning $\{ldrNeverHasLdr\}$	65	F=F	0.0	0.0	0.0	SeekSat
121	$\{ldrHasNeighbor\}$ BasicCarPlatooning $\{ldrHasNeighbor\}$	223	T=T	0.1	0.0	0.1	ProCon
122	$\{ldrHasFlw\}$ BasicCarPlatooning $\{ldrHasFlw\}$	143	T=T	0.0	0.0	0.0	ProCon
123	$\{uniqueLdr\}$ BasicCarPlatooning $\{uniqueLdr\}$	284	F=F	0.0	0.0	0.0	SeekSat
124	$\{flwHasLdr\}$ BasicCarPlatooning $\{flwHasLdr\}$	125	F=F	0.0	0.0	0.0	SeekSat
125	$\{flwHasSomeLdr\}$ BasicCarPlatooning $\{flwHasSomeLdr\}$	261	T=T	0.1	0.0	0.1	ProCon
126	$\{freeAgents\}$ BasicCarPlatooning $\{freeAgents\}$	80	T=T	0.0	0.0	0.0	SeekSat
127	$\{empty\}$ BasicCarPlatooning* $\{ldrNeverHasLdr\}$	102	T=T	49.0	49.0	0.0	SeekSat
128	$\{empty\}$ BasicCarPlatooning* $\{ldrHasNeighbor\}$	17	T=T	0.1	0.1	0.0	SeekSat
129	$\{empty\}$ BasicCarPlatooning* $\{ldrHasFlw\}$	17	T=T	0.1	0.1	0.0	SeekSat
130	$\{empty\}$ BasicCarPlatooning* $\{uniqueLdr\}$	6	F=F	2.0	2.0	0.0	SeekSat
131	$\{empty\}$ BasicCarPlatooning* $\{flwHasLdr\}$	6	F=F	184.4	184.4	0.0	SeekSat
132	$\{empty\}$ BasicCarPlatooning* $\{flwHasSomeLdr\}$	29	T=T	0.1	0.1	0.0	SeekSat
133	$\{empty\}$ BasicCarPlatooning* $\{freeAgents\}$	14	T=T	0.0	0.0	0.0	SeekSat
134	$\{empty\}$ CarPlatooning* $\{ldrNeverHasLdr\}$	102	T=T	200.1	200.1	0.0	SeekSat
135	$\{empty\}$ CarPlatooning* $\{ldrHasNeighbor\}$	6	F=F	643.9	643.9	0.0	SeekSat
136	$\{empty\}$ CarPlatooning* $\{ldrHasFlw\}$	6	F=F	374.2	374.2	0.0	SeekSat
137	$\{empty\}$ CarPlatooning* $\{uniqueLdr\}$	6	F=F	371.0	371.0	0.0	SeekSat
138	$\{empty\}$ CarPlatooning* $\{flwHasLdr\}$	6	F=F	384.3	384.3	0.0	SeekSat
139	$\{empty\}$ CarPlatooning* $\{flwHasSomeLdr\}$	29	T=T	1.6	1.6	0.0	SeekSat
140	$\{empty\}$ CarPlatooning* $\{freeAgents\}$	14	T=T	0.4	0.4	0.0	SeekSat
μ	arithmetic mean	59.1		15.9	15.8	0.1	
$\mu_{1/2}$	median	43		0.0	0.0	0.0	

Table 7.7: Car platoons case study: results 71-140

for instance, “waypoint” is the only node label and “track” and “train” are the main edge labels, resulting in a higher number of possible combinations that every decider has to consider. So while the railroad control looks like a simpler transformation system, it is actually harder to verify due to its model.

Furthermore, a distinct characteristic of the car platoons rules is the fact that the “partner constraints” considered by Bauer really correspond to strong assertions (for some rules at least) making the verification of this transformation system relatively easy.

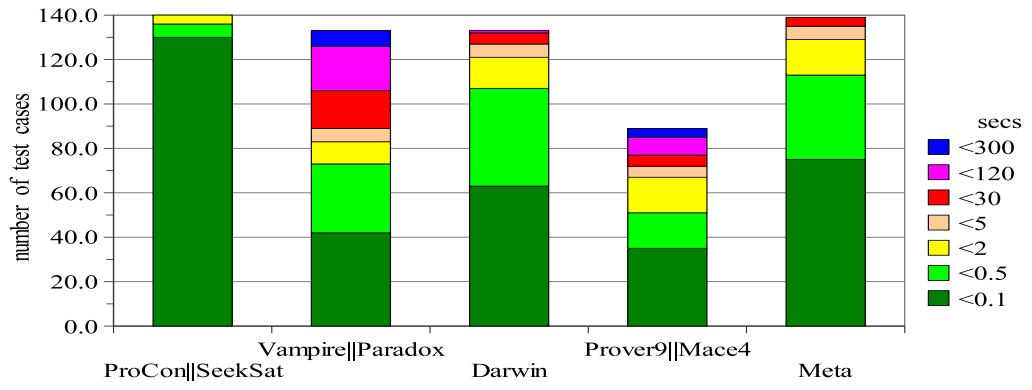


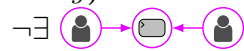
Figure 7.7: Car platoons: comparison of deciders (“ t_{\models} ”)

7.4 Access control for computer systems

In this section we briefly repeat the graph conditions and programs of the access control case study and present the results of the verification of its test specifications.

Example 7.9 (access control conditions). Consider the access control graphs as introduced in Example 2.4. The following graph conditions over the empty graph are used in the access control case study:

(*noSharing*) No session is shared between two users:

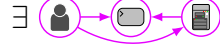


(*noTwoRights*) No user has more than one access right to one and the same system:



	result	t_{\perp} decoder	result	t_{\perp} decoder	result	t_{\perp} decoder	#	result	t_{\perp} decoder	result	t_{\perp} decoder	result	t_{\perp} decoder
1	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.0 Prover9	71	T=T	142.2 Vampire	T=T	0.9 DarwinProver	T≠E	--
2	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.4 Prover9	72	T≠E	--	T≠E	--	T=T	13.0 Prover9
3	T=T	0.0 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	73	T=T	32.6 Vampire	T=T	0.4 DarwinProver	T=T	12.0 Prover9
4	T=T	0.0 Vampire	T=T	0.1 DarwinSolver	T=T	0.0 Prover9	74	F=F	1.2 Paradox	F=F	1.6 DarwinSolver	F=F	43.8 Mace4
5	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T=T	0.4 Prover9	75	F=F	1.6 Paradox	F=F	1.2 DarwinSolver	F=F	86.9 Mace4
6	T=T	21.3 Vampire	T=T	0.1 DarwinProver	T=T	69.4 Prover9	76	T=T	22.3 Vampire	T=T	0.9 DarwinProver	T≠E	--
7	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.6 Prover9	77	T=T	41.8 Vampire	T=T	0.9 DarwinProver	T≠E	--
8	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	78	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T=T	2.0 Prover9
9	T=T	0.9 Vampire	T=T	0.1 DarwinProver	T=T	0.6 Prover9	79	F=F	0.2 Paradox	F=F	0.2 DarwinSolver	F=F	0.4 Mace4
10	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	80	F=F	0.1 Paradox	F=F	0.1 DarwinSolver	F=F	0.2 Mace4
11	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	81	T=T	43.9 Vampire	T=T	6.4 DarwinProver	T=T	8.1 Prover9
12	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	82	T=T	41.8 Vampire	T=T	2.0 DarwinProver	T≠E	--
13	T=T	22.3 Vampire	T=T	0.1 DarwinProver	T=T	76.6 Prover9	83	T=T	22.3 Vampire	T=T	0.2 DarwinProver	T≠E	--
14	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.6 Prover9	84	T=T	171.0 Vampire	T=T	1.6 DarwinProver	T≠E	--
15	T=T	0.0 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	85	T=T	0.0 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
16	T=T	246.8 Vampire	T=T	0.1 DarwinProver	T=T	3.6 Prover9	86	T=T	2.5 Vampire	T=T	0.1 DarwinProver	T=T	0.9 Prover9
17	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	87	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
18	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F=F	127.9 Mace4	88	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F=F	0.6 Mace4
19	F=F	0.2 Paradox	F=F	0.4 DarwinSolver	F≠E	--	89	F=F	0.2 Paradox	F=F	0.1 DarwinSolver	F=F	0.1 Mace4
20	T=T	21.2 Vampire	T=T	0.4 DarwinProver	T=T	224.2 Prover9	90	T=T	41.8 Vampire	T=T	0.2 DarwinProver	T=T	82.8 Prover9
21	T=T	41.8 Vampire	T=T	0.1 DarwinProver	T≠E	--	91	T=T	22.3 Vampire	T=T	0.1 DarwinProver	T≠E	--
22	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	92	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
23	T=T	4.2 Vampire	T=T	2.0 DarwinProver	T=T	0.9 Prover9	93	T=T	2.0 Vampire	T=T	1.2 DarwinProver	T=T	0.6 Prover9
24	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	94	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
25	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F≠E	--	95	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
26	F=F	0.1 Paradox	F=F	0.2 DarwinSolver	F≠E	--	96	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
27	T=T	40.7 Vampire	T=T	0.2 DarwinProver	T=T	176.1 Prover9	97	T=T	46.9 Vampire	T=T	0.4 DarwinProver	T≠E	--
28	T=T	41.8 Vampire	T=T	0.2 DarwinProver	T≠E	--	98	T≠E	--	T=T	0.6 DarwinProver	T≠E	--
29	T=T	0.0 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	99	T=T	0.1 Vampire	T=T	0.6 DarwinProver	T=T	0.6 Prover9
30	T=T	3.7 Vampire	T=T	0.6 DarwinProver	T=T	0.6 Prover9	100	T=T	0.9 Vampire	T=T	0.4 DarwinProver	T=T	1.2 Prover9
31	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	101	T=T	7.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9
32	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F≠E	--	102	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T≠E	--
33	F=F	0.2 Paradox	F=F	0.2 DarwinSolver	F=F	163.7 Mace4	103	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T=T	19.2 Prover9
34	T=T	40.7 Vampire	T=T	0.1 DarwinProver	T=T	88.9 Prover9	104	T=T	22.4 Vampire	T=T	0.4 DarwinProver	T≠E	--
35	T=T	41.8 Vampire	T=T	0.2 DarwinProver	T≠E	--	105	T=T	40.8 Vampire	T=T	3.0 DarwinProver	T≠E	--
36	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.0 Prover9	106	T=T	27.4 Vampire	T=T	0.2 DarwinProver	T≠E	--
37	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.9 Prover9	107	T=T	3.6 Vampire	T=T	0.2 DarwinProver	T=T	2.5 Prover9
38	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.0 Prover9	108	T=T	0.3 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
39	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F=F	9.0 Mace4	109	F=F	19.2 Paradox	F≠E	--	F≠E	--
40	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.9 Prover9	110	F=F	0.6 Paradox	F≠E	--	F≠E	--
41	T=T	41.7 Vampire	T=T	0.1 DarwinProver	T=T	38.7 Prover9	111	T=T	22.3 Vampire	T=T	0.4 DarwinProver	T≠E	--
42	T=T	12.0 Vampire	T=T	0.1 DarwinProver	T≠E	--	112	T≠E	--	T=T	0.4 DarwinProver	T≠E	--
43	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	113	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	1.2 Prover9
44	T=T	0.4 Vampire	T=T	5.6 DarwinProver	T=T	2.5 Prover9	114	F=F	0.2 Paradox	F=F	0.1 DarwinSolver	F=F	0.1 Mace4
45	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T=T	0.1 Prover9	115	T=T	0.6 Vampire	T=T	0.1 DarwinProver	T=T	0.9 Prover9
46	T≠E	--	T=T	0.6 DarwinProver	T≠E	--	116	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9
47	T=T	87.9 Vampire	T=T	0.2 DarwinProver	T≠E	--	117	T=T	90.0 Vampire	T=T	2.0 DarwinProver	T≠E	--
48	T=T	41.8 Vampire	T=T	0.2 DarwinProver	T≠E	--	118	T=T	24.4 Vampire	T=T	0.4 DarwinProver	T≠E	--
49	T=T	22.3 Vampire	T=T	0.4 DarwinProver	T≠E	--	119	T≠E	--	T=T	2.0 DarwinProver	T≠E	--
50	F=F	0.1 Paradox	F=F	1.2 DarwinSolver	F=F	0.2 Mace4	120	F=F	0.4 Paradox	F=F	58.2 DarwinSolver	F≠E	--
51	T=T	0.6 Vampire	T=T	0.2 DarwinProver	T=T	0.6 Prover9	121	T≠E	--	T≠E	--	T≠E	--
52	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	122	T=T	196.7 Vampire	T=T	2.0 DarwinProver	T≠E	--
53	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.6 Prover9	123	F=F	0.9 Paradox	F=F	0.4 DarwinSolver	F≠E	--
54	F=F	0.9 Paradox	F≠E	--	F≠E	--	124	F=F	0.4 Paradox	F≠E	--	F≠E	--
55	T=T	21.3 Vampire	T=T	0.4 DarwinProver	T≠E	--	125	T≠E	--	T=T	12.0 DarwinProver	T≠E	--
56	T=T	25.4 Vampire	T=T	0.9 DarwinProver	T≠E	--	126	T=T	181.3 Vampire	T=T	0.6 DarwinProver	T≠E	--
57	T=T	0.0 Vampire	T=T	0.1 DarwinProver	T=T	0.0 Prover9	127	T=T	0.3 Vampire	T=T	0.3 DarwinProver	T≠E	--
58	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	128	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
59	T=T	35.6 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	129	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
60	T=T	142.3 Vampire	T=T	0.2 DarwinProver	T≠E	--	130	F=F	0.2 Paradox	F=F	0.2 DarwinProver	F=F	0.1 Mace4
61	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.4 Prover9	131	F=F	0.3 Paradox	F=F	0.2 DarwinProver	F=F	0.1 Mace4
62	T=T	21.3 Vampire	T=T	0.1 DarwinProver	T=T	61.2 Prover9	132	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.1 Prover9
63	T=T	40.7 Vampire	T=T	0.1 DarwinProver	T≠E	--	133	T=T	0.1 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9
64	T=T	45.8 Vampire	T=T	0.4 DarwinProver	T≠E	--	134	T=T	0.2 Vampire	T=T	0.4 DarwinProver	T≠E	--
65	T=T	3.0 Vampire	T=T	9.1 DarwinProver	T=T	3.0 Prover9	135	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F=F	0.1 Mace4
66	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	136	F=F	0.2 Paradox	F=F	0.1 DarwinProver	F=F	0.1 Mace4
67	F=F	0.6 Paradox	F=F	9.0 DarwinSolver	F≠E	--	137	F=F	0.3 Paradox	F=F	0.2 DarwinProver	F=F	0.2 Mace4
68	F=F	0.2 Paradox	F≠E	--	F≠E	--	138	F=F	0.4 Paradox	F=F	0.1 DarwinProver	F=F	0.2 Mace4
69	T=T	41.8 Vampire	T=T	0.4 DarwinProver	T≠E	--	139	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
70	T=T	151.5 Vampire	T=T	0.4 DarwinProver	T≠E	--	140	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
							$\mu/2$	19.2		1.1		15.0	
							$\mu_1/2$	0.3		0.2		0.2	

(*someAccess*) There is some user that has an established session to some system:



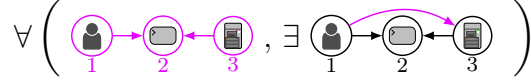
(*noMultiSession*) No session is associated to more than one system:



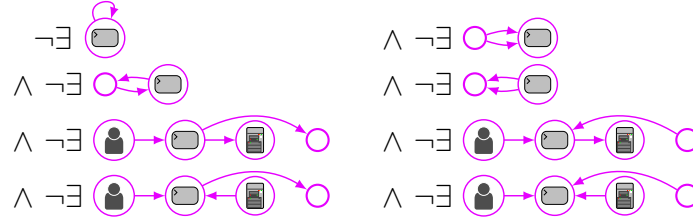
(*sessionStates*) Every session is either proposed or established: etch1.4



(*secure*) Every user logged into a system has the corresponding access right:



(*sessionDeletable*) Every session is deletable, that is, there exists no additional edges that prevent deletion of a session node:



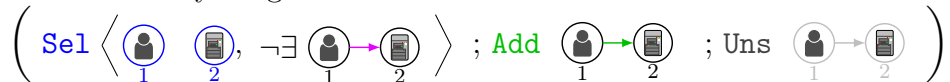
Let $secureInvariant = secure \wedge noSharing \wedge sessionDeletable$, let $decidable = noSharing \wedge noTwoRights \wedge noMultiSession \wedge sessionDeletable$ and let $ACConstraint = secureInvariant \wedge sessionStates$.

Example 7.10 (access control system). The following graph programs constitute the dynamic part of our access control and model the addition and deletion of users, the grant and removal of access rights and the login/logout procedure.

(AddUser) Adds a user to the system. A user node is created and unselected:



(Grant) Grants a user access to a system. Selects a user and a system (for which not an access right already exists), adds an access right, and unselects everything:



(Login) A user requests to log into a system. The program selects a user and a system, adds a session node with its edges and unselects everything:

$$\left(\text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} ; \text{Add } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 3 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} ; \text{Uns } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 3 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} \right)$$

(Logout) A user is logged out. A session is selected and whether it is established or proposed, it is closed:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} ; \\ \left\{ \begin{array}{l} \left(\text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 4 \end{array} ; \text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 4 \end{array} \right), \\ \left(\text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 4 \end{array} ; \text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 4 \end{array} \right) \end{array} \right\} ; \\ \text{Uns } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 4 \end{array} \end{array} \right)$$

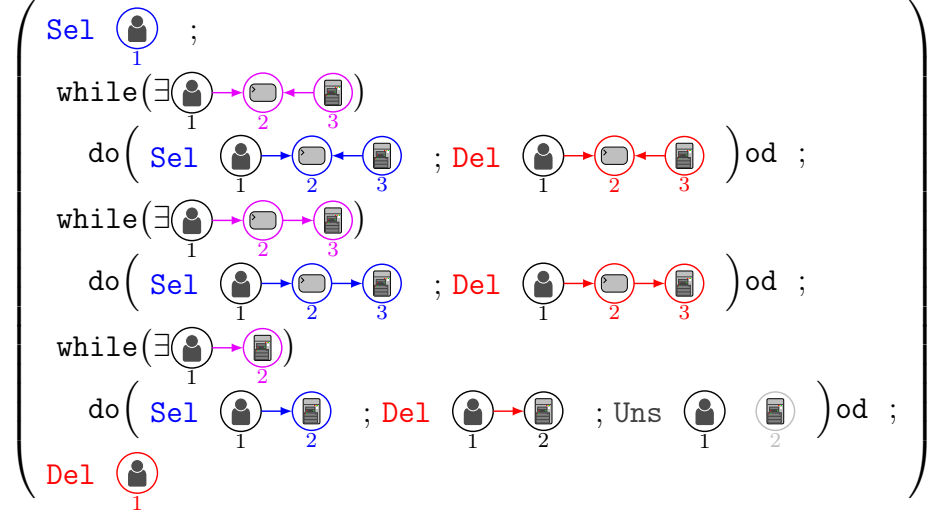
(Process) The system reacts to a log in. The program selects a proposed session. If the user has the appropriate access right, the session is established. Otherwise, the session is closed:

$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} ; \\ \text{if } \left(\exists \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} \right) \\ \text{then } \left(\text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} ; \text{Add } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} ; \text{Uns } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} \right) \\ \text{else } \left(\text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 2 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} ; \text{Uns } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 3 \end{array} \right) \text{fi} \end{array} \right)$$

(Revoke) The access right of a user to a system is revoked. Beforehand, the user's established sessions to that system are closed:

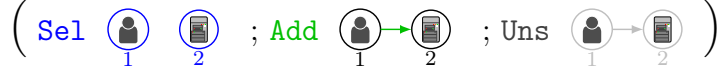
$$\left(\begin{array}{l} \text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} ; \\ \text{while } \left(\exists \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 4 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} \right) \\ \text{do } \left(\text{Sel } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 4 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} ; \text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{Session} \\ 4 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} \right) \text{od} ; \\ \text{Del } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} ; \\ \text{Uns } \begin{array}{c} \text{User} \\ 1 \end{array} \begin{array}{c} \text{System} \\ 2 \end{array} \end{array} \right)$$

(Delete) A user is deleted. Beforehand, the user's sessions are closed and the user's access rights revoked:

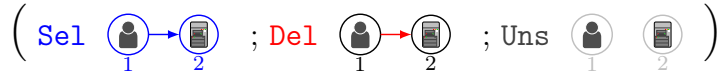


Let $\text{AccessControl} = \{\text{AddUser}, \text{Grant}, \text{Login}, \text{Logout}, \text{Process}, \text{Revoke}, \text{Delete}\}$ be the non-deterministic choice of the above graph programs. Aside the access control programs defined above, we consider two similar programs where we deliberately induce certain errors.

(GrantWrong) Grants a user access to a system, but does not check whether an access right already exists. This is expected to violate the condition *noTwoRights*.

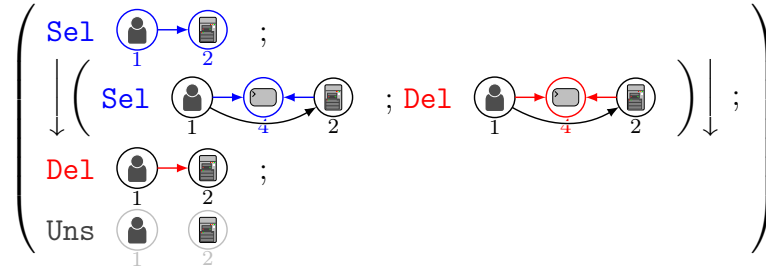


(RevokeWrong) The access right of a user to a system is revoked, but it is assumed, the user has exactly one session established to that computer system. This is expected to violate the *secure* as well as the *secureInvariant* condition.

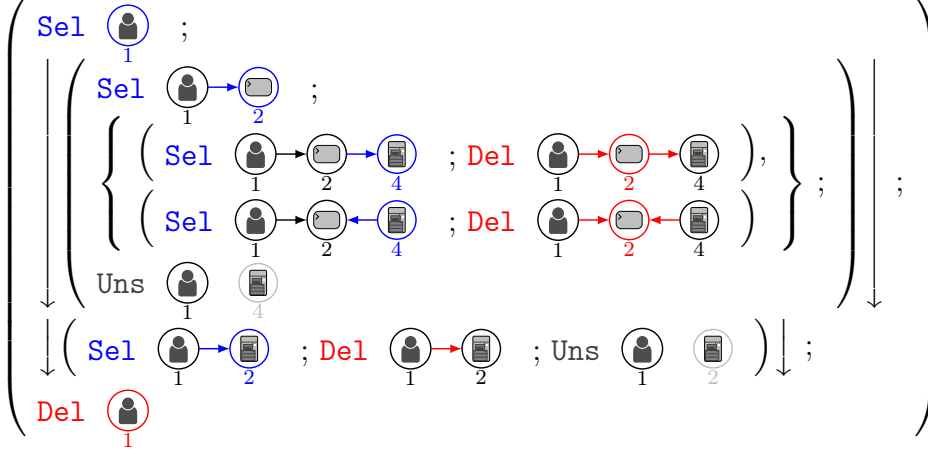


We also consider alternative programs of **Revoke** and **Delete** using the as-long-as-possible iteration instead of a while-loop.

(RevokeAlt) The access right of a user to a system is revoked. Beforehand, the established sessions of the user to that system are closed as long as there are such sessions:



(DeleteAlt) A user is deleted. Beforehand, the user's sessions are closed and the user's access rights revoked as long as possible:



We now want to verify whether or not the access control programs are correct with respect to the access control conditions of Example 7.9. We will use ENFORCE to automatically verify or refute test specifications of the form $\{c\}P\{c\}$, where c is a condition and P is a program.

Example 7.11 (access control verification). Table 7.9 and 7.10 on page 139f show a log of the access control case study. The second column states the specification of each test case. The column “ $|wlp|$ ” states the complexity of the constructed weakest preconditions, that is, the number of logical symbols. The column “result” compares the expected result with the computed result in the sense of an automated unit test. Here, **True** stands for tautology, **False** stands for contradiction, and an **Exception** indicates no decision. The following columns state the number of seconds it took to decide each specification “ t_Σ ”, which is roughly the sum of the time it took to construct the weakest precondition “ t_{wlp} ” and the time to decide the implication problem “ t_\models ”. The last column “decider” states which algorithm contributed the decision of the implication problem.

The results can be interpreted as follows: Most importantly, 117 of 120 test specifications can be automatically decided by ENFORCE. In test case #110, the approximation of an invariant weak enough to conduct a proof fails. In cases #88 and #92, the alleged counterexample cannot be retraced. Concerning performance, 87 test cases can be decided in less than 2 seconds, the majority in less than 0.2 seconds. Concerning correctness of **AccessControl**, we have proven that *noSharing*, *noTwoRights*, *noMultiSession*, *decidable* and *secureInvariant* are invariants, while *someAccess* and *secure* are not (see test case #112 or #118). Concerning the implication problem, Figure 7.8 shows

#	specification	$ wlp $	result	t_Σ	t_{wlp}	t_{\models}	decider
1	$\{noSharing\}$ AddUser $\{noSharing\}$	9	T=T	0.0	0.0	0.0	SeekSat
2	$\{noTwoRights\}$ AddUser $\{noTwoRights\}$	9	T=T	0.0	0.0	0.0	SeekSat
3	$\{someAccess\}$ AddUser $\{someAccess\}$	7	T=T	0.0	0.0	0.0	SeekSat
4	$\{noMultiSession\}$ AddUser $\{noMultiSession\}$	25	T=T	0.0	0.0	0.0	SeekSat
5	$\{sessionDeletable\}$ AddUser $\{sessionDeletable\}$	60	T=T	0.0	0.0	0.0	SeekSat
6	$\{decidable\}$ AddUser $\{decidable\}$	97	T=T	0.0	0.0	0.0	SeekSat
7	$\{sessionStates\}$ AddUser $\{sessionStates\}$	20	T=T	0.1	0.0	0.1	ProCon
8	$\{secure\}$ AddUser $\{secure\}$	12	T=T	0.0	0.0	0.0	ProCon
9	$\{secureInvariant\}$ AddUser $\{secureInvariant\}$	78	T=T	1.2	0.0	1.2	ProCon
10	$\{ACConstraint\}$ AddUser $\{ACConstraint\}$	96	T=T	1.2	0.0	1.2	ProCon
11	$\{noSharing\}$ Grant $\{noSharing\}$	18	T=T	0.0	0.0	0.0	SeekSat
12	$\{noTwoRights\}$ Grant $\{noTwoRights\}$	22	T=T	0.0	0.0	0.0	SeekSat
13	$\{someAccess\}$ Grant $\{someAccess\}$	28	T=T	0.0	0.0	0.0	SeekSat
14	$\{noMultiSession\}$ Grant $\{noMultiSession\}$	35	T=T	0.0	0.0	0.0	SeekSat
15	$\{sessionDeletable\}$ Grant $\{sessionDeletable\}$	96	T=T	0.1	0.0	0.1	SeekSat
16	$\{decidable\}$ Grant $\{decidable\}$	139	T=T	0.2	0.0	0.2	SeekSat
17	$\{sessionStates\}$ Grant $\{sessionStates\}$	35	T=T	0.1	0.0	0.1	ProCon
18	$\{secure\}$ Grant $\{secure\}$	29	T=T	0.1	0.0	0.1	ProCon
19	$\{secureInvariant\}$ Grant $\{secureInvariant\}$	122	T=T	6.5	0.0	6.5	ProCon
20	$\{ACConstraint\}$ Grant $\{ACConstraint\}$	146	T=T	11.0	0.0	11.0	ProCon
21	$\{noSharing\}$ GrantWrong $\{noSharing\}$	9	T=T	0.0	0.0	0.0	SeekSat
22	$\{noTwoRights\}$ GrantWrong $\{noTwoRights\}$	9	F=F	0.0	0.0	0.0	SeekSat
23	$\{someAccess\}$ GrantWrong $\{someAccess\}$	24	T=T	0.0	0.0	0.0	SeekSat
24	$\{noMultiSession\}$ GrantWrong $\{noMultiSession\}$	25	T=T	0.0	0.0	0.0	SeekSat
25	$\{sessionDeletable\}$ GrantWrong $\{sessionDeletable\}$	84	T=T	0.0	0.0	0.0	SeekSat
26	$\{decidable\}$ GrantWrong $\{decidable\}$	121	F=F	0.1	0.0	0.1	SeekSat
27	$\{sessionStates\}$ GrantWrong $\{sessionStates\}$	28	T=T	0.0	0.0	0.0	ProCon
28	$\{secure\}$ GrantWrong $\{secure\}$	27	T=T	0.1	0.0	0.1	ProCon
29	$\{secureInvariant\}$ GrantWrong $\{secureInvariant\}$	116	T=T	3.6	0.0	3.6	ProCon
30	$\{ACConstraint\}$ GrantWrong $\{ACConstraint\}$	142	T=T	4.3	0.0	4.3	ProCon
31	$\{noSharing\}$ Login $\{noSharing\}$	9	T=T	0.0	0.0	0.0	SeekSat
32	$\{noTwoRights\}$ Login $\{noTwoRights\}$	9	T=T	0.0	0.0	0.0	SeekSat
33	$\{someAccess\}$ Login $\{someAccess\}$	24	T=T	0.0	0.0	0.0	SeekSat
34	$\{noMultiSession\}$ Login $\{noMultiSession\}$	25	T=T	0.0	0.0	0.0	SeekSat
35	$\{sessionDeletable\}$ Login $\{sessionDeletable\}$	84	T=T	0.0	0.0	0.0	SeekSat
36	$\{decidable\}$ Login $\{decidable\}$	121	T=T	0.1	0.0	0.1	SeekSat
37	$\{sessionStates\}$ Login $\{sessionStates\}$	28	T=T	0.1	0.0	0.1	ProCon
38	$\{secure\}$ Login $\{secure\}$	34	T=T	0.1	0.0	0.1	ProCon
39	$\{secureInvariant\}$ Login $\{secureInvariant\}$	123	T=T	3.6	0.0	3.6	ProCon
40	$\{ACConstraint\}$ Login $\{ACConstraint\}$	149	T=T	2.5	0.0	2.5	ProCon
41	$\{noSharing\}$ Logout $\{noSharing\}$	80	T=T	0.0	0.0	0.0	SeekSat
42	$\{noTwoRights\}$ Logout $\{noTwoRights\}$	88	T=T	0.0	0.0	0.0	SeekSat
43	$\{someAccess\}$ Logout $\{someAccess\}$	101	F=F	0.0	0.0	0.0	SeekSat
44	$\{noMultiSession\}$ Logout $\{noMultiSession\}$	107	T=T	0.0	0.0	0.0	SeekSat
45	$\{sessionDeletable\}$ Logout $\{sessionDeletable\}$	214	T=T	0.3	0.1	0.2	SeekSat
46	$\{decidable\}$ Logout $\{decidable\}$	283	T=T	0.6	0.2	0.4	SeekSat
47	$\{sessionStates\}$ Logout $\{sessionStates\}$	111	T=T	0.2	0.0	0.2	ProCon
48	$\{secure\}$ Logout $\{secure\}$	112	T=T	0.4	0.0	0.4	ProCon
49	$\{secureInvariant\}$ Logout $\{secureInvariant\}$	269	T=T	16.2	0.1	16.1	ProCon
50	$\{ACConstraint\}$ Logout $\{ACConstraint\}$	311	T=T	18.3	0.1	18.2	ProCon
51	$\{noSharing\}$ RevokeWrong $\{noSharing\}$	14	T=T	0.0	0.0	0.0	SeekSat
52	$\{noTwoRights\}$ RevokeWrong $\{noTwoRights\}$	22	T=T	0.0	0.0	0.0	SeekSat
53	$\{someAccess\}$ RevokeWrong $\{someAccess\}$	24	F=F	0.0	0.0	0.0	SeekSat
54	$\{noMultiSession\}$ RevokeWrong $\{noMultiSession\}$	41	T=T	0.0	0.0	0.0	SeekSat
55	$\{sessionDeletable\}$ RevokeWrong $\{sessionDeletable\}$	148	T=T	0.1	0.0	0.1	SeekSat
56	$\{decidable\}$ RevokeWrong $\{decidable\}$	217	T=T	0.1	0.0	0.1	SeekSat
57	$\{sessionStates\}$ RevokeWrong $\{sessionStates\}$	28	T=T	0.0	0.0	0.0	ProCon
58	$\{secure\}$ RevokeWrong $\{secure\}$	34	F=F	0.0	0.0	0.0	SeekSat
59	$\{secureInvariant\}$ RevokeWrong $\{secureInvariant\}$	191	F=F	0.1	0.0	0.1	SeekSat
60	$\{ACConstraint\}$ RevokeWrong $\{ACConstraint\}$	217	F=F	0.1	0.0	0.1	SeekSat

Table 7.9: Access control case study: results 1-60

#	specification	$ wlp $	result	t_{Σ}	t_{wlp}	t_{\models}	decider
61	$\{noSharing\}$ Process $\{noSharing\}$	19	T=T	0.0	0.0	0.0	SeekSat
62	$\{noTwoRights\}$ Process $\{noTwoRights\}$	21	T=T	0.0	0.0	0.0	SeekSat
63	$\{someAccess\}$ Process $\{someAccess\}$	91	T=T	0.0	0.0	0.0	SeekSat
64	$\{noMultiSession\}$ Process $\{noMultiSession\}$	38	T=T	0.0	0.0	0.0	SeekSat
65	$\{sessionDeletable\}$ Process $\{sessionDeletable\}$	109	T=T	0.3	0.1	0.1	SeekSat
66	$\{decidable\}$ Process $\{decidable\}$	152	T=T	0.4	0.2	0.2	SeekSat
67	$\{sessionStates\}$ Process $\{sessionStates\}$	34	T=T	0.0	0.0	0.0	ProCon
68	$\{secure\}$ Process $\{secure\}$	76	F=F	2.0	0.0	2.0	ProCon
69	$\{secureInvariant\}$ Process $\{secureInvariant\}$	143	T=T	9.3	0.2	9.1	ProCon
70	$\{ACConstraint\}$ Process $\{ACConstraint\}$	167	T=T	18.5	0.2	18.3	ProCon
71	$\{noSharing\}$ Revoke $\{noSharing\}$	27	T=T	0.0	0.0	0.0	SeekSat
72	$\{noTwoRights\}$ Revoke $\{noTwoRights\}$	27	T=T	0.0	0.0	0.0	SeekSat
73	$\{someAccess\}$ Revoke $\{someAccess\}$	29	F=F	4.6	4.6	0.0	SeekSat
74	$\{noMultiSession\}$ Revoke $\{noMultiSession\}$	64	T=T	0.1	0.1	0.0	SeekSat
75	$\{sessionDeletable\}$ Revoke $\{sessionDeletable\}$	217	T=T	1.2	0.8	0.4	SeekSat
76	$\{decidable\}$ Revoke $\{decidable\}$	300	T=T	1.3	1.2	0.1	SeekSat
77	$\{sessionStates\}$ Revoke $\{sessionStates\}$	28	T=T	0.0	0.0	0.0	ProCon
78	$\{secure\}$ Revoke $\{secure\}$	56	T=T	0.4	0.3	0.1	ProCon
79	$\{secureInvariant\}$ Revoke $\{secureInvariant\}$	277	T=T	23.4	11.3	12.0	ProCon
80	$\{ACConstraint\}$ Revoke $\{ACConstraint\}$	321	T=T	37.0	23.9	13.1	ProCon
81	$\{noSharing\}$ RevokeAlt $\{noSharing\}$	55	T=T	0.1	0.1	0.0	SeekSat
82	$\{noTwoRights\}$ RevokeAlt $\{noTwoRights\}$	55	T=T	0.1	0.1	0.0	SeekSat
83	$\{someAccess\}$ RevokeAlt $\{someAccess\}$	61	F=F	23.7	23.7	0.0	SeekSat
84	$\{noMultiSession\}$ RevokeAlt $\{noMultiSession\}$	92	T=T	0.1	0.1	0.0	SeekSat
85	$\{sessionDeletable\}$ RevokeAlt $\{sessionDeletable\}$	245	T=T	0.8	0.6	0.2	SeekSat
86	$\{decidable\}$ RevokeAlt $\{decidable\}$	328	T=T	1.3	1.1	0.2	SeekSat
87	$\{sessionStates\}$ RevokeAlt $\{sessionStates\}$	28	T=T	0.1	0.1	0.0	ProCon
88	$\{secure\}$ RevokeAlt $\{secure\}$	91	F \neq E	-	9.9	0.2	ProCon
89	$\{secureInvariant\}$ RevokeAlt $\{secureInvariant\}$	305	T=T	47.3	21.8	25.4	ProCon
90	$\{ACConstraint\}$ RevokeAlt $\{ACConstraint\}$	349	T=T	44.3	19.8	24.4	ProCon
91	$\{noSharing\}$ Delete $\{noSharing\}$	33	T=T	0.2	0.2	0.0	SeekSat
92	$\{someAccess\}$ Delete $\{someAccess\}$	45	F \neq E	-	147.7	0.0	SeekSat
93	$\{noTwoRights\}$ Delete $\{noTwoRights\}$	33	T=T	0.2	0.2	0.0	SeekSat
94	$\{noMultiSession\}$ Delete $\{noMultiSession\}$	72	T=T	0.3	0.3	0.0	SeekSat
95	$\{sessionDeletable\}$ Delete $\{sessionDeletable\}$	185	T=T	1.4	1.4	0.0	SeekSat
96	$\{decidable\}$ Delete $\{decidable\}$	260	T=T	2.0	1.9	0.1	SeekSat
97	$\{sessionStates\}$ Delete $\{sessionStates\}$	20	T=T	0.4	0.3	0.1	ProCon
98	$\{secure\}$ Delete $\{secure\}$	74	T=T	4.1	4.0	0.1	ProCon
99	$\{secureInvariant\}$ Delete $\{secureInvariant\}$	239	T=T	382.6	377.7	4.9	ProCon
100	$\{ACConstraint\}$ Delete $\{ACConstraint\}$	293	T=T	426.4	420.8	5.6	ProCon
101	$\{noSharing\}$ DeleteAlt $\{noSharing\}$	83	T=T	0.4	0.4	0.0	SeekSat
102	$\{someAccess\}$ DeleteAlt $\{someAccess\}$	101	F=F	583.7	583.7	0.0	SeekSat
103	$\{noTwoRights\}$ DeleteAlt $\{noTwoRights\}$	83	T=T	0.4	0.4	0.0	SeekSat
104	$\{noMultiSession\}$ DeleteAlt $\{noMultiSession\}$	120	T=T	0.7	0.6	0.1	SeekSat
105	$\{sessionDeletable\}$ DeleteAlt $\{sessionDeletable\}$	205	T=T	2.2	2.2	0.0	SeekSat
106	$\{decidable\}$ DeleteAlt $\{decidable\}$	266	T=T	3.4	3.3	0.1	SeekSat
107	$\{sessionStates\}$ DeleteAlt $\{sessionStates\}$	20	T=T	0.6	0.5	0.1	ProCon
108	$\{secure\}$ DeleteAlt $\{secure\}$	118	T=T	8.8	8.2	0.6	ProCon
109	$\{secureInvariant\}$ DeleteAlt $\{secureInvariant\}$	245	T=T	531.3	518.2	13.1	ProCon
110	$\{ACConstraint\}$ DeleteAlt $\{ACConstraint\}$	223	T \neq E	-	1152.7	0.1	SeekSat
111	$\{noSharing\}$ AccessControl $\{noSharing\}$	147	T=T	0.4	0.4	0.0	SeekSat
112	$\{someAccess\}$ AccessControl $\{someAccess\}$	171	F=F	128.9	128.9	0.0	SeekSat
113	$\{noTwoRights\}$ AccessControl $\{noTwoRights\}$	165	T=T	0.4	0.4	0.0	SeekSat
114	$\{noMultiSession\}$ AccessControl $\{noMultiSession\}$	238	T=T	0.7	0.7	0.0	SeekSat
115	$\{sessionDeletable\}$ AccessControl $\{sessionDeletable\}$	579	T=T	2.5	2.2	0.3	SeekSat
116	$\{decidable\}$ AccessControl $\{decidable\}$	784	T=T	4.5	3.8	0.6	SeekSat
117	$\{sessionStates\}$ AccessControl $\{sessionStates\}$	228	T=T	0.7	0.6	0.1	ProCon
118	$\{secure\}$ AccessControl $\{secure\}$	339	F=F	7.5	3.8	3.7	ProCon
119	$\{secureInvariant\}$ AccessControl $\{secureInvariant\}$	786	T=T	454.1	377.4	76.7	ProCon
120	$\{ACConstraint\}$ AccessControl $\{ACConstraint\}$	984	T=T	475.2	428.0	47.2	ProCon
μ	arithmetic mean	131.9		28.3	35.8	2.8	
$\mu_{1/2}$	median	89		0.2	0.0	0.0	

Table 7.10: Access control case study: results 61-120

a comparison between the decision times t_{\models} of PROCON||SeekSat (see Table 7.9 and 7.10) and selected first-order theorem provers and satisfiability solvers (see Table 7.11). In contrast to the car platooning case study, all off-the-shelf tools combined (META) can only decide 77 of 120 specifications, given 5 minutes of time per specification (INTEL T5600, 1.83GHz). Again, VAMPIRE and DARWIN are able to solve different test cases.

The differences to the car platooning results may be explained by the fact that the car platooning case study only consists of transformation rules, while the access control case study considers more complex programs. This explanation is supported by the higher mean complexity of the weakest pre-conditions.

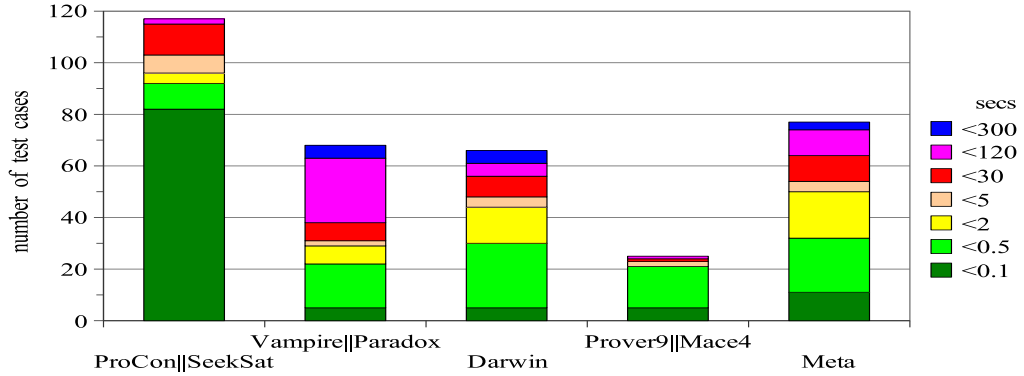


Figure 7.8: Access control: comparison of deciders (“ t_{\models} ”)

7.5 Summary and discussion

We have successfully evaluated our approach to the verification of graph transformation systems and programs by modeling and verifying selected aspects of three real-world systems: a railroad control, a platoon maneuver protocol, and an access control for computer systems. In each of these cases, we were able to formalize important properties that gave rise to a number of graphical specifications. For instance, in case of the access control system, we were able to express the security property “Every user logged into a system has the appropriate access right”, which is an invariant of the system only in conjunction with additional conditions such as the property “Never is a session shared between two users”. Of 330 considered test specifications, 327 can be automatically decided using our implementation ENFORCE. In two cases, the alleged counterexample can not be retraced. In the remaining case, the verification fails as an approximated invariant is not weak enough

#	result	t_{\perp} decoder	result	t_{\perp} decoder	result	t_{\perp} decoder	#	result	t_{\perp} decoder	result	t_{\perp} decoder	result	t_{\perp} decoder
1	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	61	T=T	41.1 Vampire	T=T	4.2 DarwinProver	T \neq E	--
2	T=T	0.1 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	62	T=T	97.4 Vampire	T=T	1.6 DarwinProver	T \neq E	--
3	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	63	T=T	2.6 Vampire	T=T	6.4 DarwinProver	T \neq E	--
4	T=T	33.6 Vampire	T=T	0.5 DarwinProver	T \neq E	--	64	T \neq E	--	T=T	29.5 DarwinProver	T \neq E	--
5	T \neq E	--	T \neq E	--	T \neq E	--	65	T \neq E	--	T \neq E	--	T \neq E	--
6	T \neq E	--	T \neq E	--	T \neq E	--	66	T \neq E	--	T \neq E	--	T \neq E	--
7	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9	67	T=T	34.6 Vampire	T=T	0.3 DarwinProver	F=T	0.3 Prover9
8	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	68	F \neq E	--	F=F	12.2 DarwinSolver	F \neq E	--
9	T=T	98.4 Vampire	T \neq E	--	T \neq E	--	69	T \neq E	--	T \neq E	--	T \neq E	--
10	T \neq E	--	T \neq E	--	T \neq E	--	70	T \neq E	--	T \neq E	--	T \neq E	--
11	T=T	0.1 Vampire	T=T	0.4 DarwinProver	T=T	2.5 Prover9	71	T=T	95.5 Vampire	T=T	1.2 DarwinProver	T \neq E	--
12	T=T	166.8 Vampire	T=T	0.4 DarwinProver	T \neq E	--	72	T=T	94.4 Vampire	T=T	0.9 DarwinProver	T \neq E	--
13	T=T	0.1 Vampire	T=T	0.3 DarwinSolver	T=T	0.2 Prover9	73	F=F	0.5 Paradox	F=F	0.9 DarwinSolver	F=F	16.1 Mace4
14	T=T	157.3 Vampire	T=T	0.4 DarwinProver	T \neq E	--	74	T=T	108.8 Vampire	T=T	5.7 DarwinProver	T \neq E	--
15	T \neq E	--	T \neq E	--	T \neq E	--	75	T \neq E	--	T \neq E	--	T \neq E	--
16	T \neq E	--	T \neq E	--	T \neq E	--	76	T \neq E	--	T \neq E	--	T \neq E	--
17	T=T	6.5 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	77	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9
18	T=T	41.0 Vampire	T=T	2.5 DarwinProver	T \neq E	--	78	T=T	22.3 Vampire	T \neq E	--	T \neq E	--
19	T \neq E	--	T \neq E	--	T \neq E	--	79	T \neq E	--	T \neq E	--	T \neq E	--
20	T \neq E	--	T \neq E	--	T \neq E	--	80	T \neq E	--	T \neq E	--	T \neq E	--
21	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	81	T=T	93.3 Vampire	T=T	1.3 DarwinProver	T \neq E	--
22	F=F	0.3 Paradox	F=F	0.2 DarwinSolver	F=F	0.2 Mace4	82	T=T	91.4 Vampire	T=T	0.5 DarwinProver	T \neq E	--
23	T=T	0.1 Vampire	T=T	0.3 DarwinSolver	T=T	0.2 Prover9	83	F=F	1.0 Paradox	F=F	2.5 DarwinSolver	F \neq E	--
24	T=T	23.3 Vampire	T=T	0.4 DarwinProver	T \neq E	--	84	T \neq E	--	T=T	12.1 DarwinProver	T \neq E	--
25	T \neq E	--	T \neq E	--	T \neq E	--	85	T \neq E	--	T \neq E	--	T \neq E	--
26	F=F	0.9 Paradox	F \neq E	--	F \neq E	--	86	T \neq E	--	T \neq E	--	T \neq E	--
27	T=T	0.4 Vampire	T=T	0.2 DarwinProver	T=T	0.1 Prover9	87	T=T	0.4 Vampire	T=T	0.2 DarwinProver	T=T	0.3 Prover9
28	T=T	0.2 Vampire	T=T	1.3 DarwinProver	T \neq E	--	88	F=F	61.4 Paradox	F \neq E	--	F \neq E	--
29	T \neq E	--	T \neq E	--	T \neq E	--	89	T \neq E	--	T \neq E	--	T \neq E	--
30	T \neq E	--	T \neq E	--	T \neq E	--	90	T \neq E	--	T \neq E	--	T \neq E	--
31	T=T	0.3 Vampire	T=T	0.2 DarwinProver	T=T	0.1 Prover9	91	T=T	97.5 Vampire	T=T	0.3 DarwinProver	T \neq E	--
32	T=T	0.3 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	92	F=F	0.6 Paradox	F=F	33.7 DarwinSolver	F \neq E	--
33	T=T	0.1 Vampire	T=T	0.3 DarwinProver	T=T	0.2 Prover9	93	T=T	95.5 Vampire	T=T	0.2 DarwinProver	T \neq E	--
34	T=T	21.5 Vampire	T=T	0.3 DarwinProver	T \neq E	--	94	T=T	168.5 Vampire	T=T	1.0 DarwinProver	T \neq E	--
35	T \neq E	--	T \neq E	--	T \neq E	--	95	T \neq E	--	T \neq E	--	T \neq E	--
36	T \neq E	--	T \neq E	--	T \neq E	--	96	T \neq E	--	T \neq E	--	T \neq E	--
37	T=T	0.6 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9	97	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.1 Prover9
38	T=T	0.3 Vampire	T=T	2.0 DarwinProver	T \neq E	--	98	T=T	41.1 Vampire	T=T	150.5 DarwinProver	T \neq E	--
39	T=T	108.8 Vampire	T \neq E	--	T \neq E	--	99	T \neq E	--	T \neq E	--	T \neq E	--
40	T \neq E	--	T \neq E	--	T \neq E	--	100	T \neq E	--	T \neq E	--	T \neq E	--
41	T=T	169.4 Vampire	T=T	133.1 DarwinProver	T \neq E	--	101	T=T	93.5 Vampire	T=T	0.5 DarwinProver	T \neq E	--
42	T=T	168.5 Vampire	T=T	206.9 DarwinProver	T \neq E	--	102	F=F	7.5 Paradox	F \neq E	--	F \neq E	--
43	F=F	1.7 Paradox	F=F	35.7 DarwinSolver	F \neq E	--	103	T=T	82.0 Vampire	T=T	0.3 DarwinProver	T \neq E	--
44	T \neq E	--	T=T	29.5 DarwinProver	T \neq E	--	104	T=T	110.6 Vampire	T=T	1.6 DarwinProver	T \neq E	--
45	T \neq E	--	T \neq E	--	T \neq E	--	105	T \neq E	--	T=T	49.1 DarwinProver	T \neq E	--
46	T \neq E	--	T \neq E	--	T \neq E	--	106	T \neq E	--	T=T	86.9 DarwinProver	T \neq E	--
47	T \neq E	--	T=T	23.4 DarwinProver	T \neq E	--	107	T=T	0.2 Vampire	T=T	0.2 DarwinProver	T=T	0.2 Prover9
48	T=T	31.7 Vampire	T \neq E	--	T \neq E	--	108	T=T	57.5 Vampire	T \neq E	--	T \neq E	--
49	T \neq E	--	T \neq E	--	T \neq E	--	109	T \neq E	--	T \neq E	--	T \neq E	--
50	T \neq E	--	T \neq E	--	T \neq E	--	110	T \neq E	--	T \neq E	--	T \neq E	--
51	T=T	40.8 Vampire	T=T	0.5 DarwinProver	T=T	38.8 Prover9	111	T \neq E	--	T \neq E	--	T \neq E	--
52	T=T	87.2 Vampire	T=T	0.5 DarwinProver	T \neq E	--	112	F=F	4.3 Paradox	F=F	27.5 DarwinSolver	F \neq E	--
53	F=F	0.3 Paradox	F=F	0.3 DarwinSolver	F=F	2.0 Mace4	113	T=T	112.9 Vampire	T=T	207.9 DarwinProver	T \neq E	--
54	T=T	99.5 Vampire	T=T	1.6 DarwinProver	T \neq E	--	114	T \neq E	--	T=T	112.6 DarwinProver	T \neq E	--
55	T \neq E	--	T \neq E	--	T \neq E	--	115	T \neq E	--	T \neq E	--	T \neq E	--
56	T \neq E	--	T \neq E	--	T \neq E	--	116	T \neq E	--	T \neq E	--	T \neq E	--
57	T=T	0.2 Vampire	T=T	0.1 DarwinProver	T=T	0.2 Prover9	117	T \neq E	--	T=T	137.2 DarwinProver	T \neq E	--
58	F=F	0.5 Paradox	F \neq E	--	F \neq E	--	118	F \neq E	--	F \neq E	--	F \neq E	--
59	F=F	14.2 Paradox	F \neq E	--	F \neq E	--	119	T \neq E	--	T \neq E	--	T \neq E	--
60	F=F	8.2 Paradox	F \neq E	--	F \neq E	--	120	T \neq E	--	T \neq E	--	T \neq E	--
							μ	42.7		20.2		2.5	
							$\mu_{1/2}$	11.2		0.5		0.2	

Table 7.11: Access control: decision times of first-order provers and solvers

to carry out the proof, which may be a starting point for further research on this topic.

On the other hand, the railroad control case study, originally considered in [Pen04, HP05], and the external car platooning case study [Bau06, HESV91] are both fully verified. Also, the verification of the access control system, to which the 3 unverified program specifications belong, can be seen as successful, as we were able to show the invariance of the conjunction *ACConstraint* with respect to *AccessControl*.

We conducted a comparison of our developed prover and solver components with pairs of existing first-order theorem provers and satisfiability solvers. *SeekSat* and *PROCON* outperform any tool pair both in terms of performance and completeness, including the hypothetical *META* representing the minimal response time of the off-the-self tools. This tendency is obvious for the railroad and access control case studies, but even for the car platooning case study, a translation of the implication problem from conditions to first-order graph formulas is not on a par with the developed components.

The good performance of *PROCON* and *SeekSat* may be explained as follows: Both components are high-level, that is, for the most part, they are structure-independent and rely only on a small number of structure-specific operations such as the pushout operation. For these operations, it is asserted that the output (morphisms and objects) always belong to the given category. In this sense, *PROCON* and *SeekSat* become structure-specific in a constructive way, once the necessary operations are provided. In contrast, theorem prover and satisfiability solver for general first-order logic necessarily consider arbitrary structures and have to be restricted by a set of axioms to a target structure, which adds to the complexity of the problem. Even worse, translations of problem instances originally belonging to a decidable fragment of conditions may be outside of a decidable fragment of first-order logic, if the axioms themselves do not belong to a decidable fragment.

Furthermore, an algorithm on conditions can and should use the fact that conditions make quantifications and statements in bulks, that is, a quantifier may introduce a number of elements. In this sense, conditions may have a lower logical complexity when compared to their translations in first-order logic, see Example 3.32 on page 38.

Finally, the transformation of graph conditions into graph formulas itself may add to the problem: distinct elements in \mathcal{M} -satisfiable conditions are implicitly mapped onto distinct elements in the test object. For formulas, it remains open if the values of variables are equal or distinct, unless it is explicitly stated. If the nodes and edges of a graph condition are not distinct by their labels, inequations have to be introduced during the translation.

8. Conclusion

We have researched, implemented and evaluated a method that aims at verifying or refuting the correctness of graph transformation systems and programs with respect to pre- and postconditions, as far as possible. While the correctness problem is undecidable for the considered program specifications, almost every test specification of the case studies considered in the evaluation of our approach can be automatically decided. No other approach dedicated to graph transformation is ready to reproduce the results of our implementation, mostly due to limitations on the kind of specifications that can be handled, see Section 5.5. Only the approach of Strecker [Str08] is capable of encoding all considered test specifications, however it is not automatized. Therefore, our research constitutes a valuable method for the development of correct graph transformation systems and programs.

We showed that our specifications consisting of graph conditions and graph programs with interface capture important system requirements, for instance, the security of an access control for computer systems. Conditions are the basis for solving the correctness problem of these specifications. By constructing weakest preconditions, we reduce the correctness problem to the implication problem of conditions. Despite some difficulties, the evaluation shows that conditions do not only provide an intuitive formalism for first-order structural properties, but are also suited to infer knowledge about the behavior of graph transformation systems and programs. Moreover, the introduced and investigated language of programs with interface is an important generalization of existing programs on graph transformation rules [HP01, PS04]. The practical relevance of programs with interface is demonstrated by the fact that one of the key components in our approach, the satisfiability solver **SeekSat**, is such a program.

8.1 Results

Our main contributions are the following:

1. We investigated nested conditions [Pen04, HP05] and showed that nested graph conditions and first-order logic on graphs are expressively equivalent [HP06, HP09]. The second part of the proof is similar to translations between first-order logic and predicates on edge-labeled graphs with single edges [Ren04a].
2. We introduced programs with interface to describe structural transformations within a category of objects. Programs with interface allow explicit control over the selection of elements and are capable of handing over selections between computations steps, which can be used to restrict the execution of programs to a previously selected context [Pen08a].
3. We defined weakest liberal preconditions of programs with interface and (nested) postconditions. We presented a construction for weakest liberal preconditions [HPR06, HP09] and considered algorithms for the construction of (weakest) invariants.
4. We related the decidability of the implication, the tautology and the satisfiability problem of conditions. We lifted the undecidability of the satisfiability problem of first-order logic on graphs [Cou90] to the satisfiability problem of graph conditions [HP09]. Following a dual approach, we investigated a sound satisfiability solver for conditions that is complete for a class of weak adhesive HLR categories [Pen08a]. We investigated a fragment of conditions for which the solver terminates and decides. On the other hand, we presented a calculus for proving conditions and showed its soundness [Pen08b].
5. We evaluated our approach and the implementations of the aforementioned transformations and algorithms by modeling and successfully verifying three case studies: a railroad control [Pen04, HP05], an access control for computer systems [HPR06], and, as an external example, a car platoon maneuver protocol [Bau06, HESV91]. We showed that our implementation is able to decide 327 of 330 considered test specifications. We also showed that the developed prover and solver for conditions are superior in terms of performance and coverage to existing first-order theorem provers and satisfiability solvers applied onto straightforward translations of graph conditions into first-order graph logic.

We have successfully presented our work on renowned international conferences in the research area of graph transformation [HPR06, HP06, Pen08a, Pen08b] and published our results in established journals in theoretical computer science [EEHP06, HP09]. Our work revived the interest in graph-based

conditions, and was picked up, for instance, in [OEP08, Ore08].

The most important take-home points are:

First, algorithms based on operations that preserve a given target structure \mathcal{C} can be expected to outperform algorithms working on a more general structure restricted to \mathcal{C} by subsequent assertions. For instance, a satisfiability algorithm on graph-preserving operations will only consider graphs. In contrast, a first-order logic based satisfiability solver used with graph axioms will still generate all kinds of intermediate structures, many of which may be much later discarded as they violate the graph axioms. If performance is the ultimate criterion and all other options have been exploited, the idea of basing computations onto structure-preserving operations seems to be a viable strategy to improve results.

Second, approximation of invariants *really is* the hardest part of program verification. All the presented algorithms are refinement based, therefore rely on deciding possibly multiple instances of the implication problem.

We conclude that category-theoretic approaches for describing structural transformations, and graph transformation in particular, are a suitable basis for modeling and verifying discrete aspects of real-world systems.

8.2 Open problems and future work

For every question answered there are new ones to ask. We distinguish between open questions concerning our work, and future work intended to extend the coverage of the presented approach. The following topics remain open:

1. A proof of the completeness of our calculus for proving conditions, see Section 6.3: While we conjecture the completeness of our calculus, a formal proof has yet to be devised.
2. A formal definition of a proof format and an independent implementation of a proof checker: A proof format should be investigated that documents the essential deductions leading to the empty clause. A proof checker based on PROCON's calculus should be implemented that is capable to retrace such proofs.

To increase the attractiveness of the presented approach to the verification of graphical program specifications, we suggest to research the following topics:

3. An extension of the approach to typed attributed graphs [EEPT06]: As a theorem prover for attributed graph conditions has been already investigated in [Ore08], this goal has come within reach.
 4. An extension of the notion of conditions to capture monadic second-order properties, or at least path properties in the sense of a reflexive, transitive closure: While monadic second order logic is more expressive than first-order logic, the model checking problem remains decidable, although it becomes more complex.
 5. An investigation of the logic that is tractable by weakest preconditions: In terms of expressiveness, a suitable logic such as dynamic logic [HKT84] may extend beyond the inflexible concept of program specifications.
 6. An evaluation of our method with respect to *liveness* properties: Liveness properties are dual to *safety* properties such as correctness. The dual of weakest liberal preconditions ensuring correctness are weakest preconditions ensuring the existence of results, as investigated in [HPR06] for programs over double pushout transformation rules.
 7. A lifting of programs with interface from morphism to conditions with the intent to describe PROCON and other transformation on conditions as such programs: Programs with interface turned out to be a suitable basis for the satisfiability solver *SeekSat*, as they are close to an implementation, yet allow elegant proofs of algorithmic properties.
-

A. Partial monomorphisms

For the definition of programs with interface, we require partial monomorphisms. Given a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$, these partial monomorphisms can be introduced as spans of \mathcal{M} -morphisms without the need of further axioms. There is a composition of partial monomorphisms, based on pullback construction, that is associative and for which identity morphisms are neutral elements.

Definition A.1 (partial monomorphisms). For a given weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$, a *partial monomorphism* $p = \langle a, b \rangle$ is a 2-tuple of \mathcal{M} -morphisms $a, b \in \mathcal{M}$ with $\text{dom}(a) = \text{dom}(b)$. The *domain* of p is codomain of a , that is, $\text{dom}(p) = \text{codom}(a)$, the *codomain* of p is the codomain of b , that is, $\text{codom}(p) = \text{codom}(b)$, and the *interface* of p refers to the common domain of a and b , that is, $\text{iface}(p) = \text{dom}(a) = \text{dom}(b)$. We write $p: \text{dom}(p) \hookrightarrow \text{codom}(p)$ to denote a partial monomorphism. The set of all partial monomorphisms is denoted by \mathcal{P} , while $\mathcal{P}(A, B)$ denotes the class of all partial monomorphisms with domain A and codomain B . Two partial monomorphism $p, q: A \hookrightarrow B$ are *commutative*, denoted by $p = q$, if there is a (total) isomorphism $\text{iface}(p) \leftrightarrow \text{iface}(q)$ such that the resulting triangles commute. For a partial monomorphism $p = \langle a, b \rangle$, $\langle b, a \rangle$ is the *inverse* partial monomorphism, denoted by p^{-1} . A partial monomorphism $p = \langle a, b \rangle$ is said to be in \mathcal{M} , if a is an isomorphism.

Every \mathcal{M} -morphism represents a partial monomorphism.

Fact A.2. $\mathcal{M} \subseteq \mathcal{P}$.

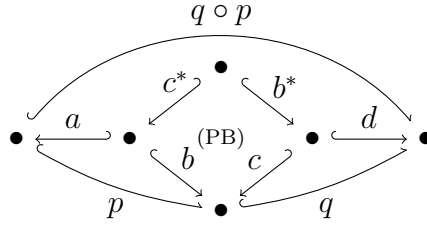
Proof. For every $m \in \mathcal{M}$, $m \circ \text{id}_{\text{dom}(m)} = m$ is in \mathcal{M} and $\langle \text{id}_{\text{dom}(m)}, m \rangle \in \mathcal{P}$. \square

Partial monomorphism are closed under composition, that is, $A \hookrightarrow B$ and $B \hookrightarrow C$ can be composed to $A \hookrightarrow C$. No further axioms are required.

Fact A.3 (closure under composition). For every objects $A, B, C \in \mathcal{O}$, there is a binary operation $\mathcal{P}(A, B) \times \mathcal{P}(B, C) \rightarrow \mathcal{P}(A, C)$ called *composition of partial monomorphisms*, denoted by $q \circ p$ for morphisms p, q with $\text{codom}(p) = \text{dom}(q)$ and the following holds:

- *associativity*: for every morphisms p, q, r with $\text{codom}(p) = \text{dom}(q)$ and $\text{codom}(q) = \text{dom}(r)$, we have $r \circ (q \circ p) = (r \circ q) \circ p$.
- *identity*: for every object A , there exists a morphism $\text{id}_A: A \rightarrow A$, called the *identity morphism for A* , such that for every partial monomorphism p , we have $\text{id}_{\text{codom}(p)} \circ p = p = p \circ \text{id}_{\text{dom}(p)}$.

Proof. The composition is obtained by pullback construction: For $p = \langle a, b \rangle$ and $q = \langle c, d \rangle$ with $\text{codom}(p) = \text{dom}(q)$, construct $\langle c^*, b^* \rangle$ as the pullback of $\langle b, c \rangle$. Then let be $q \circ p = \langle a \circ c^*, d \circ b^* \rangle$. The result is unique up to isomorphism.



Associativity: Follows from the uniqueness of pullbacks.

Identity: Follows from the identity axiom in Def. 2.1 and the fact that $\langle \text{id}_{\text{dom}(b)}, b \rangle$ is the pullback of $\langle b, \text{id}_{\text{codom}(b)} \rangle$: We have $\text{id}_{\text{codom}(p)} \circ p = \text{id}_{\text{codom}(b)} \circ \langle a, b \rangle = \langle a \circ \text{id}_{\text{dom}(b)}, \text{id}_{\text{codom}(b)} \circ b \rangle = \langle a, b \rangle = p$. Analogously, for the composition $\langle a, b \rangle \circ \langle \text{id}_{\text{codom}(a)}, \text{id}_{\text{dom}(a)} \rangle$, one yields $\langle a, \text{id}_{\text{dom}(a)} \rangle$ as the pullback of $\langle \text{id}_{\text{codom}(a)}, a \rangle$. Then let be $p \circ \text{id}_{\text{dom}(p)} = \langle a, b \rangle \circ \text{id}_{\text{codom}(a)} = \langle \text{id}_{\text{codom}(a)} \circ a, b \circ \text{id}_{\text{dom}(a)} \rangle = \langle a, b \rangle = p$. \square

Equality of partial monomorphisms is a reflexive property.

Fact A.4 (reflexivity). For $p, q \in \mathcal{P}$, we have $p = q$ implies $q = p$.

Proof. Let $p = \langle a, b \rangle$ and $q = \langle c, d \rangle$. We have $\langle a, b \rangle = \langle c, d \rangle$ if and only if there is an isomorphism $m: \text{iface}(\langle a, b \rangle) \leftrightarrow \text{iface}(\langle c, d \rangle)$ such that $a = c \circ m$ and $b = d \circ m$ if and only if there is an isomorphism $m^{-1}: \text{iface}(\langle c, d \rangle) \leftrightarrow \text{iface}(\langle a, b \rangle)$ such that $c = a \circ m^{-1}$ and $d = b \circ m^{-1}$ if and only if $\langle c, d \rangle = \langle a, b \rangle$. \square

The inverse of composition q after p equals the composition of the inverse of p after the inverse of q .

Fact A.5. For $p, q \in \mathcal{P}$, we have $(q \circ p)^{-1} = p^{-1} \circ q^{-1}$.

Proof. Let $p = \langle a, b \rangle$ and $q = \langle c, d \rangle$. Then $(\langle c, d \rangle \circ \langle a, b \rangle)^{-1} = \langle d \circ b^*, a \circ c^* \rangle = \langle b, a \rangle \circ \langle d, c \rangle = \langle a, b \rangle^{-1} \circ \langle c, d \rangle^{-1}$. \square

Two partial monomorphisms are equal if and only if their inverses are equal.

Fact A.6. For $p, q \in \mathcal{P}$, we have $p = q$ if and only if $p^{-1} = q^{-1}$.

Proof. Let $p = \langle a, b \rangle$ and $q = \langle c, d \rangle$. We have $\langle a, b \rangle = \langle c, d \rangle$ if and only if there is an isomorphism $m: \text{iface}(\langle a, b \rangle) \leftrightarrow \text{iface}(\langle c, d \rangle)$ such that $a = c \circ m$ and $b = d \circ m$ if and only if there is an isomorphism $m: \text{iface}(\langle a, b \rangle^{-1}) \leftrightarrow \text{iface}(\langle d, c \rangle^{-1})$ such that $b = d \circ m$ and $a = c \circ m$ if and only if $\langle b, a \rangle = \langle d, c \rangle$. \square

Every \mathcal{M} -morphism can be neutralized by its inverse via composition.

Fact A.7 (special cases of composition). $\langle b, c \rangle \circ \langle a, b \rangle = \langle a, c \rangle$. For $m \in \mathcal{M}$, we have $m^{-1} \circ m = \text{id}_{\text{dom}(m)}$.

Proof. For the composition $\langle b, c \rangle \circ \langle a, b \rangle$, one yields $\langle \text{id}, \text{id} \rangle$ with $\text{dom}(\text{id}) = \text{codom}(\text{id}) = \text{codom}(a) = \text{dom}(b) = \text{codom}(c)$ as the pullback of $\langle b, b \rangle$. Then, we have $\langle b, c \rangle \circ \langle a, b \rangle = \langle a \circ \text{id}, c \circ \text{id} \rangle = \langle a, c \rangle$. Special case $m \in \mathcal{M}$: $m^{-1} \circ m = \langle \text{id}, m \rangle^{-1} \circ \langle \text{id}, m \rangle = \langle m, \text{id} \rangle \circ \langle \text{id}, m \rangle = \langle \text{id}, \text{id} \rangle = \text{id}$. \square

The following fact states the requirements such that a composition of two partial monomorphisms yields an \mathcal{M} -morphism.

Fact A.8 (\mathcal{M} -decomposition). For all partial monomorphisms $\langle a, b \rangle, \langle c, d \rangle$ with $\text{codom}(b) = \text{dom}(c)$ we have: $\langle c, d \rangle \circ \langle a, b \rangle \in \mathcal{M}$ if and only if a is an isomorphism, $\langle a, b \rangle \in \mathcal{M}$ and there is an \mathcal{M} -morphism $m: \text{iface}(\langle a, b \rangle) \hookrightarrow \text{iface}(\langle c, d \rangle)$ such that $c \circ m = b$.

Proof. If. First, we show $\langle c, d \rangle \circ \langle a, b \rangle = \langle a, d \circ m \rangle$. Construct the composition $\langle c, d \rangle \circ \langle a, b \rangle$, that is, construct the pullback $\langle c^*, b^* \rangle$ of $\langle b, c \rangle$. We have $c \circ b^* = b \circ c^* = c \circ m \circ c^*$, which implies $b^* = m \circ c^*$ (Def. 2.1). For the morphism c^* , we have $\text{dom}(c^*) = \text{iface}(\langle c, d \rangle \circ \langle a, b \rangle)$ and $\text{codom}(c^*) = \text{dom}(m) = \text{codom}(a) = \text{iface}(\langle a, d \circ m \rangle)$. Moreover, there exists an inverse morphism $(c^*)^{-1}$: Construct the pullback of $\langle m, m \rangle$ and yield $\langle \text{id}_{\text{dom}(m)}, \text{id}_{\text{dom}(m)} \rangle$. As $\text{id}_{\text{dom}(m)} \circ b = \text{id}_{\text{dom}(m)} \circ m \circ c$, the universal property of pullback guarantees the existence of a morphism $(c^*)^{-1}: \text{dom}(m) \rightarrow \text{dom}(c^*)$ such that $c \circ (c^*)^{-1} = \text{id}_{\text{codom}(c^*)}$ and $(c^*)^{-1} \circ c = \text{id}_{\text{dom}(c^*)}$. Finally, $\langle c, d \rangle \circ \langle a, b \rangle = \langle a, d \circ m \rangle$ is in \mathcal{M} as a is an isomorphism.

Only if. Conversely, assume $\langle c, d \rangle \circ \langle a, b \rangle = \langle a \circ c^*, d \circ b^* \rangle$ is in \mathcal{M} , then $a \circ c^*$ is an isomorphism (Def. A.1). According to Fact 2.6, a is an isomorphism and c^* is an isomorphism. Then $\langle a, b \rangle$ is in \mathcal{M} (Def. A.1). Moreover, there exists a morphism $m: \text{iface}(\langle a, b \rangle) \hookrightarrow \text{iface}(\langle c, d \rangle)$ which is $b^* \circ (c^*)^{-1}$. As c^* is an isomorphism and pullback squares are commutative, $b = c \circ m$. \square

B. Proof

In this section we prove the soundness of the transformation of a program with interface into a **Fix**-free program.

Proof of Theorem 4.16. We show the equivalence for every equation of the construction. Obviously, $\text{Fix}(\mathbf{P}) \equiv \text{Fix}(\mathbf{P}, \text{id}_C)$ for programs \mathbf{P} with interface C .

First, we show $\text{Fix}(\text{Skip}, p) \equiv \text{Uns}(p)$ if $p \in \mathcal{M}$, and $\text{Fix}(\text{Skip}, p) \equiv \text{Abort}$.

Case $p \in \mathcal{M}$.

$$\begin{aligned}
 & \llbracket \text{Fix}(\text{Skip}, p) \rrbracket \\
 = & \{ \langle m, m^* \circ (m_{\text{Skip}} \circ p), p^{-1} \rangle \mid \\
 & \quad \langle m, m^*, m_{\text{Skip}} \rangle \in \llbracket \text{Skip} \rrbracket \text{ and } (m_{\text{Skip}} \circ p) \in \mathcal{M} \} & (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
 = & \{ \langle m, m \circ (\text{id} \circ p), p^{-1} \rangle \mid (\text{id} \circ p) \in \mathcal{M} \} & (\text{Def. } \llbracket \text{Skip} \rrbracket) \\
 = & \{ \langle m, m \circ p, p^{-1} \rangle \mid p \in \mathcal{M} \} & (\text{Fact A.3}) \\
 = & \{ \langle m, m \circ p, p^{-1} \rangle \mid \text{true} \} & (p \in \mathcal{M}) \\
 = & \llbracket \text{Uns}(p) \rrbracket & (\llbracket \text{Uns} \rrbracket)
 \end{aligned}$$

Case $p \notin \mathcal{M}$. Then

$$\begin{aligned}
 & \llbracket \text{Fix}(\text{Skip}, p) \rrbracket \\
 = & \{ \langle m, m^* \circ (m_{\text{Skip}} \circ p), p^{-1} \rangle \mid \\
 & \quad \langle m, m^*, m_{\text{Skip}} \rangle \in \llbracket \text{Skip} \rrbracket \text{ and } (m_{\text{Skip}} \circ p) \in \mathcal{M} \} & (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
 = & \{ \langle m, m \circ (\text{id} \circ p), p^{-1} \rangle \mid (\text{id} \circ p) \in \mathcal{M} \} & (\text{Def. } \llbracket \text{Skip} \rrbracket) \\
 = & \{ \langle m, m \circ p, p^{-1} \rangle \mid p \in \mathcal{M} \} & (\text{Fact A.3}) \\
 = & \{ \langle m, m \circ p, p^{-1} \rangle \mid \text{false} \} & (p \notin \mathcal{M}) \\
 = & \emptyset & (\{\dots \mid \dots\}) \\
 = & \llbracket \text{Abort} \rrbracket & (\text{Def. } \llbracket \text{Abort} \rrbracket)
 \end{aligned}$$

Second, $\text{Fix}(\mathbf{P}, p) \equiv \text{Fix}((\mathbf{P}; \text{Skip}), p)$ by Fact 4.12.

Finally, we prove the remaining equations of the construction by induction over structure of conditions \mathbf{P} in a sequential composition $(\mathbf{P}; \mathbf{Q})$:

Basis.

Case Sel.

$$\llbracket \text{Fix}((\text{Sel}(x, c); \mathbf{R}), p) \rrbracket$$

$$\begin{aligned}
&= \{ \langle m, m^* \circ (m_{\text{Sel};\mathbf{R}} \circ p), p^{-1} \rangle \mid \langle m, m^*, m_{\text{Sel};\mathbf{R}} \rangle \in \llbracket (\text{Sel}(x, c); \mathbf{R}) \rrbracket \\
&\quad \text{and } (m_{\text{Sel};\mathbf{R}} \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \{ \langle m, m^* \circ ((m_{\mathbf{R}} \circ x) \circ p), p^{-1} \rangle \mid \langle m, m', x \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \\
&\quad \text{and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } ((m_{\mathbf{R}} \circ x) \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (x \circ p)), p^{-1} \circ (x^{-1} \circ x) \rangle \mid \\
&\quad \langle m, m', x \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \text{ and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \\
&\quad \text{and } (m_{\mathbf{R}} \circ (x \circ p)) \in \mathcal{M} \} \quad (\text{Fact A.7}) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \circ x \rangle \mid \\
&\quad \langle m, m', x \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \text{ and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \\
&\quad \text{and } (m_{\mathbf{R}} \circ (x \circ p)) \in \mathcal{M} \} \quad (\text{Fact A.5}) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \circ x \rangle \mid \\
&\quad \langle m, m', x \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \text{ and } \\
&\quad \langle m', (m^* \circ m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \rangle \in \\
&\quad \{ \langle m', m^* \circ (m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \rangle \mid \\
&\quad \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } (m_{\mathbf{R}} \circ (x \circ p)) \in \mathcal{M} \} \quad (\{ \dots \mid \dots \}) \\
&= \{ \langle m, m^* \circ (m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \circ x \rangle \mid \\
&\quad \langle m, m', x \rangle \in \llbracket \text{Sel}(x, c) \rrbracket \text{ and } \\
&\quad \langle m', (m^* \circ m_{\mathbf{R}} \circ (x \circ p)), (x \circ p)^{-1} \rangle \in \llbracket \text{Fix}(\mathbf{R}, (x \circ p)) \rrbracket \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \llbracket (\text{Sel}(x, c); \text{Fix}(\mathbf{R}, (x \circ p))) \rrbracket \quad (\text{Def. } \llbracket (_; _) \rrbracket)
\end{aligned}$$

Case Del.

$$\begin{aligned}
&\llbracket \text{Fix}((\text{Del}(l); \mathbf{R}), p) \rrbracket \\
&= \{ \langle m, m^* \circ (m_{\text{Del};\mathbf{R}} \circ p), p^{-1} \rangle \mid \\
&\quad \langle m, m^*, m_{\text{Del};\mathbf{R}} \rangle \in \llbracket (\text{Del}(l); \mathbf{R}) \rrbracket \text{ and } (m_{\text{Del};\mathbf{R}} \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \{ \langle m, m^* \circ ((m_{\mathbf{R}} \circ l^{-1}) \circ p), p^{-1} \rangle \mid \langle m, m', l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \\
&\quad \text{and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } ((m_{\mathbf{R}} \circ l^{-1}) \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (l^{-1} \circ p)), p^{-1} \circ ((l^{-1})^{-1} \circ l^{-1}) \rangle \mid \\
&\quad \langle m, m', l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \\
&\quad \text{and } (m_{\mathbf{R}} \circ (l^{-1} \circ p)) \in \mathcal{M} \} \quad (*, \text{ see below}) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \circ l^{-1} \rangle \mid \\
&\quad \langle m, m', l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \\
&\quad \text{and } (m_{\mathbf{R}} \circ (l^{-1} \circ p)) \in \mathcal{M} \} \quad (\text{Fact A.5}) \\
&= \{ \langle m, (m^* \circ m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \circ l^{-1} \rangle \mid \\
&\quad \langle m, m', l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \text{ and } \\
&\quad \langle m', (m^* \circ m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \rangle \in \\
&\quad \{ \langle m', m^* \circ (m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \rangle \mid \\
&\quad \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } (m_{\mathbf{R}} \circ (l^{-1} \circ p)) \in \mathcal{M} \} \quad (\{ \dots \mid \dots \}) \\
&= \{ \langle m, m^* \circ (m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \circ l^{-1} \rangle \mid \\
&\quad \langle m, m', l^{-1} \rangle \in \llbracket \text{Del}(l) \rrbracket \\
&\quad \text{and } \langle m', (m^* \circ m_{\mathbf{R}} \circ (l^{-1} \circ p)), (l^{-1} \circ p)^{-1} \rangle \in \\
&\quad \llbracket \text{Fix}(\mathbf{R}, (l^{-1} \circ p)) \rrbracket \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket)
\end{aligned}$$

$= \llbracket (\text{Del}(l); \text{Fix}(\mathbf{R}, (l^{-1} \circ p))) \rrbracket$ (Def. $\llbracket (_; _) \rrbracket$)
 (*): Note that $m_{\mathbf{R}} \circ (l^{-1} \circ p) \in \mathcal{M}$ implies $(l^{-1} \circ p)$ in \mathcal{M} (Fact A.8).
 Instead of showing $p^{-1} \circ l \circ l^{-1} = p^{-1}$, we show $l \circ l^{-1} \circ p = p$ (Facts A.6 and A.5). Let $p = \langle a, b \rangle$. Consider the composition $l \circ l^{-1} \circ p = \langle l \circ a^*, b \circ l^* \rangle$ where $\langle a^*, l^* \rangle$ is the pullback of $\langle l, a \rangle$. According to the definition of equality (Def. A.1), it suffices to prove that l^* is an isomorphism. As $l^{-1} \circ p$ is in \mathcal{M} , we know $b \circ l^*$ is an isomorphism (Fact A.8) and l^* is an isomorphism (Fact 2.6).

Case **Add**. Analogously to the program **Sel**.

Case **Uns**. Analogously to the program **Del**.

Case **(Assert(c); \mathbf{R})**.

$$\begin{aligned}
 & \text{Fix}((\text{Assert}(c); \mathbf{R}), p) \\
 \equiv & \text{Fix}((\text{Sel}(\text{id}, c); \mathbf{R}), p) && \text{(Def. Assert)} \\
 \equiv & (\text{Sel}(\text{id}, c); \text{Fix}(\mathbf{R}, \text{id} \circ p)) && \text{(Hypothesis)} \\
 \equiv & (\text{Sel}(\text{id}, c); \text{Fix}(\mathbf{R}, p)) && \text{(Fact A.3)} \\
 \equiv & (\text{Assert}(c); \text{Fix}(\mathbf{R}, p)) && \text{(Def. Assert)}
 \end{aligned}$$

Case **(Skip; \mathbf{R})**.

$$\begin{aligned}
 & \text{Fix}((\text{Skip}; \mathbf{R}), p) \\
 \equiv & \text{Fix}((\text{Assert}(\text{true}); \mathbf{R}), p) && \text{(Def. Skip)} \\
 \equiv & (\text{Assert}(\text{true}); \text{Fix}(\mathbf{R}, p)) && \text{(Hypothesis)} \\
 \equiv & (\text{Skip}; \text{Fix}(\mathbf{R}, p)) && \text{(Def. Skip)} \\
 \equiv & \text{Fix}(\mathbf{R}, p) && \text{(Fact 4.12)}
 \end{aligned}$$

Case **(Abort; \mathbf{R})**.

$$\begin{aligned}
 & \text{Fix}((\text{Abort}; \mathbf{R}), p) \\
 \equiv & \text{Fix}((\text{Assert}(\text{false}); \mathbf{R}), p) && \text{(Def. Abort)} \\
 \equiv & (\text{Assert}(\text{false}); \text{Fix}(\mathbf{R}, p)) && \text{(Hypothesis)} \\
 \equiv & (\text{Abort}; \text{Fix}(\mathbf{R}, p)) && \text{(Def. Abort)} \\
 \equiv & \text{Abort} && \text{(Fact 4.12)}
 \end{aligned}$$

Hypothesis.

Step.

Case $(\{P, \dots, Q\}; \mathbf{R})$. Let $\mathcal{S} = \{P, \dots, Q\}$.

$$\begin{aligned}
 & \llbracket \text{Fix}((\mathcal{S}; \mathbf{R}), p) \rrbracket \\
 = & \llbracket \text{Fix}(\bigcup_{S \in \mathcal{S}} (S; \mathbf{R}), p) \rrbracket && \text{(Fact 4.14)} \\
 = & \{ \langle m, m^* \circ m_{\bigcup} \circ p, p^{-1} \rangle \mid \langle m, m^*, m_{\bigcup} \rangle \in \llbracket \bigcup_{S \in \mathcal{S}} (S; \mathbf{R}) \rrbracket \\
 & \text{and } (m_{\bigcup} \circ p) \in \mathcal{M} \} && \text{(Def. } \llbracket \text{Fix}(_, _) \rrbracket \text{)} \\
 = & \{ \langle m, m^* \circ m_{\bigcup} \circ p, p^{-1} \rangle \mid \langle m, m^*, m_{\bigcup} \rangle \in \bigcup_{S \in \mathcal{S}} \llbracket (S; \mathbf{R}) \rrbracket \\
 & \text{and } (m_{\bigcup} \circ p) \in \mathcal{M} \} && \text{(Def. } \llbracket \{ _, \dots, _ \} \rrbracket \text{)}
 \end{aligned}$$

$$\begin{aligned}
&= \bigcup_{S \in \mathcal{S}} \{ \langle m, m^* \circ m_{\bigcup} \circ p, p^{-1} \rangle \mid \langle m, m^*, m_{\bigcup} \rangle \in \llbracket (S; R) \rrbracket \\
&\quad \text{and } (m_{\bigcup} \circ p) \in \mathcal{M} \} \quad (\{\dots \mid \dots\}) \\
&= \bigcup_{S \in \mathcal{S}} \llbracket \text{Fix}((S; R), p) \rrbracket \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \llbracket \bigcup_{S \in \mathcal{S}} \text{Fix}((S; R), p) \rrbracket \quad (\text{Def. } \llbracket \{ _, \dots, _ \} \rrbracket)
\end{aligned}$$

Case $(\text{Fix}(P); R)$.

$$\begin{aligned}
&\text{Fix}((\text{Fix}(P); R), p) \\
&= \{ \langle m, m^* \circ (m_{\text{Fix}; R} \circ p), p^{-1} \rangle \mid \\
&\quad \langle m, m^*, m_{\text{Fix}; R} \rangle \in \llbracket (\text{Fix}(P); R) \rrbracket \text{ and } (m_{\text{Fix}; R} \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \{ \langle m, m^* \circ ((m_R \circ \text{id}) \circ p), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \\
&\quad \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \text{ and } ((m_R \circ \text{id}) \circ p) \in \mathcal{M} \} \quad (\text{Def. } \llbracket (_; _) \rrbracket) \\
&= \{ \langle m, m^* \circ (m_R \circ p), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \\
&\quad \text{and } \langle m', m^*, m_R \rangle \in \llbracket R \rrbracket \text{ and } (m_R \circ p) \in \mathcal{M} \} \quad (\text{Fact A.3}) \\
&= \{ \langle m, m^* \circ (m_R \circ p), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \text{Fix}(P) \rrbracket \\
&\quad \text{and } \langle m', m^* \circ (m_R \circ p), p^{-1} \rangle \in \llbracket \text{Fix}(R, p) \rrbracket \} \quad (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&\equiv \text{Fix}(P); \text{Fix}(R, p) \quad (\text{Def. } \llbracket (_; _) \rrbracket)
\end{aligned}$$

Case $((P; Q); R)$.

$$\begin{aligned}
&\text{Fix}(((P; Q); R), p) \\
&\equiv \text{Fix}((P; (Q; R)), p) \quad (\text{Fact 4.13})
\end{aligned}$$

Case $(\text{if } c \text{ then } P \text{ fi}; R)$.

$$\begin{aligned}
&\text{Fix}((\text{if } c \text{ then } P \text{ fi}; R), p) \\
&\equiv \text{Fix}(\{(\text{Assert}(c); P), \text{Assert}(\neg c)\}; R), p) \quad (\text{Def. if-then}) \\
&\equiv \{ \text{Fix}(((\text{Assert}(c); P); R), p), \text{Fix}((\text{Assert}(\neg c); R), p) \} \quad (\text{Hypothesis}) \\
&\equiv \{ \text{Fix}((\text{Assert}(c); (P; R)), p), \text{Fix}((\text{Assert}(\neg c); R), p) \} \quad (\text{Fact 4.13}) \\
&\equiv \{ (\text{Assert}(c); \text{Fix}((P; R), p)), (\text{Assert}(\neg c); \text{Fix}(R, p)) \} \quad (\text{Hypothesis}) \\
&\equiv \text{if } c \text{ then } \text{Fix}((P; R), p) \text{ else } \text{Fix}(R, p) \text{ fi} \quad (\text{Def. if-else})
\end{aligned}$$

Case $(\text{if } c \text{ then } P \text{ else } Q \text{ fi}; R)$.

$$\begin{aligned}
&\text{Fix}((\text{if } c \text{ then } P \text{ else } Q \text{ fi}; R), p) \\
&\equiv \text{Fix}(\{(\text{Assert}(c); P), (\text{Assert}(\neg c); Q)\}; R), p) \quad (\text{Def. if-else}) \\
&\equiv \{ \text{Fix}(((\text{Assert}(c); P); R), p), \text{Fix}(((\text{Assert}(\neg c); Q); R), p) \} \quad (\text{Hypothesis}) \\
&\equiv \{ \text{Fix}((\text{Assert}(c); (P; R)), p), \text{Fix}((\text{Assert}(\neg c); (Q; R)), p) \} \quad (\text{Fact 4.13}) \\
&\equiv \{ (\text{Assert}(c); \text{Fix}((P; R), p)), (\text{Assert}(\neg c); \text{Fix}((Q; R), p)) \} \quad (\text{Hypothesis}) \\
&\equiv \text{if } c \text{ then } \text{Fix}((P; R), p) \text{ else } \text{Fix}((Q; R), p) \text{ fi} \quad (\text{Def. if-else})
\end{aligned}$$

Case $(P^0; R)$.

$$\begin{aligned}
&\text{Fix}((P^0; R), p) \\
&\equiv \text{Fix}((\text{Skip}; R), p) \quad (\text{Def. } P^0) \\
&\equiv \text{Fix}(R, p) \quad (\text{Hypothesis})
\end{aligned}$$

Case $(P^j; R)$, $j > 0$.

$$\text{Fix}((P^j; R), p)$$

$$\begin{aligned}
&\equiv \text{Fix}(((\text{Fix}(\mathbf{P}); \mathbf{P}^{j-1}); \mathbf{R}), p) && (\text{Def. } \mathbf{P}^j) \\
&\equiv \text{Fix}((\text{Fix}(\mathbf{P}); (\mathbf{P}^{j-1}; \mathbf{R})), p) && (\text{Fact 4.13}) \\
&\equiv (\text{Fix}(\mathbf{P}); \text{Fix}((\mathbf{P}^{j-1}; \mathbf{R}), p)) && (\text{Hypothesis}) \\
&\equiv (\text{Fix}(\mathbf{P}); \dots; \text{Fix}(\mathbf{P}); \text{Fix}((\mathbf{P}^0; \mathbf{R}), p)) && (\dots) \\
&\equiv (\text{Fix}(\mathbf{P}); \dots; \text{Fix}(\mathbf{P}); \mathbf{P}^0; \text{Fix}(\mathbf{R}, p)) && (\text{Hypothesis}) \\
&\equiv (\mathbf{P}^j; \text{Fix}(\mathbf{R}, p)) && (\text{Def. } \mathbf{P}^j)
\end{aligned}$$

Case $(\mathbf{P}^*; \mathbf{R})$.

$$\begin{aligned}
&\text{Fix}((\mathbf{P}^*; \mathbf{R}), p) \\
&\equiv \text{Fix}((\bigcup_{j=0}^{\infty} \mathbf{P}^j; \mathbf{R}), p) && (\text{Def. } \mathbf{P}^*) \\
&\equiv \bigcup_{j=0}^{\infty} \text{Fix}((\mathbf{P}^j; \mathbf{R}), p) && (\text{Hypothesis}) \\
&\equiv \bigcup_{j=0}^{\infty} (\mathbf{P}^j; \text{Fix}(\mathbf{R}, p)) && (\text{Hypothesis}) \\
&\equiv ((\bigcup_{j=0}^{\infty} \mathbf{P}^j); \text{Fix}(\mathbf{R}, p)) && (\text{Fact 4.13}) \\
&\equiv (\mathbf{P}^*; \text{Fix}(\mathbf{R}, p)) && (\text{Def. } \mathbf{P}^*)
\end{aligned}$$

Case $(\downarrow \mathbf{P} \downarrow; \mathbf{R})$.

$$\begin{aligned}
&\llbracket \text{Fix}((\downarrow \mathbf{P} \downarrow; \mathbf{R}), p) \rrbracket \\
&= \{ \langle m, m^* \circ (m_{\downarrow \mathbf{P} \downarrow; \mathbf{R}} \circ p), p^{-1} \rangle \mid \langle m, m^*, m_{\downarrow \mathbf{P} \downarrow; \mathbf{R}} \rangle \in \llbracket (\downarrow \mathbf{P} \downarrow; \mathbf{R}) \rrbracket \\
&\quad \text{and } (m_{\downarrow \mathbf{P} \downarrow; \mathbf{R}} \circ p) \in \mathcal{M} \} && (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \{ \langle m, m^* \circ (m_{\mathbf{R}} \circ \text{id} \circ p), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \downarrow \mathbf{P} \downarrow \rrbracket \\
&\quad \text{and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } (m_{\mathbf{R}} \circ \text{id} \circ p) \in \mathcal{M} \} && (\text{Def. } (_; _)) \\
&= \{ \langle m, m^* \circ (m_{\mathbf{R}} \circ p), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \downarrow \mathbf{P} \downarrow \rrbracket \\
&\quad \text{and } \langle m', m^*, m_{\mathbf{R}} \rangle \in \llbracket \mathbf{R} \rrbracket \text{ and } (m_{\mathbf{R}} \circ p) \in \mathcal{M} \} && (\text{Fact A.3}) \\
&= \{ \langle m, (m^* \circ (m_{\mathbf{R}} \circ p)), p^{-1} \rangle \mid \langle m, m', \text{id} \rangle \in \llbracket \downarrow \mathbf{P} \downarrow \rrbracket \\
&\quad \text{and } \langle m', (m^* \circ (m_{\mathbf{R}} \circ p)), p^{-1} \rangle \in \llbracket \text{Fix}(\mathbf{R}, p) \rrbracket \} && (\text{Def. } \llbracket \text{Fix}(_, _) \rrbracket) \\
&= \llbracket (\downarrow \mathbf{P} \downarrow; \text{Fix}(\mathbf{R}, p)) \rrbracket && (\text{Def. } (_; _))
\end{aligned}$$

Case $(\text{while } c \text{ do } \mathbf{P} \text{ od}; \mathbf{R})$.

$$\begin{aligned}
&\text{Fix}((\text{while } c \text{ do } \mathbf{P} \text{ od}; \mathbf{R}), p) \\
&\equiv \text{Fix}(((\text{if } c \text{ then } \mathbf{P} \text{ fi}^*; \text{Assert}(\neg c)); \mathbf{R}), p) && (\text{Def. while}) \\
&\equiv \text{Fix}((\text{if } c \text{ then } \mathbf{P} \text{ fi}^*; (\text{Assert}(\neg c); \mathbf{R})), p) && (\text{Fact 4.13}) \\
&\equiv (\text{if } c \text{ then } \mathbf{P} \text{ fi}^*; \text{Fix}((\text{Assert}(\neg c); \mathbf{R}), p)) && (\text{Hypothesis}) \\
&\equiv (\text{if } c \text{ then } \mathbf{P} \text{ fi}^*; (\text{Assert}(\neg c); \text{Fix}(\mathbf{R}, p))) && (\text{Hypothesis}) \\
&\equiv ((\text{if } c \text{ then } \mathbf{P} \text{ fi}^*; \text{Assert}(\neg c)); \text{Fix}(\mathbf{R}, p)) && (\text{Fact 4.13}) \\
&\equiv (\text{while } c \text{ do } \mathbf{P} \text{ od}; \text{Fix}(\mathbf{R}, p)) && (\text{Def. while})
\end{aligned}$$

□

Bibliography

- [AHPZ07] Karl Azab, Annegret Habel, Karl-Heinz Pennemann, and Christian Zuckschwerdt. ENFORCE: A system for ensuring formal correctness of high-level programs. In *Workshop on Graph Based Tools (Proc. GraBaTs'06)*, volume 1, pages 82–93. Electronic Communications of EASST, 2007.
 - [AHS90] Jiri Adámek, Horst Herrlich, and George Strecker. *Abstract and Concrete Categories*. John Wiley, New York, 1990.
 - [AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures*. MIT Press, 1991.
 - [AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors*. Academic Press, 1975.
 - [AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2nd edition, 1997.
 - [Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
 - [Bau06] Jörg Bauer. *Analysis of communication topologies by partner abstraction*. PhD thesis, Universität des Saarlandes, 2006.
 - [BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A modal-logic based graph abstraction. In *International Conference on Graph Transformation (Proc. ICGT'08)*, volume 5214 of *LNCS*, pages 321–335. Springer, 2008.
 - [BFT06] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *International Journal on Artificial Intelligence Tools*, 15(1):21–52, 2006.
-

- [BK02] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *International Conference on Graph Transformation (Proc. ICGT'02)*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
 - [BKK03] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Symposium on Static Analysis (Proc. SAS'03)*, volume 2694 of *LNCS*, pages 255–272. Springer, 2003.
 - [BKR05] Paolo Baldan, Barbara König, and Arend Rensink. Graph grammar verification through abstraction. In B. König, U. Montanari, and Ph. Gardner, editors, *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*, number 04241 in Dagstuhl Seminar Proceedings, 2005.
 - [BT03] Peter Baumgartner and Cesare Tinelli. The model evolution calculus. In *19th International Conference on Automated Deduction (Proc. CADE'03)*, volume 2741 of *LNAI (LNCS)*, pages 350–364. Springer, 2003.
 - [Bus04] Giorgio Busatto. GraJ: A system for executing graph programs in Java. Berichte aus dem Fachbereich Informatik 3/04, Universität Oldenburg, 2004.
 - [BW07] Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems by partner abstraction. In *Symposium on Static Analysis (Proc. SAS'07)*, volume 4634 of *LNCS*, pages 249–264. Springer, 2007.
 - [CMR⁺97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation. In *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, pages 163–245. World Scientific, 1997.
 - [Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logical approach. In *Handbook of Theoretical Computer Science*, volume B of *volume*, pages 193–242. Elsevier, 1990.
 - [Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of Graph Grammars and Computing by Graph Transformations*, volume 1, pages 313–400. World Scientific, 1997.
-

-
- [CS03] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19 Workshop on Model Computation (Proc. MODEL'03)*, 2003.
- [DFRdS03] Fernando L. Dotti, Luciana Foss, Leila Ribeiro, and Osmar M. dos Santos. Verification of distributed object-based systems. In *International Conference on Formal Methods for Open Object-Based Distributed Systems (Proc. FMOODS'03)*, volume 2884 of *LNCS*, pages 261–275. Springer, 2003.
- [Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [DS89] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1989.
- [dSDR04] Osmar M. dos Santos, Fernando L. Dotti, and Leila Ribeiro. Verifying object-based graph grammars. In *Workshop on Graph Transformation and Visual Modelling Techniques (Proc. GT-VMT'04)*, volume 109, pages 125–136. Elsevier, 2004.
- [Ecl] Eclipse Foundation. Homepage. <http://eclipse.org/>.
- [EEHP04] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Constraints and Application Conditions: From Graphs to High-Level Structures. In *International Conference on Graph Transformation (Proc. ICGT'04)*, volume 3256 of *LNCS*, pages 287–303. Springer, 2004.
- [EEHP06] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 74:135–166, 2006.
- [EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*.
-

- EATCS Monographs of Theoretical Computer Science. Springer, Berlin, 2006.
- [EHKP91] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. Parallelism and concurrency in high level replacement systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.
- [EHPP06] Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundamenta Informaticae*, 74:1–29, 2006.
- [Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1979.
- [EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Symposium on Foundations of Computer Science (Proc. FOCS’73)*, pages 167–180. IEEE, 1973.
- [Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proc. Symposium on Applied Mathematics*, volume 19 of *Aspects in Computer Science*, pages 19–32. AMS, Providence, R.I., 1967.
- [HESV91] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. The design of platoon maneuver protocols for IVHS. Technical Report UCBITS-PRR-91-6, University of California, Berkley, 1991.
- [HHT96] Annegret Habel, Reiko Heckel, and Gaby Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.
- [HKT84] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.
- [HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
-

-
- [Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *International Conference on Foundations of Software Science and Computation Structures (Proc. FoSSaCS’01)*, volume 2030 of *LNCS*, pages 230–245. Springer, 2001.
- [HP05] Annegret Habel and Karl-Heinz Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and System Modeling*, volume 3393 of *LNCS*, pages 293–308. Springer, 2005.
- [HP06] Annegret Habel and Karl-Heinz Pennemann. Satisfiability of high-level conditions. In *International Conference on Graph Transformation (Proc. ICGT’06)*, volume 4178 of *LNCS*, pages 430–444. Springer, 2006.
- [HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:1–52, 2009.
- [HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *International Conference on Graph Transformation (Proc. ICGT’06)*, volume 4178 of *LNCS*, pages 445–460. Springer, 2006.
- [HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars. In *Workshop on Graph Rewriting and Computation (Proc. SEGRAGRA’95)*, volume 2 of *ENTCS*, pages 95–104. Elsevier, 1995.
- [KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Proc. TACAS’06)*, 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. In *Workshop on Graph Transformation and Visual Modeling Techniques (Proc. GT-VMT’06)*, volume 211 of *ENTCS*, pages 201–210. Elsevier, 2008.
-

- [KMP05] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Graph-based specification of access control policies. *Journal of Computer and System Sciences (JCSS)*, 71:1–33, 2005.
 - [KR06] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In *Proc. SPIN Workshop on Model Checking Software*, volume 3925 of *LNCS*, pages 299–305. Springer, 2006.
 - [LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 2nd edition, 1998.
 - [LS04] Stephen Lack and Pawel Sobociński. Adhesive categories. In *International Conference on Foundations of Software Science and Computation Structures (Proc. FoSSaCS'04)*, volume 2987 of *LNCS*, pages 273–288. Springer, 2004.
 - [McC] William McCune. Homepage of Prover9. <http://prover9.org/>.
 - [McC03] William McCune. Mace4 reference manual and guide. Tech. Memo ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2003.
 - [McC06] William McCune. Semantic guidance for saturation provers. In *International Conference on Artificial Intelligence and Symbolic Computation (Proc. AISC'06)*, volume 4120 of *LNCS*, pages 18–24. Springer, 2006.
 - [Möl03] Andreas Möller. Eine virtuelle Maschine für Graphprogramme. Berichte aus dem Fachbereich Informatik 5/03, Universität Oldenburg, 2003.
 - [OEP08] Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. A logic of graph constraints. In *International Conference on Fundamental Approaches to Software Engineering (Proc. FASE'08)*, volume 4961 of *LNCS*, pages 179–19. Springer, 2008.
 - [Ore08] Fernando Orejas. Attributed graph constraints. In *International Conference on Graph Transformation (Proc. ICGT'08)*, volume 5214 of *LNCS*, pages 274–288. Springer, 2008.
 - [Pen04] Karl-Heinz Pennemann. Generalized constraints and application conditions for graph transformation systems. Master's thesis, Department of Computing Science, University of Oldenburg, Oldenburg, 2004.
-

-
- [Pen08a] Karl-Heinz Pennemann. An algorithm for approximating the satisfiability problem of high-level conditions. In *Graph Transformation for Verification and Concurrency (Proc. GT-VC'07)*, volume 213 of *ENTCS*, pages 75–94. Elsevier, 2008.
- [Pen08b] Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In *International Conference on Graph Transformation (Proc. ICGT'08)*, volume 5214 of *LNCS*, pages 289–304. Springer, 2008.
- [Pfe07] Frank Pfenning, editor. *21th International Conference on Automated Deduction (Proc. CADE'07)*, volume 4603 of *LNAI (LNCS)*. Springer, 2007.
- [Plu93] Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In *Term Graph Rewriting: Theory and Practice*, pages 201–213. John Wiley, New York, 1993.
- [Plu95] Detlef Plump. On termination of graph rewriting. In *Workshop on Graph-Theoretic Concepts in Computer Science (Proc. WG'95)*, volume 1017 of *LNCS*, pages 88–100. Springer, 1995.
- [Plu05] Detlef Plump. Confluence of graph transformation revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, volume 3838 of *LNCS*, pages 280–308. Springer, 2005.
- [PS04] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *International Conference on Graph Transformation (Proc. ICGT'04)*, volume 3256 of *LNCS*, pages 128–143. Springer, 2004.
- [RD06] Arend Rensink and Dino S. Distefano. Abstract graph transformation. In *Proc. Workshop on Software Verification and Validation*, volume 157 of *ENTCS*, pages 39–59. Elsevier, 2006.
- [Ren04a] Arend Rensink. Representing first-order logic by graphs. In *International Conference on Graph Transformation (Proc. ICGT'04)*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.
- [Ren04b] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (Proc. AGTIVE'03)*, volume 3062 of *LNCS*, page 485. Springer, 2004.
-

- [Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
 - [RSV04] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *International Conference on Graph Transformation (Proc. ICGT'04)*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.
 - [RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.
 - [Sch08] Uwe Schöning. *Theoretische Informatik - kurzgefaßt*. Spektrum Akademischer Verlag, 5th edition, 2008. In German.
 - [SG06] Martin Strecker and Mathieu Giorgino. Towards a formalisation of graph transformations in proof assistants. In *Workshop on Automated Verification of Critical Systems (Proc. AVoCS'06)*, 2006.
 - [SS98] Geoff Sutcliffe and Christian B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
 - [Str08] Martin Strecker. Modeling and verifying graph transformations in proof assistants. In *Workshop on Computing with Terms and Graphs (Proc. Termgraph'07)*, volume 203 of *ENTCS*, pages 135–148. Elsevier, 2008.
 - [Tra50] Boris A. Trakhtenbrot. The impossibility of an algorithm for the decision problem on finite classes (In Russian). *Doklady Akademii Nauk SSSR*, 70:569–572, 1950. English translation in: Nine Papers on Logic and Quantum Electrodynamics, *AMS Translation Series 2*, 23:1–5, 1963.
 - [Var03] Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02)*, volume 72 of *ENTCS*, pages 51–64. Elsevier, 2003.
 - [Var04] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 3(2):85–113, 2004.
-

-
- [Zha96] Jian Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17(1):1–22, 1996.
- [Zuc06] Christian Zuckschwerdt. Ein System zur Transformation von Konsistenz in Anwendungsbedingungen. Berichte aus dem Department für Informatik 11/06, 114 Seiten, Universität Oldenburg, 2006.
- [ZZ95] Jian Zhang and Hantao Zhang. SEM: A system for enumerating models. In *International Joint Conferences on Artificial Intelligence (Proc. IJCAI'95)*, volume 1 of , pages 298–303. Morgan Kaufmann, 1995.
-

List of Figures

1.1	Example specification of an access control system	1
1.2	Overview of this thesis	3
2.1	A graph morphism $g: G \rightarrow H$	9
2.2	A set-theoretic depiction of a pushout	12
4.1	Triple $\langle m, m^*, m_p \rangle$ of the semantics of a program P	46
4.2	Semantics of $\text{Sel}(x, c)$, $\text{Del}(l)$, $\text{Add}(r)$ and $\text{Uns}(y)$	46
4.3	Semantics of $\text{Add}(\emptyset \hookrightarrow \bigcirc)$ and $\text{Fix}(\text{Add}(\emptyset \hookrightarrow \bigcirc))$	49
5.1	Decider for the correctness problem	68
6.1	A fragment of the semantics of $\text{SeekSat}(c)$	89
7.1	Execution of the access control JUNIT test in ECLIPSE	117
7.2	Concrete graph and its graphical representation	118
7.3	Railroad network with a train and two switches	118
7.4	Railroad control: comparison of deciders (“ t_{\models} ”)	124
7.5	Car platoon	124
7.6	Car platoons state graphs	125
7.7	Car platoons: comparison of deciders (“ t_{\models} ”)	133
7.8	Access control: comparison of deciders (“ t_{\models} ”)	141

List of Tables

2.1	Correspondence of categorical and graph notions	11
3.1	Conditions and their meaning	19
4.1	Program constructs and their meaning	44
5.1	Summary of dedicated GT verification approaches	81
6.3	Equivalences for conjunctive normal form	110
7.1	Correspondence of case studies and JUNIT tests	116
7.2	Considered pairs of theorem provers and satisfiability solvers .	116
7.3	Railroad control case study: results	122
7.4	Railroad: decision times of first-order provers and solvers . . .	123
7.5	Car platoons case study: alphabet	125
7.6	Car platoons case study: results 1-70	131
7.7	Car platoons case study: results 71-140	132
7.8	Car platoons: decision times of first-order provers and solvers	134
7.9	Access control case study: results 1-60	139
7.10	Access control case study: results 61-120	140
7.11	Access control: decision times of first-order provers and solvers	142

Abbreviations

CTL	c omputation t ree l ogic
GL	g raph l ogic
MSOGL	m onadic ulsecond-ulorder g raph l ogic
CADE	c onference on a utomated d eduction
CNF	c onjunctive n ormal f orm
DPLL	D avis- P utnam- L ogemann- L oveland algorithm
DPO	d ouble p ushout approach
ENFORCE ..	e nsuring f ormal c orrectness
GT	g raph t ransformation
HLR	h igh-level r eplacement
ME calculus ..	m odel e volution c alculus
MNF	M -normal f orm
NAC	n egative a pplication c ondition
ProCon	p roving c onditions
SeekSat	s eeking s atisfiability
Wlp	w eakest l iberal p reconditions

Index

Symbols	
\emptyset	8
I	16
i_G	16
id_G	7
\rightarrow	7
\hookrightarrow	8
\hookleftarrow	149
\leftrightarrow	8
\mathcal{O}	7
\mathcal{A}	7
Epi	7
Iso	7
Mon	7
\mathcal{M}	7
\mathcal{P}	149
$Cond$	20
\Rightarrow	20
\models	20
$\models_{\mathcal{A}}$	24
\Rightarrow	20
\equiv	
on conditions	20
on programs	46
$\equiv_{\mathcal{M}}$	20
$\equiv_{\mathcal{A}}$	24
$\text{dom}(m)$	7
$\text{codom}(m)$	7
$\text{iface}(pm)$	149
A	
access control case study ..	9, 21, 133
alphabet	8, 118, 125
application condition	21, 44, 119, 127
C	
calculus	102
car platooning case study	124
case study	115
access control	9, 21, 133
car platooning	124
railroad control	117
category	7
Graphs	8
weak adhesive HLR	15
codomain	7
condition	20, 24
application condition	21, 44, 119, 127
graph condition	118, 125, 133
non-negated subcondition ...	101
constraint	20
correctness	57
cospans	11
D	
deduction rule	101
domain	7
E	
edge	8
epimorphism	7
equivalence	
of conditions	20, 46
G	
graph	8

graph morphism	8	pullback	14
I		pushout	11, 46, 65
implication problem	83	\mathcal{M} -pushout	99
invariant	74	pushout complement	13, 46, 61
isomorphism	7	R	
L		railroad control case study	117
label	8	rule	
M		deduction rule	101
matching		transformation rule	44, 119, 127
\mathcal{M} -matching	53	S	
monomorphism	7	satisfiability notion	
partial	149	\mathcal{A} -satisfiability	24
morphism	7	\mathcal{M} -satisfiability	24
epimorphic	7	satisfiability problem	84
graph morphism	8	SeekSat	87, 110
in <i>Epi</i>	7	source node	8
in <i>Iso</i>	7	span	11
in \mathcal{M}	15	specification	57
injective	8	structural equivalence	109
isomorphic	7	T	
monomorphic	7	target node	8
surjective	8	tautology problem	84
N		transformation	
node	8	$A(m, c)$	59
normal form		$Add(m, c)$	65
\mathcal{M} -normal form	22	$Asat(c)$	28
conjunctive normal form	100	$Cond(F)$	33
O		$Del(m, c)$	61
object	7	$Deletable(m)$	63
P		$Ffree(P)$	51
partial monomorphism	149	$Form(c)$	36
postcondition	57	$Msat(c)$	25
precondition	57	$SeekSat(c)$	87
ProCon	98, 102	$Wlp(P, c)$	69
program specification	57	transformation rule	44, 119, 127
program with interface 43 ff., 47, 135		transformation system	45
		V	
		vertex	8

W

- weak adhesive HLR category 15
 - weakest
 - invariant 74
 - liberal precondition . . . 67, 69, 74
 - precondition 67, 82
-

Curriculum Vitae

- December 2005 – July 2009 PhD student and scholarship holder at the DFG graduate school TrustSoft, University of Oldenburg
- October 2004 – August 2005 Research associate with teaching duties; Formal Languages Group of Prof. Annegret Habel, University of Oldenburg
- September 2004 Diploma (MSc.) degree (with honors) in Computer Science
- October 1997 – September 2004 Computer Science studies, University of Oldenburg
- June 1997 A-Level (Abitur), Fachgymnasium Wirtschaft Papenburg (Grammar School for Business Education)
- October 1977 Born in Papenburg, Germany
-