

Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

A Theory of HR* Graph Conditions and their Application to Meta-Modeling

Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt von

Dipl.-Inform. Hendrik Radke

Datum der Disputation: 18.03.2016

Gutachter: Prof. Dr. Annegret Habel, Prof. Dr. Andy Schürr

Abstract

As software systems grow more complex, there is a growing need for design concepts that facilitate an intuitive overview of a system. This is usually achieved through visual modeling techniques. Graph transformation systems are an established visual modeling approach, modeling systems as graphs. Structural properties of the graphs can be expressed by nested conditions as described by (Habel and Pennemann, 2009). However, nested conditions are not expressive enough to formulate non-local properties often encountered in real-world problems, like the existence of arbitrary-length paths, connectedness or circle-freeness.

We thus propose HR^* conditions, an extension to nested conditions. HR^* conditions are enriched with hyperedges, which are then replaced by graphs according to a hyperedge replacement system. The expressiveness of HR^* conditions lies between counting monadic second-order logic and second-order logic. Several variants of HR^* conditions are introduced and their respective advantages and disadvantages are discussed.

The correctness of a specification, i.e. a triple of graph program, pre- and postcondition in the form of HR^* conditions, can be checked. Basic transformations are used on the graph program and the postcondition to generate a weakest precondition. This can then be compared with the original precondition to check the correctness of the specification.

HR^* conditions are applied to the problem of instance generation for UML meta-models with OCL constraints. The meta-model's type graph can be transformed into a graph grammar, as shown e.g. in (K. Ehrig et al., 2009). Essential OCL constraints belonging to the type graph are transformed into HR^* conditions. Using HR^* conditions enables the translation of OCL constraints that go beyond first-order. The conditions are then transformed into application conditions for the graph grammar's rules. This ensures that the grammar only generates instances which satisfy the OCL constraints. The grammar-based approach enables the simple generation of a large number of instances.

Zusammenfassung

Die steigende Komplexität von Software-Systemen bedingt einen steigenden Bedarf an Designkonzepten, die einen intuitiven Überblick auf das System erlauben. Dies wird üblicherweise durch visuelle Modellierungstechniken erreicht. Einen solchen Ansatz liefern Graphtransformationssysteme, die Systeme durch Graphen modellieren. Strukturelle Eigenschaften des Systems können durch geschachtelte Graphbedingungen, wie in (Habel und Pennemann, 2009) gezeigt, beschrieben werden. Die Ausdruckskraft geschachtelter Graphbedingungen reicht jedoch nicht zur Beschreibung nichtlokaler Eigenschaften aus, wie sie in praktischen Anwendungen häufig zu finden sind. Beispielsweise ist es nicht möglich, Eigenschaften wie die Existenz beliebig langer Pfade, Verbundenheit oder Kreisfreiheit mit geschachtelten Graphbedingungen auszudrücken.

Diese Arbeit führt HR^* -Bedingungen ein, eine Erweiterung von geschachtelten Graphbedingungen. HR^* -Bedingungen sind mit Hyperkanten angereichert, die anhand eines Hyperkantenersetzungssystems durch Graphen ersetzt werden. Die Ausdruckskraft von HR^* -Bedingungen liegt zwischen Counting Monadic Second-Order-Logik und Second-Order-Logik. Verschiedene Varianten von HR^* -Bedingungen werden vorgestellt und ihre Vor- und Nachteile diskutiert.

Die Korrektheit einer Spezifikation, d.h. eines Tripels aus einem Graphprogramm sowie Vor- und Nachbedingung in Form von HR^* -Bedingungen, kann geprüft werden. Dazu wird aus Nachbedingung und Graphprogramm durch grundlegende Transformationen eine schwächste Vorbedingung erzeugt. Diese kann dann mit der ursprünglichen Vorbedingung verglichen werden, um die Korrektheit der Spezifikation zu prüfen.

HR^* -Bedingungen werden auf das Problem der Erzeugung von Instanzen von UML-Metamodellen mit OCL-Constraints angewandt. Der Typgraph des Metamodells kann in eine Graphgrammatik umgewandelt werden, wie beispielsweise in (K. Ehrig u. a., 2009) gezeigt. Die zum Typgraphen gehörigen Essential OCL-Constraints werden zu HR^* -Bedingungen transformiert. Die Benutzung von HR^* -Bedingungen ermöglicht dabei das Ausdrücken von OCL-Constraints jenseits von First-Order-Logik. Mit Hilfe der eingeführten grundlegenden Transformationen werden die HR^* -Bedingungen anschließend in Anwendungsbedingungen für die Graphgrammatik umgewandelt. So wird sichergestellt, dass alle von der Grammatik erzeugten Graphen den OCL-Constraints genügen. Die Verwendung einer solchen Grammatik ermöglicht die einfache Erzeugung von Instanzen des Metamodells.

Acknowledgements

Writing this thesis was a challenge I could not have mastered without the help of many different people.

First and foremost, I thank my advisors, Annegret Habel and Ernst-Rüdiger Olderog, for their continuous support and guidance during my time as a Ph.D. student. A big thank you also goes to my second examiner, Andy Schürr, for providing helpful and in-depth feedback, and to the other members of the board of examiners, Andreas Winter and Stephanie Kemper.

During my time in Oldenburg, many people provided help and advice. I am especially thankful to Berthold Hoffmann for providing advice and helpful questions in all phases of my thesis, and to Sven Linker for his ongoing encouragement and our little “race into space”. I would like to thank Jan Steffen Becker, Thomas Hujsa, Stephanie Kemper, Christoph Peuser and Maike Schwammbberger for proofreading different chapters of my thesis, for valuable feedback and the many hours that sometimes unwelcome task took out of their schedule. Thanks also go to Jan Steffen Becker, Evgeny Erofeev, Maik Fischer and Heinrich Ody for helping me prepare for my defense. I thank the members of the Marburg-Oldenburg DFG project "Meta-modeling and graph grammars": Thank you Gabriele Taentzer, Thorsten Arendt and Nebras Nassar, for fruitful discussions and inspiration.

The pleasant working atmosphere I experienced in the theoretical computer science groups is due to all of my former and current colleagues (in alphabetical order): Jan Steffen Becker, Marion Bramkamp, Björn Engelmann, Evgeny Erofeev, Johannes Faber, Hans Fleischhack, Nils-Erik Flick, Sibylle Fröschle, Thomas Hujsa, Manuel Giesecking, Andrea Göken, Martin Hilscher, Stephanie Kemper, Heinrich Ody, Christoph Peuser, Jan-David Quesel, Uli Schlachter, Maike Schwammbberger, Valentin Spreckels, Thomas Stratmann, Tim Strazny, Mani Swaminathan, Patrick Uven and Elke Wilkeit. Thank you for making my time in Oldenburg such a pleasant experience.

A big influence and motivation is due to my fellows in the graduate schools “TrustSoft” and “SCARE”. Thank you for instructive and inspiring meetings, lively discussions and helpful advice. A special thanks goes to Ira Wempe for being a great coordinator of both graduate schools and for always having an open ear for all the big and small problems.

Where would I be without the neverending support of my family? A heartfelt “thank you” goes to my parents, Oskar and Margit Radke, to Holger and Christine, and to Yani. Without your care, your advice and help, I could never have even dreamed of achieving this.

Contents

1	Introduction	1
1.1	State of the art	3
1.2	Thesis outline	5
2	Nested graph conditions and graph transformation	7
2.1	Graphs and graph morphisms	7
2.2	Nested graph conditions	8
2.3	Graph transformation	10
2.4	A containment operator for nested conditions	13
3	HR* graph conditions	15
3.1	HR systems	15
3.2	HR* graph conditions	21
3.3	Decidability of HR* conditions	24
3.4	Case study: car platooning	25
4	Normal forms and variants of HR* conditions	31
4.1	Normal forms	31
4.2	HR* conditions with arbitrary satisfaction	34
4.3	HR* conditions with replacement semantics	42
5	Expressive power of HR* conditions	47
5.1	Graph formulas	47
5.2	Transforming counting monadic second-order formulas into HR* conditions	52
5.3	Transforming HR* conditions into second-order formulas	56
6	Correctness relative to HR* conditions	65
6.1	Shifting for path-like conditions	67
6.2	Shifting for arbitrary HR* conditions	73
6.3	From right to left application conditions	82
6.4	From left application conditions to preconditions	86
6.5	Expressing the applicability of a rule as a condition	86
6.6	Correctness of graph programs	87
7	Application to meta-modeling	93
7.1	An overview of the Object Constraint Language	94
7.2	Graphs and conditions for OCL	101

Contents

7.3	Translating Essential OCL to graph conditions	110
7.4	Translating OCL constraints beyond first-order expressiveness	120
7.5	Integration of graph constraints into graph grammars	124
8	Conclusion	129
8.1	Summary	129
8.2	Open problems and future work	130
	Bibliography	131
	Index	142
	Symbol Glossary	142
	Subject Index	147

Chapter 1

Introduction

As soft- and hardware grow in size and complexity, visual modeling techniques that give an intuitive overview of a system are more and more important. Moreover, with the increasing reliance on hard- and software systems in safety- and security-critical areas, there is also a growing need for the verification of these systems. Therefore, it is desirable to combine visual modeling with formal verification.

The approach taken in this PhD thesis is to model the states of a system with graphs. Graphs are a well-known and established representation for the visualization of relations between a set of elements. The dynamics of a system are modeled using graph transformation rules (H. Ehrig, K. Ehrig, Prange, et al., 2006). Graph transformation has many application areas in software engineering and in the design of structure-changing or distributed systems.

Structural properties of a system are traditionally described using textual logical formulas. This work focuses on a more visual notation to describe these properties. In (Habel and Pennemann, 2009), structural properties are described visually by nested conditions, equivalent to first-order logic on graphs. Nested graph conditions are expressively equivalent to first-order graph formulas and can express local properties in the sense of (Gaifman, 1982). However, there are many interesting non-local graph properties like the existence of an arbitrary-length path between two nodes, connectedness or circle-freeness of the graph.

We propose HR^* conditions, an extension to nested conditions that facilitates the formulation of such properties in a visual way. HR^* conditions use hyperedge replacement systems to describe recurring structures of arbitrary size, like paths or circles. Since HR^* conditions can express such non-local properties, they are clearly more expressive than nested conditions. We will show in this PhD thesis that HR^* conditions can express any counting monadic second-order property.

HR^* conditions can be used together with graph programs to build graph specifications with pre- and postconditions in the form of a Hoare triple. Our goal is to check the correctness of such specifications, i.e. whether for all graphs that satisfy the precondition, any graph that results from application of the program satisfies the postcondition. Following the approach of (Dijkstra, 1976), we transform the postcondition into a weakest

precondition. We can then check whether the original precondition implies the weakest precondition.

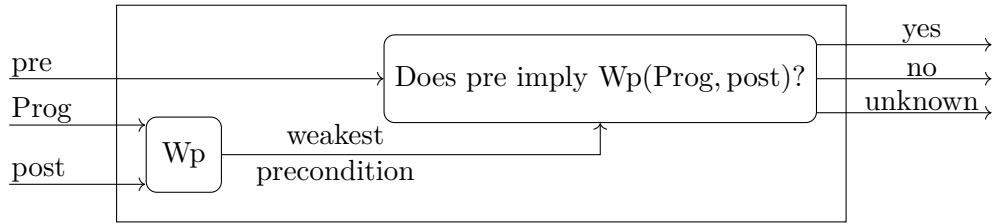


Figure 1.1: Correctness relative to HR* conditions

HR* conditions fulfill a double role as general constraints for graphs and application conditions for graph transformation rules. The basic transformations used to calculate weakest preconditions also serve to transform constraints into application conditions and vice versa. This enables us to transform e.g. a postcondition into an application condition for a rule, ensuring that the rule can only be applied to a graph if the result satisfies the postcondition.

Graph transformation has many applications in the area of software engineering. In this thesis, we use graph grammars and HR* conditions to generate instances of a UML meta-model. The combination of meta-modeling with graph grammars promises several useful applications, including the testing of model transformations and the generation of edit operations or model repair actions.

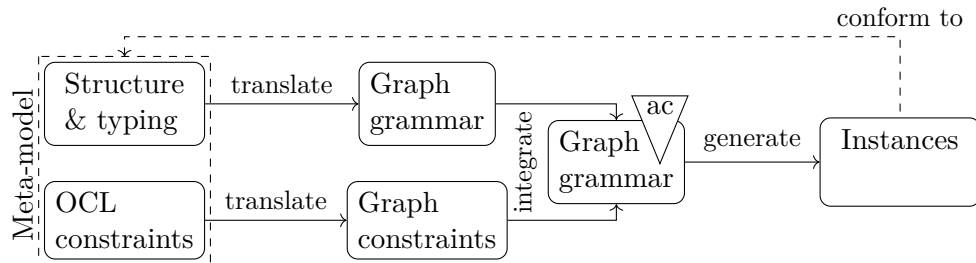


Figure 1.2: Application to meta-modeling

The approach translates the meta-model into a graph grammar, with all its constraints and restrictions. Instances can then be generated by deriving a terminal graph with the rules of the graph grammar. Since meta-models are often further restricted with Essential OCL (Object Constraint Language) invariants, the grammar has to be restricted accordingly. We thus present a translation from Essential OCL invariants to HR* constraints, establishing a formal link between OCL and graph transformation.

The HR* constraints are further transformed into application conditions for the rules

of the grammar, guaranteeing that only valid models are generated that satisfy the OCL constraints.

1.1 State of the art

In the following, we present a short state of the art. This subchapter consists of three parts: The first part discusses various concepts for expressing graph properties in a mostly visual way. In the second part, approaches for the verification of graph transformation systems are commented. The third part surveys works that relate meta-models, and especially OCL constraints, to graph grammars and graph constraints. At the end of each chapter, there will be a more detailed discussion of the chapter's relation to other publications.

Languages for expressing graph properties

Visual languages Many concepts have been developed to express graph properties in a formal way. They differ in their expressiveness, their exact purpose and style. While some concepts stay close to logical formulas in order to facilitate reasoning about these properties, other languages aim for a more visual, intuitive formalism.

The expressiveness of application conditions for graph transformation rules has been steadily increased over the years. Some early graphical query languages supporting transitive closure on edge relations can be found in (Cruz et al., 1987; Angelaccio et al., 1990). In (Heckel and Wagner, 1995; Habel, Heckel, et al., 1996), *consistency conditions* and their transformation into *negative application conditions* were presented. Negative application conditions were more thoroughly regarded in (Lambers, 2010), and the concept was lifted to weak adhesive HLR categories in (H. Ehrig, K. Ehrig, Habel, et al., 2006). (Habel, Pennemann, and Rensink, 2006) introduced nested conditions, equivalent to first-order logic on graphs, and showed how they could be used as application conditions. The concept was generalized and equipped with basic transformations in (Habel, Pennemann, and Rensink, 2006). The idea to use graph grammars in conditions was explored in (B. König and Esparza, 2010).

However, many interesting graph properties lie beyond the scope of first-order logic, as shown in (Gaifman, 1982). To address such properties, several extensions have been proposed. (Poskitt and Plump, 2013) enhanced nested conditions to *E-conditions*, facilitating the handling of nodes with multiple, typed labels and to perform operations on these labels, e.g. integer arithmetic and string operations. An extension into a different direction was proposed in (Bruggink and B. König, 2010; Bruggink, Hülsbusch, et al., 2012): The *logic on subobjects* includes reasoning about subobjects in the regarded category. In the category of graphs, these subobjects are subgraphs, and the resulting formalism is exactly as expressive as monadic second-order logic on graphs. The *M-conditions* defined in (Poskitt and Plump, 2014) are an extension of nested conditions to monadic second-order logic, handling graphs in the visual way of nested conditions and sets in a more abstract, textual way. The μ -conditions presented in (Flick, 2016) take a recursive approach to express paths of arbitrary length.

Non-visual languages Several logics on graphs, especially monadic second-order logic, are presented in (Courcelle and Engelfriet, 2012). An insightful comparison of such logics is given in (Courcelle, 1996). For the case of separation logic, (Dodds and Plump, 2009) show that a fragment of separation logic exactly corresponds to a subclass of the languages defined by hyperedge replacement grammars. (Gadducci et al., 2012) unites concepts from temporal logic with monadic second-order logic, in order to describe properties of graphs and graph transformation systems.

Verification of graph transformation systems

A prominent application of languages expressing graph properties lies in the verification of graph transformation systems. One approach to verification is model checking, where a finite state space is systematically and exhaustively checked against constraints.

In (Varró, 2003), typed, attributed graph transformation rules with (negative) application conditions are checked against reachability of “property graphs” by SPIN after a prior translation into PROMELA. The GROOVE tool of (Kastenberg and Rensink, 2006) performs model checking of graph transformation rules and can verify properties formulated in a modal logic over graphs, which is enhanced by a transitive closure operator. In (Bauer, 2006), abstraction is used on a fixpoint approximation of graph relabeling rules. This approach is implemented in the HIRALYSIS tool.

However, model checking can only check programs relative to a finite number of starting states (i.e. graphs). Therefore, another approach is to translate the transformation rules into logical formulas, as proposed in (Courcelle, 1990), and to use theorem proving techniques on them. In (M. Strecker, 2008; M. Strecker, 2011), graph transformation rules and programs, with respect to the GREAT language for graph properties, are translated into formulas that can be checked with the ISABELLE proof assistant.

The calculus developed in (Poskitt and Plump, 2013) can be used to verify properties relative to the graph programming language GP. The PROCON theorem prover presented in (Pennemann, 2008b) uses a similar technique and tries to solve implication problems for logical formulas on graphs. The program SEEKSAT presented in (Pennemann, 2008a) tries to find counter-examples to such formulas using SAT solving techniques. The AUGUR2 tool (B. König and Kozioura, 2008a) analyzes and verifies graph transformation systems by approximating them with Petri nets. In (Hildebrandt et al., 2012), model transformation using triple graph grammars is enhanced by an invariant checker for constraints that restrict the source and target graphs.

Meta-models and graph grammars

The idea to use graph transformation in the context of meta-modeling in general and OCL in particular is not new. In (Lara and Vangheluwe, 2004), a framework and a tool (ATOM3) are presented that use graph grammars to manipulate meta-models. (Klar et al., 2007) explored special techniques to perform model transformation on large models, and (Varró and Balogh, 2007) introduced the VIATRA2 framework integrating abstract space machines with graph transformation. Techniques for dealing with attributed

graph transformation were refined in (B. König and Kozioura, 2008b). (Hermann et al., 2010) discussed general techniques for model transformation and the use of triple graph grammars. A transformation from meta-models, along with some constraints, was presented in (Taentzer, 2012). Another option is explored in (Kuhlmann and Gogolla, 2012), where OCL constraints are transformed to and from a relational logic.

In (Winkelmann et al., 2008), it was shown how a restricted subset of OCL constraints can be transformed into constraints for graph grammars. This work only translates a subset of navigation expressions, size and attribute constraints and Boolean combinations thereof. This work was extended to larger subclasses of OCL in (Bergmann, 2014), (Richa, Borde, Pautet, et al., 2014; Richa, Borde, and Pautet, 2015) as well as (Arendt, Habel, et al., 2014; Radke et al., 2015); a more detailed look at these papers will be given at the end of Chapter 7.

1.2 Thesis outline

In Chapter 2, we recapitulate nested conditions. HR^* conditions are introduced in Chapter 3, and Chapter 4 discusses different variants of them. In Chapter 5, we look at the expressiveness of HR^* conditions in comparison to several logical formalisms and show that it lies between monadic second-order and second-order graph formulas. Chapter 6 is dedicated to transformations of HR^* conditions over rules and programs, and to checking correctness of programs with respect to HR^* conditions. The developed concepts are applied to UML meta-models with OCL constraints in Chapter 7, where HR^* conditions are used as application conditions in an instance-generating graph grammar. Chapter 8 summarizes the results of the thesis and discusses topics for future work.

Chapter 2

Nested graph conditions and graph transformation

Contents

2.1	Graphs and graph morphisms	7
2.2	Nested graph conditions	8
2.3	Graph transformation	10
2.4	A containment operator for nested conditions	13

In this chapter, we recall the basic notions of graphs and graph morphisms, nested graph conditions, and graph transformation. For a more detailed introduction, we refer to (H. Ehrig, K. Ehrig, Prange, et al., 2006; Habel and Pennemann, 2009).

2.1 Graphs and graph morphisms

Graphs consist of labeled nodes and edges. Edges have one source and one target.

Definition 2.1 (graph). Let $L = L_V \uplus L_E$ ¹ be a fixed, finite alphabet of node and edge labels. A *graph* over L is a system $G = (V_G, E_G, s_G, t_G, lv_G, le_G)$ consisting of finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, *source* and *target functions* $s_G, t_G: E_G \rightarrow V_G$, and *labeling functions* $lv_G: V_G \rightarrow L_V$, $le_G: E_G \rightarrow L_E$. A node or edge is *a-labeled* if its label is a . The set of all graphs is denoted by \mathcal{G} . A graph G is *empty*, denoted by \emptyset , iff $V_G = \emptyset$. We denote the disjoint union of two graphs G, H by $G + H$. For graphs G, H with $H \subseteq G$, $G - H$ denotes the difference. \triangle

¹For two sets A and B , $A \uplus B := \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}$ denotes the disjoint union of A and B .

Graph morphisms consist of a pair of mappings between the node and edge sets of graphs.

Definition 2.2 (graph morphism). A *graph morphism* $g: G \rightarrow H$ consists of functions $g_V: V_G \rightarrow V_H$ and $g_E: E_G \rightarrow E_H$ that preserve sources, targets and labels, i.e. $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $lv_H \circ g_V = lv_G$ and $le_H \circ g_E = le_G$, where \circ denotes the composition of functions. The graph G is called the *domain* of g , denoted $\text{Dom}(g)$. H is called the *codomain* of g , denoted $\text{Ran}(g)$. The *composition* $h \circ g$ of g with a graph morphism $h: H \rightarrow M$ consists of the composed functions $h_V \circ g_V$ and $h_E \circ g_E$.

A graph morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is both injective and surjective. Injective/surjective morphisms are also called *monomorphisms* and *epimorphisms*, respectively. Let \mathcal{M} and \mathcal{E} be the set of all injective and surjective morphisms, respectively. If g is an isomorphism, its domain and codomain G and H are *isomorphic*, which is denoted by $G \cong H$.

An injective graph morphism $m: G \hookrightarrow H$ is an *inclusion*, written $G \subseteq H$, if $V_G \subseteq V_H$ and $E_G \subseteq E_H$. For a graph G , the *identity* $\text{id}_G: G \rightarrow G$ consists of the identities id_{G_V} and id_{G_E} on G_V and G_E , respectively. \triangle

Notation. Morphisms are written with a plain arrow \rightarrow ; injective morphisms are written with a hooked arrow (\hookrightarrow). The mapping of nodes by a morphism is conveyed by adding small numbers to the side of the nodes; a node in the domain is mapped to the node with the same number in the codomain.

Example 2.1. The following example shows an injective morphism g from a graph G to a graph H . Graph G consists of five nodes with labels a and c and seven edges (including one loop) with the (invisible) label \square . Additionally, graph H contains an isolated node with label c and another edge with label \square .

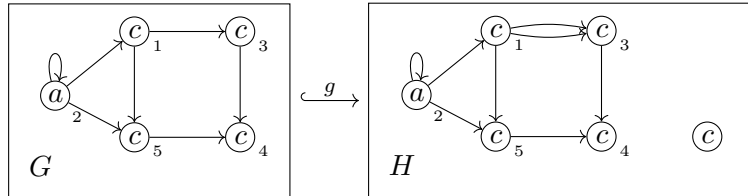


Figure 2.1: A graph morphism

2.2 Nested graph conditions

We can combine graph morphisms with first-order logic to define nested graph conditions (Habel and Pennemann, 2009). These conditions contain quantifiers over graph morphisms, providing an intuitive, graphical way to formulate graph properties.

Definition 2.3 (nested graph condition). *Nested (graph) conditions*, or short *conditions*, are inductively defined.

- (1) For any graph P , **true** is a condition over P .
- (2) For an injective morphism $a: P \hookrightarrow C$ and a condition c over C , $\exists(a, c)$ is a condition over P .
- (3) For conditions c, c' over P , $\neg c$ and $c \wedge c'$ are conditions over P .

A *constraint* is a condition over the empty graph \emptyset . △

Remark. In contrast to (Habel and Pennemann, 2009), we consider finite nested conditions only, i.e. conditions with finite conjunctions.

Abbreviations. To facilitate the use of nested conditions, we define some abbreviations, similar to logics.

- $\#$ abbreviates $\neg \text{exists}$,
- $\exists a$ abbreviates $\exists(a, \text{true})$,
- $\forall(a, c)$ abbreviates $\neg \exists(a, \neg c)$,
- **false** abbreviates $\neg \text{true}$,
- $c \vee c'$ abbreviates $\neg(\neg c \wedge \neg c')$.
- The domain of a morphism may be omitted if no confusion arises: $\exists(C)$ or $\exists C$ can replace $\exists(P \rightarrow C)$ in this case.

Example 2.2. The nested condition

$$\exists(\emptyset \hookrightarrow \textcircled{a}_1, \#(\textcircled{a}_1 \hookrightarrow \textcircled{a}_1 \xrightarrow{e} \textcircled{c}_2, \text{true}))$$

expresses the property “There exists an a-labeled node, which has no e-labeled, outgoing edge to a c-labeled node”. With the above abbreviations, this can be shortened to

$$\exists(\textcircled{a}_1, \#(\textcircled{a}_1 \xrightarrow{e} \textcircled{c}_2)). \quad \diamond$$

We now give a formal semantics for nested conditions.

Definition 2.4 (satisfaction of nested conditions). The *satisfaction* of a nested condition c over P by a morphism $p: P \rightarrow G$, written $p \models c$, is inductively defined as follows.

- (1) p satisfies **true**.
- (2) p satisfies $\exists(a, c)$ for a morphism $a: P \hookrightarrow C$ if there is an injective morphism $q: C \hookrightarrow G$ such that $q \circ a = p$ and q satisfies c .

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

(3) p satisfies $\neg c$ if p does not satisfy c . p satisfies $c \wedge c'$ if p satisfies c and c' .

A graph G *satisfies* a constraint c if the morphism $\emptyset \hookrightarrow G$ satisfies c . We write $G \models c$ to denote that G satisfies c . \triangle

Example 2.3. The nested condition in Example 2.2 is satisfied by every graph which has an a -labeled node without an e -labeled edge to a c -labeled node. \diamond

Remark. Non-injective nested conditions and non-injective semantics can be defined with non-injective morphisms a and q in rule (2). As shown by (Habel and Pennemann, 2009, Fact 6), the expressiveness is not changed by this choice. In the following, we only consider injective nested conditions.

The expressive power of nested conditions can be compared with traditional logical formulas on graphs.

Fact 2.1 (Habel and Pennemann (2009)). Nested graph conditions are expressively equivalent to first-order graph formulas.

In (Gaifman, 1982) it is shown that first-order graph formulas can only express local properties. However, many interesting properties of graphs are non-local. For example, nested conditions are unable to express the properties “the graph contains an arbitrary-length path from a node to another node”, “the graph is cycle-free” or modulo-counting properties like EVEN (Libkin, 2004, p. 24) stating “the graph has an even number of nodes”.

2.3 Graph transformation

We now explain the double-pushout (DPO) approach to graph transformation, as explained in (H. Ehrig, K. Ehrig, Prange, et al., 2006, Chapter 1.2.1). Some knowledge of category theory, especially pushouts and pullbacks, might help the understanding and can be found e.g. in the appendix of (H. Ehrig, K. Ehrig, Prange, et al., 2006). For more details on category theory in general, consider e.g. (Adámek et al., 2004; Simmons, 2011).

Definition 2.5 (graph transformation rule). A *plain (graph transformation) rule* $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$ is a pair of injective morphisms l, r and a common domain K called *interface*. L is called the *left-hand side* and R the *right-hand side*. A *left (right) application condition* is a condition over L (R). A *(graph transformation) rule* $\rho = \langle p, ac_L, ac_R \rangle$ consists of a plain rule p together with a left and a right application condition ac_L and ac_R , respectively.

Given a plain rule $p = \langle L \leftarrow K \hookrightarrow R \rangle$ and an injective morphism $m: L \hookrightarrow G$, a *direct derivation* consists of two pushouts (1) and (2). We write $G \Rightarrow_{p,m,m'} H$, $G \Rightarrow_p H$, or short $G \Rightarrow H$. Morphism m is called *match* and m' is called *comatch*. Given a rule $\rho = \langle p, ac_L, ac_R \rangle$, there is a direct derivation $G \Rightarrow_{\rho,m,m'} H$ if $G \Rightarrow_{p,m,m'} H$, $m \models ac_L$, and $m' \models ac_R$. A *derivation* is a sequence of direct derivations.

$$\begin{array}{ccccc}
 L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 \downarrow m & & \downarrow & & \downarrow m' \\
 G & \xrightarrow{\quad} & D & \xrightarrow{\quad} & H
 \end{array}
 \quad (1) \quad (2)$$

△

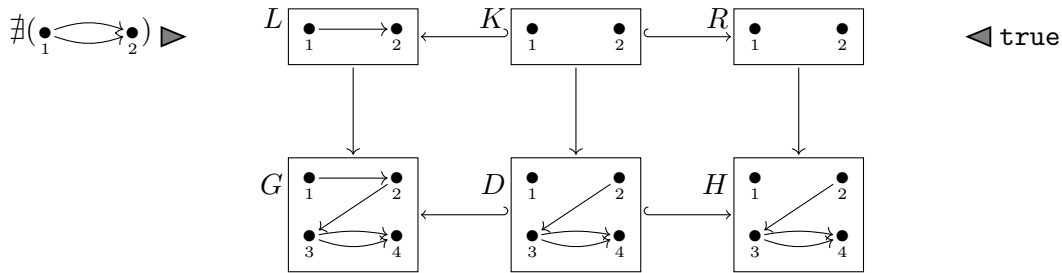
For brevity, the interface graph K of a rule can be omitted, writing $L \Rightarrow R$ instead of $\langle L \leftarrow K \hookrightarrow R \rangle$, provided that the corresponding nodes in L and R are marked by indices to convey the mapping of items. A rule $\rho = \langle p, ac_L, ac_R \rangle$ can also be written as $ac_L \blacktriangleright p \blacktriangleleft ac_R$, with the application conditions indicated by gray triangles. For $ac_R = \text{true}$, we write $\langle p, ac_L \rangle$ instead of $\langle p, ac_L, \text{true} \rangle$.

Pushout (1) dictates that the match m must satisfy the *dangling condition*. This means that any node to be deleted (i.e. in $G - D$) must not have an edge to a node which is not deleted (i.e. a node in D). Otherwise, trying to construct pushout (1) would leave D with “dangling” edges which have no source or target node and D would not be a proper graph in \mathcal{G} .

Remark. Application of a rule can also be explained with the use of set theory. The set-theoretic construction yields the same results as the category-theoretic one. The construction proceeds in two steps: (1) remove all vertices and edges in $m(L - K)$ from G . The resulting structure $D = (G - m(L - K))$ might not be a graph. In this case, the dangling condition is not met and the application fails. If D is a valid graph, proceed by (2) gluing graph D together with $R - K$ to obtain graph $H = D + (R - K)$.

Remark. In Definition 2.5, the match m is an injective morphism. One can also use arbitrary matches instead: As shown by (Habel, Müller, et al., 2001), both approaches are expressively equivalent.

Example 2.4. The rule $\text{delEdge} = \langle \langle \bullet_1 \rightarrow \bullet_2 \leftarrow \bullet_1 \bullet_2 \leftrightarrow \bullet_1 \bullet_2 \rangle, \nexists(\bullet_1 \rightleftarrows \bullet_2), \text{true} \rangle$ deletes an edge between two nodes 1 and 2 if there is no second edge from node 1 to node 2. The application of this rule on a graph is shown below. The left application condition forbids the deletion of the edges from 3 to 4.



◇

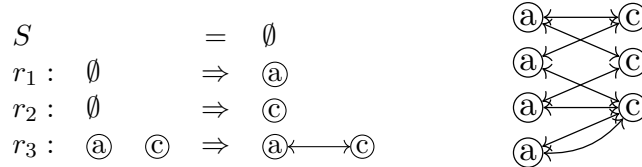
Rules can be combined to form graph transformation systems, graph grammars and graph programs.

Definition 2.6 (graph transformation system, graph grammar). A *graph transformation system* is a finite set R of rules. Together with a starting graph S , a graph transformation system R forms a *graph grammar* $GG = (R, S)$. The *language* $\mathcal{L}(GG)$ of a graph grammar $GG = (R, S)$ consists of all graphs that can be derived from S using the rules from R in an arbitrary number of steps:

$$\mathcal{L}((R, S)) = \{G \mid S \Rightarrow_R^* G\}.$$

△

Example 2.5. Regard the graph transformation system $\{r_1, r_2, r_3\}$ displayed below left. Together with the starting graph $S = \emptyset$, it constitutes the graph grammar $GG = (\{r_1, r_2, r_3\}, \emptyset)$. The language $\mathcal{L}(GG)$ is the set of bipartite graphs (e.g. the one displayed below right) consisting of two disjoint sets M_a, M_c of a- and c-labeled nodes, respectively, where each edge connects a node from M_a with a node from M_c .



◇

Graph programs are defined as in (Habel and Pennemann, 2009). The combination of rules into programs allows for a better control of the program flow between execution of the rules.

Definition 2.7 (graph program (Habel and Plump, 2001)).

Graph programs are defined as in (Habel and Plump, 2001):

1. Every rule is a program.
2. Every finite set S of programs is a program.
3. Given programs P and Q , sequential composition $(P; Q)$ and as-long-as-possible iteration $P \downarrow$ are programs.

We also write P^i for the i -fold sequential composition $\underbrace{P; \dots; P}_{i \text{ times}}$ of program P . △

Example 2.6. The program **reverse** \downarrow ; **cleanup** \downarrow with

$$\begin{aligned} \text{reverse} &= \textcircled{a} \longrightarrow \textcircled{c} \Rightarrow \textcircled{a} \xrightarrow{\text{rev}} \textcircled{c} \text{ and} \\ \text{cleanup} &= \textcircled{a} \xrightarrow{\text{rev}} \textcircled{c} \Rightarrow \textcircled{a} \leftarrow \textcircled{c} \end{aligned}$$

reverses all unlabeled edges from a- to c-labeled nodes. First, **reverse** \downarrow deletes each unlabeled edge and replaces it with a **rev**-labeled edge in the opposite direction. When all unlabeled edges are reversed and labeled, **cleanup** \downarrow replaces the temporary **rev**-labeled edges with unlabeled ones. ◇

Definition 2.8 (semantics of graph programs). The semantics of a program is a binary relation $\llbracket P \rrbracket \subseteq \mathcal{G} \times \mathcal{G}$. For every rule ρ , every non-empty set S of programs, and every pair of programs P and Q ,

1. $\llbracket \rho \rrbracket = \{\langle G, H \rangle \mid G \Rightarrow_\rho H\}$,
2. $\llbracket S \rrbracket = \bigcup_{P \in S} \llbracket P \rrbracket$,
3. $\llbracket P; Q \rrbracket = \llbracket Q \rrbracket \circ \llbracket P \rrbracket$, where \circ is the composition of relations,
4. $\llbracket P \Downarrow \rrbracket = \{\langle G, H \rangle \mid \langle G, H \rangle \in \llbracket P \rrbracket^* \wedge \nexists M. \langle H, M \rangle \in \llbracket P \rrbracket\}$,
where $\llbracket P \rrbracket^*$ is the reflexive-transitive closure of \circ .

We also write $G \Rightarrow_P H$ instead of $(G, H) \in \llbracket P \rrbracket$. △

2.4 A containment operator for nested conditions

By the nature of nested conditions, in a condition of the form $\exists(P \hookrightarrow C, c)$, the codomain C is usually bigger than the domain P . The deeper the nesting gets, the bigger the codomain gets, although only a small part of the graph is actually changing. One might wonder whether a *containment operator* $\exists(P \sqsupseteq C, c)$ that “forgets” part of the graph might be useful for nested conditions, as introduced in (Arendt, Habel, et al., 2014). This could reduce the size of a condition, e.g. instead of

$$\forall(\bullet_1 \bullet_2 \bullet_3, \exists(\bullet_1 \rightarrow \bullet_2 \bullet_3)) \vee \exists(\bullet_1 \leftarrow \bullet_2 \bullet_3) \vee \exists(\bullet_1 \bullet_2 \rightarrow \bullet_3) \vee \exists(\bullet_1 \bullet_2 \leftarrow \bullet_3) \vee \exists(\bullet_1 \bullet_2 \curvearrowright \bullet_3) \vee \exists(\bullet_1 \curvearrowleft \bullet_2 \bullet_3))$$

meaning “For every triple of nodes, there is an edge between two of them”, we could write

$$\forall(\bullet_1 \bullet_2 \bullet_3, \exists(\bullet_1 \bullet_2 \bullet_3 \sqsupseteq \bullet_4 \bullet_5, \exists(\bullet_4 \rightarrow \bullet_5))).$$

Before presenting this abbreviation, we repeat the definition of construction Shift from (H. Ehrig et al., 2012).

Construction (Shift). Let $\text{Shift}(b, \exists(a, c)) = \bigvee_{a', b' \in \mathcal{F}} \exists(a', \text{Shift}(b', c))$, where $\mathcal{F} = \{(a', b') \mid (a', b') \text{ jointly surjective, } b \text{ injective, (1) commutes}\}^2$.

$$\begin{array}{ccc} P & \xleftarrow{a} & C \\ \downarrow b & (1) & \downarrow b' \\ P' & \xleftarrow{a'} & C' \end{array}$$

For Boolean conditions and **true**, the construction is straightforward: $\text{Shift}(b, \mathbf{true}) = \mathbf{true}$, $\text{Shift}(b, \neg c) = \neg \text{Shift}(c)$ and $\text{Shift}(b, c \wedge c') = \text{Shift}(b, c) \wedge \text{Shift}(b, c')$. 🍃


With the help of Shift, we can reduce nested conditions with containment operator into “pure” nested conditions, i.e. without containment operator.

²For graphs, the set \mathcal{F} is always non-empty, as one can always construct a pushout.

Fact 2.2 (nested + containment \equiv nested). There is a transformation Pure from nested conditions with containment operator into equivalent (pure) nested conditions.

Construction.

$$\text{Pure}(c) = \begin{cases} \bigvee_{b \in \mathcal{B}} \text{Shift}(b, c') & \text{if } c = \exists(P \sqsupseteq C, c') \\ c & \text{otherwise} \end{cases}$$

where \mathcal{B} is the set of injective morphisms from C to P . 

Example 2.7. Regard the condition $c = \exists(\bullet_1 \bullet_2 \sqsupseteq \bullet_4, \exists(\bullet_4 \rightarrow \bullet_3))$, meaning “There are two nodes 1, 2, and one of them has an outgoing edge to another node”. We transform it into a pure nested condition:

$$\text{Pure}(c) = \exists\left(\bullet_1 \bullet_2, \exists(\bullet_1 \bullet_2 \rightarrow \bullet_3) \vee \exists(\bullet_3 \leftarrow \bullet_1 \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_2) \vee \exists(\bullet_1 \leftarrow \bullet_2)\right) \quad \diamond$$

Bibliographic notes

In the literature, many formalisms have been proposed to express properties of graphs as constraints and as application conditions for graph transformation rules.

The concept of application conditions for rules of a graph transformation system was introduced by (H. Ehrig and Habel, 1986). Conditions of the form $\neg\exists(a)$ were first introduced in (Habel, Heckel, et al., 1996) as *negative application conditions*, and (Heckel and Wagner, 1995) introduced *consistency conditions* of the form $\forall(P, \exists(P \rightarrow C))$. (Koch et al., 2005) investigated how sets of positive and negative conditions can be checked for consistency. A detailed study on negative application conditions can be found in (Lambers, 2010). Application conditions and constraints were combined with Boolean operations and merged into the first-order equivalent *nested graph conditions* by (H. Ehrig, K. Ehrig, Habel, et al., 2006; Habel, Pennemann, and Rensink, 2006; Habel and Pennemann, 2009).

An extensive study of nested conditions has been performed in (Pennemann, 2009). The results include basic transformations of conditions over rules and the computation of weakest preconditions (and strongest postconditions) for a specification.

Chapter 3

HR* graph conditions

Contents

3.1	HR systems	15
3.2	HR* graph conditions	21
3.3	Decidability of HR* conditions	24
3.4	Case study: car platooning	25

In this chapter, we introduce the main concept of this thesis: HR* *conditions*. As noted in the previous chapter, nested conditions cannot express paths of arbitrary length or other non-local conditions. HR* conditions use hyperedge replacement systems, short *HR systems*, to remedy this deficiency while retaining the intuitive visual representation of graphs in the conditions. We show that HR* conditions can be used to express the existence or non-existence of paths or cycles, and that the problem whether a given graph satisfies a given condition is decidable. The contents of this chapter are oriented on (Habel and Radke, 2010; Radke, 2013).

3.1 HR systems

In order to express non-local structures of arbitrary size in a finite way, we need some kind of placeholder for those structures. We thus enhance graphs with hyperedges, which serve as placeholders and can later be replaced by graphs of arbitrary size.

Graphs with variables are similar to graphs as in Definition 2.1. They consist of labeled nodes, edges and hyperedges. Edges have one source and one target and are labeled by a symbol of an alphabet; hyperedges have an arbitrary long sequence of attachment nodes (indicated by *tentacles* between the hyperedge and the attachment node) and are labeled by ranked variables.

Definition 3.1 (graph with variables). Let $L = L_V \uplus L_E \uplus \text{Var}$ be a fixed, finite label alphabet where Var is a set of variables with a mapping $\text{rank}: \text{Var} \rightarrow \mathbb{N}^1$ assigning a rank to each variable.

A graph (with variables) over L is a system $G = (V_G, E_G, Y_G, s_G, t_G, \text{att}_G, \text{lv}_G, \text{le}_G, \text{ly}_G)$ consisting of finite sets V_G , E_G , and Y_G of nodes (or vertices), edges and hyperedges, source and target functions $s_G, t_G: E_G \rightarrow V_G$, an attachment function $\text{att}_G: Y_G \rightarrow V_G^*$ ², and labeling functions $\text{lv}_G: V_G \rightarrow L_V$, $\text{le}_G: E_G \rightarrow L_E$, $\text{ly}: Y_G \rightarrow \text{Var}$ such that, for all $y \in Y_G$, $|\text{att}_G(y)| = \text{rank}(\text{ly}_G(y))$. We call the set of all graphs with variables \mathcal{G}_{Var} and write $D_G = V_G \cup E_G \cup Y_G$ to denote all items of a graph. A graph G is empty, denoted \emptyset , if $V_G = \emptyset$ and $Y_G = \emptyset$. We denote the disjoint union of two graphs G, H by $G + H$ and their difference by $G - H$ for any $H \subseteq G$. For a graph G and item o , let $G + o$ designate a graph consisting of G with o disjointly added to the graph (if o is a (hyper)edge, it is attached to nodes in G). \triangle

A hyperedge $y \in Y$ has $\text{rank}(y)$ tentacles. The i^{th} tentacle of a variable, where $1 \leq i \leq \text{rank}(y)$, connects the variable with the i^{th} node of the sequence $\text{att}(y)$. This number i is also called the index of a tentacle, and the i^{th} tentacle of y can be referred to with $\text{att}(y)_i$.

Notation. Nodes are drawn by circles carrying the node label inside. Edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow. Hyperedges are drawn as boxes with attachment nodes where the i^{th} tentacle has its number i written next to it and is attached to the i^{th} attachment node and the label of the hyperedge is inscribed in the box. Nodes with the invisible \square label are drawn as points (\bullet). For visibility reasons, we may abbreviate hyperedges of rank 2 by writing $\bullet \xrightarrow{x} \bullet$ instead of $\bullet^1 \text{---} \square \text{---}^2 \bullet$ (i.e. as an x -labeled arrow going from the first to the second attachment node).

Example 3.1. Consider the graphs G, H in Figure 3.1 over the label alphabet $L = \{a, c\} \uplus \{\square\} \uplus \text{Var}$ where the symbol \square stands for the invisible edge label and is not drawn and $\text{Var} = \{u, v\}$ is a set of variables that have rank 4 and 2, respectively. The graph G contains five nodes with the labels a and c , drawn as circles with the label inside, seven edges with (invisible) label \square , drawn as arrows, and one hyperedge of rank 4 with label u , drawn as a square with the label inside. Additionally, the graph H contains a node, an edge, and a hyperedge of rank 2 with label v . See also Figure 2.1 for a similar example without variables. \diamond

¹ \mathbb{N} denotes the set of natural numbers, including 0.

²The *-operator denotes a sequence of arbitrary length. This also includes hyperedges with zero tentacles.

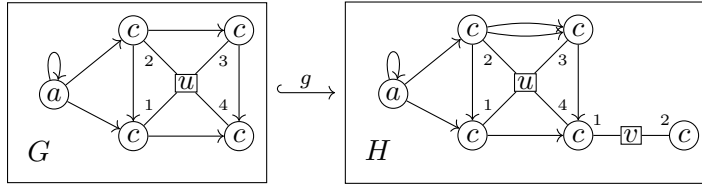


Figure 3.1: A morphism between two graphs with variables

Remark. The definition extends the well-known definition of graphs (H. Ehrig, 1979) by the concept of hyperedges in the sense of (Habel, 1992). Graphs with variables form a special case of labeled hypergraphs. While edge variables resemble arbitrary hyperedges, the “common” edges in E_G are hyperedges with two tentacles. The labeling functions for hyperedges and edges can be combined into a single hyperedge labeling function, keeping the label sets for hyperedges and common edges disjoint.

Graph morphisms with variables consist of mappings between the sets of nodes, edges and hyperedges of graphs, and are defined analogously to the graph morphisms in Definition 2.2.

Definition 3.2 (graph morphism with variables). Let G and H be graphs with variables. A *graph morphism (with variables)*, short *morphism*, $g: G \rightarrow H$ consists of functions $g_V: V_G \rightarrow V_H$, $g_E: E_G \rightarrow E_H$, and an injective mapping $g_Y: Y_G \hookrightarrow Y_H$ that preserve sources, targets, attachment nodes and labels, i.e. $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $att_H \circ g_Y = g_V^* \circ att_G$ ³, $lv_H \circ g_V = lv_G$, $le_H \circ g_E = le_G$, and $ly_H \circ g_Y = ly_G$. The graph G is called the *domain* of g , denoted $\text{Dom}(g)$. H is called the *codomain* of g , denoted $\text{Ran}(g)$. The *composition* $h \circ g$ of g with a graph morphism $h: H \rightarrow H'$ consists of the composed functions $h_V \circ g_V$, $h_E \circ g_E$, and $h_Y \circ g_Y$.

A morphism g is *injective* (*surjective*) if g_V , g_E , and g_Y are injective (surjective), and an *isomorphism* if it is both injective and surjective. Injective/surjective morphisms are also called *monomorphisms* and *epimorphisms*, respectively. Let \mathcal{M} and \mathcal{E} be the set of all injective and surjective morphisms, respectively. If g is an isomorphism, its domain and codomain G and H are *isomorphic*, which is denoted by $G \cong H$. An injective graph morphism $m: G \hookrightarrow H$ is an *inclusion*, written $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$ and $Y_G \subseteq Y_H$. For a graph G , the *identity* $\text{id}_G: G \rightarrow G$ consists of the identities id_{G_V} , id_{G_E} , and id_{G_Y} on G_V , G_E , and G_Y , respectively. \triangle

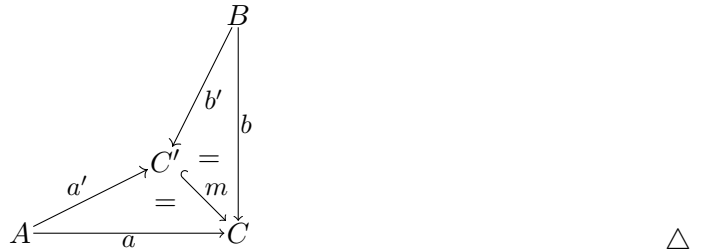
Example 3.2. The graph morphism $g: G \hookrightarrow H$ in Figure 3.1 maps all nodes, edges and hyperedges in G to corresponding objects in H , as indicated by the small numbers next to the nodes. \diamond

³For a mapping $g: A \rightarrow B$, the free symbol-wise extension $g^*: A^* \rightarrow B^*$ is defined by $g^*(a_1 \dots a_k) = g(a_1) \dots g(a_k)$ for all $k \in \mathbb{N}$.

Notation. Arbitrary graph morphisms are drawn by the usual arrows “ \rightarrow ”; the use of “ \hookrightarrow ” indicates an injective graph morphism. The actual mapping of items is conveyed by indices, if necessary. The mapping of nodes by a morphism is displayed by adding small numbers to the side of the nodes; a node in the domain is mapped to the node with the same number in the codomain.

Further properties deal with pairs of morphisms.

Definition 3.3 (spans and cospans). A pair of morphisms (a, b) is called a *span*, written $A \xleftarrow{a} C \xrightarrow{b} B$, if a and b have a common domain C . Likewise, (a, b) is called a *cospan*, written $A \xrightarrow{a} C \xleftarrow{b} B$, if a and b have a common codomain C . A cospan $A \xrightarrow{a} C \xleftarrow{b} B$ is *jointly surjective* if for each item $v \in C$, there is a preimage $u \in A$ with $a(u) = v$ or $u \in B$ with $b(u) = v$. Let \mathcal{E}' denote the class of jointly surjective morphism pairs. For a cospan $A \xrightarrow{a} C \xleftarrow{b} B$, an \mathcal{E}' - \mathcal{M} *pair factorization* is a cospan $A \xrightarrow{a'} C' \xleftarrow{b'} B$ and a morphism $m: C' \hookrightarrow C$ with $(a', b') \in \mathcal{E}'$, $m \in \mathcal{M}$ and $a = m \circ a'$ and $b = m \circ b'$ (see diagram below). A *partial morphism* $P \rightarrow C$ is a span $P \leftarrow I \rightarrow C$.



Hyperedges do not only play a static part as building blocks of graphs with variables, but also a more dynamic part as place holders for graphs. Before a graph can take the place of a hyperedge, it needs some preparation. While a hyperedge is attached to a sequence of attachment nodes, a graph has to be equipped with a sequence of nodes.

Definition 3.4 (pointed graph). A *pointed graph* $\langle G, \text{pin}_G \rangle$ is a graph with variables G together with a sequence $\text{pin}_G = v_1 \dots v_n$ of pairwise disjoint nodes from G called *pinpoints*. We write $\text{rank}(\langle G, \text{pin}_G \rangle)$ for the number n of nodes in pin_G . For $x \in \text{Var}$ with $\text{rank}(x) = n$, x^\bullet denotes the pointed graph with the nodes v_1, \dots, v_n , an x -labeled hyperedge attached to $v_1 \dots v_n$, and pinpoints $v_1 \dots v_n$, and $\langle x \rangle$ denotes x^\bullet with the hyperedge removed, i.e. the pointed graph consisting of the node sequence $v_1 \dots v_n$ only (see Figure 3.2). △



Figure 3.2: Graphs x^\bullet and $\langle x \rangle$ for an x -labeled hyperedge with $\text{rank}(x) = n$.

With hyperedges and pointed graphs, we can now define the replacement of a hyperedge in some graph G by a graph R . The replacement process connects the replacement graph R with G , according to the tentacles of the hyperedge. Hyperedge replacement systems can be combined with a start graph to form a grammar, generating a language of graphs, see Chapter 2.3.

Definition 3.5 (hyperedge replacement system). A *hyperedge replacement system* \mathcal{R} , short *HR system*, is a finite set of replacement pairs of the form x/R , also written x^\bullet/R , where $x \in \text{Var}$ is a variable and R a pointed graph with $\text{rank}(x) = \text{rank}(R)$.

Given a graph G , the *application* of the replacement pair $x/R \in \mathcal{R}$ to a hyperedge y with label x proceeds in two steps (see Figures 3.3 and 3.4):

1. Remove the hyperedge y from G , yielding the graph $G - \{y\}$ ⁴.
2. Construct the disjoint union $(G - \{y\}) + R$ and fuse the i^{th} node in $\text{att}_G(y)$ with the i^{th} attachment point of R , for $i = 1, \dots, \text{rank}(y)$, yielding the graph H . \triangle

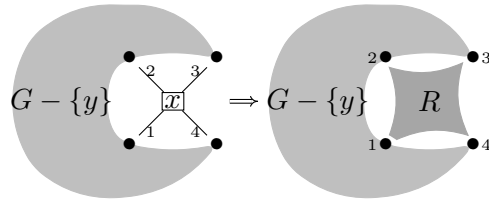
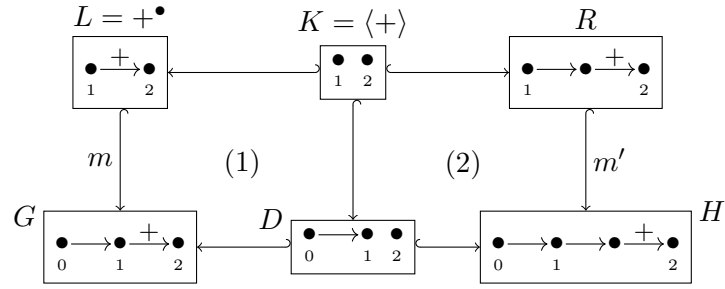


Figure 3.3: Application of replacement pair x/R .

Rules can also be represented in Backus-Naur form $x ::= R_1 \mid R_2 \mid \dots R_n$.

Remark. A hyperedge replacement rule $+/R$ can also be formulated as a special form of double pushout graph transformation rules (see Chapter 2.3), where $L = +^\bullet$ and $K = \langle + \rangle$, so rules have the form $\langle +^\bullet \leftrightarrow \langle + \rangle \hookrightarrow R \rangle$.

⁴For a graph G and a set $Y \subseteq Y_G$ of hyperedges, $G - Y$ denotes G without the hyperedges in Y .


 Figure 3.4: Application of rule $+/\overset{\bullet}{1} \rightarrow \overset{\bullet}{2}$.

Definition 3.6 (derivation). G directly derives H by $x/R \in \mathcal{R}$, denoted by $G \Rightarrow_{x/R} H$ or $G \Rightarrow_{\mathcal{R}} H$. A sequence of direct derivations $G \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} H$ is called a *derivation* from G to H , denoted by $G \Rightarrow_{\mathcal{R}}^* H$. For every variable x , $\mathcal{R}(x) = \{G \in \mathcal{G}_{\text{Var}} \mid x^{\bullet} \Rightarrow_{\mathcal{R}}^* G\}$ denotes the set of all graphs derivable from x^{\bullet} by \mathcal{R} . \triangle

Example 3.3. Starting from $+^{\bullet} = \overset{\bullet}{1} \rightarrow \overset{\bullet}{2}$, the hyperedge replacement system \mathcal{R} with the rules given in Backus-Naur form

$$\overset{\bullet}{1} \rightarrow \overset{\bullet}{2} ::= \overset{\bullet}{1} \rightarrow \overset{\bullet}{2} \mid \overset{\bullet}{1} \rightarrow \overset{\bullet}{\bullet} \rightarrow \overset{\bullet}{2}$$

can derive the set of all directed paths from node 1 to node 2. \diamond

In HR* conditions, we substitute all variables by graphs, which are generated according to a hyperedge replacement system. Contrary to the replacement process defined above, this is based on variables instead of individual hyperedges, i.e. all hyperedges with the same label are replaced by isomorphic graphs *simultaneously*.

Definition 3.7 (substitution). A *substitution* induced by a hyperedge replacement system \mathcal{R} is a mapping $\sigma: \text{Var} \rightarrow \mathcal{G}$ with $\sigma(x) \in \mathcal{R}(x)$ for all $x \in \text{Var}$. The set of all substitutions induced by \mathcal{R} is denoted by Σ . The substitution of all hyperedges with label x in a graph G by $\sigma(x)$ is obtained from G by applying the rule $x/\sigma(x)$ to every hyperedge with label x in G . Application of σ to a graph G , denoted $\sigma(G)$ or G^{σ} , is obtained by substitution of all hyperedges in G according to σ . \triangle

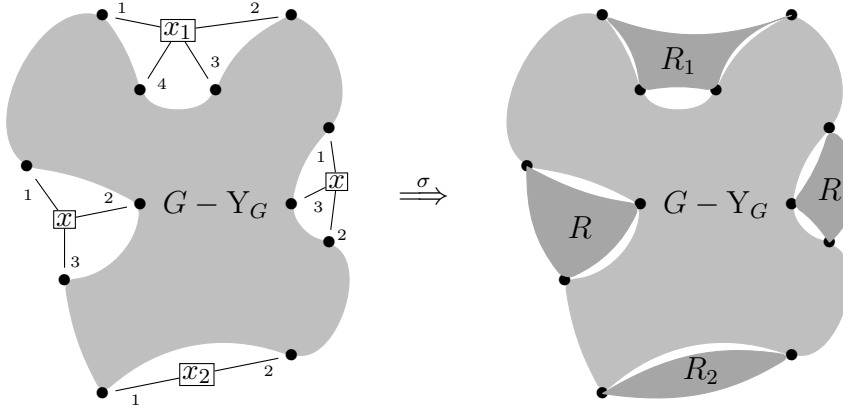


Figure 3.5: Substitution of hyperedges.

The simultaneous substitution of hyperedges plays an important role in the next part, defining the semantics of HR* conditions.

3.2 HR* graph conditions

Having defined hyperedge replacement, we can now extend nested conditions to contain variables, which are replaced by graphs according to a hyperedge replacement system. This provides a finite way to express structures of arbitrary size, with the variables acting as placeholders. HR* conditions combine graphical notation with the usual first-order logical operators. In addition, the *containment operator* \sqsubseteq provides a means to focus on a subgraph of a previously-defined graph and formulate constraints for this subgraph.

Assumption. In this PhD thesis, we consider injective HR* conditions, i.e. HR* conditions with injective morphisms, and injective / \mathcal{M} -satisfaction. For technical reasons, in Chapters 4 and 5, we also consider arbitrary / \mathcal{A} -satisfaction to prove results on the expressive power of HR* conditions.

Definition 3.8 (HR* condition). HR* (graph) conditions over an HR system \mathcal{R} , short *conditions*, are inductively defined:

1. For a graph P , **true** is a condition over P .
2. For an injective morphism $a: P \hookrightarrow C$ and a condition c over C , $\exists(a, c)$ is a condition over P .
3. For an injective partial morphism $a: C \rightarrow P$ and a condition c over C , $\exists(P \sqsubseteq C, c)$ is a condition over P ⁵.
4. For conditions c, c' over P , $\neg c$ and $c \wedge c'$ are conditions over P .

⁵Morphism a is usually conveyed by indices on the nodes of P and C .

A (graph) constraint is an HR* condition over the empty graph \emptyset .

HR* conditions c over \mathcal{R} are denoted by $\langle c, \mathcal{R} \rangle$, or c if \mathcal{R} is clear from the context. \triangle

Notation. The following abbreviations are used: $\exists(a)$ abbreviates $\exists(a, \mathbf{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$ and likewise for $\forall(P \sqsubseteq C, c)$, **false** abbreviates $\neg\mathbf{true}$, and $c \vee c'$ abbreviates $\neg(\neg c \wedge \neg c')$. The domain of a morphism may be omitted if no confusion arises: $\exists(C, c)$ can replace $\exists(P \rightarrow C, c)$ in this case. Nodes with an arbitrary label are represented by \circ , e.g. for node label alphabet L_V , $\exists(\circ)$ abbreviates $\bigvee_{a \in L_V} \exists(@_1)$. For readability, we sometimes omit parentheses and write e.g. $\exists_1 \circ \xrightarrow{+} \circ_2$ instead of $\exists(\circ \xrightarrow{+} \circ_2)$.

Example 3.4. The following example shows an HR* condition intuitively expressing the property “There exists a path from node 1 to node 2”.

$$\exists(\circ_1 \xrightarrow{+} \circ_2) \text{ with } \circ_1 \xrightarrow{+} \circ_2 ::= \circ_1 \longrightarrow \circ_2 \mid \circ_1 \longrightarrow \circ \xrightarrow{+} \circ_2.$$

HR* conditions can also express the fact that a graph contains an even number of nodes:

$$\exists(\boxed{\circ}, \#(\boxed{\circ} \bullet)) \text{ with } \boxed{\circ} ::= \emptyset \mid \boxed{\circ} \bullet.$$

Here, the zero-tentacle hyperedge $\boxed{\circ}$ is substituted by a discrete graph with an even number of nodes, and the condition states that the graph must contain this as a subgraph and no further node. \diamond

A more complicated example shows the use of the \sqsubseteq operator.

Example 3.5. The following HR* condition has the intuitive meaning “There is a path, and every inner node (i.e. every node that is part of the path except the first and the last) has at least three more outgoing edges, in addition to the path’s edges”.

$$\begin{aligned} & \exists(\circ_1 \xrightarrow{+} \circ_2, \quad \text{there is a path} \\ & \forall(\circ_1 \xrightarrow{+} \circ_2 \sqsubseteq \circ_1 \xrightarrow{+_1} \circ_3 \xrightarrow{+_2} \circ_2, \quad \text{where every inner node} \\ & \exists(\circ_1 \xrightarrow{+_1} \circ_3 \xrightarrow{+_2} \circ_2)) \quad \text{has (at least) 3 more outgoing edges} \\ & \text{with } \circ_1 \xrightarrow{+} \circ_2 ::= \circ_1 \longrightarrow \circ_2 \mid \circ_1 \longrightarrow \circ \xrightarrow{+} \circ_2 \text{ and } \circ_1 \xrightarrow{+_1} \circ_2, \circ_1 \xrightarrow{+_2} \circ_2 \text{ defined likewise.} \end{aligned}$$

The \sqsubseteq operator is used here to “peek into” the graph generated from the $+$ -labeled hyperedge. It splits the path from node 1 to 2 into two subpaths from 1 to 3 and from 3 to 2. The universal quantifier ensures that the “three more outgoing edges” property holds for every such decomposition of the path. We use three different hyperedge symbols to represent three paths of independent length. \diamond

To understand how the above examples are evaluated, we define the formal semantics of HR* conditions. An HR* condition is checked by first substituting the occurring hyperedges with graphs generated according to the HR system. The resulting condition can then be checked in a way similar to nested conditions.

Definition 3.9 (satisfaction of HR* conditions). Let $p: P \hookrightarrow G$ be an injective morphism with P, G containing no hyperedges. The *satisfaction* of a condition c , written $p \models c$, is inductively defined as follows.

1. p satisfies **true**.
2. p satisfies $\exists(P \xrightarrow{a} C, c)$ if there is a substitution σ with $\text{Dom}(\sigma) = \text{Var}(C)$ and an injective morphism $q: C^\sigma \hookrightarrow G$ such that $q \circ a^\sigma = p$ ⁶ and q satisfies c^σ (left image below).

$$\begin{array}{ccc} \exists(P \xrightarrow{a^\sigma} C^\sigma, c^\sigma) & & \exists(P \xrightarrow{b} C^\sigma, c^\sigma) \\ \begin{array}{ccc} P & \xrightarrow{a^\sigma} & C^\sigma \\ p \searrow & = & \nearrow q \\ & G & \end{array} & & \begin{array}{ccc} P & \xrightarrow{b} & C^\sigma \\ p \searrow & = & \nearrow q \\ & G & \end{array} \end{array}$$

3. p satisfies $\exists(P \sqsupseteq C, c)$ with partial morphism $C \xrightarrow{a} P$ if there are a substitution σ with $\text{Dom}(\sigma) = \text{Var}(C)$, an injective morphism $b: C^\sigma \hookrightarrow P$ with b restricted to $C - Y_C$ commuting with a and an injective morphism $q: C^\sigma \hookrightarrow G$ such that $p \circ b = q$ and q satisfies c^σ (right image above).
4. p satisfies $\neg c$ if p does not satisfy c . p satisfies $c \wedge c'$ if p satisfies c and c' .

A morphism p satisfies a condition c by substitution σ , written $p \models^\sigma c$, iff $p \models c^\sigma$. A graph G *satisfies* a constraint c if the morphism $\emptyset \hookrightarrow G$ satisfies c . We write $G \models c$ to denote that a graph G satisfies c . △

Example 3.6. Recall the HR* condition from Example 3.4 expressing the property “There exists a path from node 1 to node 2”:

$$c = \exists(\bullet_1 \xrightarrow{+} \bullet_2) \text{ with } \mathcal{R} = \bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2.$$

We check whether the graph $G = \bullet_a \xrightarrow{b} \bullet_c \xrightarrow{d} \bullet_d$ satisfies c .

We first expand the abbreviations and substitute G by morphism $p: \emptyset \rightarrow G$:

$$p \models \exists(\emptyset \xrightarrow{a} \bullet_1 \xrightarrow{+} \bullet_2, \text{true}).$$

By Definition 3.9, p satisfies the condition if there are a substitution σ and an injective morphism $q: (\bullet_1 \xrightarrow{+} \bullet_2)^\sigma \hookrightarrow G$ such that $q = p \circ a^\sigma$ and $q \models \text{true}$. Using the rules of the HR system, we expand $\bullet_1 \xrightarrow{+} \bullet_2 \Rightarrow_{\mathcal{R}}^* \bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \rightarrow \bullet_2$. This yields the condition

$$\exists(\emptyset \xrightarrow{a^\sigma} \bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \rightarrow \bullet_2, \text{true}).$$

⁶For $a: P \rightarrow C$, $a^\sigma: P \rightarrow C^\sigma$ is the morphism induced from a with $a^\sigma(o) = a(o)$ for every vertex or edge $o \in P$.

We define morphism q with $q(1)=a, q(3)=b, q(4)=c, q(2)=d$ and map the edges accordingly. Now $q = p \circ a^\sigma$ and, trivially, $q \models \text{true}$, so G satisfies c . \diamond

One might ask why the above semantics is based on the simultaneous substitution of variables and not on their individual replacement. Substitution makes it very easy to formulate an HR* condition for a property like “there are two paths of equal length between two nodes 1 and 2”:

$$\exists(\overset{+}{\underset{+}{\bullet_1 \rightleftarrows \bullet_2}}) \text{ with } \bullet_1 \overset{+}{\rightarrow} \bullet_2 ::= \bullet_1 \rightarrow \bullet_2 \mid \bullet_1 \rightarrow \bullet \overset{+}{\rightarrow} \bullet_2$$

Substitution and replacement as basis for the semantics of HR* conditions are discussed in detail in Chapter 4.3.

Remark. HR* conditions extend both the HR conditions from (Habel and Radke, 2010) and the nested conditions from (Habel and Pennemann, 2009). The extension is straightforward, as the definition of HR conditions is equal to Definition 3.8 without item 3, and nested conditions are defined equally to Definition 3.8 without item 3 and without hyperedges. For nested conditions, it is possible to add a \sqsubseteq -operator, see Chapter 2.4.

3.3 Decidability of HR* conditions

We now show that the satisfaction of an HR* condition is decidable. This is based on the monotonicity of HR systems. A replacement system is *monotone* if $|x^\bullet| \leq |R|$ ⁷ for each rule $x^\bullet/R \in \mathcal{R}$, i.e. each derivation step increases the size of the derived graph.

Fact 3.1 ((Habel, 1992)). For every hyperedge replacement system, there is an equivalent monotone one.

Using the above fact, we show that checking the validity of a condition for a graph is decidable.

Theorem 3.1 (decidability of HR* conditions).

The validity problem for HR* conditions is decidable, i.e. there is an algorithm that determines, for a given HR* condition c and a graph G , whether $G \models c$.

Proof. Let c be a finite HR* condition with replacement system \mathcal{R} and $|G| = |V_G| + |E_G|$ be the size of a graph G . Without loss of generality, assume \mathcal{R} to be monotone, so we can enumerate the set $\mathcal{R}^n(C) = \{H \in \mathcal{G} \mid C \Rightarrow_{\mathcal{R}}^* H \wedge |H| \leq n\}$ of all graphs derivable from C by \mathcal{R} which are not bigger than n .

For graph morphisms $p: P \hookrightarrow G$ with $|G|=n$ and HR* conditions of the form $\exists(P \hookrightarrow C, c)$, we have $p \models \exists(P \hookrightarrow C, c) \Leftrightarrow p \models \exists(P \hookrightarrow C^\sigma, c^\sigma)$ for some substitution σ iff there is an injective morphism $q: C^\sigma \hookrightarrow G$ such that $p = a \circ q$. For any σ with $C^\sigma \geq |G|$, there

⁷For graph G , $|G|$ denotes G 's size, i.e. the sum of the number of G 's nodes, edges and hyperedges.

is no injective morphism $q: C^\sigma \hookrightarrow G$. Thus, it suffices to check $p \models \exists(P \hookrightarrow C^\sigma, c^\sigma)$ for every σ such that $|C^\sigma| \leq n$. The same argument applies for HR^* conditions of the form $\exists(P \sqsupseteq C, c)$. Thus, the validity problem for HR^* is decidable. \square

Example 3.7. Suppose we want to check whether the graph $G = \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$ satisfies the HR^* condition $\exists(\overset{+}{\bullet}_1 \rightarrow \bullet_2)$ with $\overset{+}{\bullet}_1 \rightarrow \bullet_2 ::= \bullet_1 \rightarrow \bullet_2 \mid \bullet_1 \rightarrow \bullet \overset{+}{\rightarrow} \bullet_2$. By the semantics from Definition 3.9, we need to find a substitution σ such that G satisfies $\exists(\overset{+}{\bullet}_1 \rightarrow \bullet_2)^\sigma$. Since the replacement system \mathcal{R} is monotone, we can check different substitutions σ in order of size:

- | | | | | |
|-----|---|---|--|---|
| (1) | $\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$ | $\models^? \exists(\overset{+}{\bullet}_1 \rightarrow \bullet_2)$ | | ✓ |
| (2) | $\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$ | $\models^? \exists(\overset{+}{\bullet}_1 \rightarrow \bullet \rightarrow \bullet_2)$ | | ✓ |
| (3) | $\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$ | $\models^? \exists(\overset{+}{\bullet}_1 \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet_2)$ | | ✓ |
| (4) | $\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$ | $\models^? \exists(\overset{+}{\bullet}_1 \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet_2)$ | | ✗ |

Any condition with a graph bigger than (3) cannot be satisfied by G , since there is no injective mapping from a graph with more than four nodes to G . Thus, we do not need to check any bigger substitutions. \diamond

Theorem 3.1 is not only valid for HR^* conditions. One can also define conditions with a different replacement formalism, as long as the replacement systems are monotone. This does not influence the decidability of the validity problem.

Fact 3.2 (decidability of conditions with monotone replacement system).

The validity problem for graph conditions with variables and an arbitrary monotone replacement system is decidable, i.e. there is an algorithm that determines, for a given condition c with monotone replacement system \mathcal{R} and a graph G , whether $G \models c$.

3.4 Case study: car platooning

As a case study, we look at the *car platooning* protocol suggested by (Hsu et al., 1991). In order to reduce traffic jams on highways, cars are organized into platoons. These platoons consist of cars driving on the same lane, at the same speed, and with narrow distance to each other in order to conserve road space and energy by slipstreaming.

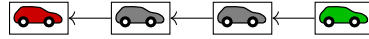


Figure 3.6: A car platoon

Figure 3.6 shows a typical car platoon consisting of four cars: a leader, two followers and a terminator. The different types of cars are distinguished by their color. White color is used to represent an arbitrary car regardless of its type. Each type of car has

certain constraints, which are expressed with HR* conditions. Internally, the colors are modeled by a looping edge with a label signifying the color (which also means a white car can be represented by a node without such a looping edge). For visual clarity, we draw the car in the appropriate color instead of drawing the loop edge.

Free Agent: A single car not associated with any platoon, drawn in blue color. Free agents may have no in- or outgoing edges, including edges to itself:

$$\#(\text{blue car} \rightarrow \text{white car}) \wedge \#(\text{white car} \leftarrow \text{blue car}) \wedge \#(\text{blue car} \rightarrow \text{blue car})$$

Leader: The leading car of a platoon. Leaders are drawn in green. Leaders have a path from a terminator (see below) and no further edges:

$$\begin{aligned} &\forall(\text{green car}), \\ &\exists(\text{red car} \rightarrow \text{green car}), \\ &\exists(\text{red car} \rightarrow \text{green car} \rightarrow \text{white car}), \\ &\#(\text{white car} \rightarrow \text{green car}) \wedge \#(\text{white car} \leftarrow \text{green car}) \wedge \\ &\#(\text{white car} \rightarrow \text{green car} \rightarrow \text{white car}) \wedge \\ &\#(\text{white car} \leftarrow \text{green car} \rightarrow \text{white car}) \wedge \\ &\#(\text{white car} \rightarrow \text{green car} \rightarrow \text{white car} \rightarrow \text{white car}) \end{aligned}$$

for all leaders,
there is a path from a terminator, and
for the leader and its immediate follower,
exactly one edge from follower to leader,
no further outgoing edge,
no further incoming edge and
no loops on the leader.

Terminator: The last car of a platoon, drawn in red. Terminators have a path to a leader and no further edges:

$$\begin{aligned} &\forall(\text{red car}), \\ &\exists(\text{red car} \rightarrow \text{green car}), \\ &\exists(\text{red car} \rightarrow \text{green car} \rightarrow \text{white car}), \\ &\#(\text{red car} \leftarrow \text{white car}) \wedge \#(\text{red car} \rightarrow \text{white car}) \wedge \\ &\#(\text{white car} \rightarrow \text{red car} \rightarrow \text{white car}) \wedge \\ &\#(\text{white car} \leftarrow \text{red car} \rightarrow \text{white car}) \wedge \\ &\#(\text{white car} \rightarrow \text{red car} \rightarrow \text{white car} \rightarrow \text{white car}) \end{aligned}$$

for all terminators,
there is a path to a leader, and
from the terminator and the car before,
there is exactly one edge,
no further incoming edge,
no further outgoing edge and
no loops on the terminator.

Follower: Any member of a platoon except leader and terminator. Followers are drawn in gray. Followers have an incoming path from a terminator and an outgoing path to a

leader, and no further edges:

$$\begin{aligned}
 & \forall (\text{grey car}), && \text{all followers} \\
 & \exists (\text{red car} \xrightarrow{+} \text{grey car} \xrightarrow{+} \text{green car}) \wedge && \text{are on a path from a terminator to a leader, and} \\
 & \nexists (\text{white car} \rightarrow \text{grey car}) \wedge && \text{have no two incoming edges,} \\
 & \nexists (\text{grey car} \rightarrow \text{white car}) \wedge && \text{no two outgoing edges and} \\
 & \nexists (\text{white car} \rightarrow \text{grey car} \rightarrow \text{white car}) && \text{no loop edge.}
 \end{aligned}$$

The above constraints can be combined into one by forming a conjunction.

$$\begin{aligned}
 & \nexists (\text{blue car} \rightarrow \text{white car}) \wedge \nexists (\text{blue car} \leftarrow \text{white car}) \wedge \nexists (\text{blue car} \rightarrow \text{blue car}) \wedge \\
 & \forall (\text{green car}, \exists (\text{red car} \xrightarrow{+} \text{green car}), \exists (\text{red car} \xrightarrow{+} \text{green car} \xrightarrow{+} \text{white car}), \nexists (\text{white car} \rightarrow \text{green car}) \wedge \nexists (\text{white car} \leftarrow \text{green car}) \wedge \\
 & \quad \nexists (\text{white car} \rightarrow \text{green car} \rightarrow \text{white car}) \wedge \nexists (\text{white car} \rightarrow \text{green car} \leftarrow \text{white car}) \wedge \nexists (\text{white car} \rightarrow \text{green car} \rightarrow \text{white car} \rightarrow \text{white car})) \wedge \\
 & \forall (\text{red car}, \exists (\text{red car} \xrightarrow{+} \text{green car}), \exists (\text{red car} \xrightarrow{+} \text{green car} \xrightarrow{+} \text{white car}), \nexists (\text{white car} \rightarrow \text{red car}) \wedge \nexists (\text{white car} \leftarrow \text{red car}) \wedge \\
 & \quad \nexists (\text{white car} \rightarrow \text{red car} \rightarrow \text{white car}) \wedge \nexists (\text{white car} \leftarrow \text{red car} \rightarrow \text{white car}) \wedge \nexists (\text{white car} \rightarrow \text{red car} \rightarrow \text{white car} \rightarrow \text{white car})) \wedge \\
 & \forall (\text{grey car}, \exists (\text{red car} \xrightarrow{+} \text{grey car} \xrightarrow{+} \text{green car}) \wedge \nexists (\text{white car} \rightarrow \text{grey car}) \wedge \nexists (\text{grey car} \rightarrow \text{white car}) \wedge \nexists (\text{white car} \rightarrow \text{grey car} \rightarrow \text{white car}))
 \end{aligned}$$

This constraint can be simplified to

$$\begin{aligned}
 & \nexists (\text{blue car} \rightarrow \text{white car}) \wedge \nexists (\text{blue car} \leftarrow \text{white car}) \wedge \nexists (\text{blue car} \rightarrow \text{blue car}) \wedge && \text{free agents have no edges,} \\
 & \forall (\text{green car}, \exists (\text{red car} \xrightarrow{+} \text{green car})) \wedge && \text{leaders have a path to a terminator,} \\
 & \forall (\text{red car}, \exists (\text{red car} \xrightarrow{+} \text{green car})) \wedge && \text{terminators have a path to a leader,} \\
 & \forall (\text{grey car}, \exists (\text{red car} \xrightarrow{+} \text{grey car} \xrightarrow{+} \text{green car})) \wedge && \text{followers are on a path between a} \\
 & \forall (\text{white car}, \nexists (\text{white car} \rightarrow \text{white car}) \wedge \nexists (\text{white car} \leftarrow \text{white car})) && \text{leader and a terminator, and} \\
 & && \text{no car has two outgoing or incoming edges.}
 \end{aligned}$$

The paths represented by “+”-labeled-edges are generated by the HR system

$$\text{white car}_1 \xrightarrow{+} \text{white car}_2 ::= \text{white car}_1 \rightarrow \text{white car}_2 \mid \text{white car}_1 \xrightarrow{+} \text{grey car} \xrightarrow{+} \text{white car}_2.$$

Note that the replacement rules can be applied to non-white cars because a node’s color is modeled by a loop edge labeled with the color.

In the car platooning protocol, the cars can perform certain maneuvers. Cars can enter or leave the road, form platoons or split them. These maneuvers are represented as graph transformations rules.

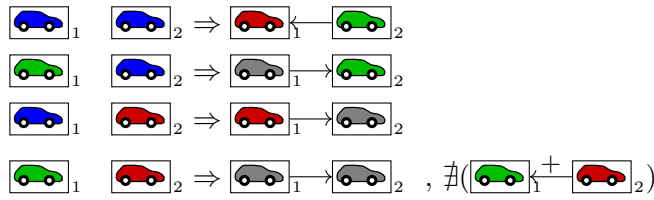
CreateFA This rule creates a new free agent.

$$\emptyset \Rightarrow \text{car}$$

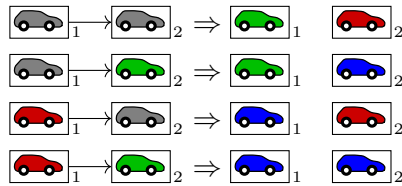
DestroyFA Deletes a free agent. Since free agents have no in- or outgoing edges, deletion of the node always succeeds.

$$\text{car} \Rightarrow \emptyset$$

Merge Merges two platoons (or free cars), given two suitable actors (i.e. leaders or free agents). One of the nodes becomes leader of the new platoon; the other node is (together with any followers it might have) appended to the end of the platoon. Since both nodes might be leaders, terminators or free agents, there are four different **Merge** rules.

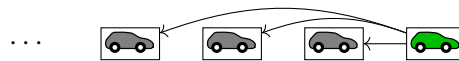


PerformSplit Performs a split of a platoon. Since a split can happen at the first, the last or an intermediate car, there are three rules, plus one for the fringe case of a platoon of 2 cars.



Remark. In some of the above rules, the color of the car is changed. This is a kind of relabeling and can either be solved by using a category with morphisms that support relabeling like \mathcal{M}, \mathcal{N} -adhesive categories (Habel and Plump, 2012), or, as done here, by representing the color as a loop edge attached to the node.

The car platooning protocol introduced by (Hsu et al., 1991) was tackled and improved in several papers, among them (Bauer, 2006) and (Pennemann, 2009). The mentioned approaches have in common that they represent a platoon using a star topology, linking every follower in a platoon directly to its leader:



In this thesis, we take another, more realistic approach, linking the cars in a chain. In practice, this facilitates the formation of longer platoons without the need to increase transmission power of the wireless communication links and with less telecommunicative collisions.



Bibliographic notes

HR* graph conditions generalize nested graph conditions (Habel and Pennemann, 2009) as well as HR conditions (Habel and Radke, 2010). While HR conditions support the use of variables in the graphs of a condition, HR* conditions are built with an additional operator $\exists(P \sqsupseteq C, c)$. This enables HR* conditions to “look into variables”, i.e. in an HR* condition $\exists(P \sqsupseteq C, c)$, morphism $b: C^\sigma \hookrightarrow P^\sigma$ may map nodes and edges in C to nodes and edges generated by a variable in P . Nested conditions, in turn, are based on negative application conditions (Habel, Heckel, et al., 1996).

To describe languages of graphs which replace the variables, HR* conditions use hyperedge replacement (Habel, 1992). More on the hyperedge replacement can be found in (Habel, 1992; Drewes, Habel, et al., 1997). Furthermore, (Plump and Habel, 1996) deals with graph unification and matching when substituting hyperedges.

As mentioned, any class of monotone replacement systems can be used for conditions with variables. A particularly interesting replacement mechanism is *contextual hyperedge replacement* (Drewes and Hoffmann, 2015), where the left-hand side of a rule might have a context. This makes the formulation of some languages of graphs easier; the author supposes that a variant of HR* conditions using contextual hyperedge replacement would be more expressive than the hyperedge-replacement based approach of HR* conditions. However, such an extension would also need more complicated matching algorithms and constructions.

Other extensions of nested conditions have been proposed. E-conditions (Poskitt and Plump, 2013) extend nested conditions with label variables, typed multi-labels and expressions over label variables, forming a simple attribution concept. The logic on subobjects from (Bruggink and B. König, 2010) introduces reasoning about subobjects and an operator \sqsubseteq which works similar to the \sqsupseteq operator in HR* conditions. The logic on subobjects is exactly as expressive as monadic second-order logic on graphs. In (Poskitt and Plump, 2014), M-conditions were introduced as an extension to E-Conditions which are also expressively equivalent to monadic second-order logic. The μ -conditions of (Flick, 2016) use a fixpoint semantics to express non-local properties like the existence of arbitrary-length paths. More on the expressiveness of HR* conditions can be found in the next chapter.

Finally, HR* conditions have been applied in different contexts: (Kutz et al., 2012) suggest the use of HR* conditions to represent classes of molecules in a chemical ontology. The abstract syntax of the traffic diagrams in (Linker, 2015) is based on graph transformation rules with HR* application conditions.

Chapter 4

Normal forms and variants of HR^* conditions

Contents

4.1	Normal forms	31
4.2	HR^* conditions with arbitrary satisfaction	34
4.3	HR^* conditions with replacement semantics	42

HR^* conditions can be used in different application scenarios. Each of these scenarios may have different requirements or preferences regarding the expressiveness, matching or substitution process. This part introduces several variants of HR^* conditions, compares them and shows, if possible, how they can be transformed into one another.

First, normal forms for HR^* conditions will be studied. Then, we will compare injective with arbitrary matching semantics, and substitution with replacement of variables.

4.1 Normal forms

Although the structure of HR^* conditions is clearly defined, it is sometimes advisable to have more rigid assumptions about the structure. This may make proofs and constructions easier, as well as lead to shorter and easier to understand conditions. Implementations of HR^* conditions may also benefit from improved performance if the formula is in some kind of normal form. For normal forms of nested conditions, see also (Pennemann, 2004, Chapter 3.3) and (Pennemann, 2009, Table 6.3).

Equivalences for HR^* conditions. The table below shows several (rather trivial) equivalences for HR^* conditions, which can be used to simplify conditions. Such simplifications are especially useful to reduce the size of a condition blown up by some transformation

(e.g. the Shift* construction from Chapter 6.1). Let P be a graph, a, a' be morphisms, c, c' be HR* conditions and $Q \in \{\forall, \exists\}$ be a quantifier.

$$\begin{aligned}
 Q(\text{id}_P, c) &\equiv c \\
 Q(a, Q(a', c)) &\equiv Q(a' \circ a, c) \\
 \neg\neg c &\equiv c \\
 c \wedge c &\equiv c \\
 c \vee c &\equiv c
 \end{aligned}$$

To improve the readability of an HR* condition and to simplify it, one can try to “compress” it, i.e. aggregate nested subconditions $\exists(a, \exists(b, c))$ to $\exists(b \circ a, c)$ (since $\text{Ran}(a) = \text{Dom}(b)$ per definition).

Definition 4.1 (compressed normal form). An HR* condition is in *compressed normal form*, or *compressed*, if it contains no subformulas of the form $\exists(a, \exists(b))$ or $\neg\neg c$. \triangle

It is easy to bring a formula into compressed normal form. Directly nested quantifiers are combined and double negations eliminated.

Construction. We define the Compress operation inductively over the structure of HR* conditions. Let a, b be morphisms and c, c' be HR* conditions.

$$\begin{aligned}
 \text{Compress}(\mathbf{true}) &= \mathbf{true}, \\
 \text{Compress}(\exists(a, \exists(b, c))) &= \text{Compress}(\exists(b \circ a, c)) \\
 \text{Compress}(\exists(a, c)) &= \exists(a, \text{Compress}(c)) \text{ if } c \neq \exists(b, c') \\
 \text{Compress}(\exists(P \sqsupseteq C, c)) &= \exists(P \sqsupseteq C, \text{Compress}(c)), \\
 \text{Compress}(\neg c) &= \text{Compress}(c') \text{ if } c = \neg c' \text{ and } \neg \text{Compress}(c) \text{ otherwise,} \\
 \text{Compress}(c \wedge c') &= \text{Compress}(c) \wedge \text{Compress}(c').
 \end{aligned}$$



To be useful as a normal form, using Compress on an HR* condition must not change the semantics of the condition.

Lemma 4.1 (Well-formedness of compressed normal form).

For every HR* condition c , $\text{Compress}(c)$ is equivalent to c , i.e. for all morphisms p , $p \models c \Leftrightarrow p \models \text{Compress}(c)$.

Proof. By induction over the structure of an HR* condition d .

Basis. Case **true**: Trivial. By construction, $p \models \text{Compress}(\mathbf{true}) \Leftrightarrow p \models \mathbf{true}$.

Hypothesis. For condition c and morphism p , assume that $p \models c \Leftrightarrow p \models \text{Compress}(c)$.

Step.

Case $\exists(a, c)$: Assume $p \models \text{Compress}(\exists(a, c))$. For $c \neq \exists(b, c')$, by construction and Definition 3.9, $p \models \text{Compress}(\exists(a, c))$ iff $p \models \exists(a, \text{Compress}(c))$ iff $\exists q. p = q \circ a \wedge q \models \text{Compress}(c)$. By the hypothesis, $\exists q. p = q \circ a \wedge q \models c$ and by Definition 3.9, $p \models \exists(a, c)$. For $c = \exists(b, c')$, by construction, $p \models \text{Compress}(\exists(a, \exists(b, c')))$ iff $p \models \text{Compress}(\exists(b \circ a, c'))$.

$a, c')$). Without loss of generality, assume $c' \neq \exists(b', c'')$; otherwise, repeatedly apply the construction. Then $p \models \exists(b \circ a, \text{Compress}(c'))$, and by Definition 3.9 and the hypothesis, $\exists q.p = q \circ b \circ a \wedge q \models \text{Compress}(c') \Leftrightarrow \exists q.p = q \circ b \circ a \wedge q \models c'$. Let $q' = q \circ b$. Then $\exists q'.p = q' \circ a \wedge \exists q.q' = q \circ b \wedge q \models c'$. By the semantics of HR* conditions, $p \models \exists(a, \exists(b, c'))$.

Case $\exists(P \sqsupseteq C, c)$: By construction and HR* semantics, $p \models \exists(P \sqsupseteq C, \text{Compress}(c))$ iff $\exists a: C \hookrightarrow P, q.q = p \circ a \wedge q \models \text{Compress}(c)$. By the hypothesis, $\exists a: C \hookrightarrow P, q.q = p \circ a \wedge q \models c$, and by HR* semantics, $p \models \exists(P \sqsupseteq C, c)$.

Case $\neg c$. For $c = \neg c'$, by construction and the hypothesis, $p \models \text{Compress}(\neg \neg c') \Leftrightarrow p \models \text{Compress}(c') \Leftrightarrow p \models c' \Leftrightarrow p \models \neg \neg c'$. For $c \neq \neg c'$, by construction and hypothesis, $p \models \text{Compress}(\neg c) \Leftrightarrow p \models \neg \text{Compress}(c) \Leftrightarrow p \models \neg c$.

Case $c \wedge c'$: By construction and the hypothesis, $p \models \text{Compress}(c \wedge c') \Leftrightarrow p \models \text{Compress}(c) \wedge p \models \text{Compress}(c') \Leftrightarrow p \models c \wedge p \models c' \Leftrightarrow p \models (c \wedge c')$. \square

Example 4.1. Let $d = \exists(\bullet_1 \bullet_2, \neg \neg \exists(\bullet_1 \rightarrow \bullet_2))$. Then $\text{Compress}(d) = \exists(\bullet_1 \rightarrow \bullet_2)$. \diamond

The compressed normal form shortens conditions $\exists(a, \exists(b, c))$ to $\exists(b \circ a, c)$. For proofs and constructions (such as the proof about substitution and replacement in Chapter 4.3), the opposite strategy may be beneficial: Every $\exists(P \hookrightarrow C, c)$ just adds a single node, edge or hyperedge to C . A similar idea is used by (Pennemann, 2009, Chapter 3.3) in the transformation Forms to translate nested conditions into first-order formulas.

Definition 4.2 (decompressed normal form). An HR* condition is in *decompressed normal form*, or *decompressed*, if, for every subcondition $\exists(P \hookrightarrow C, c)$, $C \cong P + o$, where o is a single item of a graph (i.e. a node, edge or hyperedge). \triangle

Note that uniqueness is not claimed for decompressed normal form. In fact, the decompressed normal form is not deterministic, since a graph can generally be “decompressed” into components in different ways. However, for a unique and deterministic normal form, one can define an order over the items of each graph and perform the decompression along this order.

It is easy to bring a formula into decompressed normal form by decomposing the morphism a in a condition $\exists(a, c)$ into several morphisms, each of whose codomains add only a single item to their respective domain.

Construction. We define the Decom operation inductively over the structure of HR* conditions.

$$\begin{aligned} \text{Decomp}(\exists(P \hookrightarrow C, c)) &= \exists(P \hookrightarrow P + o, \text{Decomp}(\exists(P + o \hookrightarrow C, c))) \text{ if } P \not\cong C \text{ and} \\ &\quad \text{Decomp}(c) \text{ else} \\ &\quad \text{where } o \text{ is a single node, edge or hyperedge in } C \\ \text{Decomp}(\mathbf{true}) &= \mathbf{true} \\ \text{Decomp}(\exists(P \sqsupseteq C, c)) &= \exists(P \sqsupseteq C, \text{Decomp}(c)) \\ \text{Decomp}(\neg c) &= \neg \text{Decomp}(c) \\ \text{Decomp}(c \wedge c') &= \text{Decomp}(c) \wedge \text{Decomp}(c'). \end{aligned}$$



Lemma 4.2 (Well-formedness of decompressed normal form).

For every HR* condition c , $\text{Decomp}(c)$ is equivalent to c , i.e. for all morphisms p , $p \models c \Leftrightarrow p \models \text{Decomp}(c)$.

Proof. By induction over the structure of an HR* condition d .

Basis. Case **true**: Trivial. By construction, $p \models \text{Decomp}(\text{true}) \Leftrightarrow p \models \text{true}$.

Hypothesis. For condition c and morphism p , assume that $p \models c \Leftrightarrow p \models \text{Decomp}(c)$.

Step.

Case $\exists(P \xrightarrow{a} C, c)$: For $P \cong C$, by the hypothesis, $p \models \text{Decomp}(\exists(P \xrightarrow{a} C, c))$ iff $p \models c \Leftrightarrow p \models \exists(P \xrightarrow{a} C, c)$.

For $P \not\cong C$, by the construction and Definition 3.9, $p \models \text{Decomp}(\exists(P \xrightarrow{a} C, c))$ iff $p \models \exists(P \xrightarrow{b} P + o, \text{Decomp}(\exists(P + o \xrightarrow{b'} C, c)))$. By the hypothesis, $p \models \exists(P \xrightarrow{b} P + o, \exists(P + o \xrightarrow{b'} C, c))$. Since $a = b' \circ b$, $p \models \exists(P \xrightarrow{a} C, c)$.

Case $\exists(P \sqsupseteq C, c)$: By construction and HR* semantics, $p \models \exists(P \sqsupseteq C, \text{Decomp}(c))$ iff $\exists a: C \hookrightarrow P, q. q = p \circ a \wedge q \models \text{Decomp}(c)$. By the hypothesis, $\exists a: C \hookrightarrow P, q. q = p \circ a \wedge q \models c$, and by HR* semantics, $p \models \exists(P \sqsupseteq C, c)$.

Case $\neg c$. By construction and hypothesis, $p \models \text{Decomp}(\neg c) \Leftrightarrow p \models \neg \text{Decomp}(c) \Leftrightarrow p \models \neg c$.

Case $c \wedge c'$: By construction and the hypothesis, $p \models \text{Decomp}(c \wedge c') \Leftrightarrow p \models \text{Decomp}(c) \wedge p \models \text{Decomp}(c') \Leftrightarrow p \models c \wedge p \models c' \Leftrightarrow p \models (c \wedge c')$. \square

Example 4.2. Let $d = \exists(\bullet_1 \bullet_2, \neg \exists(\bullet_1 \rightarrow \bullet_2))$. Then $\text{Decomp}(d) = \exists(\bullet_1, \neg \exists(\bullet_1 \bullet_2, \exists(\bullet_1 \rightarrow \bullet_2)))$, meaning that there are no two nodes 1, 2 with an edge from 1 to 2. \diamond

4.2 HR* conditions with arbitrary satisfaction

In Definition 3.9, the semantics of HR* conditions is defined using injective morphisms. An alternative satisfaction definition for HR* conditions uses arbitrary instead of injective morphisms. This provides a means to identify nodes or edges in the condition. The concept of defining variables which may later be identified is nearer to logic formulas, where constructions like “ $\exists x, y. x = y \Rightarrow \dots$ ” are common. We show that, for a subset of HR* conditions, both the injective semantics from Definition 3.9 and the alternative definition are expressively equivalent. The results of this chapter will be used in Chapter 5 to establish a relationship between HR* conditions and logical formulas.

Assumption. Let \mathcal{A} be the class of (arbitrary) graph morphisms and recall that \mathcal{M} is the class of injective graph morphisms.

Definition 4.3 (\mathcal{A} -satisfaction). For an HR system \mathcal{R} , a substitution $\sigma \in \Sigma^*$ and a morphism $p: P^\sigma \rightarrow G$, HR* condition c is \mathcal{A} -satisfied by p , written $p \models_{\mathcal{A}} c$, as defined in Definition 3.9, except that morphisms p and q are in \mathcal{A} (i.e. arbitrary). Let \mathcal{M} -satisfaction for HR* conditions be synonymous to the satisfaction in Definition 3.9. \triangle

Notation. In order to distinguish between arbitrary and injective satisfaction, we will denote them as \mathcal{A} -satisfaction and \mathcal{M} -satisfaction, respectively. We write $p \models_{\mathcal{A}} c$ and $p \models_{\mathcal{M}} c$ to denote that p \mathcal{A} - or \mathcal{M} -satisfies c , respectively.

For nested conditions, \mathcal{A} - and \mathcal{M} -satisfaction are equivalent (Habel and Pennemann, 2009). For HR* conditions in general, this seems not to be the case: With \mathcal{A} -satisfiable HR* conditions, it is quite easy to let a variable match an arbitrary graph by unifying nodes and edges after substituting the hyperedges. Since the set of all graphs is not expressible by hyperedge replacement, this is not possible with \mathcal{M} -satisfiable HR* conditions. However, we can restrict hyperedges to represent sets of nodes only, without any edges. These restricted HR* conditions are dubbed *Set conditions*. For Set conditions, we will show that every \mathcal{A} -satisfiable condition can be transformed into an equivalent \mathcal{M} -satisfiable condition.

Definition 4.4 (Set condition). An HR* condition is a *Set condition* if its replacement system \mathcal{R} generates, for each variable x , only discrete graphs, i.e. graphs without edges (or hyperedges). \triangle

Remark. A Set condition can be regarded as an HR* condition $\langle c, \mathcal{R} \rangle$, where \mathcal{R} consists only of *set rules* of the form $\boxtimes ::= \emptyset \mid \bullet \boxtimes \mid \boxtimes \boxtimes$. These rules only generate discrete graphs.

We will now show that it is possible to transform an \mathcal{A} -satisfiable Set condition into an \mathcal{M} -satisfiable Set condition.

Theorem 4.1 (from \mathcal{A} - to \mathcal{M} -satisfaction for Set conditions).

For every Set condition c , there is a Set condition $\text{Cond}_{\mathcal{M}}(c)$ such that for every morphism n ,

$$n \models_{\mathcal{A}} c \iff n \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(c).$$

Likewise, for every graph G ,

$$G \models_{\mathcal{A}} c \iff G \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(c).$$

Before we can prove the above theorem, we need to prove an auxiliary lemma, which ensures that a substitution σ followed by an arbitrary morphism $b: G^\sigma \rightarrow H$ can be reached “the other way round”, i.e. by having a “small” substitution (in the sense that it consists of a bounded number of derivation steps), followed by a morphism and another substitution (with no bounds on the number of derivation steps).

Lemma 4.3 (commutativity of morphisms with set rules).

Let G be a graph, \mathcal{R} a replacement system consisting of set rules only, σ a substitution and $b: G^\sigma \rightarrow H$ a morphism. Then there are substitutions τ and σ' consisting of set rules only, and a morphism $b': G^\tau \rightarrow G'$ such that $b'(\sigma(x)) = \sigma'(b(\tau(x)))$ for all nodes and edges x in G and τ consists of no more than $|G|$ replacement steps.

$$\begin{array}{ccc}
 G & \xrightarrow{\sigma} & G^\sigma \\
 \downarrow \tau & & \downarrow b \\
 G^\tau & = & \\
 \downarrow b' & \xrightarrow{\sigma'} & H \\
 G' & \xrightarrow{\sigma'} & H
 \end{array}$$

Every pair of items u, v that are identified by morphism b also have to be identified by b' . The role of τ is to generate all items which are unified by b with another item not part of G (i.e. generated from a hyperedge by σ).

Construction. Without loss of generality, we assume only a single pair u, v of items to be identified by b ; we can construct b' and σ' step-by-step for every such pair of identified items.

If b is injective for a pair of items $u^\sigma, v^\sigma \in G^\sigma$ with preimages u, v in G , then $\tau(u) = u$, $b'(u) = b(u)$, $\sigma'(b'(u)) = b(u^\sigma)$ and likewise for v .

For every pair of items u, v in G^σ with $b(u) = b(v)$, before the application of σ , we have either

- (a) $u, v \in G$. Then let $\tau(u) = u$, $\tau(v) = v$, $b'(u) = b'(v) = b(u)$ and $\sigma'(b(u)) = b(u)$:

$$\begin{array}{ccccc}
 \bullet & \bullet & \xrightarrow{\tau} & \bullet & \bullet & \xrightarrow{b'} & \bullet \\
 \downarrow \sigma & & & & & & \downarrow \sigma' \\
 \bullet & \bullet & \xrightarrow{b} & & & & \bullet \\
 u & v & & & & & u=v
 \end{array}$$

- (b) a hyperedge $y \in G$ with both $u, v \in \sigma(y)$. Then let $\tau(y) = y$, $b'(y) = y$ and $\sigma'(y) = b(\sigma(y))$:

$$\begin{array}{ccccc}
 \boxed{y} & \xrightarrow{\tau} & \boxed{y} & \xrightarrow{b'} & \boxed{y} \\
 \downarrow \sigma & & & & \downarrow \sigma' \\
 \sigma(y) & \xrightarrow{b} & & & b(\sigma(y))
 \end{array}$$

- (c) a hyperedge $y \in G$ with $u \in \sigma(y)$ and $v \in G$ (or vice versa). Then let $\tau(y) = y' + u$, $\tau(v) = v$, $b'(u) = b'(v) = b(u)$, $b'(y') = y'$, $\sigma'(y') = b(\sigma(y) - u)$ and $\sigma'(b(u)) = b(u)$:

$$\begin{array}{ccccc}
 \boxed{y} \bullet & \xrightarrow{\tau} & \boxed{y'} \bullet & \bullet & \xrightarrow{b'} & \boxed{y'} \bullet \\
 \downarrow \sigma & & & & & \downarrow \sigma' \\
 \sigma(y) + \bullet & \xrightarrow{b} & & & & \sigma(y) - \bullet + \bullet \\
 & & & & & u \quad u=v
 \end{array}$$

- (d) two non-identical hyperedges $y_u, y_v \in G$ with $u \in \sigma(y_u)$ and $v \in \sigma(y_v)$. Then let $\tau(y_u) = y'_u + u$, $b'(y'_u) = y'_u$, $\sigma'(y'_u) = b(\sigma(y_u) - u)$ and likewise for y_v , and $b'(u) = b'(v) = b(u)$, $\sigma'(b(u)) = b(u)$:

$$\begin{array}{ccc}
 \boxed{y_u} \boxed{y_v} & \xrightarrow{\tau} & \boxed{y'_u} \bullet_u \quad \boxed{y'_v} \bullet_v \xrightarrow{b'} \boxed{y'_{u=v}} \bullet_{u=v} \\
 \downarrow \sigma & & \downarrow \sigma' \\
 \sigma(y_u) + \sigma(y_v) & \xrightarrow{b} & \sigma(y_u) - \bullet_u + \sigma(y_v) - \bullet_v + \bullet_{u=v}
 \end{array}$$

Proof (of Lemma 4.3). We show that for every item u in G , $\sigma'(b'(\tau(u))) = b(\sigma(u))$. The proof proceeds along the cases given in the construction.

If u is not identified with another item v by b , by construction, $\sigma'(b'(\tau(u))) = \sigma'(b'(u)) = \sigma'(b(u)) = b(\sigma(u))$.

- (a) By construction, $\sigma'(b'(\tau(u))) = \sigma'(b(u)) = b(u) = \sigma(b(u))$. Likewise, $\sigma'(b'(\tau(v))) = \sigma'(b(v)) = b(v) = \sigma(b(v))$.
- (b) By construction, $\sigma'(b'(\tau(y))) = \sigma'(y) = b(\sigma(y))$.
- (c) By construction, $\sigma'(b'(\tau(y))) = \sigma'(b'(y' + u)) = \sigma'(y' + b(u)) = b(\sigma(y) - u) + b(u) = b(\sigma(u))$. For v , we have $\sigma'(b'(\tau(v))) = \sigma'(b(u)) = b(u) = \sigma(b(u))$.
- (d) By construction, $\sigma'(b'(\tau(y_u))) = \sigma'(b'(y'_u + u)) = \sigma'(y'_u - b(u)) = b(\sigma(y_u) + u) - b(u) = b(\sigma(y_u))$. For y_v , the proof is analogous.

Every item u that is unified with another item v adds zero (cases a,b) or one (cases c,d) step from \mathcal{R} to τ . A maximum of $|G| - 1$ items can be identified in a graph of size $|G|$, so τ consists of no more than $|G|$ derivation steps. \square

Example 4.3. Let $G = \boxed{A} \boxed{B}$, $\sigma(A) = \bullet_1 \bullet_2$, $\sigma(B) = \bullet_3 \bullet_4 \bullet_5$. Let $b: G^\sigma \rightarrow H$ be the identity except for $b(5) = 4$ and $b(2) = 3$. The diagram below shows the construction of τ , b' and σ' .

$$\begin{array}{ccc}
 G & \boxed{A} \boxed{B} \xrightarrow{\sigma} & \boxed{\bullet_1 \bullet_2} \quad \boxed{\bullet_3 \bullet_4 \bullet_5} & G^\sigma \\
 & \downarrow \tau & \downarrow b & \\
 G^\tau & \boxed{A-B} \boxed{A \cap B} \boxed{B-A} = & & \\
 & \downarrow b' & & \\
 G' & \boxed{A-B} \boxed{A \cap B} \boxed{B-A} \xrightarrow{\sigma'} & \boxed{\bullet_1} \quad \boxed{\bullet_2 = \bullet_3} \quad \boxed{\bullet_4 = \bullet_5} & H
 \end{array}$$

We use “speaking” variable names to emphasize the way in which the construction splits up the hyperedges labeled A and B into three hyperedges containing the nodes of $A-B$, $A \cap B$ and $B-A$, respectively. \diamond

We continue to prove that \mathcal{A} -satisfiable Set conditions can be transformed into \mathcal{M} -satisfiable Set conditions.

Idea. Under \mathcal{A} -satisfaction, items generated by a hyperedge may be identified with items generated by another hyperedge or with items present before substitution. Hence, a substitution τ is used to split up hyperedges into different parts (each represented by a hyperedge with a fresh name), each of which may be identified with another part of the graph. Replacement rules for the new hyperedge names are added as needed.

Construction. Let $\mathcal{E}(P)$ denote the set of all surjective morphisms with domain P . For spans $C \xleftarrow{a} P \xrightarrow{b} P'$, let $\mathcal{E}'(a, b)$ be the set of all jointly surjective cospans $C \xrightarrow{b'} C' \xleftarrow{a'} P'$.

For every condition c over P , let $\text{Cond}_{\mathcal{M}}(c) = \text{Cond}_{\mathcal{M}}(P \rightarrow P, c)$ where the latter is defined inductively as follows:

$\text{Cond}_{\mathcal{M}}(b, \exists(P \xrightarrow{a} C, c)) = \bigvee_{\tau \in \Sigma_k, (a', b') \in \mathcal{E}'(a\tau, b)} \exists(a', \text{Cond}_{\mathcal{M}}(b', c))$, where $k = |C - P|$ and Σ_k is the set of all substitutions $C \Rightarrow^{\leq k} C^\tau$ over C with no more than k steps,

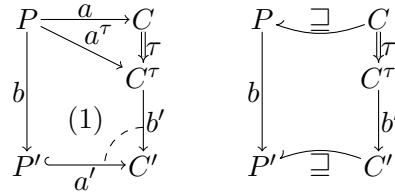
$\text{Cond}_{\mathcal{M}}(b, \exists(P \sqsupseteq C, c)) = \bigvee_{\tau \in \text{Sigma}_k, b' \in \mathcal{E}(C)} \exists(P \sqsupseteq b(C^\tau), \text{Cond}_{\mathcal{M}}(b', c^\tau))$.

The rest is straightforward:

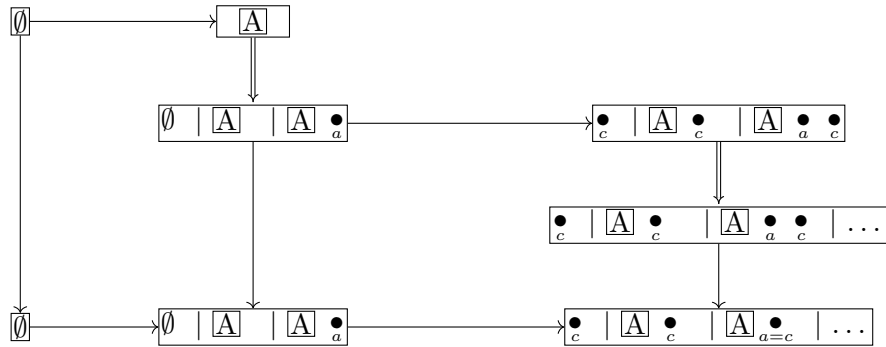
$\text{Cond}_{\mathcal{M}}(b, \text{true}) = \text{true}$

$\text{Cond}_{\mathcal{M}}(b, \neg c) = \neg \text{Cond}_{\mathcal{M}}(b, c)$, and

$\text{Cond}_{\mathcal{M}}(b, c \wedge d) = \text{Cond}_{\mathcal{M}}(b, c) \wedge \text{Cond}_{\mathcal{M}}(b, d)$.



Example 4.4. In the following example, we convert the condition $\forall(\underline{\mathbb{A}}, \exists(\underline{\mathbb{A}} \bullet_c))$ from \mathcal{A} -satisfiable into \mathcal{M} -satisfiable form. There are many duplicate cases and cases which are subsumed by others; we leave most of these out to clarify what is happening in the construction.



$$\begin{aligned} \text{Cond}_{\mathcal{M}}(\emptyset \rightarrow \emptyset, \forall(\underline{\mathbb{A}}, \exists(\underline{\mathbb{A}} \bullet_c))) &= \\ \forall(\underline{\mathbb{A}}, \text{Cond}_{\mathcal{M}}(\underline{\mathbb{A}} \rightarrow \underline{\mathbb{A}}, \exists(\underline{\mathbb{A}} \bullet_c))) &\wedge \forall(\emptyset, \text{Cond}_{\mathcal{M}}(\emptyset \rightarrow \emptyset, \exists(\bullet_c))) = \\ \forall(\underline{\mathbb{A}}, \exists(\underline{\mathbb{A}} \bullet_c)) &\wedge \forall(\emptyset, \exists(\bullet_c)) \end{aligned}$$

4.2 HR* conditions with arbitrary satisfaction

In a similar fashion, we can convert $\exists(\underline{A} \ \underline{B})$ into \mathcal{M} -satisfiable form:

$$\begin{aligned} \text{Cond}_{\mathcal{M}}(\emptyset \rightarrow \emptyset, \exists(\underline{A} \ \underline{B})) &= \\ \exists(\emptyset) \vee \exists(\underline{A}) \vee \exists(\underline{B}) \vee \exists(\underline{A} \ \underline{B}) \vee \exists(\underline{A-B} \ \underline{A \cap B} \ \underline{B-A}) & \\ \equiv \exists(\underline{A-B} \ \underline{A \cap B} \ \underline{B-A}) & \quad \diamond \end{aligned}$$

Proof (of Theorem 4.1). We proceed by induction over the structure of Set conditions. The proof is straightforward for conditions of the form **true**, $c \wedge d$ as well as $\neg c$.

Our induction hypothesis is that for any subcondition c and morphisms $q': P' \rightarrow G$ and $b': P \rightarrow P'$, $q' \circ b \models_{\mathcal{A}} c \iff q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b, c)$.

Case $\exists(a, c)$, “ \Leftarrow ”:

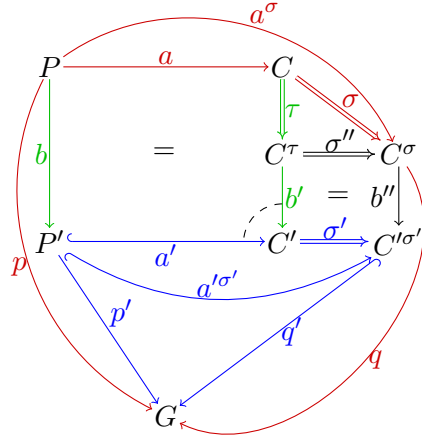
Assume $p' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(\exists(a, c))$. By construction of $\text{Cond}_{\mathcal{M}}$, this equals $p' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b, \exists(a, c)) \Leftrightarrow p' \models_{\mathcal{M}} \bigvee_{\tau \in \Sigma^k, (a', b') \in \mathcal{E}'(C^\tau, P')} \exists(a', \text{Cond}_{\mathcal{M}}(b', c^\tau))$.

By the semantics of HR* conditions, this is equivalent to

$$\exists \sigma', a'^{\sigma'} : P' \hookrightarrow C^{\sigma'}, b', q' : C^{\sigma'} \hookrightarrow G. p' = q' \circ a'^{\sigma'} \wedge q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau)^{\sigma'}.$$

We construct a substitution σ'' and a morphism b'' such that $\sigma''(b'(x)) = b''(\sigma'(x))$ for all items x in C^τ . Now, we can construct $\sigma = \sigma'' \circ \tau$, $a^\sigma : P \rightarrow C^\sigma$, $q = q' \circ b''$. Then $q' \circ b'' \circ a^\sigma = p' \circ b$, thus $p = q \circ a^\sigma$.

By $\sigma'' \circ b' = \sigma' \circ b''$ and the induction hypothesis, we have $q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau)^{\sigma'} \Leftrightarrow q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b'', c^{\sigma'' \circ \tau}) \Leftrightarrow q' \circ b'' \models_{\mathcal{A}} c^{\sigma'' \circ \tau}$. Together, this yields $\exists \sigma = \sigma'' \circ \tau, a^\sigma : P \rightarrow C^\sigma, q = q' \circ b'' . p = q \circ a^\sigma \wedge q \circ b' \models_{\mathcal{A}} c^{\sigma'' \circ \tau} \Leftrightarrow \exists \sigma, a^\sigma : P \rightarrow C^\sigma, q : C^\sigma \rightarrow G. p = q \circ a^\sigma \wedge q \models_{\mathcal{A}} c^\sigma \Leftrightarrow p \models_{\mathcal{A}} \exists(a, c)$.



Case $\exists(a, c)$, “ \Rightarrow ”: Assume $p' \circ b \models_{\mathcal{A}} \exists(a, c)$.

By the definition of \mathcal{A} -satisfaction, this equals $\exists \sigma, a^\sigma, q. p' \circ b = q \circ a^\sigma \wedge q \models c^\sigma$.

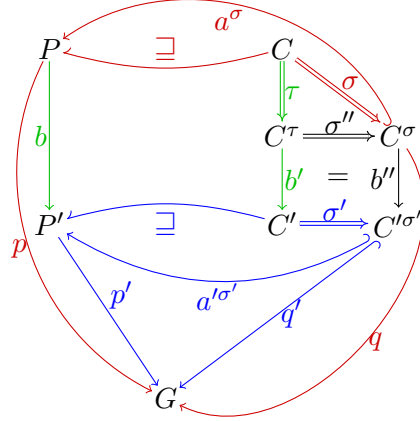
Using epi-mono-factorization (see (H. Ehrig, K. Ehrig, Prange, et al., 2006)) on q , we split $q = q'_h \circ b''_h$. Let $(a'^{\sigma'}, b'')$ be the pushout of the span $(b, b_h \circ a^\sigma)$ with pushout object $C'^{\sigma'}$. Note that $(a'^{\sigma'}, b'')$ are jointly surjective. By Lemma 4.3, we construct morphism $b_h : C \rightarrow C_h$ and substitution σ'_h such that $\sigma'_h(b_h(x)) = b''(\sigma(x))$ for all x in C . We split

$\sigma'_h = \sigma' \circ \tau_h$, as per the Lemma. Note that τ_h consists of no more than $|C|$ replacement steps. Let $C' = \tau_h(C_h)$. We can now construct substitution τ and morphism $b': C^\tau \rightarrow C'$ such that $\sigma'_h(b_h(x)) = b''(\sigma(x))$ for all x in C . Since τ_h consists of no more than $|C|$ replacement steps, τ does, too, and $\tau \in \sigma_{|C|}$. Since the pair $(a'^{\sigma'}, b')$ is jointly surjective, p' satisfies $\exists(a', \text{Cond}_{\mathcal{M}}(b', c^\tau))$ with $(a', b') \in \mathcal{E}(C)$. By the induction hypothesis, we have $q \models_{\mathcal{A}} c \Leftrightarrow q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau)$. Together, this yields $\exists \sigma', a'^{\sigma'}: P \rightarrow C'^{\sigma'}, q': C'^{\sigma'} \rightarrow G. p' = q' \circ a'^{\sigma'} \wedge q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau) \equiv p' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b, \exists(a, c))$.

Case $\exists(P \sqsupseteq C, c)$, “ \Leftarrow ”:

$p' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b, \exists(P \sqsupseteq C, c)) \Leftrightarrow p' \models_{\mathcal{M}} \bigvee_{\tau \in \Sigma_k, b' \in \mathcal{E}(C)} \exists(P \sqsupseteq b(C^\tau), \text{Cond}_{\mathcal{M}}(b', c^\tau)) \Leftrightarrow \exists \tau \in \Sigma_k, b' \in \mathcal{E}(C). p' \models \exists(P \sqsupseteq b(C^\tau), \text{Cond}_{\mathcal{M}}(b', c^\tau))$. Let $C' = b(C^\tau)$. Then $\exists \tau \in \Sigma_k, b' \in \mathcal{E}(C), \sigma' \in \Sigma, a': C'^{\sigma'} \hookrightarrow P^{\sigma'}, q': C'^{\sigma'} \rightarrow G. q' = p' \circ a' \wedge q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c)$. We construct substitution σ'' and morphism $b'': (C^\tau)^{\sigma''} \rightarrow C'^{\sigma'}$ such that $\sigma'(b''(x)) = b''(\sigma''(x))$ for all x in C^τ . Let $\sigma = \sigma'' \circ \tau$ and $q = q' \circ b''$.

By the induction hypothesis, $q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau) \Leftrightarrow q \models_{\mathcal{A}} c$. Then there is an injective morphism $a: C^\sigma \hookrightarrow P$ such that $p' \circ b \circ a = q$ and $q \models_{\mathcal{A}} c \equiv p' \circ b \models_{\mathcal{A}} \exists(P \sqsupseteq C, c)$.



Case $\exists(P \sqsupseteq C, c)$, “ \Rightarrow ”:
 Assume $p' \circ b \models_{\mathcal{A}} \exists(P \sqsupseteq C, c)$.

By the definition of \mathcal{A} -satisfaction, this equals $\exists \sigma, a: C^\sigma \hookrightarrow P, q, q = p' \circ b \circ a \wedge q \models c^\sigma$. Using epi-mono-factorization on q , we split $q = q'_h \circ b''_h$. By Lemma 4.3, we construct morphism $b_h: C \rightarrow C_h$ and substitution σ'_h such that $\sigma'_h(b_h(x)) = b''(\sigma(x))$ for all x in C . We split $\sigma'_h = \sigma' \circ \tau_h$, as per the Lemma. Note that τ_h consists of no more than $|C|$ replacement steps. Let $C' = \tau_h(C_h)$. We can now construct substitution τ and morphism $b': C^\tau \rightarrow C'$ such that $\sigma'_h(b_h(x)) = b''(\sigma(x))$ for all x in C . Since τ_h consists of no more than $|C|$ replacement steps, τ does, too, and $\tau \in \sigma_{|C|}$. By the induction hypothesis, we have $q \models_{\mathcal{A}} c \Leftrightarrow q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau)$. Together, this yields $\exists \sigma', a': C'^{\sigma'} \rightarrow P', q': C'^{\sigma'} \rightarrow G. q' = p' \circ a' \wedge q' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b', c^\tau) \Leftrightarrow p' \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(b, \exists(P \sqsupseteq C, c))$.

For $b = \text{id}_P$, we thus have $n \models_{\mathcal{A}} c \Leftrightarrow n \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(c)$.

$G \models_{\mathcal{A}} c \Leftrightarrow G \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(c)$ follows directly from the definition of satisfaction by choosing $n: \emptyset \hookrightarrow G$. \square

Theorem 4.1 shows that for Set conditions, \mathcal{M} -satisfaction is (at least) as expressive as \mathcal{A} -satisfaction. This fact will be used in Chapter 5.2 to show that counting monadic second-order formulas can be expressed as HR* conditions.

The consequence of Definition 4.3 is that nodes and edges in \mathcal{A} -satisfiable HR* conditions no longer have a disjoint image in graph G by default, but may be identified. We showed \mathcal{A} -satisfaction to be as expressive as \mathcal{M} -satisfaction. We now show that the converse is also true: it is possible to find an equivalent \mathcal{A} -satisfiable HR* condition for every \mathcal{M} -satisfiable HR* condition.

Theorem 4.2 (from \mathcal{M} - to \mathcal{A} -satisfaction).

For every HR* condition c over P , there is an HR* condition $\text{Cond}_{\mathcal{A}}(c)$ such that for every morphism $p: P \hookrightarrow G$,

$$p \models_{\mathcal{M}} c \iff p \models_{\mathcal{A}} \text{Cond}_{\mathcal{A}}(c).$$

Likewise, for every graph G ,

$$G \models_{\mathcal{M}} c \iff G \models_{\mathcal{A}} \text{Cond}_{\mathcal{A}}(c).$$

Idea. Each subcondition of the form $\exists(P \hookrightarrow C, c)$ in the \mathcal{A} -satisfiable condition is supplemented with a condition $\text{noId}(C)$, which ensures that no items in C are identified.

Construction ($\text{Cond}_{\mathcal{A}}$). For a condition $\exists(P \hookrightarrow C, c)$, let

$$\begin{aligned} \text{Cond}_{\mathcal{A}}(\exists(P \hookrightarrow C, c)) &= \exists(P \rightarrow C, \text{Cond}_{\mathcal{A}}(c) \wedge \text{noId}(C)) \\ \text{with } \text{noId}(C) &:= \forall(C \sqsupseteq \bullet \bullet, \#(\bullet \rightarrow \bullet)) \wedge \forall(C \sqsupseteq \downarrow \downarrow, \#(\downarrow \rightarrow \downarrow)). \end{aligned}$$

noId is a condition over C ensuring that no two nodes or edges in C are identified in G by a non-injective match q . For all other conditions, $\text{Cond}_{\mathcal{A}}$ is straightforwardly passed on: $\text{Cond}_{\mathcal{A}}(\text{true}) = \text{true}$, $\text{Cond}_{\mathcal{A}}(\exists(P \sqsupseteq C, c)) = \exists(P \sqsupseteq C, \text{Cond}_{\mathcal{A}}(c))$, $\text{Cond}_{\mathcal{A}}(\neg c) = \neg \text{Cond}_{\mathcal{A}}(c)$ and $\text{Cond}_{\mathcal{A}}(c \wedge c') = \text{Cond}_{\mathcal{A}}(c) \wedge \text{Cond}_{\mathcal{A}}(c')$. ◊

Example 4.5. The HR* condition $\text{EVEN} = \exists(\mathbb{2}, \#(\mathbb{2} \bullet))$ with $\mathbb{2} ::= \emptyset \mid \bullet \bullet$ expresses the property “the graph has an even number of nodes” with \mathcal{M} -satisfaction. With \mathcal{A} -satisfaction, the same condition would express “the graph has any number of nodes (including zero)”, i.e. be equivalent to true . Using the construction of $\text{Cond}_{\mathcal{A}}$, we get

$$\begin{aligned} \text{Cond}_{\mathcal{A}}(\exists(\mathbb{2}, \#(\mathbb{2} \bullet))) &= \exists(\mathbb{2}, \text{noId}(\mathbb{2}) \wedge \#(\mathbb{2} \bullet, \text{noId}(\mathbb{2} \bullet))) \\ &\equiv \exists(\mathbb{2}, \forall(\mathbb{2} \sqsupseteq \bullet \bullet, \#(\bullet \rightarrow \bullet)) \wedge \forall(\mathbb{2} \sqsupseteq \downarrow \downarrow, \#(\downarrow \rightarrow \downarrow))) \\ &\quad \wedge \#(\mathbb{2} \bullet, \forall(\mathbb{2} \bullet \sqsupseteq \bullet \bullet, \#(\bullet \rightarrow \bullet)) \wedge \forall(\mathbb{2} \bullet \sqsupseteq \downarrow \downarrow, \#(\downarrow \rightarrow \downarrow))) \quad \diamond \end{aligned}$$

Proof (of Theorem 4.2). For conditions true , $\exists(P \sqsupseteq C, c)$, $\neg c$ and $c \wedge c'$, the proof is trivial as $\text{Cond}_{\mathcal{A}}$ does not change the condition and just “passes on” the construction. For a condition $\exists(P \hookrightarrow C, c)$ and morphism $p: P \hookrightarrow G$, we can directly transform the statement that two objects d, d' must be disjoint into a condition that fits our construction:

By the semantics of HR* conditions, G satisfies $\exists(P \hookrightarrow C, c)$ iff
 $\exists \sigma, q: C^\sigma \hookrightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{M}} c^\sigma$, which is equivalent to
 $\exists \sigma, q: C^\sigma \rightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{M}} c^\sigma \wedge \nexists d, d' \in D_C.d \neq d' \wedge q(d) = q(d')$.
 Since q is injective, this is equivalent to
 $\exists \sigma, q: C^\sigma \rightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{M}} c^\sigma$
 $\wedge \nexists v, v' \in V_C.v \neq v' \wedge q(v) = q(v') \wedge \nexists e, e' \in E_C.e \neq e' \wedge q(e) = q(e')$
 By the semantics of HR* conditions, this yields
 $\Leftrightarrow \exists \sigma, q: C^\sigma \rightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{M}} c^\sigma$

$$\wedge q \models_{\mathcal{M}} \forall (C \sqsupseteq \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \xrightarrow{v} \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array}, \nexists (\begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \xrightarrow{v} \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \rightarrow \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} = \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array})) \wedge \forall (C \sqsupseteq \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \xrightarrow{e} \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array}, \nexists (\begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \xrightarrow{e} \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} \rightarrow \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array} = \begin{array}{c} \bullet \\ \bullet \\ \bullet \end{array})),$$

the latter part matches the construction of noId , so we get

$$\Leftrightarrow \exists \sigma, q: C^\sigma \rightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{M}} c^\sigma \wedge q \models_{\mathcal{M}} \text{noId}.$$

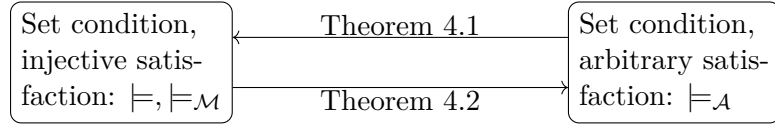
By the definitions of $\text{Cond}_{\mathcal{A}}$ and \mathcal{A} -satisfaction, this yields

$$\Leftrightarrow G \models_{\mathcal{A}} \text{Cond}_{\mathcal{A}}(\exists(P \hookrightarrow C, c)).$$

$G \models_{\mathcal{M}} c \iff G \models_{\mathcal{A}} \text{Cond}_{\mathcal{A}}(c)$ follows directly from the definition of satisfaction by choosing $p: \emptyset \hookrightarrow G$. \square

With HR* conditions under \mathcal{A} -satisfaction, one can express any property expressible with \mathcal{M} -satisfaction as per Definition 3.9. As a corollary from Theorems 4.1 and 4.2, for Set conditions, \mathcal{A} - and \mathcal{M} -satisfaction are equivalent.

Corollary 4.1. Every \mathcal{A} -satisfiable Set condition can be transformed into an equivalent \mathcal{M} -satisfiable Set condition and vice versa.



4.3 HR* conditions with replacement semantics

In Definition 3.9, we defined the semantics of HR* conditions using the simultaneous substitution of all variables. Hyperedges with the same label are replaced by isomorphic graphs. However, it is also possible to replace each variable separately, such that two hyperedges with the same label might be replaced by different graphs.

As mentioned in the previous chapter, a substitution-based semantics has the advantage of providing an easy way to formulate conditions like “there are two paths of equal length between two nodes 1 and 2”. On the other hand, when using substitution-based semantics, a property like “there are two paths of arbitrary length between two nodes 1 and 2” has to be formulated with two different hyperedge labels for the path. With replacement semantics, this can be formulated with the simple intuition of “the $+$ -hyperedge stands for a path of arbitrary length”.

We will show that for HR* conditions under \mathcal{A} -satisfaction, both semantics can be used interchangeably, with the same expressiveness. This result will be used in Chapter 5.2 to help express logical formulas with HR* conditions.

Definition 4.5 (replacement semantics for HR* conditions). The definition for *replacement semantics* for HR* conditions follows Definition 3.9, with σ being a replacement instead of a substitution. \triangle

This can be illustrated with a simple example.

Example 4.6. Regard the condition $\exists(\bullet_1 \overset{+}{\curvearrowright} \bullet_2)$. Using substitution semantics as in Definition 3.9, the condition means “there are two paths of equal length from node 1 to node 2”. Using replacement semantics instead, the two +-labeled hyperedges may be replaced by paths of different length, so the meaning becomes “there are two paths of arbitrary length from node 1 to node 2”. \diamond

For HR* conditions under \mathcal{A} -satisfaction, the substitution of hyperedges (i.e. all edges with the same label are replaced by isomorphic graphs) is equivalent to replacement of hyperedges (i.e. edges with the same label may be replaced by different graphs).

Theorem 4.3 (equivalence of substitution and replacement).

For \mathcal{A} -satisfiability, HR* conditions with replacement semantics are expressively equivalent to HR* conditions with substitution semantics.

$$\boxed{\text{substitution}}_{\mathcal{A}\text{-satisfaction}} \equiv \boxed{\text{replacement}}_{\mathcal{A}\text{-satisfaction}}$$

Remark. Note that the above theorem is valid only for \mathcal{A} -satisfaction. For \mathcal{M} -satisfaction, it is unknown whether replacement and substitution are equivalent.

We prove Theorem 4.3 by proving a lemma for each direction.

Lemma 4.4 (from replacement to substitution).

For every HR* condition c , there is a condition $\text{Rep2Sub}(c)$ such that any graph $G \in \mathcal{G}$ \mathcal{A} -satisfies c by replacement iff G \mathcal{A} -satisfies $\text{Rep2Sub}(c)$ by substitution.

We define $\text{Rep2Sub}(c)$ simply by giving each hyperedge occurring in condition c a unique label and cloning the rules accordingly.

Proof. The proof is straightforward: If every hyperedge in $\text{Rep2Sub}(c)$ has a unique name, it satisfies exactly the same graphs under substitution which are satisfied by c under replacement. \square

We now show that the converse of Lemma 4.4 is also true: We can convert any HR* condition with substitution semantics into an equivalent formula with replacement semantics.

Lemma 4.5 (from substitution to replacement).

There is a transformation Sub2Rep such that for any condition c and any graph $G \in \mathcal{G}$, \mathcal{A} -satisfies c by substitution iff G \mathcal{A} -satisfies Sub2Rep(c) by replacement.

The construction has to simulate the substitution of several hyperedges with the same label by the same graph, using replacement. To ensure that the graphs for same-labeled hyperedges are identical, the same derivation steps have to be performed for both hyperedges. Two hyperedges with the same label are combined into “2-hyperedges” with the double amount of tentacles to grab all the attachment points of both hyperedges. Replacement rules are added for this hyperedge which copy the rule for the single hyperedge twice. For three or more hyperedges with identical labels, the construction works the same way as for two hyperedges.

Construction. Without loss of generality, all HR* conditions are in decompressed normal form as per Definition 4.2. For a variable $x \in \text{Var}$, let x^2 be a variable with $\text{rank}(x^2) = n * \text{rank}(x)$. For a graph G , let $\text{clone}(G, n)$ be an n -fold copy of G :

$$\text{clone}(G, n) := \langle V', E', Y', s_{G'}, t_{G'}, \text{att}_{G'}, \text{lv}_{G'}, \text{le}_{G'}, \text{ly}_{G'} \rangle$$

where V' is the set of pairs (k, v) with $k \in [n]$ and $v \in V$ (analogous for E' and Y'), and $s_{G'}((k, e)) = (k, s_G(e))$ (analogous for the other mappings).

Let $P_x = P + \boxed{x}$, $P_{xx} = P + \boxed{x} + \boxed{x}$ and $P_{x^2} = P + \boxed{x^2}$.

- $\text{Sub2Rep}(\langle \exists(P_x \hookrightarrow P_{xx}, c), \mathcal{R} \rangle) = \langle \exists(P_x \hookrightarrow P_{xx}, \exists(P_{xx} \sqsupseteq P_{2x} \wedge \text{Sub2Rep}(c))), \mathcal{R}' \rangle$
with $\mathcal{R}' = \mathcal{R} \uplus \{x^2 / \text{clone}(R, 2) \mid x/R \in \mathcal{R}\}$.
- $\text{Sub2Rep}(\langle \exists(P \hookrightarrow C, c), \mathcal{R} \rangle) = \exists(P \hookrightarrow C, \text{Sub2Rep}(\langle c, \mathcal{R} \rangle))$ if C contains no pair of identically-labeled hyperedges
- $\text{Sub2Rep}(\langle \exists(P \sqsupseteq C, c), \mathcal{R} \rangle) = \exists(P \sqsupseteq C, \text{Sub2Rep}(\langle c, \mathcal{R} \rangle))$,
- $\text{Sub2Rep}(\langle \text{true}, \mathcal{R} \rangle) = \text{true}$,
- $\text{Sub2Rep}(\langle \neg c, \mathcal{R} \rangle) = \neg \text{Sub2Rep}(\langle c, \mathcal{R} \rangle)$ and
- $\text{Sub2Rep}(\langle c \wedge c', \mathcal{R} \rangle) = \text{Sub2Rep}(\langle c, \mathcal{R} \rangle) \wedge \text{Sub2Rep}(\langle c', \mathcal{R} \rangle)$.



Proof. By structural induction over HR* conditions. Assume that the proposition holds for c, c_i . Let $p: P_x \rightarrow G$. In this proof, $p \models_{\mathcal{A}}^r c$ stands for “ p \mathcal{A} -satisfies c ” and $p \models_{\mathcal{A}}^s c$ stands for “ p \mathcal{A} -satisfies c by substitution”.

Basis. $p \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle \text{true}, \mathcal{R} \rangle) \Leftrightarrow \text{true} \Leftrightarrow p \models_{\mathcal{A}} c$.

Hypothesis. Assume that $q \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle c, \mathcal{R} \rangle)$ iff $q \models_{\mathcal{A}}^s \langle c, \mathcal{R} \rangle$.

Step.

4.3 HR* conditions with replacement semantics

Case $\exists(P_x \hookrightarrow P_{xx}, c)$.

$p \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle \exists(P_x \hookrightarrow P_{xx}, c), \mathcal{R} \rangle)$	Construction
$\Leftrightarrow p \models_{\mathcal{A}}^r \langle \exists(P_x \xrightarrow{\alpha} P_{x2}, \exists(P_{x2} \sqsupseteq P_{xx}, \text{Sub2Rep}(c))), \mathcal{R}' \rangle$	Def. 4.3
$\Leftrightarrow \exists \tau \in \mathcal{R}'^*, q: P_{x2}^\tau \rightarrow G.p = q \circ a^\tau \wedge \exists b: P_{xx}^\tau \hookrightarrow P_{x2}^\tau,$	
$q': P_{xx}^\tau \rightarrow G.q' = q \circ b \wedge q' \models_{\mathcal{A}}^r \text{Sub2Rep}(c)^\tau$	$P_{x2}^\sigma \cong P_{xx}^{\sigma^{-1}}$
$\Leftrightarrow \exists \tau \in \mathcal{R}'^*, q: P_{xx}^\tau \rightarrow G.p = q \circ a^\tau \wedge \exists b: P_{xx}^\tau \hookrightarrow P_{xx}^\tau,$	
$q': P_{xx}^\tau \rightarrow G.q' = q \circ b \wedge q' \models_{\mathcal{A}}^r \text{Sub2Rep}(c)^\tau$	$b = \text{id}_{P_{xx}}, q' = q$
$\Leftrightarrow \exists \tau \in \mathcal{R}'^*, q: P_{xx}^\tau \rightarrow G.p = q \circ a^\tau \wedge q \models_{\mathcal{A}}^r \text{Sub2Rep}(c)^\tau$	$\tau(x) = \sigma(x)$ ²
$\Leftrightarrow \exists \sigma \in \mathcal{R}^*. \exists q: P_{xx}^\sigma \rightarrow G.p = q \circ a^\sigma \wedge \tau \in \mathcal{R}'^*, q \models_{\mathcal{A}}^s \text{Sub2Rep}(c)^\tau$	Ind. hyp.
$\Leftrightarrow \exists \sigma \in \mathcal{R}^*. \exists q: P_{xx}^\sigma \rightarrow G.p = q \circ a^\sigma \wedge q \models_{\mathcal{A}}^s c^\sigma$	Def. 4.3
$\Leftrightarrow p \models_{\mathcal{A}, \sigma}^s \exists(P_x \rightarrow P_{xx}, c)$.	

Case $\exists(P \sqsupseteq C, \langle c, \mathcal{R} \rangle)$.

$p \models_{\mathcal{A}}^r \text{Sub2Rep}(\exists(P \sqsupseteq C, \langle c, \mathcal{R} \rangle))$	Construction
$\Leftrightarrow p \models_{\mathcal{A}}^r \exists(P \sqsupseteq C, \text{Sub2Rep}(\langle c, \mathcal{R} \rangle))$	Def. 4.3
$\Leftrightarrow \exists \sigma' \in \mathcal{R}'^*, b': C^{\sigma'} \rightarrow P.q' = p \circ b' \wedge q' \models \text{Sub2Rep}(\langle c, \mathcal{R} \rangle)$	Ind. hyp.
$\Leftrightarrow \exists \sigma \in \mathcal{R}^*, b: C^\sigma \rightarrow P.q = p \circ b \wedge q \models \langle c, \mathcal{R} \rangle$	Def. 4.3
$\Leftrightarrow p \models_{\mathcal{A}}^s \langle \exists(P \sqsupseteq C, c), \mathcal{R} \rangle$.	

Case $c \wedge c'$.

$p \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle c \wedge c', \mathcal{R} \rangle)$	Construction
$\Leftrightarrow p \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle c, \mathcal{R} \rangle) \wedge p \models_{\mathcal{A}}^r \langle c', \mathcal{R} \rangle$	Ind. hyp.
$\Leftrightarrow p \models_{\mathcal{A}}^s \langle c, \mathcal{R} \rangle \wedge p \models_{\mathcal{A}} \langle c', \mathcal{R} \rangle$	Def. 4.3
$\Leftrightarrow p \models_{\mathcal{A}}^s \langle c \wedge c', \mathcal{R} \rangle$.	

Case $\neg c$.

$p \models_{\mathcal{A}}^r \text{Sub2Rep}(\langle \neg c, \mathcal{R} \rangle)$	Construction
$\Leftrightarrow p \models_{\mathcal{A}}^r \neg \text{Sub2Rep}(\langle c, \mathcal{R} \rangle)$	Ind. hyp.
$\Leftrightarrow p \models_{\mathcal{A}}^s \langle \neg c, \mathcal{R} \rangle$.	

This completes the proof. \square

Example 4.7. The HR* condition $\exists(\bullet \xrightarrow{+} \bullet, \#(\bullet \xrightarrow{+} \bullet, \bullet \xrightarrow{+} \bullet))$ with $\bullet \xrightarrow{+} \bullet ::= \bullet \longrightarrow \bullet \mid$

$\bullet \xrightarrow{+} \bullet$ is \mathcal{A} -satisfied by any graph which has a path between two nodes 1 and 2, but no second path of the same length, since both “+” hyperedges are substituted by a path of the same length. Using replacement instead of substitution, this is equivalent to the HR* condition

$$\exists(\bullet \xrightarrow{+} \bullet, \#(\bullet \xrightarrow{+} \bullet, \bullet \xrightarrow{+} \bullet)) \text{ with } \bullet \xrightarrow{+} \bullet ::= \bullet \longrightarrow \bullet \mid \bullet \xrightarrow{+} \bullet$$

With replacement, the substitution of the two hyperedges by isomorphic graphs is simulated by combining both hyperedges into a single one, where two isomorphic graphs are generated in parallel. \diamond

¹This is valid since pinpoints in P_{xx} and P_{x2} are identical and b is injective.

²By construction of Sub2Rep , \mathcal{R}' contains the same replacement rules for x as \mathcal{R} .

Since Lemmata 4.4 and 4.5 showed that HR* conditions with replacement semantics can be transformed into equivalent HR* conditions with substitution semantics and vice versa, Theorem 4.3 follows: Under \mathcal{A} -satisfaction, HR* conditions with replacement semantics are expressively equivalent to HR* condition with substitution semantics.

Conclusion. As we have just shown, substitution and replacement are both suitable semantics for \mathcal{A} -satisfaction of HR* conditions. Substitution offers some advantages over replacement:

- Substitution is the more “natural” semantics for variables and corresponds to the way variables are replaced e.g. in logic formulas.
- Using substitution, it is very easy to express properties like “two paths have equal length”.
- Converting a condition from replacement semantics to substitution semantics is far easier than the other way round.

However, replacement semantics has its benefits, too:

- Unique symbols can be used that stand for a certain kind of replacement system. For example, a simple path can always be expressed by a hyperedge labeled with $+$, even if several paths of independent length exists. For condition $\exists(\bullet \xrightarrow{+}_1 \bullet \xrightarrow{+}_2 \bullet)$ with $\bullet \xrightarrow{+}_1 \bullet \xrightarrow{+}_2 \bullet ::= \bullet \xrightarrow{+}_1 \bullet \mid \bullet \xrightarrow{+}_1 \bullet \xrightarrow{+}_2 \bullet$, with replacement semantics, both paths may be of different length, as opposed to substitution, where one would have to use different symbols (and according HR systems). For such cases, this notation might be more intuitive.
- In larger HR* conditions, a user might forget about previous use of a variable and involuntarily introduce dependencies into an HR* condition with substitution semantics.

It is thus left up to the reader which semantics are preferable for one use or another.

Open question. For \mathcal{M} -satisfaction, the relationship between substitution and replacement semantics is still unclear.

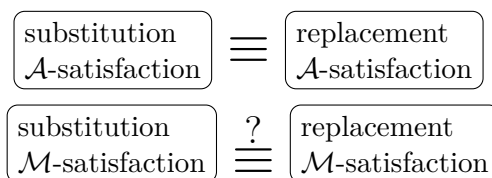


Figure 4.1: The relation of substitution and replacement semantics.

Chapter 5

Expressive power of HR^* conditions

Contents

5.1	Graph formulas	47
5.2	Transforming counting monadic second-order formulas into HR^* conditions	52
5.3	Transforming HR^* conditions into second-order formulas . . .	56

In the following, we will investigate the expressive power of HR^* conditions. This chapter is based on the paper (Radke, 2013) by the author. We start with the definition of first-order, (counting) monadic second-order and second-order formulas on graphs. Chapter 5.2 then shows how to express any counting monadic second-order formula as an HR^* condition. In Chapter 5.3, we show how HR^* conditions can be transformed into second-order formulas.

5.1 Graph formulas

System properties are often expressed with logical formulas. The logic formalisms presented here are all defined over the universe of directed graphs. However, the formalisms differ in the sets of variables over which quantifiers range, yielding different expressive power. In ascending expressive power, we recall first-order (FO), monadic second-order (MSO), counting monadic second-order (CMSO) and second-order (SO) formulas on graphs.

(Courcelle, 1994) compares the languages of graphs definable by monadic second-order formulas, hyperedge replacement grammars and vertex replacement grammars, and (Courcelle, 1996) investigates the expressive power of several logics between first-order and

monadic second-order. For a comprehensive overview on the monadic second-order logic of graphs, see (Courcelle and Engelfriet, 2012). For a general overview on second-order logic, see e.g. (Dalen, 2004) or (Manzano, 2005).

The following presentation of the syntax for graph formulas is oriented at (Courcelle, 1990; Courcelle, 1997).

Notation. In the following, let L be a set of node and edge labels, and let $D_G = V_G \uplus E_G$ be the set of nodes and edges in a graph G over L . Let

\mathcal{V}_0 be the set of individual variables in graph formulas,

\mathcal{V}_1 be the set of set variables in graph formulas,

\mathcal{V}_2 be the set of second-order variables in graph formulas, together with a function $\text{rank}: \mathcal{V}_2 \rightarrow \mathbb{N}$ that maps each variable in \mathcal{V}_2 to its *rank*.

Individual variables are designated by lowercase latin letters x, y, z , while set and second-order variables are designated by uppercase latin letters X, Y, Z . Note that second-order variables include set variables.

Furthermore, let $\{\text{lab}_b \mid b \in L\} \cup \{\text{inc}, \doteq\}$ be the set of predicate symbols over \mathcal{V}_0 , where the unary predicate symbol lab_b assigns a label to a variable, the ternary predicate symbol inc assigns to an edge its source and target, and the binary \doteq states the equality of its elements. We are only interested in logical formulas on graphs, so we do not need other predefined predicate symbols.

For any formula F , $\text{Free}(F)$ denotes the set of all free variables of F , as defined for first-order formulas in (Huth and Ryan, 2004, p. 106). A formula is *closed* iff $\text{Free}(F)$ is empty.

The *existential closure* of a formula F with free variables $\text{Free}(F) = \{x_1, \dots, x_n\}$, written $\exists F$, is the formula F with an existential quantifier added for each free variable, i.e. $\exists x_1, \dots, x_n. F$, yielding a closed formula. The *universal closure* of F is defined analogously for the \forall quantifier.

First-order graph formulas

First-order formulas are a very common way to formulate properties of systems in computer science. With first-order graph formulas, many local properties of graphs can be expressed.

Definition 5.1 (first-order graph formula). The set of *first-order (graph) formulas*, short *FO formulas*, is inductively defined as follows:

- (1) For $b \in L$ and $e, x, y \in \mathcal{V}_0$, $\text{inc}(e, x, y)$, $x \doteq y$ and $\text{lab}_b(x)$ are FO formulas.
- (2) For FO formulas F, F' and $x \in \mathcal{V}_0$, true , $\neg F$, $F \wedge F'$ and $\exists x.F$ are FO formulas.

The semantics $G \llbracket F \rrbracket (\theta)$ of a FO formula F in a non-empty graph G over L under assignment $\theta: \mathcal{V}_0 \rightarrow D_G$ is inductively defined as follows:

- (1) $G[\text{lab}_b(x)](\theta) = \mathbf{true}$ iff $\text{lv}_G(\theta(x)) = b$ or $\text{le}_G(\theta(x)) = b$,
 $G[\text{inc}(e, x, y)](\theta) = \mathbf{true}$ iff $\theta(e) \in E_G$, $s_G(\theta(e)) = \theta(x)$, and $t_G(\theta(e)) = \theta(y)$,
 $G[x \doteq y](\theta) = \mathbf{true}$ iff $\theta(x) = \theta(y)$.
- (2) $G[\mathbf{true}](\theta) = \mathbf{true}$, $G[\neg F](\theta) = \neg G[F](\theta)$, $G[F \wedge F'](\theta) = G[F](\theta) \wedge G[F'](\theta)$,
 $G[\exists x.F](\theta) = \mathbf{true}$ iff $G[F](\theta\{x/d\}) = \mathbf{true}$ for some $d \in D_G$, where $\theta\{x/d\}$ is the modified assignment with $\theta\{x/d\}(x) = d$ and $\theta\{x/d\}(y) = \theta(y)$ otherwise.

A non-empty graph G *satisfies* a FO formula F , denoted by $G \models F$, if for all assignments $\theta: \mathcal{V}_0 \rightarrow D_G$, $G[F](\theta) = \mathbf{true}$. \triangle

Example 5.1. The FO formula

$$\exists e, x, y. \text{inc}(e, x, y) \wedge \neg x \doteq y$$

is satisfied for all graphs which contain an edge from one node to another, disjoint node. Without the “ $\neg x \doteq y$ ” part, it is also satisfied for all graphs which contain a loop edge. \diamond

Notation. In the following, we use the following abbreviations: $\text{edge}(e)$ abbreviates $\exists y. \exists z. \text{inc}(e, x, y)$, $\text{node}(v)$ abbreviates $\neg \text{edge}(v)$, $\text{edg}(x, y)$ abbreviates $\exists e. \text{inc}(e, x, y)$. $\exists x_1, \dots, x_n. F$ abbreviates $\exists x_1. \exists x_2. \dots. \exists x_n. F$ and likewise for the universal quantifier, **false** abbreviates $\neg \mathbf{true}$, $F \vee F'$ abbreviates $\neg(\neg F \wedge \neg F')$, $F \Rightarrow F'$ abbreviates $\neg F \vee F'$, $\forall x. F$ abbreviates $\neg \exists x. \neg F$, and $x \neq y$ abbreviates $\neg x \doteq y$.

Monadic second-order graph formulas

As shown by (Gaifman, 1982), FO formulas are unable to express non-local properties. Therefore, we look at stronger formalisms. Many interesting properties can be formulated by extending the quantifiers: Instead of only quantifying over individual nodes and edges, we allow them to quantify over sets of nodes and edges, and also add a predicate $x \in X$ to relate individual and set variables.

Definition 5.2 (monadic second-order graph formula). Let $\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_1$ be the set of all individual and set variables. The set of *monadic second-order (graph) formulas*, short *MSO formulas*, is inductively defined as follows:

- (1) Every FO formula is an MSO formula.
- (2) For variables $x \in \mathcal{V}_0, X \in \mathcal{V}_1$, the expressions $\exists X.F$ and $x \in X$ are MSO formulas.
- (3) For MSO formulas F and F' , $\neg F$ and $F \wedge F'$ are MSO formulas.

The semantics $G[F](\theta)$ of an MSO formula F in a non-empty graph G under assignment $\theta: \mathcal{V}_0 \cup \mathcal{V}_1 \rightarrow D_G \cup 2^{V_G} \cup 2^{E_G}$ ¹ is inductively defined as follows:

- (1) For an FO formula, the semantics is as for FO formulas.

¹For a set M , 2^M denotes the powerset of M , i.e. the set of all subsets of M .

- (2) $G[\exists X.F](\theta) = \mathbf{true}$ iff $G[F](\theta\{X/D\}) = \mathbf{true}$ for some $D \subseteq D_G$.
 $G[x \in X](\theta) = \mathbf{true}$ iff $\theta(x) \in \theta(X)$.
- (3) $G[\neg F](\theta) = \neg G[F](\theta)$ and $G[F \wedge F'](\theta) = G[F](\theta) \wedge G[F'](\theta)$.

A non-empty graph G *satisfies* an MSO formula F , denoted by $G \models F$, if for all assignments θ , $G[F](\theta) = \mathbf{true}$.

Node-MSO formulas are the subset of MSO formulas where quantification is only allowed over nodes and the expression $\text{inc}(e, x, y)$ is replaced by $\text{edg}(x, y)$ (meaning “there is an edge from node x to node y ”). \triangle

Notation. We abbreviate $\neg\exists X.\neg F$ with $\forall X.F$ and use the other abbreviations as defined for FO formulas.

Example 5.2. The MSO formula

$$\exists X.(\forall y, z.y \in X \wedge \text{edg}(y, z) \Rightarrow z \in X) \wedge (\forall y.(\text{edg}(x_1, y) \Rightarrow y \in X) \Rightarrow x_2 \in X)$$

expresses the property “There is a non-empty path of arbitrary length from node x_1 to node x_2 ” (Courcelle, 1997). \diamond

Remark. MSO formulas quantifying over nodes are less expressive than MSO formulas quantifying over nodes and edges, even for the case of simple² graphs: As shown by (Courcelle, 1997), the property that an (undirected) graph has a Hamiltonian cycle can not be expressed with an MSO formula with quantification over the universe of nodes, but only with an MSO formula which quantifies over the universe of nodes and edges.

Counting monadic second-order graph formulas

With MSO formulas, many interesting graph properties can be formulated. However, the inability of MSO formulas to “count” nodes or edges still leaves room for improvement. We recall counting MSO formulas, which can count nodes and edges modulo some natural number. This enables the formulation of properties like “The graph has an even number of nodes”. Counting monadic second-order formulas are an extension of MSO formulas.

Definition 5.3 (counting monadic second-order formula). The set of *counting monadic second-order (graph) formulas*, short *CMSO formulas*, is inductively defined as follows:

- (1) Every MSO formula is a CMSO formula.
- (2) For a free variable $x \in \mathcal{V}_0$ and $m \geq 1 \in \mathbb{N}$, $\exists^{(m)}x.F(x)$ is a CMSO formula.
- (3) For CMSO formulas F and F' , $\neg F$ and $F \wedge F'$ are CMSO formulas.

The semantics of CMSO formulas under assignment $\theta: \mathcal{V}_0 \cup \mathcal{V}_1 \rightarrow D_G \cup 2^{\mathcal{V}_G} \cup 2^{\mathcal{E}_G}$ is inductively defined as follows.

²A graph is *simple* if it has neither multiple edges nor loops.

- (1) For an MSO formula, the semantics is defined as for MSO formulas.
- (2) $G[\exists^{(m)}x.F(x)](\theta) = \mathbf{true}$ iff $|\{u \in V_G \cup E_G : G[F(x)](\theta\{x/u\})\}| \equiv 0 \pmod{m}$,
for any formula $F(x)$ over free variable x .
- (3) $G[\neg F](\theta) = \neg G[F](\theta)$ and $G[F \wedge F'](\theta) = G[F](\theta) \wedge G[F'](\theta)$.

Node-CMSO formulas are the subset of CMSO formulas for which counting is only allowed over nodes, i.e. $G[\exists^{(m)}x.F(x)](\theta) = \mathbf{true}$ iff $|\{u \in V_G : G[F(x)](\theta\{x/u\})\}| \equiv 0 \pmod{m}$. \triangle

Example 5.3. The CMSO formula $\exists^{(2)}x.\text{node}(x)$ is satisfied for every non-empty graph with an even number of nodes. \diamond

Second-order graph formulas

Further increasing the expressive power is possible by allowing quantification over relations of arbitrary arity.

Definition 5.4 (second-order graph formula). Let $\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_2$ be the set of all individual and second-order variables. The set of *second-order (graph) formulas*, short *SO formulas*, is inductively defined.

- (1) Every MSO formula is an SO formula.
- (2) For a relation variable $X \in \mathcal{V}_2$ with $\text{rank}(X) = k$, an SO formula F and variables $x_1, \dots, x_k \in \mathcal{V}_0$, $\exists X.F$ and $X(x_1, \dots, x_k)$ are SO formulas.
- (3) For SO formulas F and F' , $\neg F$ and $F \wedge F'$ are SO formulas.

For a non-empty graph G , let D_G^\times be the set of all relations over D_G . The semantics $G[F](\theta)$ of an SO formula F under assignment $\theta: \mathcal{V} \rightarrow D_G \cup D_G^\times$ is inductively defined as follows:

- (1) For an MSO formula, the semantics is defined as for MSO formulas.
- (2) $G[\exists X.F](\theta) = \mathbf{true}$ iff $G[F](\theta\{X/D\}) = \mathbf{true}$ for some relation $D \in D_G^\times$
 $G[X(x_1, \dots, x_k)](\theta) = \mathbf{true}$ iff $(G[x_1](\theta), \dots, G[x_k](\theta)) \in \theta(X)$.
- (3) $G[\neg F](\theta) = \neg G[F](\theta)$ and $G[F \wedge F'](\theta) = G[F](\theta) \wedge G[F'](\theta)$.

A non-empty graph G *satisfies* an SO formula F , denoted by $G \models F$, iff, for all assignments $\theta: \mathcal{V} \rightarrow D_G \cup D_G^\times$, $G[F](\theta) = \mathbf{true}$. \triangle

Example 5.4 (non-trivial automorphism). In graph theory, an *automorphism* is an isomorphism from a graph to itself. We recall from Definition 2.2 that an isomorphism is a total, surjective and injective mapping that preserves labels, nodes and edges. An automorphism is *non-trivial* if it is not an identity morphism. The SO formula below is true for every graph which has a non-trivial automorphism:

$$\exists X. [\beta_{\text{inj}}(X) \wedge \beta_{\text{tot}}(X) \wedge \beta_{\text{surj}}(X) \wedge \beta_{\text{ntniv}}(X) \wedge \beta_{\text{predg}}(X) \wedge \beta_{\text{prlab}}(X)]$$

where the subformulas are defined as follows:

- $\beta_{\text{inj}}(X) = \forall x, y, z. (X(x, y) \wedge X(x, z)) \Rightarrow y \doteq z \wedge (X(x, z) \wedge X(y, z)) \Rightarrow x \doteq y$ expresses the fact that relation X is injective and functional,
- $\beta_{\text{tot}}(X) = \forall x \exists y. X(x, y)$ expresses the fact that X is total,
- $\beta_{\text{surj}}(X) = \forall x \exists y. X(y, x)$ expresses the fact that X is surjective,
- $\beta_{\text{ntniv}}(X) = \exists x, y. x \neq y \wedge X(x, y)$ expresses the fact that X is non-trivial,
- $\beta_{\text{predg}}(X) = \forall e, x, y, \bar{e}, \bar{x}, \bar{y}. (\text{inc}(e, x, y) \wedge X(e, \bar{e}) \wedge X(x, \bar{x}) \wedge X(y, \bar{y})) \Rightarrow \text{inc}(\bar{e}, \bar{x}, \bar{y})$ expresses the fact that X preserves edges, i.e. for every pair of nodes x, y connected by an edge and related to nodes \bar{x}, \bar{y} by relation X , \bar{x} and \bar{y} are connected by an edge,
- $\beta_{\text{prlab}}(X) = \forall x, y. X(x, x') \Rightarrow \bigvee_{b \in \mathbb{L}} \text{lab}_b(x) \wedge \text{lab}_b(x')$ expresses the fact that X preserves labels, i.e. x and x' have identical labels whenever they are related by X .
◇

5.2 Transforming counting monadic second-order formulas into HR* conditions

One may ask about the relation between logical formulas and graph conditions. We show that every CMSO formula can be translated into an equivalent HR* condition.

Theorem 5.1 (From CMSO to HR*).

There is a transformation Cond_F such that for every CMSO graph formula F , there is an HR* condition $\text{Cond}_F(F)$ such that for all graphs G ,

$$G \models F \iff G \models \text{Cond}_F(F).$$

Instead of a direct translation from a CMSO graph formula into an HR* condition, we first transform it into an \mathcal{A} -satisfiable Set condition and use Theorem 4.1 to show that this can be transformed into an (\mathcal{M} -satisfiable) HR* condition. Figure 5.1 shows an illustration of the proof idea.

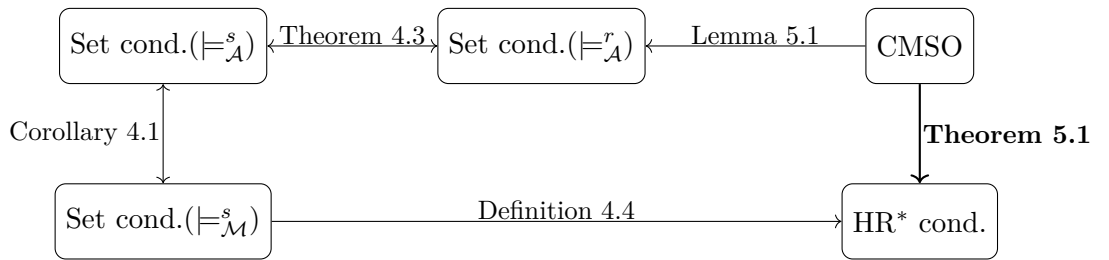


Figure 5.1: From CMSO graph formula to HR* condition: Proof idea

Lemma 5.1 (From CMSO to \mathcal{A} -satisfiable Set conditions).

There is a transformation Cond such that for every CMSO formula F , there is an HR* condition $\text{Cond}(F)$ such that for all graphs G ,

$$G \models F \iff G \models_{\mathcal{A}} \text{Cond}(F).$$

Idea. The following construction uses the inc and lab relations from a formula to construct a graph. For counting CMSO formulas of the form $\exists^{(m)}v.F$, hyperedge replacement is used to represent the set and count its members: We construct a grammar which generates the language $\{G \in \mathcal{G} \mid |E_G| = 0 \text{ and } \exists k \in \mathbb{N}. |V_G| = k * m\}$ of all discrete graphs (i.e. with no edges) with $k * m$ nodes, where m is a fixed number given by the formula and $k \in \mathbb{N}$ is variable. We then check that the property F to be counted holds for all nodes inside and for none outside the generated subgraph. The existence of a node or edge, represented by $\exists x.F$ in the CMSO formula, is represented using the abbreviation $\exists(\circ_1) := \bigvee_{a \in L_V} \exists(\textcircled{a}_1)$ from Chapter 3.2 to range over all possible labels. When the label is fixed by a formula $\text{lab}_b(x)$ in F , disjunctions with other labels evaluate to **false** and can be removed from the resulting HR* condition.

Construction (Cond). For any natural number $m \in \mathbb{N} - \{0\}$, let D_m be a discrete graph with m nodes, and let $\boxed{Y_m} ::= \emptyset \mid \boxed{Y_m} D_m$ be a hyperedge replacement system generating graphs with $m * k$ nodes for some $k \in \mathbb{N}$.

For a node-CMSO formula F , $\text{Cond}(F) = \text{Cond}(\emptyset, F)$. For a graph P and a formula F , $\text{Cond}(P, F)$ is defined inductively as follows:

$$\begin{aligned} \text{Cond}(P, \text{inc}(e, x, y)) &= \mathbf{true} \text{ if } e \in E_P, s_P(e) = x, t_P(e) = y \text{ and } \mathbf{false} \text{ otherwise} \\ \text{Cond}(P, \text{lab}_b(x)) &= \mathbf{true} \text{ if } \text{lv}_P(x) = b \text{ or } \text{le}_P(x) = b \text{ and } \mathbf{false} \text{ otherwise} \\ \text{Cond}(P, x \doteq y) &= \mathbf{true} \text{ if } x, y \in P \text{ and } x = y, \text{ and } \mathbf{false} \text{ otherwise} \\ \text{Cond}(P, \mathbf{true}) &= \mathbf{true} \\ \text{Cond}(P, \exists x.F) &= \exists(P + \circ_x, \text{Cond}(P + \circ_x, F)) \vee \exists(P + \circ_1 \xrightarrow{\circ_2}, \text{Cond}(P + \circ_1 \xrightarrow{\circ_2}, F)) \\ \text{Cond}(P, \exists X.F) &= \exists(P + \boxed{Y_1}, \text{Cond}(P + \boxed{Y_1}, F)) \\ \text{Cond}(P, x \in X) &= \exists(P \sqsupseteq (x + (P - X))) \\ \text{Cond}(P, \exists^{(m)}v.F) &= \exists(\boxed{Y_m}, \forall(\boxed{Y_m} \sqsupseteq \circ_v, \text{Cond}(F(v))) \wedge \nexists(\boxed{Y_m} \circ_v, \text{Cond}(F(v)))) \\ \text{Cond}(P, \neg F) &= \neg \text{Cond}(P, F) \\ \text{Cond}(P, F \wedge F') &= \text{Cond}(P, F) \wedge \text{Cond}(P, F') \end{aligned}$$



Remark. Concerning labels, the construction creates a disjoint union of every possible label for every node or edge. When the formula gives a concrete label to an item x using $\text{lab}_b(x)$, all differently-labeled nodes x in the HR* condition evaluate to **false** and can be removed from the condition. While theoretically sound, this is not particularly efficient in practice. Instead, an implementation should keep the label of an item in a special “uncertain” state and set it to a concrete label when given by the formula. An item in the “uncertain” state can later be matched to an item with any label.

The following proof proceeds by induction over the structure of CMSO formulas. Aside from the definition of CMSO and HR* satisfaction (Definitions 5.3 and 3.9, respectively) and the induction hypothesis (called (ihyp) in the proof), it uses the fact that inclusions are injective morphisms, as well as some simple arithmetic and set theory. For node-MSO formulas, the proof is similar to the one in (Habel and Radke, 2010).

Proof (of Lemma 5.1). We proceed by induction over the structure of the formula. Without loss of generality, we assume that the HR* conditions are interpreted with \mathcal{A} -satisfiable replacement semantics (see Definitions 4.3 and 4.5 and Theorems 4.2 and 4.3). In the following inductive proof, let $P = \text{Ran}(\theta)$ and let (ihyp) stand for the inductive hypothesis.

Basis. By the definition of HR* conditions (Def. 3.9) and construction Cond, we have: $G \llbracket \text{true} \rrbracket(\theta) \Leftrightarrow G \models_{\mathcal{A}} \text{true} \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, \text{true})$.

Hypothesis. Assume that $G \llbracket F \rrbracket(\theta) \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, F)$ with $P = \text{Ran}(\theta)$. (ihyp)
Step.

$$\begin{aligned}
 G \llbracket \text{inc}(e, x, y) \rrbracket(\theta) &= \text{true} && \text{Def. 5.3} \\
 \Leftrightarrow \theta(e) \in E_G, s_G(\theta(e)) = \theta(x) \text{ and } t_G(\theta(e)) = \theta(y) &&& m(D_P) := \theta(D_P) \\
 \Leftrightarrow m(e) \in E_G, s_G(m(e)) = m(x) \text{ and } t_G(m(e)) = m(y) &&& \text{Def. 2.2} \\
 \Leftrightarrow e \in E_G \wedge x, y \in V_G \wedge m(s_G(x)) = m(y) \wedge m(t_G(x)) = m(x) &&& \text{Constr., (ihyp)} \\
 \Leftrightarrow m \models_{\mathcal{A}} \text{Cond}(P, \text{inc}(e, x, y)). &&&
 \end{aligned}$$

$$\begin{aligned}
 G \llbracket \text{lab}_b(x) \rrbracket(\theta) &= \text{true} && \text{Def. 5.3} \\
 \Leftrightarrow \text{lv}_G(\theta(x)) = b \text{ or } \text{le}_G(\theta(x)) = b &&& m(D_P) := \theta(D_P) \\
 \Leftrightarrow \text{lv}_P(m(x)) = b \vee \text{le}_P(m(x)) = b &&& \text{Constr., (ihyp)} \\
 \Leftrightarrow m \models_{\mathcal{A}} \text{Cond}(P, \text{lab}_b(x)). &&&
 \end{aligned}$$

$$\begin{aligned}
 G \llbracket x \doteq y \rrbracket(\theta) &= \text{true} \Leftrightarrow \theta(x) = \theta(y) && m(P) := \theta(D_P) \\
 \Leftrightarrow x, y \in P \wedge m(x) = m(y) &&& \text{Constr., (ihyp)} \\
 \Leftrightarrow \text{Cond}(P, x \doteq y). &&&
 \end{aligned}$$

$$\begin{aligned}
 G \llbracket \exists x.F \rrbracket(\theta) &= \text{true} \Leftrightarrow \exists \theta, d \in D_G. G \llbracket F \rrbracket(\theta\{x/d\}) = \text{true} && \text{set theory} \\
 \Leftrightarrow \exists P \subseteq G, x.x \notin P \wedge (x \in V_G \vee x \in E_G) \wedge (P+x) \llbracket F \rrbracket(\theta\{x/d\}) &&& \text{Def. 3.9, (ihyp)} \\
 \Leftrightarrow G \models_{\mathcal{A}} \exists(P + \overset{\bullet}{x}, \text{Cond}(P + \overset{\bullet}{x}, F)) \vee \exists(P + \overset{\bullet}{1} \xrightarrow{x} \overset{\bullet}{2}, \text{Cond}(P + \overset{\bullet}{1} \xrightarrow{x} \overset{\bullet}{2}, F)) &&& \text{Construction} \\
 \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, \exists x.F). &&&
 \end{aligned}$$

$$\begin{aligned}
 G \llbracket \exists X.F \rrbracket(\theta) &= \text{true} \Leftrightarrow \exists \theta, D \subseteq D_G. G \llbracket F \rrbracket(\theta\{X/D\}) && \text{set theory} \\
 \Leftrightarrow \exists P, X \subseteq G. P \cap X = \emptyset \wedge (X \subseteq V_G \vee X \subseteq E_G) \wedge (P+X) \llbracket F \rrbracket(\theta\{X/D\}) &&& \text{Def. 3.9, (ihyp)} \\
 \Leftrightarrow G \models_{\mathcal{A}} \exists(P + \overline{Y}_1, \text{Cond}(P + \overline{Y}_1, F)) &&& \text{Construction} \\
 \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, \exists X.F). &&&
 \end{aligned}$$

$$\begin{aligned}
 G \llbracket x \in X \rrbracket(\theta) &= \text{true} \Leftrightarrow \theta(x) \in \theta(X) && \text{Def. 5.3} \\
 \Leftrightarrow \exists P. \theta(x) \subseteq \theta(X) \subseteq P &&& \text{set theory} \\
 \Leftrightarrow \exists P. \theta(x) + (P - \theta(X)) \subseteq P &&& \text{Def. 2.2} \\
 \Leftrightarrow \exists b: C \hookrightarrow P \text{ with } C = \theta(x) + (P - \theta(X)) &&& \text{Def. 3.9} \\
 \Leftrightarrow G \models_{\mathcal{A}} \exists(P \supseteq C) \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, x \in X). &&&
 \end{aligned}$$

5.2 Transforming counting monadic second-order formulas into HR* conditions

$G[\neg F](\theta) = \mathbf{true} \Leftrightarrow \neg G[F](\theta)$	Def. 3.9, (ihyp)
$\Leftrightarrow G \models_{\mathcal{A}} \neg \text{Cond}(P, F) \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, \neg F)$.	
$G[F \wedge F'](\theta) = \mathbf{true} \Leftrightarrow G[F](\theta) \wedge G[F'](\theta)$	Def. 3.9, (ihyp)
$\Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, F) \wedge \text{Cond}(P, F') \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(P, F \wedge F')$.	
$G[\exists^{(m)}x.F(x)](\theta) = \mathbf{true}$	Construction
$\Leftrightarrow \{v \in V_G : G \models \phi(v)\} \equiv 0 \pmod{m}$	Arithmetic
$\Leftrightarrow \exists n \in \mathbb{N}. \{v \in V_G : G \models \phi(v)\} = n * m$	Ind. hyp.
$\Leftrightarrow \exists n \in \mathbb{N}. \{v \in V_G : G \models_{\mathcal{A}} \text{Cond}(\phi(v))\} = n * m$	Arithmetic
$\Leftrightarrow \exists n \in \mathbb{N}. \{\bullet_x \subseteq V_G \mid \bullet_x \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma\} \geq n * m$	
$\quad \wedge \neg \{\bullet_x \subseteq V_G \mid \bullet_x \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma\} \geq n * m + 1$	Set theory
$\Leftrightarrow \exists n \in \mathbb{N}. \exists D_{n * m} \subseteq G. \forall (\bullet_x \subseteq D_{n * m}. \bullet_x \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma)$	
$\quad \wedge \nexists (D_{n * m} + \bullet_x \subseteq G. \bullet_x \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma)$	Def. 2.2
$\Leftrightarrow \exists n \in \mathbb{N}. \exists q_a : D_{n * m} \hookrightarrow G. p = q_a \circ a \wedge$	
$\quad \forall q_b : \emptyset \hookrightarrow \bullet_x. q_b(\bullet_x) \subseteq q_a(D_{n * m}) \wedge q_b \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma \wedge$	
$\quad \nexists (q_c : D_{n * m} + \bullet_x \hookrightarrow G. q_a = q_c \circ c \wedge q_c \models_{\mathcal{A}} \text{Cond}(\phi(x))^\sigma)$	Def. 3.9
$\Leftrightarrow \exists n \in \mathbb{N}. p \models_{\mathcal{A}} \exists (D_{n * m}, \forall (D_{n * m} \supseteq \bullet_x,$	
$\quad \text{Cond}(\phi(x))) \wedge \nexists (D_{n * m} \bullet_x, \text{Cond}(\phi(x)))$	$\boxed{Y}^\sigma = D_{n * m}$
$\Leftrightarrow p \models_{\mathcal{A}} \exists (\boxed{Y}, \forall (\boxed{Y} \supseteq \bullet_x, \text{Cond}(\phi(x))) \wedge \nexists (\boxed{Y} \hookrightarrow \boxed{Y} \bullet_x, \text{Cond}(\phi(x))))$	Construction
$\Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(\exists^{(m)}x.\phi(x)) \Leftrightarrow p \models_{\mathcal{A}} \text{Cond}(\exists^{(m)}x.\phi(x))$	

This concludes the proof. □

We can now use Theorem 4.1 and the definition of Set conditions to prove Theorem 5.1.

Proof (of Theorem 5.1). Given a CMSO graph formula F , we let $\text{Cond}_F(F) := \text{Cond}_{\mathcal{M}}(\text{Cond}(F))$. By Lemma 5.1, for any graph G , $G \models F \Leftrightarrow G \models_{\mathcal{A}} \text{Cond}(F)$, and by Theorem 4.1, $G \models_{\mathcal{M}} \text{Cond}_{\mathcal{M}}(\text{Cond}(F))$. Since, by Definition 4.4, every Set condition is also an HR* condition, we have $G \models F \Leftrightarrow G \models \text{Cond}_{\mathcal{M}}(\text{Cond}(F)) \Leftrightarrow G \models \text{Cond}_F(F)$. □

Using node-counting, it is possible to simulate edge-counting. A property $P(e)$ is valid for a number $n \equiv 0 \pmod{m}$ edges iff it is valid for n outgoing edges of k nodes. Thus, one can group the nodes by the number of outgoing edges which fulfill P , and count the nodes in each group.

We already showed that HR* condition can count nodes modulo m . (Outgoing) edges can also be counted modulo m using an HR system which generates “stars”, i.e. graphs with a node v_0 in the middle, from which edges go out to otherwise isolated nodes, as seen in Figure 5.2. HR systems for generating stars are described in more detail in (Habel, 1992). In our case, however, the star graphs may be “collapsed”, i.e. a node may have more than one outgoing edge to a single node. This can also be described with an HR system.



Figure 5.2: A star-generating HR system and some exemplary (undirected) stars.

Example 5.5 (edge-counting modulo 2). For $m = 2$, the edge-CMSO formula $\exists^{(2)}e. \text{edge}(e) \wedge P(e)$ is valid if $P(e)$ is valid for an even number of edges. The number of edges satisfying $P(e)$ is even iff (a) there is an arbitrary number k_0 of nodes with an even number of edges e satisfying $P(e)$ and (b) an even number $k_1 \equiv 0 \pmod{2}$ of nodes with an odd number of edges e satisfying $P(e)$. This scheme can be translated into an HR* condition, using the construction for node-counting and HR systems which equip a node with an even (or uneven) number of edges, similar to the HR system in Figure 5.2.

$$\begin{aligned} & \exists(\overline{N^0} \mid \overline{N^1}), \quad \exists(\overline{N^0} \mid \overline{N^1} \supseteq \overline{N^0}, \forall(\overline{N^0} \supseteq \bullet, e_0(\bullet))) \\ & \wedge \exists(\overline{N^0} \mid \overline{N^1} \supseteq \overline{N^1}, \forall(\overline{N^1} \supseteq \bullet, e_1(\bullet))) \\ & \text{where } e_0(\bullet) = \exists(\bullet \text{---} \overline{E^0}), \not\exists(\bullet \text{---} \overline{E^0}) \\ & \text{and } e_1(\bullet) = \exists(\bullet \text{---} \overline{E^1}), \not\exists(\bullet \text{---} \overline{E^1}) \end{aligned}$$

The HR systems $\overline{N^0} ::= \emptyset \mid \overline{N^0} \bullet$ and $\overline{N^1} ::= \bullet \mid \overline{N^1} \bullet$ generate discrete graphs with an even (N^0) or odd (N^1) number of nodes.

The HR systems $\bullet \text{---} \overline{E^0} ::= \bullet \mid \bullet \text{---} \bullet$ and $\bullet \text{---} \overline{E^1} ::= \bullet \text{---} \bullet \mid \bullet \text{---} \bullet \text{---} \bullet$ add an even (for E^0) or uneven (E^1) number of edges to node \bullet as explained above. Subconditions $e_0(\bullet)$ and $e_1(\bullet)$ ensure that every node \bullet has an even (for e_0) or an odd (for e_1) number of edges.

A similar scheme can be used for any m , although it gets rather complicated for larger values of m . \diamond

5.3 Transforming HR* conditions into second-order formulas

With a lower bound for the expressiveness of HR* conditions established, we shift our attention to an upper bound. Second-order formulas are used to simulate HR* conditions.

Theorem 5.2 (from HR* conditions to SO formulas).

For every HR* condition c over \emptyset , there is a second-order graph formula $\text{SO}(c)$ such that for all graphs $G \in \mathcal{G}$,

$$G \models c \iff G \models \text{SO}(c).$$

Idea. The transformation SO from an HR* condition to a logical formula has to capture several things, which are done by appropriate sub-constructions:

5.3 Transforming HR^* conditions into second-order formulas

1. The logical structure of the HR^* condition has to be preserved. This is simple, as the Boolean operators and quantifiers of HR^* conditions can be represented by the same operators in SO formulas.
2. The graph morphisms and graphs in HR^* conditions have to be translated into an SO formula. This is done by sub-construction $\text{SOgra}(G, F)$, where G is a graph to be represented and F is some subformula (which may be yielded by some other part of the construction). The mappings of morphisms are preserved through the use of identical variable names.
3. The hyperedge replacement system, along with the process of hyperedge replacement, have to be encoded in SO formulas. Sub-construction SOsys fulfills this task. Hyperedges in the condition are represented by relations in the formula.
4. For HR^* conditions of the form $\exists(P \sqsupseteq C, c)$, we need to represent the sets P^σ and C^τ for some substitutions σ, τ . This is done by sub-construction SOset .

Figure 5.3 shows the dependencies of the parts of construction SO.

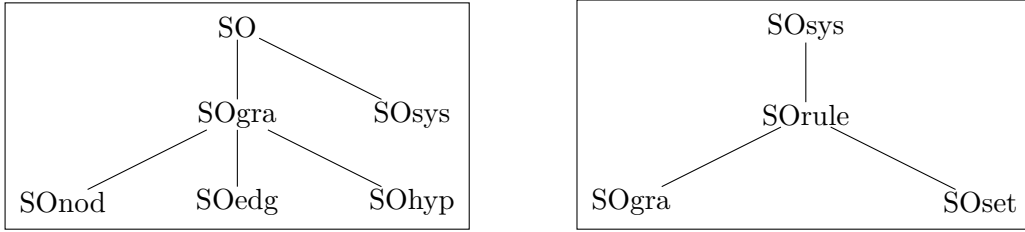


Figure 5.3: Overview of the parts of construction SO and their dependencies.

Construction. Without loss of generality, $P \hookrightarrow C$ is an inclusion. For a condition c with HR system \mathcal{R} and hyperedge label set Var , we let $\text{SO}(\langle c, \mathcal{R} \rangle) = \bigwedge_{x \in \text{Var}} (\exists x. \text{SOsys}(\mathcal{R}) \wedge \text{SO}(c))$ and define

- (1) $\text{SO}(\text{true}) = \text{true}$.
- (2) $\text{SO}(\exists(P \hookrightarrow C, c)) = \text{SOgra}(C - P, \text{SO}(c))$.
- (3) $\text{SO}(\exists(P \sqsupseteq C, c)) = \text{SOgra}(C, \exists X_P, X_C. (\text{SOset}(P, X_P) \wedge \text{SOset}(C, X_C) \wedge X_C \subseteq X_P \wedge \text{SO}(c)))$,
where X_P, X_C are fresh second-order variables of rank 1 (i.e. set variables) and the relation \subseteq is constructed in SO logic as usual: $X_C \subseteq X_P = \forall x. x \in X_C \Rightarrow \exists y \in X_P. x \doteq y$.
- (4) $\text{SO}(\neg c) = \neg \text{SO}(c)$ and $\text{SO}(c \wedge c') = \text{SO}(c) \wedge \text{SO}(c')$.



The construction is straightforward for HR* conditions of the form (1) and (4) as given in Definition 3.8, as these have equivalent constructs in SO formulas. For HR* conditions of form (2), it suffices to state the existence of the items in $C - P$ and to translate subcondition c into an SO formula, too. The construction gets a bit more complicated for case (3). The SO formula has to state that graph C exists, that the sets X_C of nodes and edges in C^τ are subsets of the set X_P of nodes and edges in P^σ for some substitutions τ and σ , and that P^σ includes C^τ .

We show the construction SOgra and its sub-constructions SONod, SOedg and SOhyp. F constitutes an arbitrary subformula which may be nested inside.

The construction is split in three parts for nodes, edges and hyperedges, respectively. The construction for nodes and edges is quite straightforward: we state the existence of every node and edge and then specify the node and edge labels and the incidence relation for the edges. For the hyperedges, we state the existence of each hyperedge label x with $\text{rank}(x) = k$ as a k -ary relation $x(v_1, \dots, v_k)$, where the elements v_1, \dots, v_k represent the attachment points of the tentacles.

Construction. For a set A and SO formula F , let $\exists F$ be the existential closure of F and $\dot{\exists}F = \exists F \wedge \bigwedge_{\substack{a,b \in A \\ a \neq b}} (-a \dot{=} b)$ be the existential closure of F with disjointness check. Define the universal closure $\forall F$ analogously. For a graph with variables G and an SO formula F , we define

$$\begin{aligned} \text{SOgra}(G, F) &= \text{SONod}(G, \text{SOedg}(G) \wedge \text{SOhyp}(G) \wedge F) \\ \text{SONod}(G, F) &= \exists \bigwedge_{v \in V_R} \text{lab}_{l_G}(v)(v) \wedge F \\ \text{SOedg}(G) &= \exists \bigwedge_{e \in E_R} \text{lab}_{l_G}(e)(e) \wedge \text{inc}(e, s_G(e), t_G(e)) \\ \text{SOhyp}(G) &= \dot{\exists} \bigwedge_{l_{y_G}(y) | y \in Y_R} \cdot l_{y_G}(y)(\text{att}_G(y)_{1, \dots, k}) \end{aligned}$$



Before we can continue, we have to make sure that every hyperedge in the HR* condition has at least one tentacle: Since we translate hyperedges of rank k into k -ary relations, we would otherwise end up with 0-ary relations. An HR* condition where each hyperedge has a minimum of one tentacle is in *one-tentacle normal form*.

Such a normal form is easily constructed: For each x -labeled hyperedge $y \subseteq C$ with zero tentacles in a condition $\exists(P \hookrightarrow C, c)$ with replacement system \mathcal{R} , replace the hyperedge x with a hyperedge $\bullet_{\Gamma} \overline{x}$ with 1 tentacle and adapt the replacement system accordingly by adding rules for x' analogous to the rules for x . This can be seen as a special case of the Integrate_n construction introduced later in Chapter 6.2.

If the replacement system contains the empty graph \emptyset as right-hand side of a rule, replace \emptyset by \bullet and $\exists(P \hookrightarrow C, c)$ by the disjunction $\exists(P \hookrightarrow C, c) \vee \exists(P \hookrightarrow C_{\overline{y}}, c_{\overline{y}})$, where $C_{\overline{y}}$ is C with the image of y removed and likewise for $c_{\overline{y}}$.

Example 5.6. The HR* condition $\text{EVEN} = \exists(\underline{\mathbb{Z}}, \#(\underline{\mathbb{Z}} \bullet))$ from Example 4.5 is transformed to the one-tentacle normal form

$$\exists(\bullet_{\Gamma} \overline{\mathbb{Z}}, \#(\bullet_{\Gamma} \overline{\mathbb{Z}} \bullet)) \vee \exists(\emptyset, \#(\bullet)) \text{ with } \overline{\mathbb{Z}} ::= \bullet \mid \overline{\mathbb{Z}} \bullet \mid \overline{\mathbb{Z}} \bullet_{\Gamma}$$



5.3 Transforming HR* conditions into second-order formulas

The following transformation $\text{SOsys}(\mathcal{R})$ expresses the process of hyperedge replacement for a given set \mathcal{R} of HR rules using SO formulas.

The main idea of the construction is to represent hyperedges as relations over nodes. A hyperedge with k nodes is represented as a k -ary predicate $x(v_1, \dots, v_k)$, where the elements v_1, \dots, v_k represent the nodes attached to the hyperedge by its k tentacles. In order to keep track of all nodes and edges that replace each hyperedge in a graph G^σ , we use $(k+1)$ -ary predicates $\text{Set}_x(v_1, \dots, v_k, o)$, representing a set of objects o which is dependent on a x -labeled hyperedge attached to points v_1, \dots, v_k . Let $o \in \text{Set}_x(v_1, \dots, v_k)$ abbreviate $\text{Set}_x(v_1, \dots, v_k, o)$, which denotes that o is element of a set dependent on x and v_1, \dots, v_k .

Construction. For any rule x/R with $\text{rank}(x) = k$ and every HR system \mathcal{R} , let

$$\begin{aligned} \text{SOsys}(\mathcal{R}) &= \bigwedge_{x \in \text{Var}} (\text{Set}_x \cdot \forall v_1, \dots, v_k. x(v_1, \dots, v_k) \Rightarrow \bigvee_{x/R \in \mathcal{R}} \text{SORule}(x/R)) \\ \text{SORule}(x/R) &= \text{SOgra}(R, \text{SOset}(R, \text{Set}_x(v_1, \dots, v_k))) \end{aligned}$$


where x and Set_x are predicates in the formula. Construction SOset is needed to keep track of the items in G^σ and will be explained in the following. 

In order to translate HR* conditions of the form $\exists(P \sqsupseteq C, c)$, we need sets of every object in P^σ and C^σ , i.e. *after* the substitution of the hyperedges by substitution σ . This is the role of transformation $\text{SOset}(R, X)$. For a graph with variables R , it ensures that every node and edge in R^σ (i.e. after replacing any hyperedges left in R) is member of the set X in the SO formula.

The construction begins by stating that all nodes and edges of graph R are in set X . Then, for every hyperedge y in R , it uses the predicate $\text{Set}_x(u_1, \dots, u_{\text{rank}(y)}, o)$ with $x = \text{ly}_R(y)$, representing the set of items o in y^σ , and ensures that every item is an element in X . Iteratively, this ensures that X contains every node and edge in R^σ .

Construction. For any graph R and unary variable X ,

$$\text{SOset}(R, X) = \bigwedge_{o \in D_R} o \in X \wedge \bigwedge_{y \in Y_R} \forall v_1, \dots, v_k. \forall o. o \in \text{Set}_x(v_1, \dots, v_k) \Rightarrow o \in X$$

where $x = \text{ly}_G(y)$ is the label and $k = \text{rank}(y)$ the rank of hyperedge y . 

Example 5.7. For the left-hand graph G from Example 3.1, SOset yields the formula below. The v_i and u_i are nodes, while the e_i are edges, and u is the hyperedge.

$$\begin{aligned} \text{SOset}(G, X) &= v_1, \dots, v_5, e_1, \dots, e_7 \in X \wedge u(v_5, v_1, v_3, v_4) \\ &\quad \wedge \forall v'_1, \dots, v'_4. o \in \text{Set}_u(v'_1, \dots, v'_4) \Rightarrow o \in X \end{aligned} \quad \diamond$$

These constructions simulate the derivation process of hyperedge replacement with an SO formula. For SORule , every hyperedge, i.e. every tuple $x(v_1, \dots, v_{\text{rank}(x)})$ implies the existence of the right-hand side of a rule x/R . Since R can itself contain hyperedges, an arbitrary number of derivations over \mathcal{R} can be simulated by SOsys . Construction SOset ensures that all equally-labeled hyperedges are substituted by identical graphs (up to isomorphism).

Example 5.8. We show the translation of the HR* condition $c = \exists(\emptyset \hookrightarrow \textcircled{a}_1 \xrightarrow{+} \textcircled{c}_2 \xrightarrow{d})$ with HR system $\bullet_1 \xrightarrow{+} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2$. This condition is satisfied for every graph which contains a path (of nodes and edges with the not-drawn \square label) from an a-labeled node to a c-labeled node with a d-labeled loop.

$$\begin{aligned} \text{SO}(c) &= \exists + . \text{SOgra}(\textcircled{a}_1 \xrightarrow{+} \textcircled{c}_2 \xrightarrow{d}, \text{true}) \wedge \\ &\quad (\forall u_1, u_2. \text{SORule}(+ / \bullet_1 \longrightarrow \bullet_2) \vee \text{SORule}(+ / \bullet_1 \longrightarrow \bullet \xrightarrow{+} \bullet_2)) \\ &\equiv \exists +, v_1, v_2. \text{lab}_a(v_1) \wedge \text{lab}_c(v_2) \wedge \exists e_1. \text{lab}_d(e_1) \wedge \text{inc}(e_1, v_2, v_2) \wedge +(v_1, v_2) \\ &\quad \wedge (\forall u_1, u_2. \\ &\quad \quad + (u_1, u_2) \Rightarrow \exists e'. \text{lab}_{\square}(e') \wedge \text{inc}(e', u_1, u_2) \vee \\ &\quad \quad + (u_1, u_2) \Rightarrow \exists u_3. \text{lab}_{\square}(u_3) \wedge \exists e'. \text{lab}_{\square}(e') \wedge \text{inc}(e', u_1, u_3) \wedge + (u_3, u_2)) \end{aligned}$$

Intuitively, this formula means that there are a relation $+$, two nodes v_1, v_2 labeled with a and c, respectively, v_2 has a d -labeled loop, v_1 and v_2 are in a relation $+(v_1, v_2)$, and for each pair u_1, u_2 of nodes in relation $+$, there is either an edge from u_1 to u_2 or an edge from u_1 to a \square -labeled node which is in $+$ -relation to u_2 , which effectively builds a path between v_1 and v_2 . Note that the set equality constraints are not needed, as there is only one hyperedge to be substituted. \diamond

We now prove the correctness of the constructions, and, therefore, Theorem 5.2. We do this by first proving a two-part lemma.

The first part of the lemma states that the formula $\text{SOgra}(R, \text{true})$ is satisfied for every graph G which has a (possibly non-injective) image of R as a subgraph, disregarding the hyperedges. This part is used to transform conditions $\exists(\emptyset \hookrightarrow R)$, where R has no hyperedges (or we just disregard them).

The second part of the lemma intuitively states that constructions SOgra and SOsys together simulate the derivation process of a graph grammar.

Lemma 5.2 (formulas for hypergraphs and replacement systems).

For every graph with variables $R \in \mathcal{G}_{\text{Var}}$, hyperedge replacement system \mathcal{R} and graph $G \in \mathcal{G}$,

$$(1) \quad G \models_{\mathcal{A}} \exists(R - Y_R) \iff G \models \text{SOgra}(R - Y_R, \text{true}) \text{ and}$$

$$(2) \quad G \models_{\mathcal{A}} \langle \exists(S), \mathcal{R} \rangle \iff G \models \text{SOgra}(S, \text{true}) \wedge \text{SOsys}(\mathcal{R}).$$

Proof. We prove both parts of the lemma separately.

Part (1): Let $G_{\overline{H}}$ be the graph G with subgraph H removed.

Assume $G \models_{\mathcal{A}} \exists(\emptyset \xrightarrow{a} R_{\overline{Y_R}})$.

By the semantics of HR* conditions, for $p: \emptyset \rightarrow G$, this is equivalent to

$$\Leftrightarrow p \models_{\mathcal{A}} \exists(\emptyset \xrightarrow{a} R_{\overline{Y_R}})$$

$$\Leftrightarrow \exists q: R_{\overline{Y_R}} \rightarrow G.p = q \circ a \wedge q \models_{\mathcal{A}} \text{true}.$$

By the definition of morphisms, this equals

$$\Leftrightarrow \exists q: R_{\overline{Y_R}} \rightarrow G. \forall o \in D_R. p(o) = q(a(o))$$

$$\Leftrightarrow \exists R' \in \mathcal{G}. \exists q': R_{\overline{Y_R}} \rightarrow R' \wedge R' \subseteq G$$

5.3 Transforming HR* conditions into second-order formulas

which can be expressed as an SO formula

$$\begin{aligned} &\Leftrightarrow \exists R' \in \mathcal{G}. \exists v \in V'_R. (\bigwedge_{v \in V'_R} (\text{lab}_{\text{lv}(v)}(v)) \wedge \exists e \in E'_R. (\bigwedge_{e \in E'_R} (\text{lab}_{\text{le}(e)}(e)) \wedge \text{inc}(e, \text{s}(e), \text{t}(e)))) \\ &\Leftrightarrow \exists R' \in \mathcal{G}. \text{SONod}(R', \text{SOedg}(R') \wedge \text{SOhyp}(R')) \end{aligned}$$

which equals the definition of SOgra:

$$\Leftrightarrow G \models \text{SOgra}(R', \text{true}) \Leftrightarrow G \models \text{SOgra}(R_{\overline{V}_R}, \text{true}).$$

Part (2): From the semantics of HR* conditions, it is clear that for $p: \emptyset \rightarrow G$,

$$\begin{aligned} G \models_{\mathcal{A}} \langle \exists(\emptyset \xrightarrow{a} S), \mathcal{R} \rangle &\iff p \models_{\mathcal{A}} \langle \exists(\emptyset \xrightarrow{a} S), \mathcal{R} \rangle \Leftrightarrow \exists \sigma, q: S^\sigma \hookrightarrow G.p = q \circ a \\ &\Leftrightarrow \exists S^\sigma, q: S^\sigma \hookrightarrow G.S \Rightarrow_{\mathcal{R}}^* S^\sigma. \end{aligned}$$

We continue by induction over the length of derivations.

Basis. By the definition of derivations,

$$\begin{aligned} \exists S^\sigma \in \mathcal{G}, q: S^\sigma \rightarrow G.S \Rightarrow_{\mathcal{R}} S^\sigma &\iff \exists S^\sigma, q: S^\sigma \rightarrow G. \exists x/R \in \mathcal{R}. S \Rightarrow_{x/R} S^\sigma \\ &\Leftrightarrow \exists S^\sigma, q: S^\sigma \rightarrow G. \exists y \in Y_S. \text{ly}(y) = x \wedge S^\sigma \cong S_{\overline{y}} \cup R \wedge \forall i \in [k]. \text{pin}_{R_i} = \text{att}_S(y)_i \end{aligned}$$

By Lemma 5.2, we can reduce this to

$$\Leftrightarrow \exists y \in Y_S. \text{ly}(y) = x \wedge G \models \text{SOgra}(S_{\overline{y}} \cup R_{\overline{\text{Pin}(R)}}, \bigwedge_{i \in [k]} \text{pin}_{R_i} \doteq \text{att}_S(y)_i).$$

Since $k \geq 1$ and $v_i = \text{pin}_{R_i}$ for $i \in [k]$, we include the formula for $\text{SOgra}(S, \text{true})$:

$$\Leftrightarrow G \models \text{SOgra}(S, \text{true}) \wedge \forall_{i \in [k]} v_i. x(v_1, \dots, v_k) \Rightarrow \text{SOgra}(R_{\overline{\text{Pin}(R)}}, \bigwedge_{i \in [k]} v_i \doteq \text{att}_S(y)_i).$$

and by the definition of SOsys, we get

$$\Leftrightarrow G \models \text{SOgra}(S, \text{true}) \wedge \forall_{i \in [k]} v_i. \text{SORule}(x/R).$$

Since S has only a single hyperedge, $x'(v_1, \dots, v_{\text{rank } x'})$ is false for every $x' \neq x$,

$$\Leftrightarrow G \models \text{SOgra}(S, \text{true}) \wedge \text{SOsys}(\mathcal{R}).$$

Hypothesis. For some $S' \in \mathcal{G}_{\text{Var}}$ with $S \Rightarrow_{\mathcal{R}} S'$, assume

$$\exists S', q': S' \rightarrow G.S' \Rightarrow_{\mathcal{R}}^* S^\sigma \iff G \models \text{SOgra}(S', \text{true}) \wedge \text{SOsys}(\mathcal{R}).$$

Step. Then

$$\exists S^\sigma, q: S^\sigma \rightarrow G.S \Rightarrow_{\mathcal{R}}^* S^\sigma \iff \exists S^\sigma, q: S^\sigma \rightarrow G. \exists S'. S \Rightarrow_{\mathcal{R}} S' \Rightarrow_{\mathcal{R}}^* S^\sigma.$$

By Lemma 5.2, we can express S' as an SO formula

$$\Leftrightarrow \exists S'. G \models \text{SOgra}(S', \text{true}) \wedge \bigwedge_{x \in \text{Var}} \bigvee_{x/R \in \mathcal{R}} \forall v_i. \text{SORule}(x/R)$$

$$\Leftrightarrow \exists S'. G \models \text{SOgra}(S', \text{true}) \wedge \text{SOsys}(\mathcal{R}) \wedge S \Rightarrow_{\mathcal{R}} S' \Leftrightarrow G \models \text{SOgra}(S, \text{true}) \wedge \text{SOsys}(\mathcal{R}).$$

This completes the inductive proof. \square

We can now prove Theorem 5.2: For every HR* condition c and for all graphs $G \in \mathcal{G}$,

$$G \models c \iff G \models \text{SO}(c).$$

Proof (of Theorem 5.2). We first show that $G \models_{\mathcal{A}} c \iff G \models \text{SO}(c)$ and conclude by Theorem 4.1 that $G \models c \iff G \models \text{SO}(c)$. The proof proceeds by induction over the structure of HR* conditions. The proofs for conditions true , $\neg c$ and $c \wedge c'$ are straightforward. For conditions $\exists(a, c)$, we use Lemma 5.2 to show that graph morphisms and substitution can be simulated by our construction. For conditions $\exists(P \sqsupseteq C, c)$, Lemma 5.2 is used to show that the inclusion of C^σ in P^σ is simulated by the constructed formula.

Basis. $c = \text{true}$. Then $\text{SO}(c) = \text{true} \Leftrightarrow G \models_{\mathcal{A}} \text{true} \Leftrightarrow \text{true} \Leftrightarrow \text{SO}(c) \models \text{true}$.

Hypothesis. Assume that for HR* conditions $c_i, i \in \mathbb{N}$, the statement holds:

$$G \models_{\mathcal{A}} c_i \Leftrightarrow G \models \text{SO}(c_i).$$

Step.

Case $c = \exists(P \xrightarrow{a} C, c_1)$. By the definition of HR* conditions and the induction hypothesis, we have $G \models_{\mathcal{A}} \exists(a, c_1) \Leftrightarrow \exists \sigma, p: P \xrightarrow{a} G, q: C^\sigma \rightarrow G. q \circ a^\sigma = p \wedge q \models_{\mathcal{A}, \sigma} c_1$.

Using constructions SOgra yields $\Leftrightarrow G \models \exists + . \text{SOgra}(C - P, \text{SO}(c_1)) \wedge \text{SOsys}(\mathcal{R})$. The graph $C - P$ has no dangling edges, since C adds only a single object to P , either a node or a (hyper-)edge connected to nodes in P . By the construction of SO, we have $G \models \text{SO}(\exists(a, c_1))$.

Case $c = \exists(P \supseteq C, c_1)$. By the definition of HR* conditions and the induction hypothesis, we have $G \models_{\mathcal{A}} \exists(P \supseteq C, c_1) \Leftrightarrow \exists p: P \rightarrow G, \sigma, b: C^\sigma \rightarrow P^\sigma, q: C^\sigma \rightarrow G. p \circ b = q \wedge q \models_{\mathcal{A}, \sigma} c_1 \Leftrightarrow \exists \sigma. P^\sigma \supseteq C^\sigma \wedge C \models_{\mathcal{A}, \sigma} c_1 \Leftrightarrow \exists \sigma. P^\sigma \supseteq C^\sigma \wedge \text{SO}(c_1)$. We can now use the constructions SOgra and then SO:

$$\Leftrightarrow G \models \text{SOgra}(C, \exists X_P, X_C. \bigwedge_{x \in D_P} (x \in X_P) \wedge \bigwedge_{y \in D_C} (y \in X_C) \wedge X_C \subseteq X_P \wedge \text{SO}(c_1))$$

$$\Leftrightarrow G \models \text{SOgra}(C, \exists X_P, X_C. \text{SOset}(P, X_P) \wedge \text{SOset}(C, X_C) \wedge X_C \subseteq X_P \wedge \text{SO}(c_1))$$

$$\Leftrightarrow G \models \text{SO}(\exists(P \supseteq C, c_1)).$$

Case $c = \neg c_1$. $\text{SO}(c) = \neg \text{SO}(c_1)$. By the induction hypothesis and Theorem 4.1, we have $G \models_{\mathcal{A}} c \Leftrightarrow G \not\models_{\mathcal{A}} c_1 \Leftrightarrow G \not\models \text{SO}(c_1) \Leftrightarrow G \models \text{SO}(c)$.

Case $c = c_1 \wedge c_2$. $\text{SO}(c) = \text{SO}(c_1) \wedge \text{SO}(c_2)$. Using the induction hypothesis and Theorem 4.1, we get: $G \models_{\mathcal{A}} c_1 \wedge c_2 \Leftrightarrow G \models \text{SO}(c_1) \wedge \text{SO}(c_2) \Leftrightarrow G \models \text{SO}(c_1 \wedge c_2)$.

This concludes the proof of Theorem 5.2. \square

Bibliographic notes

Several formalisms for graph properties have been proposed and examined with regard to their expressive power. (Gaifman, 1982) showed that first-order formulas can express only local properties. (Courcelle, 1997) compares several fragments of MSO formulas and their expressive power. The expressive power of nested conditions is shown to be equivalent to first-order formulas in (Habel and Pennemann, 2009), and several early types of conditions are shown to be equivalent to nested conditions in (H. Ehrig, K. Ehrig, Habel, et al., 2006). In (Habel and Radke, 2010), HR conditions are introduced as an extension to nested conditions, and it is shown that HR⁺ conditions, a subset of HR* conditions, are more expressive than monadic second-order formulas. In (Baldan et al., 2004), a modal logic is presented which uses monadic second-order formulas to describe state properties and temporal modalities to describe behavioral properties. Linear temporal logic is also used in (Rensink, 2003), including the monadic second-order quantification over sets of nodes. (Bruggink and B. König, 2010) present a logic on subobjects, a formalism that is similar to nested conditions, which is equivalent to MSO formulas. Other types of graph conditions going beyond first-order logic are the M-conditions of (Poskitt and Plump, 2014) and the μ -conditions of (Flick, 2016). Both of these, however, cannot express arbitrary counting monadic second-order properties. The language *GPL* of graph properties with paths recently introduced in (Navarro et al., 2016) is an extension of nested conditions that can also express properties over paths (i.e. the transitive hull of the edge relation) and is thus between first-order and monadic second-order logic.

Figure 5.4 shows a diagram comparing the expressive power of graph formulas, nested

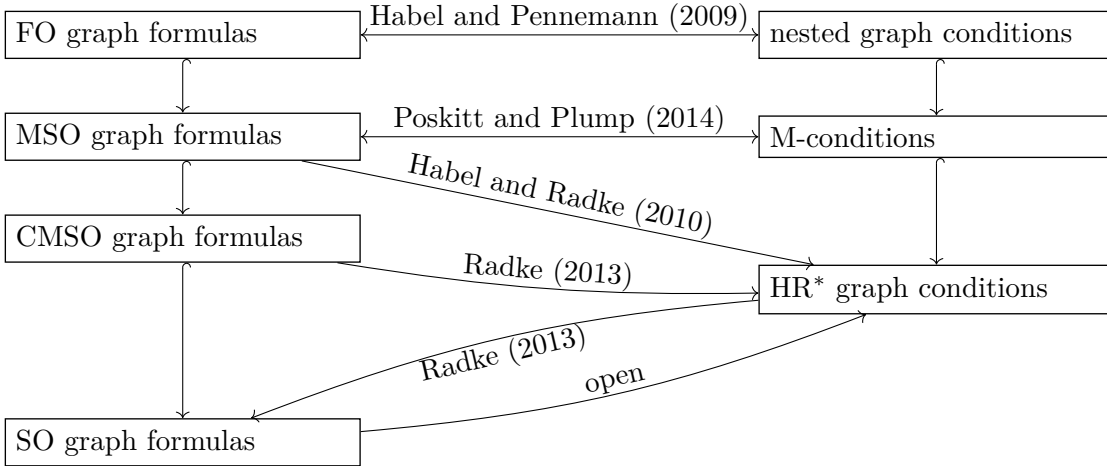


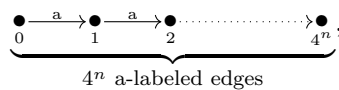
Figure 5.4: Overview of the expressive power of graph formulas and conditions.

and HR* conditions. An arrow from A to B in the diagram signifies that A is less or equally expressive than B .

1. Nested conditions are equivalent to first-order graph formulas (Habel and Pennemann, 2009).
2. HR* conditions are at least as expressive as monadic (Habel and Radke, 2010) and counting monadic second-order logic (Radke, 2013).
3. HR* conditions are at most as expressive as second-order logic (Radke, 2013).
4. M-conditions are equivalent to monadic second-order graph formulas (Poskitt and Plump, 2014).

Open questions.

1. It remains open whether HR* conditions are strictly less expressive than SO formulas. The author suspects this to be the case: SO formulas allow quantification over arbitrary relations, which seems to be more powerful than the replacing of hyperedge variables according to a hyperedge replacement system, as used in HR* conditions.
2. The exact relation between HR* conditions and the μ -conditions from (Flick, 2016) is not yet clear. While μ -conditions can express the language of string graphs (see (Habel, 1992)) of the form $a^{(4^n)}$, i.e. graphs of the form



it is probably impossible to express this language using HR* conditions. On the other hand, as stated above, μ -conditions cannot express every counting monadic second-order property. It is likely, but not yet proven, that the expressiveness of both languages is incomparable.

Chapter 6

Correctness relative to HR^* conditions

Contents

6.1	Shifting for path-like conditions	67
6.2	Shifting for arbitrary HR^* conditions	73
6.3	From right to left application conditions	82
6.4	From left application conditions to preconditions	86
6.5	Expressing the applicability of a rule as a condition	86
6.6	Correctness of graph programs	87

HR^* conditions fulfill a double role as general constraints for graphs and application conditions for graph transformation rules. The difference is easy to spot: application conditions are HR^* conditions over the left- or right-hand side of some rule. Constraints are conditions over the empty graph and are usually not linked to a specific rule.

This chapter introduces basic transformations for HR^* conditions, similar to the basic transformations for nested conditions introduced in (Habel, Pennemann, and Rensink, 2006) and generalized in (H. Ehrig et al., 2012). These transformations are used to transform constraints into application conditions for graph programs and vice versa, and serve as building blocks for the construction of weakest preconditions.

First, two transformations are presented that transform an HR^* constraint (i.e. a condition over the empty graph \emptyset) over a morphism, yielding a (right) application condition for a rule. The transformation in Chapter 6.1 is a simple one for special cases, while the transformation in Chapter 6.2 can transform any HR^* condition, but is more complicated. Both transformations are combined into compound transformation A, using the simple variant whenever possible and reverting to the general variant otherwise. In Chapter 6.3, a transformation L (for “left”) is introduced that converts right into left

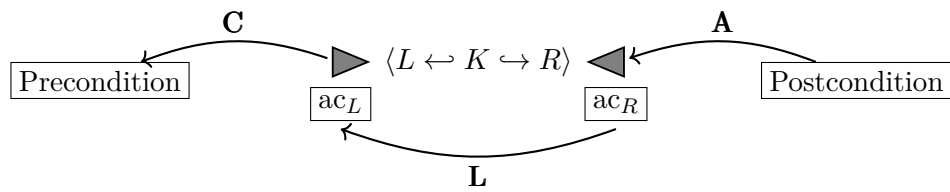


Figure 6.1: Basic transformations for HR* conditions.

application conditions. A third transformation, C, transforms a left application condition into a constraint and is introduced in Chapter 6.4. Transformation Appl in Chapter 6.5 encodes the applicability of a rule into an HR* condition. These transformations are useful on their own to convert between constraints over the empty graph and application conditions, as well as between left and right application conditions. In combination, they provide a conversion from a postcondition of some rule into a weakest precondition, which is explained in Chapter 6.6. Figure 6.2 illustrates the relation of this chapter's theorems.

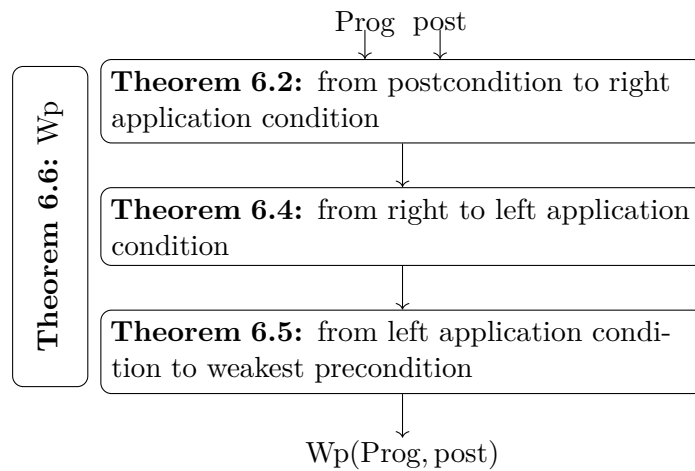


Figure 6.2: Illustration of constructions and theorems

As usual, this chapter uses HR* conditions with \mathcal{M} -satisfaction and substitution semantics.

6.1 Shifting for path-like conditions

One of the most useful constructions for graph conditions is the so-called *shifting* of a condition over an (injective) morphism. In (Pennemann, 2009), it is used on nested conditions to simplify them, is an essential part of the PROCON theorem prover and, most importantly, to translate a postcondition over the empty graph into a right application condition for some rule.

We now establish a construction for HR^* conditions that can serve the same purposes that Shift serves for nested conditions. However, the presence of variables in HR^* conditions makes this task considerably more difficult, and we restrict the kind of morphisms we shift over: *We only shift over morphisms that are injective, and also isomorphic on hyperedges*, i.e. over morphisms $b: P \hookrightarrow P'$ where $Y_P \cong Y_{P'}$.

For a certain subset of HR^* conditions, shifting over morphisms is substantially easier than for the whole class. These HR^* conditions are restricted in the HR systems allowed and are called *path-like conditions*, since they allow the construction of paths and similar structures.

Definition 6.1 (path-like condition). A hyperedge replacement system \mathcal{R} is *path-like* if, for each rule $x/R \in \mathcal{R}$, all of the following conditions hold:

1. The undirected graph R' induced by R^1 does not contain a circle involving a hyperedge (see Figure 6.3).
2. Every pinpoint in $\text{Pin}(R)$ is either directly incident to a hyperedge or has no path to any hyperedge in the undirected induced graph R' .
3. Every hyperedge of rank ≥ 1 in R has at least one tentacle adjacent to a pinpoint in $\text{Pin}(R)$.

An HR^* condition is *path-like* if its replacement system is path-like. △

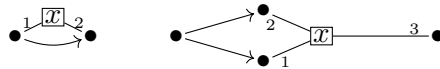


Figure 6.3: Two graphs whose undirected induced graphs both contain a circle over a hyperedge.

Example 6.1 (path-like conditions). In order to illustrate path-like conditions, we look at some examples.

- The set of discrete graphs is a very simple path-like condition:
 $\exists(\mathbb{1})$ with $\mathbb{1} ::= \emptyset \mid \bullet \mathbb{1}$

¹Every directed graph R yields an *undirected induced graph* R' by taking R , adding an edge in the opposing direction for each edge in R , and replacing each hyperedge by a node and an undirected edge for each tentacle.

- As is to be expected, paths are path-like:

$$\exists(\bullet_1^1 \text{---} \square \text{---} \bullet_2^2) \text{ with } \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1^1 \text{---} \square \text{---} \bullet_2^1 \text{---} \square \text{---} \bullet_2^2$$

- Trees can also be expressed with path-like conditions, the replacement system is only a slight extension of the one for paths:

$$\exists(\bullet_1^1 \text{---} \square \text{---} \bullet_2^2) \text{ with } \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1^1 \text{---} \square \text{---} \bullet_2^1 \text{---} \square \text{---} \bullet_2^2 \mid \begin{array}{c} \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 \\ \diagup \quad \diagdown \\ \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 \end{array}$$

- However, a similar extension where the third rule contains a loop leads to series-parallel graphs, which are *not* path-like, since the third rule includes a circle over two hyperedges:

$$\exists(\bullet_1^1 \text{---} \square \text{---} \bullet_2^2) \text{ with } \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1^1 \text{---} \square \text{---} \bullet_2^1 \text{---} \square \text{---} \bullet_2^2 \mid \begin{array}{c} \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 \\ \diagup \quad \diagdown \\ \bullet_1^1 \text{---} \square \text{---} \bullet_2^2 \end{array}$$

- The following alternative replacement system to generate trees is *not* path-like, since the single pinpoint 1 in the second rule has a path to the hyperedge.

$$\exists(\bullet_1^1 \text{---} \square) \text{ with } \bullet_1^1 \text{---} \square ::= \bullet_1 \mid \bullet_1 \text{---} \bullet_1^1 \text{---} \square \mid \begin{array}{c} \bullet_1^1 \text{---} \square \\ \diagup \quad \diagdown \\ \bullet_1^1 \text{---} \square \end{array} \quad \diamond$$

The intuition behind this definition, as can be seen in the examples, is to force the grammar to generate new items “in the middle”, i.e. a replacement step can add to a path in both directions. This makes the replacement system highly ambiguous, which is useful in this case – it provides a way to generate the items of a path, tree or other allowed structures in arbitrary sequence. Furthermore, loops that span over several tentacles of a hyperedge are disallowed, since this would also constrain the possible sequences of derivation steps.

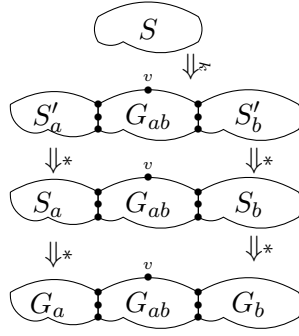
The following lemma on arbitrary separability expresses the fact that a path-like replacement system can generate the items of a graph in arbitrary order.

Lemma 6.1 (separability of path-like grammars).

Let (\mathcal{R}, S) be a hyperedge replacement grammar with a path-like replacement system \mathcal{R} and start graph S . For every graph $G \in \mathcal{L}(\langle \mathcal{R}, S \rangle)$ and every item x in G , there is a graph with variables S' and a number $k \leq |\mathcal{R}|$ limited by the number of rules, such that $x \in S'$ and $S \Rightarrow_{\mathcal{R}}^k S' \Rightarrow_{\mathcal{R}}^* G$.

$$\begin{array}{ccc} S & \xrightarrow{*} & G \ni x \\ & \searrow^k & \nearrow^* \\ & S' \ni x & \end{array}$$

Proof. Without loss of generality, we assume that \mathcal{R} contains no unreachable rules. We decompose G into graphs G_a , G_b and G_{ab} such that G_a and G_b are disjoint up to a small (i.e. no bigger than the right-hand side of a rule in \mathcal{R}) environment G_{ab} around item x . This decomposition is always possible because \mathcal{R} is path-like and there are no loops over hyperedges, thus there can be no edge between nodes in G_a and G_b .



Since x is an item in G , x (together with its environment G_{ab}) is either in S or has to be generated by some rule in \mathcal{R} . If x is in S , it is also in any derived graph S' , since \mathcal{R} is, by assumption, monotone and cannot remove items. Otherwise, let S_x be the graph in which G_{ab} is generated, i.e. the first graph in the derivation $S \Rightarrow^* S_x \Rightarrow^* G$ containing x . Again, we can decompose S_x into two graphs S_a and S_b which are disjoint up to the environment S_{ab} that contains x . This decomposition always exists for the reasons given above.

Since \mathcal{R} is path-like, pinpoints in the rule's right-hand side are either isolated from or directly connected to a hyperedge, so G_{ab} contains neither pinpoints nor hyperedges. This implies that there is a derivation $S_a \Rightarrow_{\mathcal{R}}^* G_a$, and that there is a hyperedge S'_a with $S'_a \Rightarrow_{\mathcal{R}}^* S_a$. The same goes for G_b .

Since \mathcal{R} has no unreachable rules and is (as every HR system) context-free, the rule generating $S'_a + G_{ab} + S'_b$ can be reached within no more than $k = |\mathcal{R}|$ steps². \square

Path-like conditions can be shifted over morphisms into corresponding conditions over the codomain of the morphism.

Theorem 6.1 (Shifting of path-like conditions over morphisms).

There is a transformation Shift^* such that for every path-like condition c over P and every morphism $b: P \rightarrow P'$, there is a path-like condition $\text{Shift}^*(b, c)$ such that, for all morphisms $n: P' \rightarrow G$,

$$n \circ b \models c \iff n \models \text{Shift}^*(b, c).$$

$$\begin{array}{ccc} c \triangleright P & \xrightarrow{b} & P' \triangleleft \text{Shift}^*(b, c) \\ & \searrow n \circ b & \swarrow n \\ & & H \end{array}$$

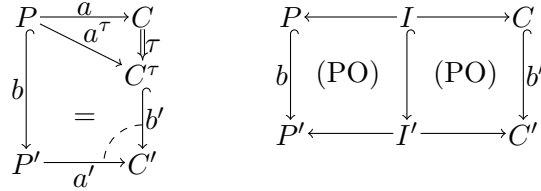
The construction of transformation Shift^* relies on the properties of path-like conditions, in particular Lemma 6.1. A partial expansion of the condition along the replacement system ensures that items in the (expanded) condition can be unified with items in the

²In the worst case, imagine hyperedges labeled A_1, \dots, A_k and rules A_i/A_{i+1} , so that to reach rule A_{n-1}/A_n from A_1 , one has to go through all rules A_1, \dots, A_n to generate A_n .

codomain of the morphism to shift over, while at the same time keeping the condition finite.

Construction. Shift^* is defined inductively on the structure of conditions. For any injective morphism $b: P \hookrightarrow P'$ and replacement system \mathcal{R} , we have:

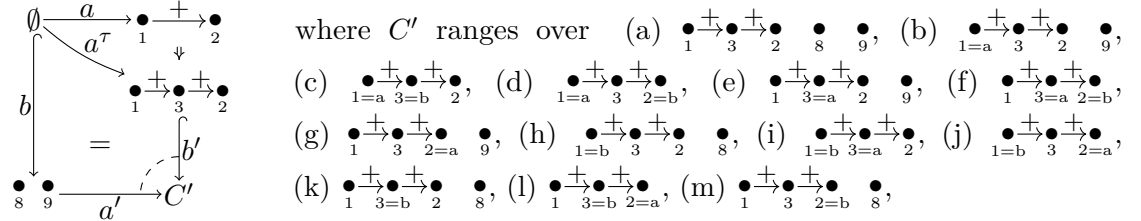
- $\text{Shift}^*(b, \text{true}) = \text{true}$.
- $\text{Shift}^*(b, \exists(P \hookrightarrow C, c)) = \bigvee_{\tau \in \Sigma_b} \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}^*(b', c^\tau))$, where Σ_b is the (finite) set of all derivations along \mathcal{R} with no more than $|\mathcal{R}|$ steps, and \mathcal{F} is the set of all jointly surjective morphism pairs (a', b') with $a' \circ b = b' \circ a^\tau$ (see left diagram below).
- $\text{Shift}^*(b, \exists(P \sqsupseteq C, c)) = \exists(P' \sqsupseteq C', \text{Shift}^*(b', c))$ as per the right diagram below.
- $\text{Shift}^*(b, \neg c) = \neg \text{Shift}^*(b, c)$.
- $\text{Shift}^*(b, c \wedge c') = \text{Shift}^*(b, c) \wedge \text{Shift}^*(b, c')$.



Example 6.2. We shift the path-like condition $\exists(\emptyset \rightarrow \bullet \xrightarrow{+} \bullet)$ with replacement system $\mathcal{R} = \bullet \xrightarrow{+} \bullet ::= \bullet \rightarrow \bullet \mid \bullet \xrightarrow{+} \bullet \xrightarrow{+} \bullet$ over morphism $b: \emptyset \hookrightarrow \bullet \bullet$.

Since $|\text{Ran}(b)| = 2$, $\Sigma_b(\bullet \xrightarrow{+} \bullet) = \{\bullet \rightarrow \bullet, \bullet \xrightarrow{+} \bullet \xrightarrow{+} \bullet, \bullet \rightarrow \bullet \xrightarrow{+} \bullet, \bullet \xrightarrow{+} \bullet \rightarrow \bullet, \bullet \xrightarrow{+} \bullet \xrightarrow{+} \bullet \xrightarrow{+} \bullet\}$.

For the second element of Σ_b , we show the set \mathcal{F} exemplary:



The condition is expanded considerably by the construction. However, many cases are equivalent to or implied by other cases, so a simplification of the condition by eliminating cases which are implied by others is advisable after using Shift^* . \diamond

Proof (of Theorem 6.1). By induction over the structure of conditions.

Basis. For the condition true , the equivalence holds trivially.

Hypothesis. Let $n \circ b \models c' \iff n \models \text{Shift}^*(b, c')$.

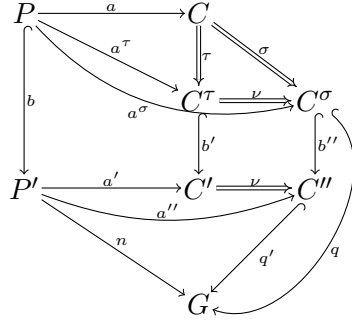
Step.

Case true. See induction basis.

Case $\exists(P \xrightarrow{a} C, c)$. We do both directions of the proof separately.

“ \Leftarrow ”. Assume that $n \models \text{Shift}^*(b, c)$. By construction, this equals

$n \models \bigvee_{\tau \in \Sigma_b} \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}^*(b' : C^\tau \hookrightarrow C', c^\tau))$. By the definition of satisfaction, this implies there is a substitution ν and an injective morphism $q' : C'^\nu \hookrightarrow G$ such that $q' \circ a'^\nu = n$ and $q' \models \text{Shift}^*(b', c^\tau)^\nu$. Let $C'' = C'^\nu$ and $\sigma = \nu \circ \tau$ and $C^\sigma = C'^{\sigma \circ \nu}$. We construct $b'' : C^\sigma \hookrightarrow C''$, with $b''(x) = b'(x)$ if $x \in C^\tau$ and $b''(x) = x$ else. Now $q' \models \text{Shift}^*(b', c^\tau)^\nu$ is equivalent to $q' \circ b'' \models \text{Shift}^*(b', c^{\nu \circ \tau})$. Let $q = q' \circ b''$. By the hypothesis, $q' \models \text{Shift}^*(b'', c^\sigma) \Leftrightarrow q' \circ b'' = q \models c^\sigma$. We now have a substitution σ such that $n \circ b \circ a^\sigma = q$ and $q \models c^\sigma$. By definition of satisfaction, this is equivalent to $n \circ b \models \exists(P \xrightarrow{a} C, c)$.



“ \Rightarrow ”. Assume $n \circ b \models \exists(P \xrightarrow{a} C, c)$.

By the semantics of HR^* conditions, there are substitution σ and an injective morphism $q : C^\sigma \hookrightarrow G$ such that $n \circ b \circ a^\sigma = q$ and $q \models c^\sigma$.

By \mathcal{E}' - \mathcal{M} -pair factorization of the cospan $P' \xrightarrow{n} G \xleftarrow{q} C^\sigma$, we have unique morphisms $a'' : P' \rightarrow C''$, $b'' : C^\sigma \rightarrow C''$ and $q' : C'' \hookrightarrow G$ such that a'' and b'' are jointly surjective, $n = q' \circ a''$ and $q = q' \circ b''$ (see diagram). Note that b'' is injective, since q and q' are injective.

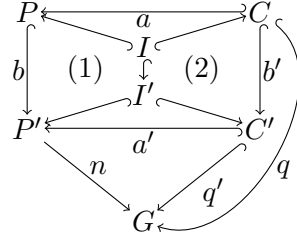
By Lemma 6.1, we can split $\sigma = \tau \circ \nu$ with τ consisting of $k \leq |\mathcal{R}|$ derivation steps of \mathcal{R} . Since b'' is injective, we can construct morphism $b' : C^\tau \hookrightarrow C'$ by restricting b'' to C^τ . This construction also implies $C'^\nu = C''$.

For a morphism $m : G \rightarrow H$, let $m_{-Y} : G - Y_G \rightarrow H - Y_H$ be the morphism restricted to nodes and (non-hyper-)edges. Note that for $C' \Rightarrow_\nu C''$, there is an underlying injective morphism $m_\nu : C' - Y_{C'} \hookrightarrow C''$ (C'' contains no hyperedges, so $C'' = C'' - Y_{C''}$).

By \mathcal{E}' - \mathcal{M} pair factorization of a''_{-Y} and b''_{-Y} , we can construct morphisms a'_{-Y} and b'_{-Y} such that $a''_{-Y} = m_\nu \circ a'_{-Y}$, $b''_{-Y} = m_\nu \circ b'_{-Y}$ and (a'_{-Y}, b'_{-Y}) are jointly surjective.

By definition, b is isomorphic on hyperedges (i.e. b restricted to hyperedges is an isomorphism); b'' does not contain any hyperedges since ν replaced all hyperedges, and by its construction, b' is also isomorphic on hyperedges. This implies that there is a morphism $a' : P' \rightarrow C'$, that $b' \circ a^\tau = a' \circ b$, that b' is also surjective on hyperedges, and that (a', b') are jointly surjective. Since this matches the construction of Shift^* , $n \models \text{Shift}^*(b, \exists(P \hookrightarrow C, c))$.

Case $\exists(P \sqsupseteq C, c)$. By the satisfaction of HR^* conditions, $n \circ b \models \exists(P \sqsupseteq C, c) \Leftrightarrow \exists a: C \hookrightarrow P, q: C \hookrightarrow G. q = n \circ b \circ a \wedge q \models c$. Since $q = n \circ b \circ a$, we also have $n \circ b \circ l = q \circ r$ and thus, there is an injective $a': C \hookrightarrow P'$ such that $q' = n \circ a'$. By the hypothesis, $q' \models \text{Shift}^*(b', c)$ and thus we gain $n \models \text{Shift}^*(b, \exists(P \sqsupseteq C, c))$.



Case $\neg c$. trivial.

Case $c_1 \wedge c_2$. trivial.

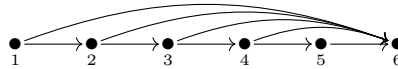
This concludes the proof. □

In order to motivate the next part, we illustrate why the above construction does not work for non-pathlike HR^* conditions, using the grammar for series-parallel graphs from Example 6.1.

Example 6.3. We first recall the replacement system for series-parallel graphs from Example 6.1.

$$\mathcal{R} = \{ \bullet_1 \xrightarrow{1} \boxed{\text{SP}} \xrightarrow{2} \bullet_2 ::= \bullet_1 \longrightarrow \bullet_2 \mid \bullet_1 \xrightarrow{1} \boxed{\text{SP}} \xrightarrow{2} \bullet_1 \xrightarrow{1} \boxed{\text{SP}} \xrightarrow{2} \bullet_2 \mid \bullet_1 \xrightarrow{1} \boxed{\text{SP}} \xrightarrow{2} \bullet_2 \xrightarrow{1} \boxed{\text{SP}} \xrightarrow{2} \bullet_2 \}$$

The graph below can be generated by alternatively using the parallel and the serial rule on the rightmost hyperedge. This construction principle leads to a subclass of series-parallel graph which all consist of a path with parallel edges to the last node from every other node (except the last-but-one).



Suppose that during Shift^* , we want to identify a node v with node 4 from the above graph. It turns out that in order to generate node 4, we first have to generate all the nodes left of it. Otherwise, it is impossible to generate the bent parallel edges. More generally, in order to generate node n of a graph with $n + 1$ nodes, it is necessary to generate n nodes before. Since the Shift^* construction only allows a limited number of derivation steps, it cannot work for series-parallel graphs. ◇

6.2 Shifting for arbitrary HR* conditions

The above construction Shift^* works well in the case of path-like replacement systems. However, HR* conditions are not path-like in general, and shifting over morphisms should be possible for arbitrary conditions. We now show a transformation that shifts arbitrary HR* conditions over morphisms. This transformation needs considerably more effort than the version for path-like conditions. Whenever possible, that construction should be used to shift conditions, and the below construction reserved for HR* conditions which are not path-like.

The final goal of this subchapter is the following theorem, stating that any HR* constraint can be transformed into a right application condition for a rule.

Theorem 6.2 (from postcondition to right application condition).

There is a transformation A such that for any rule $\rho = \langle L \leftrightarrow K \hookrightarrow R \rangle$, any HR* condition c over \emptyset , there is an HR* condition $A(\rho, c)$ such that for any injective morphism $m': R \hookrightarrow H$,

$$m' \models A(\rho, c) \iff H \models c.$$

Our goal is to identify “external” nodes with nodes generated by a substitution. We do this by “integrating” the external nodes into the hyperedge with additional tentacles. During the replacement process of the hyperedges, the additional nodes connected to these tentacles can be identified with a generated node. The following construction $\text{Shiva}(\mathcal{R}, k)$ ³ is given a natural number k and an HR system \mathcal{R} and equips the rules of \mathcal{R} with up to k additional tentacles. For hyperedge label x , we use the label (x, k) to denote a “similar” hyperedge label with k more tentacles. This “similarity” means that rules x/R and $(x, k)/R'$ should generate isomorphic graphs, minus some added tentacles and their pinpoints in R' . The purpose of this construction is to generate an HR system \mathcal{R}' such that each graph G of sufficient size (i.e. of at least k nodes) generated by an HR grammar (\mathcal{R}, x^\bullet) can also be generated by $(\text{Shiva}(\mathcal{R}, k), (x, k)^\bullet)$ and vice versa.

Let $\langle R \rangle$ be the graph R with all its hyperedges removed. The construction adds, for all rules $x/R \in \mathcal{R}$, rules x^ν/R^τ , where ν and τ are substitutions from a replacement system \mathcal{T}_k that adds from 0 up to k tentacles to hyperedges. So x^ν/R^τ is a rule x/R with some tentacles (and their pinpoints) added. The set of these rules is filtered such that each new pinpoint in x^ν should have an image in R^τ and vice versa. This means there is an injective morphism α' from $\langle x^\nu \rangle$ to R^τ and each node in R^τ has a preimage in R or $x^\nu - x$, as expressed by the diagram in the construction below.

Construction. Let the HR system \mathcal{T}_k consist of rules that add up to k tentacles to each hyperedge x occurring as a left hand side in \mathcal{R} , i.e.

$$\mathcal{T}_k = \{x/(x, i)^\bullet \mid x/R \in \mathcal{R}, i \in [0, k]\}$$

³The construction’s main characteristic is the addition of tentacles (arms) to the hyperedges, so it is named after the many-armed Hindu god Shiva.

Given an HR system \mathcal{R} , let $\text{Shiva}(\mathcal{R}, k)$ be the set of all rules x^ν/R^τ such that x/R is a rule in \mathcal{R} and $\nu, \tau \in \mathcal{T}_k$ are substitutions adding tentacles such that every node that ν adds to x , τ adds to R , i.e.

$$\text{Shiva}(\mathcal{R}, k) := \{x^\nu/R^\tau \mid x/R \in \mathcal{R}, \nu, \tau \in \mathcal{T}_k\}$$

such that the diagram below commutes (i.e. $\alpha' \circ \beta = \beta' \circ \alpha$) and α' and β' are jointly surjective.

$$\begin{array}{ccc} \langle x \rangle & \xleftarrow{\alpha} & \langle R \rangle \\ \downarrow \beta & = & \downarrow \beta' \\ \langle x^\nu \rangle & \xleftarrow{\alpha'} & \langle R^\tau \rangle \end{array}$$



An example illustrates the way the construction works.

Example 6.4. Let $k = 2$ and $\mathcal{R} = \bullet_1 \xrightarrow{+} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_2$. Then $\text{Shiva}(\mathcal{R}, k)$ yields the new set of rules

$$\mathcal{R}' = \left\{ \begin{array}{l} \bullet_1 \xrightarrow{+} \boxed{(+, 0)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 0)} \bullet_2, \\ \bullet_1 \xrightarrow{+} \boxed{(+, 1)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 1)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 0)} \bullet_2, \\ \bullet_1 \xrightarrow{+} \boxed{(+, 2)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 2)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 1)} \bullet_2 \mid \bullet_1 \xrightarrow{+} \bullet_1 \xrightarrow{+} \boxed{(+, 1)} \bullet_2 \end{array} \right\}$$



The construction essentially does not change the languages that can be generated with the help of the HR systems. The following lemma captures this property.

Lemma 6.2 (Shiva is language-invariant).

For any HR grammar (\mathcal{R}, x^\bullet) with language $L = \mathcal{L}(\mathcal{R}, x^\bullet)$ and any natural number k , the HR grammar $(\text{Shiva}(\mathcal{R}, k), (x, k)^\bullet)$ generates the language $\{G \in L \mid |G| \geq \text{rank}(x) + k\}$, i.e. the language of all graphs in L except those with less than $\text{rank}(x) + k$ nodes.

Proof (of Lemma 6.2). Both directions of the proof are done separately. Let $\mathcal{R}' = \text{Shiva}(\mathcal{R}, k)$ and $L' = \mathcal{L}((\text{Shiva}(\mathcal{R}, k), (x, k)^\bullet))$.

Case $G' \in L' \implies G' \in L$. Every graph $G' \in L'$ is the result of a derivation $(x, k)^\bullet \Rightarrow_{\mathcal{R}'}^* G'$. By the construction, for every derivation step $G'_i \Rightarrow_{x'_i/R'_i} G'_{i+1}$, the rule $x'_i/R'_i \in \mathcal{R}'$ is constructed from a rule $x_i/R_i \in \mathcal{R}$, and there is a derivation step $G_i \Rightarrow_{x_i/R_i} G_{i+1}$. Since the only difference between x'_i/R'_i and x_i/R_i is the adding of tentacles or unifying of nodes, $G'_i \cong G_i$ implies $G'_{i+1} \cong G_{i+1}$. By induction, it follows that $(x, k)^\bullet \Rightarrow_{\mathcal{R}'}^* G'$ implies $x^\bullet \Rightarrow_{\mathcal{R}}^* G$ and $G' \cong G$.

Case $G \in L \implies G \in L'$. Every graph $G \in L$ with $|G| \geq \text{rank}(x) + k$ is the result of a derivation $x^\bullet \Rightarrow_{\mathcal{R}}^* G$. By the construction, for every derivation step $G_i \Rightarrow_{x_i/R_i} G_{i+1}$, there are rules $\{(x_i, l)/R'_i \in \mathcal{R}' \mid 0 \leq l \leq k\}$, and a derivation $G'_i \Rightarrow_{(x_i, l)/R'_i}^* G'_{i+1}$. Since $|G| \geq \text{rank}(x) + k$, the derivation $x^\bullet \Rightarrow_{\mathcal{R}}^* G$ consists of enough steps such that the additional nodes in $(x, k)^\bullet$ can be unified with nodes on the RHS of the rules in \mathcal{R}' , so for $x^\bullet \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \cdots \Rightarrow_{\mathcal{R}} G_n = G$, there is a sequence $(x, k)^\bullet \Rightarrow_{\mathcal{R}'} G'_1 \Rightarrow_{\mathcal{R}'} \cdots \Rightarrow_{\mathcal{R}'} G'_n = G'$ such that $G \cong G'$. \square

We proceed with the shifting of HR* conditions over morphisms. For readability reasons, this is split into two parts. The first construction, called Integrate_n , deals only with nodes. The morphism $b_n: P \hookrightarrow P_n$ over which to shift may only add nodes, i.e. $P_n - P$ has no edges. The second construction, called Integrate_e , deals with edges, i.e. shifts over morphisms $b: P \hookrightarrow P'$ which add only edges.

The following Lemma states that HR* conditions can be shifted over injective morphisms into corresponding conditions over the codomain of the morphism, provided the codomain, compared to the domain, only contains additional nodes.

Lemma 6.3 (Integrating nodes into HR* conditions).

There is a transformation Integrate_n such that for all HR* conditions $\langle c, \mathcal{R} \rangle$ over P and morphisms $b: P \rightarrow P'$, $n: P' \rightarrow G$, where $P' - P$ is a discrete graph, there is a condition $\text{Integrate}_n(b, \langle c, \mathcal{R} \rangle)$ such that

$$n \circ b \models \langle c, \mathcal{R} \rangle \iff n \models \text{Integrate}_n(b, \langle c, \mathcal{R} \rangle).$$

Construction. Let $k = |\text{Ran}(b) - \text{Dom}(b)|$ be the number of nodes added in the codomain of b . Let \mathcal{T}_k be defined as in Construction 6.2: $\mathcal{T}_k = \{x/(x, i)^\bullet \mid x/R \in \mathcal{R}, i \in [0, k]\}$, i.e. \mathcal{T}_k consists of rules that add up to k tentacles to each hyperedge x occurring as a left hand side in \mathcal{R} .

Then $\text{Integrate}_n(b, \langle c, \mathcal{R} \rangle) = \langle \bigvee_{\tau \in \mathcal{T}_k} \text{Shift}_n(b, c^\tau), \text{Shiva}(\mathcal{R}, k) \rangle$, where Shift_n is defined like Shift in Chapter 2.4 with the addition of $\text{Shift}_n(b, \exists(P \sqsupseteq C, c)) = \exists(P' \sqsupseteq C', \text{Shift}_n(b', c))$, where $P \leftarrow I \rightarrow C$ is the partial morphism from C to P , I' is the pushout complement of $I \rightarrow P \hookrightarrow P'$ and C' the pushout of $I' \leftarrow I \rightarrow C$ (see diagram below).

$$\begin{array}{ccccc} P & \longleftarrow & I & \longrightarrow & C \\ \downarrow b & & \downarrow & & \downarrow b' \\ P' & \longleftarrow & I' & \longrightarrow & C' \end{array} \quad \begin{array}{c} \text{(PO)} \\ \text{(PO)} \end{array}$$



Some explanation of the above construction is in order. Note that each node v in $P' - P$ may be identified with a node in $C - P$, but also with a “hidden” node generated from a hyperedge y in C . For the latter case, we add another tentacle to y with a new node v' , so that v can be identified with v' . During derivation of $\sigma(C)$, v' has to be

identified with one of the generated nodes. This means that (a) this node is part of the right-hand side R of some rule x/R and (b) is not a pinpoint of R .

Assume that hyperedge y has label x . Since we added a new tentacle to y and the rank of each label is fixed, we need to use another label $(x, 1)$ (the 1 signifies we added 1 additional tentacle) and according rules $(x, 1)/R'$, which generate the same graphs (almost, except for the added node) as the rules for x , and also deal with the additional tentacle. Two things can happen to the node v' attached to that tentacle: (a) node v' is identified with a non-pinpoint node in R' , or (b) v' is “passed on” to a later derivation step, by adding a tentacle to a hyperedge in R' and connecting it to v' . The left square of the construction diagram ensures that every node in $\langle(x, k)\rangle - \langle x\rangle$ can only be identified with a non-pinpoint in R' , while the right square enumerates every combination of up to k tentacles added to the hyperedges of R .

Example 6.5. As an example, we integrate the HR* condition

$$\exists(\bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2) \text{ with } \mathcal{R} = \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2 ::= \bullet_1 \text{---} \bullet_2 \mid \bullet_1 \text{---} \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2$$

into the rule $\langle L \leftrightarrow K \leftrightarrow R \rangle = \langle \bullet_a \text{---} \bullet_c \leftrightarrow \bullet_a \quad \bullet_c \leftrightarrow \bullet_a \text{---} \bullet_c \rangle$.

Since the rule's RHS $R = \bullet_a \text{---} \bullet_c$ contains one edge, we expand \mathcal{R} to

$$\mathcal{R}'' = \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2 ::= \bullet_1 \text{---} \bullet_2 \mid \bullet_1 \text{---} \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2 \mid \bullet_1 \text{---} \bullet_1 \text{---} \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2$$

which is equivalent to \mathcal{R} .

Since R contains two nodes and \mathcal{R} has one hyperedge label, we let

$$\mathcal{T}_k = \{ \bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2 ::= \bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2 \mid \bullet_1^1 \text{---} \boxed{+,1} \text{---} \bullet_2^2 \mid \bullet_1^1 \text{---} \boxed{+,2} \text{---} \bullet_2^2 \}$$

\mathcal{R}' is defined as in Example 6.4.

The condition $\exists(\bullet_1^1 \text{---} \boxed{+} \text{---} \bullet_2^2)$ is transformed to

$$\begin{aligned} & \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \\ & \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,0} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,1} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,1} \text{---} \bullet_2^2) \vee \exists(\bullet_1^1 \text{---} \boxed{+,2} \text{---} \bullet_2^2). \end{aligned}$$

◇

Proof (of Lemma 6.3). The proof proceeds by induction over the structure of HR* conditions. Let $\langle c', \mathcal{R}' \rangle = \text{Integrate}_n(b, \langle c, \mathcal{R} \rangle)$. Colors are used in the diagram for an easier overview of the different stages of the proof.

Basis. $n \circ b \models \text{true} \Leftrightarrow \text{true} \Leftrightarrow n \models \text{true} \Leftrightarrow n \models \text{Integrate}_n(b, \langle \text{true}, \mathcal{R} \rangle)$.

Hypothesis. For any subcondition c , $n \circ b \models \langle c, \mathcal{R} \rangle \Leftrightarrow n \models \text{Integrate}_n(b, \langle c, \mathcal{R} \rangle)$.

Step.

Case true. See induction basis.

Case $\neg c$. By induction, $n \circ b \not\models c \Leftrightarrow n \not\models \text{Integrate}_n(b, \langle c, \mathcal{R} \rangle)$, which is equivalent to

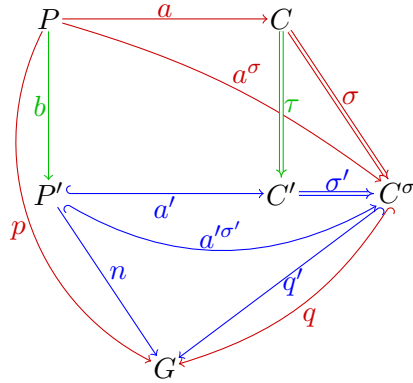
$n \models \text{Integrate}_n(b, \langle \neg c, \mathcal{R} \rangle)$.

Case $c \wedge c'$. $n \circ b \models c \wedge c' \Leftrightarrow n \circ b \models c \wedge n \circ b \models c'$. By the induction hypothesis, this is equivalent to $n \models \text{Integrate}_n(b, \langle c, \mathcal{R} \rangle) \wedge n \models \text{Integrate}_n(b, \langle c', \mathcal{R} \rangle)$, which is equivalent to $n \models \text{Integrate}_n(b, \langle c \wedge c', \mathcal{R} \rangle)$ by construction.

Case $\exists(P \xrightarrow{a} C, c)$. We prove both directions separately.

“ \Leftarrow ”. Assume that $n \models \text{Integrate}_n(b, \langle \exists(a, c), \mathcal{R} \rangle)$. By construction, this is equivalent to $n \models \langle \bigvee_{\tau, a', b'} \exists(a', \text{Integrate}_n(b', \langle c^\tau, \mathcal{R} \rangle)), \mathcal{R}' \rangle$. By the definition of \models (Def. 3.9), this implies $\exists \sigma' \in \mathcal{R}'^*$, $q': C'^{\sigma'} \hookrightarrow G. n = q \circ a'^{\sigma'} \wedge q' \models \text{Integrate}_n(b', \langle c^{\sigma'}, \mathcal{R} \rangle)$. By Lemma 6.2, there is a $\sigma \in \mathcal{R}^*$ such that for every hyperedge (x, k) in C' , there is a hyperedge x in C such that $\sigma(x) = \sigma'((x, k)^\bullet)$. By extension, we have $\sigma(C) = \sigma'(C') = \sigma'(\tau(C))$. Using σ , we construct morphism $a^\sigma: P \hookrightarrow C^\sigma$. Since $n = q' \circ a'^{\sigma'}$, $n \circ b = q' \circ a'^{\sigma'} \circ b$. Let $q = q'$, then $q' \circ a'^{\sigma'} \circ b = q \circ a^\sigma$ and, by the hypothesis, $q \models c^\sigma$. By the definition of HR* satisfaction (Def. 3.9), this implies $n \circ b \models \exists(a, c)$.

“ \Rightarrow ”. Assume that $n \circ b \models \langle \exists(P \xrightarrow{a} C, c), \mathcal{R} \rangle$. By the definition of \models (Def. 3.9), this implies $\exists \sigma \in \mathcal{R}^*$, $q: C^\sigma \hookrightarrow G. n \circ b = q \circ a^\sigma \wedge q \models c^\sigma$. By Lemma 6.2, for some $\tau \in \mathcal{T}_k$, there is a derivation $\sigma' \in \mathcal{R}'^*$ such that $\sigma'(\tau(C)) = \sigma(C)$. We construct $a'^{\sigma'}: P' \hookrightarrow C'^{\sigma'}$ such that $a^\sigma = a'^{\sigma'} \circ b$ and, since $n \circ b = q \circ a^\sigma$ and $q = q'$, get $n = a'^{\sigma'} \circ q'$ by \mathcal{E}' - \mathcal{M} pair factorization. By the hypothesis, $q' \circ b' \models \langle c^\sigma, \mathcal{R} \rangle \Leftrightarrow q \models \text{Integrate}_n(b, \langle c^\sigma, \mathcal{R} \rangle)$. Then we have $\exists \tau \in \mathcal{T}_k, \sigma' \in \mathcal{R}'^*$, $q': C'^{\sigma'} \hookrightarrow G. n = q' \circ a'^{\sigma'} \wedge q' \models \text{Shift}_n(b', c^\tau)$. By the semantics of HR* conditions, this equals $n \models \langle \bigvee_{\tau \in \mathcal{T}_k, (a', b') \in \mathcal{F}} \exists(a', \text{Shift}_n(b', c^\tau)), \text{Shiva}(\mathcal{R}, k) \rangle$. By the construction of Shift_n and Integrate_n , this equals $n \models \langle \bigvee_{\tau \in \mathcal{T}_k} \text{Shift}_n(b, \exists(a, c)^\tau), \text{Shiva}(\mathcal{R}, k) \rangle \Leftrightarrow n \models \text{Integrate}_n(b, \langle \exists(P \xrightarrow{a} C, c), \mathcal{R} \rangle)$.



Case $\exists(P \sqsupseteq C, c)$. We have $n \circ b \models \langle \exists(P \sqsupseteq C, c), \mathcal{R} \rangle \Leftrightarrow \exists \sigma \in \mathcal{R}^*$, $q: C^\sigma \hookrightarrow G$, $a: C^\sigma \hookrightarrow P. q = n \circ b \circ a \wedge q \models c^\sigma$. By Lemma 6.2, this equals $\exists \tau \in \mathcal{T}_k, \sigma' \in \mathcal{R}'^*$, $q: C'^{\sigma' \circ \tau}$, $a: C'^{\sigma' \circ \tau} \hookrightarrow P. q = n \circ b \circ a \wedge q \models c^{\sigma' \circ \tau}$. Using the induction hypothesis and the construction, this equals $\exists \tau \in \mathcal{T}_k, \sigma' \in \mathcal{R}'^*$, $q': C'^{\sigma'}$, $a': C'^{\sigma'} \hookrightarrow P. q' = n \circ a \wedge q' \models \text{Shift}_n(b', c^{\sigma'})$. By the semantics of HR* conditions, this equals $n \models \text{Integrate}_n(b, \langle \exists(P \sqsupseteq C, c), \mathcal{R} \rangle)$.

This concludes the proof. \square

Handling edges. The above construction can only handle nodes. To handle edges, the HR system \mathcal{R} has to be in a form such that all edges in $P' - P$ can be identified with edges generated by a rule in $x/R \in \mathcal{R}$. The basic idea is that whenever there is an edge in $P' - P$ between two nodes, an edge between corresponding pinpoints in a rule can be deleted. This is done by remembering all such edges during the hyperedge replacement process, as part of the hyperedge label.

Construction. Given a morphism $b: P \hookrightarrow P'$, let $k = |V_{P'-P}|$ and a HR system \mathcal{R} , expand \mathcal{R} $k * |LHS(\mathcal{R})|$ times, yielding $\mathcal{R}_{ex} = \{x/R' \mid \exists i. 0 \leq i \leq k * |LHS(\mathcal{R})| \wedge R \Rightarrow_i R'\}$. The expansion ensures any subset of edges in $P' - P$ can be integrated into a hyperedge in one step.

Then perform $\text{Integrate}_n(b, \langle c, \mathcal{R}_{ex} \rangle) = \langle c_n, \mathcal{R}_n \rangle$.

For any hyperedge y with label (x, k) in condition c , let $P_y = \langle (x, k) \rangle - \langle x \rangle$ be the discrete graph of all attachment points of y with index greater than $\text{rank}(x)$. Let G_y be the graph consisting of P_y plus all edges in c with both source and target in P_y (see the example below).

$$c = \forall(1 \bullet, \exists(\overset{1 \bullet 1}{\underset{4 \bullet}{\curvearrowright}} \overset{y}{\text{X}} \overset{2 \bullet 2}{\underset{3 \bullet}{\curvearrowright}})) \quad P_y = \overset{2 \bullet}{\bullet} \quad \overset{2 \bullet}{\bullet} \quad G_y = \overset{2 \bullet}{\bullet} \leftarrow \overset{4 \bullet}{\bullet} \overset{3 \bullet}{\bullet}$$

For a cospan $P' \xrightarrow{a'} C' \xleftarrow{b'} C$, let $\tau(a', b')$ replace every hyperedge y with label (x, k) in C' by a hyperedge (x, k, G_y) with the same rank and $G_y = P_y + \{e \mid e \in E_{C'} \wedge s(e), t(e) \in P_y \wedge \exists e' \in E_{P'} . e = a'(e')\}$ being the graph consisting of all of y 's attachment points with index $\geq \text{rank}(x)$, plus all edges between them which have a preimage in P' . We define Shift_e similar to Shift_n :

$\text{Shift}_e(b, \text{true}) = \text{true}$, $\text{Shift}_e(b, \neg c) = \neg \text{Shift}_e(b, c)$, $\text{Shift}_e(b, c \wedge c') = \text{Shift}_e(b, c) \wedge \text{Shift}_e(b, c')$. $\text{Shift}_e(b, \exists(P \hookrightarrow C, c)) = \bigvee_{(a', b') \in \mathcal{F}} \exists(a', \text{Shift}_e(b', c))^{\tau(a', b')}$, where \mathcal{F} is the set of all jointly surjective morphism pairs (a', b') with $a' \circ b = b' \circ a$ (see left diagram below) and $\tau(a', b')$ is defined as above. $\text{Shift}_e(b, \exists(P \sqsupseteq C, c)) = \exists(P' \sqsupseteq C', \text{Shift}_e(b', c))$, where $P \leftarrow I \rightarrow C$ is the partial morphism from C to P , I' is the pushout complement of $I \rightarrow P \hookrightarrow P'$ and C' the pushout of $I' \leftarrow I \rightarrow C$ as per the right diagram below.

$$\begin{array}{ccc} P & \xleftarrow{a} & C \\ \downarrow b & & \downarrow b' \\ P' & \xleftarrow{a'} & C' \end{array} \xrightarrow{\tau(a', b')} C'_n$$

$$\begin{array}{ccccc} P & \longleftarrow & I & \longrightarrow & C \\ \downarrow b & \text{(PO)} & \downarrow & \text{(PO)} & \downarrow b' \\ P' & \longleftarrow & I' & \longrightarrow & C' \end{array}$$

Let maxEdges be the maximum number of parallel edges between two nodes in c . For each hyperedge label (x, k) , let $\mathcal{G}_{(x, k)}$ be the set of all graphs with k nodes and no more than maxEdges parallel edges.

Then $\text{Integrate}_e(b, \langle c_n, \mathcal{R}_n \rangle) = \langle \text{Shift}_e(c_n), \mathcal{R}' \rangle$, where $\mathcal{R}' = \{(x, k, G_y)/R' \mid (x, k)/R \in \mathcal{R}_n \wedge G_y \in \mathcal{G}_{(x, k)}\}$ and R' being constructed from R as described below.

Given G_y and R , construct a partial morphism (symbolized by the span $G_y \leftarrow I \rightarrow R$) from G_y to R , mapping the nodes of G_y to the corresponding nodes (pinpoints) in R and the edges between them (since not all edges in G_y may have corresponding edges in R ,

the morphism is partial). Construct P_y by removing all edges from G_y and for every I' with $G_y \hookrightarrow I' \hookrightarrow I$, construct the pushout complement R' .

$$\begin{array}{ccccc} G_y & \hookrightarrow & I & \hookrightarrow & R \\ \uparrow & & \uparrow & & \uparrow \\ & = & & \text{(PO)} & \\ P_y & \hookrightarrow & I' & \hookrightarrow & R' \end{array}$$



Example 6.6. We continue Example 6.5: We integrate condition $\exists(\bullet_1^1 \boxplus \bullet_2^2)$ with HR system $\mathcal{R} = \bullet_1^1 \boxplus \bullet_2^2 ::= \bullet_1 \rightarrow \bullet_2 \mid \bullet_1 \rightarrow \bullet_1^1 \boxplus \bullet_2^2$ into rule $\langle \bullet_3 \bullet_4 \leftarrow \bullet_3 \bullet_4 \hookrightarrow \bullet_3 \bullet_4 \rangle$.

We split morphism $b: \emptyset \hookrightarrow \bullet_3 \bullet_4$ into $b = b_e \circ b_n$ with morphisms $b_n: \emptyset \hookrightarrow \bullet_3 \bullet_4$ and $b_e: \bullet_3 \bullet_4 \hookrightarrow \bullet_3 \bullet_4$.

From Example 6.5, we get $\text{Integrate}_n(b_n, \langle c, R \rangle) = \langle c_n, R_n \rangle$.

The HR system \mathcal{R}' , without superfluous rules, looks as follows:

$$\begin{aligned} \mathcal{R}' = & \bullet_1^1 \xrightarrow{3 \bullet_3 \bullet_4} \boxed{+2, \bullet_3 \bullet_4} \xrightarrow{4 \bullet_4} \bullet_2^2 ::= \bullet_1 \rightarrow \bullet_1 \xrightarrow{3 \bullet_3 \bullet_4} \boxed{+2, \bullet_3 \bullet_4} \xrightarrow{4 \bullet_4} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_3^1 \xrightarrow{4 \bullet_4} \boxed{+1, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_4^1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 \\ & \bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+0, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2^2 \mid \\ & \bullet_1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 ::= \bullet_1 \rightarrow \bullet_1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+0, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2^2 \mid \\ & \bullet_1 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 ::= \bullet_1 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2^2 \mid \bullet_1 \rightarrow \bullet_2 \end{aligned}$$

The condition $\exists(\bullet_1^1 \boxplus \bullet_2^2)$ becomes

$$\begin{aligned} & \exists(\bullet_1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_1 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+0, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2) \vee \\ & \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \\ & \exists(\bullet_1 \xrightarrow{3 \bullet_3 \bullet_4} \boxed{+2, \bullet_3 \bullet_4} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_1 \xrightarrow{3 \bullet_3 \bullet_4} \boxed{+2, \bullet_3 \bullet_4} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+1, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \\ & \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3 \bullet_4} \boxed{+2, \bullet_3 \bullet_4} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+1, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \\ & \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+1, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \\ & \exists(\bullet_1 \rightarrow \bullet_3 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_3 \xrightarrow{4 \bullet_4} \boxed{+0, \emptyset} \xrightarrow{3 \bullet_3} \bullet_2) \vee \exists(\bullet_1 \rightarrow \bullet_4 \xrightarrow{3 \bullet_3} \boxed{+0, \emptyset} \xrightarrow{4 \bullet_4} \bullet_2) \end{aligned} \quad \diamond$$

The following theorem states that we can shift HR* conditions over any injective morphism which does not add hyperedges.

Theorem 6.3 (Shifting HR^* conditions over morphisms).

There is a transformation Integrate such that for any HR^* condition $\langle c, \mathcal{R} \rangle$ over P , any injective morphism $b: P \hookrightarrow P'$ isomorphic on hyperedges, there is an HR^* condition $\text{Integrate}(b, \langle c, \mathcal{R} \rangle)$ such that for any morphism $n: P' \rightarrow G$,

$$n \circ b \models \langle c, \mathcal{R} \rangle \iff n \models \text{Integrate}(b, \langle c, \mathcal{R} \rangle).$$

$$\begin{array}{ccc}
 c \triangleright P & \xrightarrow{b} & P' \triangleleft \text{Integrate}(b, c) \\
 & \searrow n \circ b & \swarrow n \\
 & & H
 \end{array}$$

The construction splits the morphism b into two parts for nodes and edges and then uses Integrate_n and Integrate_e .

Construction. We split morphism $b = b_e \circ b_n$, where b_n adds only nodes and b_e adds only edges. Let $\text{Integrate}(b, \langle c, \mathcal{R} \rangle) = \text{Integrate}_e(b_e, \text{Integrate}_n(b_n, \langle c, \mathcal{R} \rangle))$. $\text{\textcircled{e}}$

Proof. Without loss of generality, assume that \mathcal{R} has been expanded $k * |\text{LHS}(\mathcal{R})|$ times.

As in the construction, let $b = b_e \circ b_n$, where b_n adds only nodes and b_e adds only edges. By Lemma 6.3, we have $n \circ b_n \models \langle c, \mathcal{R} \rangle \iff n \models \langle c_n, \mathcal{R}_n \rangle$ with $\langle c_n, \mathcal{R}_n \rangle = \text{Integrate}_n(b_n, \langle c, \mathcal{R} \rangle) = \text{Integrate}(b_n, \langle c, \mathcal{R} \rangle)$.

This leaves us to prove that $n' \circ b_e \models \langle c_n, \mathcal{R}_n \rangle \iff n' \models \text{Integrate}(b_e, \langle c_n, \mathcal{R}_n \rangle)$ with $n' = n \circ b_n$. This proof proceeds by induction over the structure of HR^* conditions.

Assume that for any subcondition c , $n \circ b \models \langle c, \mathcal{R} \rangle \iff n \models \text{Integrate}(b, \langle c, \mathcal{R} \rangle)$.

Cases **true**, $\neg c$ and $c \wedge c'$, proceed analogous to the proof of Lemma 6.3; see the corresponding parts of the proof of Lemma 6.3.

For $n' \circ b_e \models \exists(P \rightarrow C, c)$, we do both directions of the proof separately.

“ \Leftarrow ”: Assume that $n' \models \text{Integrate}(b_e, \langle \exists(a, c), \mathcal{R}_n \rangle)$.

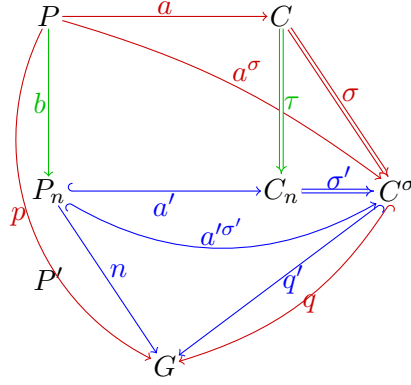
By construction of Integrate and Definition 3.9, this is equivalent to $\exists \sigma' \in \mathcal{R}_e^*, q': C'^{\sigma'} \hookrightarrow G. n = q \circ a'^{\sigma'} \wedge q' \models \text{Integrate}(b'_e, \langle c^{\sigma'}, \mathcal{R}_n \rangle)$.

For every rule $(x, k, G_y)/R'$ in \mathcal{R}' , there is a rule $(x, k)/R_n$ in \mathcal{R}_n , and R_n is isomorphic to R' up to some edges in R_n which are missing in R' . This implies that there is a substitution $\sigma \in \mathcal{R}_n$. By Lemma 6.3, this equals $n \circ b_n \circ b_e \models \langle c, \mathcal{R} \rangle$. By construction, for each edge in $R_n - R'$, there is a corresponding edge in G_y , and furthermore a corresponding edge in c' and in b_e .

This implies $\exists \sigma \in \mathcal{R}^*, q: C^\sigma \hookrightarrow G. n = q' \circ a^\sigma \wedge q' \models \text{Integrate}(b, \langle c^\sigma, \mathcal{R} \rangle)$.

By the induction hypothesis, $q \models \langle c^\sigma, \mathcal{R} \rangle$.

By the definition of HR^* satisfaction (Def. 3.9), this implies $n' \circ b_e \models \exists(a, c)$.



“ \Rightarrow ”: Assume that $n' \circ b_e \models \langle \exists(P \xrightarrow{a} C, c), \mathcal{R}_n \rangle$.

By the definition of satisfaction (Def. 3.9), this implies $\exists \sigma \in \mathcal{R}_n^*, q: C^\sigma \hookrightarrow G.n' \circ b_e = q \circ a^\sigma \wedge q \models c^\sigma$.

By construction, for each of the edges in $\text{Ran}(b_e) - \text{Dom}(b_e)$, there is a corresponding edge in c' and for each rule $(x, k)/R$, a rule in $(x, k, G_y)/R' \in \mathcal{R}_e$ with $E_{G_y} = E_{\text{Ran}(b_e) - \text{Dom}(b_e)}$ with R' being R minus the edges in G_y .

This implies $\exists \sigma' \in \mathcal{R}_e^*, q': C^{\sigma'} \hookrightarrow G.n' = q' \circ a'^{\sigma'} \wedge q' \models c^\sigma$. By the induction hypothesis, $q \models \langle c^\sigma, \mathcal{R}_n \rangle$. By the definition of HR* satisfaction (Def. 3.9), this equals $n' \models \text{Integrate}(b_e, \langle \exists(a, c), \mathcal{R}_n \rangle)$.

For $n' \circ b_e \models \exists(P \sqsupseteq C, c)$, the proof is analogous to the above for $\exists(P \hookrightarrow C, c)$.

Case $\exists(P \sqsupseteq C, c)$. Assume that $n' \circ b_e \models \langle \exists(P \sqsupseteq C, c), \mathcal{R}_n \rangle \Leftrightarrow \exists \sigma \in \mathcal{R}_n^*, q: C^\sigma \hookrightarrow G, a: C^\sigma \hookrightarrow P.q = n' \circ b_e \circ a$ and $q \models c^\sigma$. By construction, for each of the edges in $\text{Ran}(b_e) - \text{Dom}(b_e)$, there is a corresponding edge in c' and for each rule $(x, k)/R$, a rule in $(x, k, G_y)/R' \in \mathcal{R}_e$ with $E_{G_y} = E_{\text{Ran}(b_e) - \text{Dom}(b_e)}$ with R' being R minus the edges in G_y .

This means $\exists \sigma' \in \mathcal{R}_e^*, q': C^{\sigma'} \hookrightarrow G.q' = n' \circ a'^{\sigma'} \wedge q' \models c^\sigma$. By the induction hypothesis, $q \models \langle c^\sigma, \mathcal{R}_n \rangle$. By the definition of HR* satisfaction (Def. 3.9), this equals $n' \models \text{Integrate}(b_e, \langle \exists(P \sqsupseteq C, c), \mathcal{R}_n \rangle)$.

This concludes the proof. \square

With the help of Shift^* and Integrate , we can now define a construction $A(\rho, c)$ that transforms a postcondition c into a right application condition for rule ρ and prove Theorem 6.2.

Construction. For any rule $\rho = \langle \langle L \leftrightarrow K \hookrightarrow R \rangle, ac_L, ac_R \rangle$ and any HR* condition c , let

$$A(\rho, c) = \begin{cases} \text{Shift}^*(b, c) \wedge ac_R & \text{if } c \text{ is path-like} \\ \text{Integrate}(b, c) \wedge ac_R & \text{otherwise} \end{cases}$$

where $b: \emptyset \hookrightarrow R$ is the initial morphism to R . leaf

Proof (of Theorem 6.2). The proof follows directly from Theorems 6.1 and 6.3. \square

6.3 From right to left application conditions

The transformation L introduced in this chapter takes a rule $\rho = \langle L \leftrightarrow K \hookrightarrow R \rangle$ and a right application condition ac_R over R and constructs a left application condition ac_L over L . With Shift and L , one can construct a left application condition from an HR* constraint c over \emptyset : $ac_L = L(\rho, \text{Shift}(\emptyset \hookrightarrow R, c))$.

Theorem 6.4 (from right to left HR* application conditions).

There is a transformation L such that, for every HR* application condition ac of a rule $\rho = \langle L \leftrightarrow K \hookrightarrow R \rangle$ and for all direct derivations $G \xRightarrow[\rho, m, m']{=} H$, $m \models L(\rho, ac) \Leftrightarrow m' \models ac$.

For the construction of L , the HR* condition has to be in a special form, called *full-containment normal form*. This normal form concerns the containment operator, i.e. conditions of the form $\exists(P \sqsupseteq C, c)$, and enforces that C contains every node and edge of P (but not necessarily the hyperedges), so that any items in C that are not in P may only be mapped to items “hidden” in hyperedges of P .

Definition 6.2 (full-containment normal form). An HR* condition is said to be in *full-containment normal form* if, for every subcondition $\exists(P \sqsupseteq C, c)$ in the condition, $P - Y_P$ (i.e. P without its hyperedges) is a subgraph of C . \triangle

To transform an HR* condition into full-containment normal form, subconditions of the form $\exists(P \sqsupseteq C, c)$ are changed by merging all nodes and edges (but not the hyperedges) in P with the nodes and edges in C .

Construction. The construction is inductively defined over HR* conditions. For a condition $\exists(P \sqsupseteq C, c)$, let $\text{FC}(\exists(P \sqsupseteq C, c)) = \bigvee_{(a', b') \in \mathcal{F}} \exists(P \sqsupseteq C', \text{FC}(\text{Shift}(b', c)))$, where a' , b' and C' are the result of a Shift as given in Chapter 2.4, P_Y is P without hyperedges and $P_Y \leftarrow I \rightarrow C$ is the partial morphism induced by the partial morphism $P \leftarrow I \rightarrow C$ from C to P .

$$\begin{array}{ccc} I & \xrightarrow{a} & C \\ \downarrow b & = & \downarrow b' \\ P_Y & \xrightarrow{a'} & C' \end{array}$$

For all other conditions, FC is straightforwardly passed through: $\text{FC}(\mathbf{true}) = \mathbf{true}$, $\text{FC}(\exists(a, c)) = \exists(a, \text{FC}(c))$, $\text{FC}(c \wedge c') = \text{FC}(c) \wedge \text{FC}(c')$ and $\text{FC}(\neg c) = \neg \text{FC}(c)$. $\text{\textcircled{e}}$

It is easy to see that this construction puts the condition in full-containment normal form, as every node and edge in P has an image in C , thus P without hyperedges is a subgraph of C .

6.3 From right to left application conditions

Example 6.7. Regard the HR* condition $c = \forall(\bullet \xrightarrow{+} \bullet, \nexists(\bullet \xrightarrow{+} \bullet \sqsupseteq \bullet, \exists \bullet \xleftarrow{-} \bullet \rightarrow \bullet))$ expressing the property ‘‘On all paths, no inner node has two outgoing edges’’. The full-containment normal form of c would be

$$\begin{aligned} \text{FC}(c) = & \forall(\bullet \xrightarrow{+} \bullet, \neg(\exists(\bullet \xrightarrow{+} \bullet \sqsupseteq \bullet \bullet \bullet, \exists \bullet \bullet \xleftarrow{-} \bullet \rightarrow \bullet \bullet)) \\ & \vee \exists(\bullet \xrightarrow{+} \bullet \sqsupseteq \bullet \bullet \bullet, \exists \bullet \bullet \xleftarrow{-} \bullet \rightarrow \bullet \bullet)) \\ & \vee \exists(\bullet \xrightarrow{+} \bullet \sqsupseteq \bullet \bullet \bullet, \exists \bullet \bullet \xleftarrow{-} \bullet \rightarrow \bullet \bullet)) \end{aligned} \quad \diamond$$

Lemma 6.4 (Well-formedness of full-containment normal form).

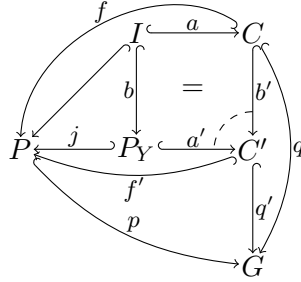
For every HR* condition c over P , $\text{FC}(c)$ is an equivalent full-containment normal form of c , i.e.

$$p \models c \iff p \models \text{FC}(c) \text{ for every } p: P \hookrightarrow G.$$

Proof. By induction over the structure of conditions.

For a condition of the form $\exists(P \sqsupseteq C, c)$, and a morphism $p: P \hookrightarrow G$, we have $p \models \text{FC}(\exists(P \sqsupseteq C, c)) = \bigvee_{(a', b') \in \mathcal{F}} \exists(P \sqsupseteq C', \text{FC}(\text{Shift}(b', c)))$. By the semantics of HR* conditions, this means there exist morphisms $f': C' \hookrightarrow P$ and $q': C' \hookrightarrow G$ such that $q' = p \circ f'$ and $q' \models \text{FC}(\text{Shift}(b', c))$. Let $q = q' \circ b'$ and $f = f' \circ b'$. Then $q = q' \circ b' = p \circ f' \circ b' = p \circ f$ and, by the induction hypothesis, $q \models c$, thus $p \models \exists(P \sqsupseteq C, c)$.

Conversely, assume that $p \models \exists(P \sqsupseteq C, c)$. By the semantics of HR* conditions, this means there exist morphisms $f: C \hookrightarrow P$ and $q: C \hookrightarrow G$ such that $q = p \circ f$ and $q \models c$. By \mathcal{E}' - \mathcal{M} -pair factorization of $(p \circ j, q)$, we get the jointly surjective morphism pair (a', b') and $q': C' \hookrightarrow G$ such that $p \circ j = q' \circ a'$ and $q = b' \circ q'$. A morphism $f': C' \hookrightarrow P$ can be constructed analogously. Then we have $q' \circ b' = q = p \circ f = p \circ f' \circ b'$, thus $q' = p \circ f'$. By the induction hypothesis, $q' \models c$, thus $p \models \text{FC}(\exists(P \sqsupseteq C, c))$.



For all other forms of conditions, FC does not change the condition. This completes the proof. \square

Intuitively, the transformation L applies the reverse of the rule to each morphism and object in the condition, yielding **false** whenever the dangling condition is not met.

Construction. L is defined inductively similar to (Habel and Pennemann, 2009): For any rule ρ with (right) application condition of the form $\text{ac}_R = \exists(a, c)$, $L(\rho, \exists(a, c)) =$

$\exists(a', L(\rho', c))$ as per the left diagram below if $\langle r, a \rangle$ has pushout complement (1) and $\rho' = \langle L' \leftarrow K' \hookrightarrow R' \rangle$ is the derived rule by constructing pushout (2), and **false** otherwise. $L(\rho, \exists(R \sqsupseteq C_R, c)) = \exists(L \sqsupseteq C_L, L(\rho', c))$, where $L \leftarrow L-Y_P \rightarrow C_L$ is the partial morphism from C' to P' as per the right diagram below, if the lower right square is a pushout, and **false** otherwise.

$$\begin{array}{ccc}
 \rho = & P' \leftarrow K_P \hookrightarrow P & \\
 & \downarrow a' \quad (2) \quad \downarrow (1) \quad \downarrow a & \\
 \rho' = & C' \leftarrow K_C \hookrightarrow C & \\
 & \blacktriangle \quad L(\rho', c) \quad \blacktriangle \quad c & \\
 \end{array}
 \qquad
 \begin{array}{ccc}
 L \leftarrow K \hookrightarrow R = \rho & & \\
 & \downarrow (PO) \quad \downarrow (PO) & \\
 L-Y_P \leftarrow K-Y_P \hookrightarrow R-Y_P & & \\
 & \downarrow (PO) \quad \downarrow (PO) & \\
 C_L \leftarrow K_C \hookrightarrow C_R = \rho' & & \\
 & \blacktriangle \quad L(\rho', c) \quad \blacktriangle \quad c & \\
 \end{array}$$

For Boolean formulas over HR* conditions, the construction is straightforward: $L(\rho, \text{true}) = \text{true}$, $L(\rho, \neg c) = \neg L(\rho, c)$ and $L(\rho, c \wedge c') = L(\rho, c) \wedge L(\rho, c')$. \(\circlearrowleft\)

Example 6.8. Regard the rule $\rho = \langle \bullet_2 \rightarrow \bullet_4 \leftarrow \bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4 \rangle$ with right application condition $\text{ac}_R = \exists(\bullet_2 \leftarrow \bullet_1 \xrightarrow{+} \bullet_2, \#(\bullet_1 \xrightarrow{+} \bullet_2 \sqsupseteq \bullet_1 \bullet_3 \bullet_2, \exists(\bullet_1 \leftarrow \bullet_3 \bullet_2)))$, which is already in full-containment normal form. Intuitively, ac_R ensures that there is a path from some node 1 to node 2 from the rule, and no node 3 on the path has an edge going back to 1.

We compute $L(\rho, \text{ac}_R)$ step by step, with the necessary constructions following the formulas.

$$\begin{aligned}
 & L(\rho, \text{ac}_R) \\
 & \equiv \exists(\bullet_2 \rightarrow \bullet_4 \leftarrow \bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4, L(\rho_1, \#(\bullet_1 \bullet_3 \bullet_2 \sqsupseteq \bullet_1 \bullet_3 \bullet_2, \exists(\bullet_1 \leftarrow \bullet_3 \bullet_2)))) \\
 & \equiv \exists(\bullet_2 \rightarrow \bullet_4 \leftarrow \bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4, \#(\bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4 \sqsupseteq \bullet_1 \bullet_3 \bullet_2 \rightarrow \bullet_4, L(\rho_2, \exists(\bullet_1 \leftarrow \bullet_3 \bullet_2)))) \\
 & \equiv \exists(\bullet_2 \rightarrow \bullet_4 \leftarrow \bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4, \#(\bullet_1 \xrightarrow{+} \bullet_2 \rightarrow \bullet_4 \sqsupseteq \bullet_1 \bullet_3 \bullet_2 \rightarrow \bullet_4, \exists(\bullet_1 \leftarrow \bullet_3 \bullet_2 \rightarrow \bullet_4)))
 \end{aligned}$$

◇

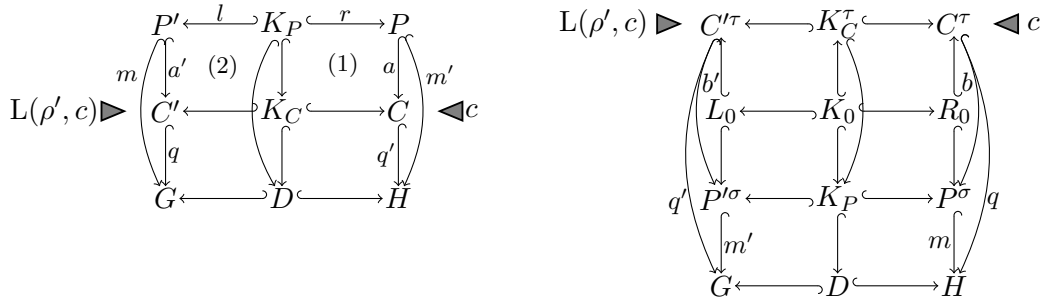
Proof (of Theorem 6.4). By induction over the structure of ac .

Basis (Case $ac = \text{true}$). For any injective morphism m , we have $m \models L(\rho, \text{true}) = \text{true} \Leftrightarrow \text{true} \Leftrightarrow m' \models \text{true}$.

Hypothesis. Assume that $m \models L(\rho, c) \Leftrightarrow m' \models c$ holds for application condition c .

Step. We proceed by case distinction over the structure of ac .

Case $ac = \exists(a, c)$. Regard the left picture below. Let $\rho = \langle P' \leftarrow K_P \hookrightarrow P \rangle$ and $\rho' = \langle C' \leftarrow K_C \hookrightarrow C \rangle$. Assume that (r, a) has a pushout complement. Then $m \models L(\rho, ac) \Leftrightarrow m \models \exists(a', L(\rho', c))$. For the derivation, we can decompose the pushouts of the derivation $G \xRightarrow[\rho, m, m']{m} H$ such that $m = q \circ a$ and $m' = q' \circ a'$. By the hypothesis, $q' \models L(\rho', c) \Leftrightarrow q' \models c$, thus $m' \models \exists(a, c)$. If (m, a) has no pushout complement, $m \models L(m, ac) \Leftrightarrow m \models \text{false}$ and there is no pushout such that $m' \models c$, i.e. $m' \models \text{false}$.



Case $ac = \exists(P \sqsupseteq C, c)$. Regard the right picture above. Assume that some graph H satisfies $\exists(P \sqsupseteq C, c)$, and that H results from applying rule ρ to G . This is equivalent to the existence of substitutions σ, τ and injective morphisms $b: C^\tau \hookrightarrow P^\sigma$ and $q: C^\tau \hookrightarrow H$ such that $P^\sigma = P^\tau$, $q = m \circ b$ and $q \models c$. The existence of $b: C^\tau \hookrightarrow P^\sigma$ and $P^\sigma = P^\tau$ imply that a (sub-)morphism $b_Y: Y_C^\tau \hookrightarrow Y_P^\tau$ from the substituted hyperedges of C to the substituted hyperedges of P exists, and that morphism $b': C'^\tau \hookrightarrow P'^\sigma$ exists. We can now construct $q': C'^\tau \hookrightarrow G$ from $q' = m' \circ b'$. Because of the induction hypothesis, $q' \models L(\rho', c)$. This is equivalent to $\exists(P' \sqsupseteq C', L(\rho', c))$, completing this part of the proof.

Case $ac = \neg c$. Then $m \models L(\rho, \neg c) \Leftrightarrow m \models \neg L(\rho, c) \Leftrightarrow \neg m \models L(\rho, c) \Leftrightarrow \neg m' \models c \Leftrightarrow m' \models \neg c$.

Case $ac = c \wedge c'$. Then $m \models L(\rho, c) \wedge c' \Leftrightarrow m \models L(\rho, c) \wedge m \models L(\rho, c') \Leftrightarrow m' \models c \wedge m' \models c' \Leftrightarrow m' \models c \wedge c'$.

By induction, we have thus proven Theorem 6.4. \square

Remark. By the symmetric nature of graph transformation rules, the above construction can be easily reversed to transform a left into a right application condition.

6.4 From left application conditions to preconditions

Having transformed a right application condition into a left one, the question remains how to transform a left application condition (over the left-hand side L of some rule) into a (pre-)condition over the empty graph. We now define such a transformation C . As in (Habel, Pennemann, and Rensink, 2006), this is done by ensuring that the left application condition is valid for every morphism $\emptyset \rightarrow L$.

Theorem 6.5 (from left application conditions to preconditions).

For every HR* application condition ac over L and all graphs G , there is an HR* condition $C(ac)$ over \emptyset such that

$$G \models C(ac) \iff \forall m: L \hookrightarrow G.m \models ac.$$

The construction is the same as in (Habel, Pennemann, and Rensink, 2006); surprisingly, it is not necessary to adapt it for graphs with variables or the containment operator.

Construction. For any condition ac over L , let $C(ac) = \forall(\emptyset \hookrightarrow L, ac)$. ✍

The proof is similar to the one given in (Habel, Pennemann, and Rensink, 2006) for nested conditions.

Proof. Let $p: \emptyset \hookrightarrow G$ be a morphism.

$$\begin{aligned} G \models C(ac) & && \text{Def. 3.9} \\ \Leftrightarrow p \models C(ac) & && \text{construction} \\ \Leftrightarrow p \models \forall(\emptyset \xrightarrow{i_L} L, ac) & && \text{Def. 3.9} \\ \Leftrightarrow \forall m: L \hookrightarrow G.p = m \circ i_L \wedge m \models ac & && p = m \circ i_L \text{ true by construction} \\ \Leftrightarrow \forall m: L \hookrightarrow G.m \models ac. & && \end{aligned}$$

□

Example 6.9. We apply transformation C to the rule $\langle \bullet_1 \bullet_2 \leftrightarrow \bullet_1 \bullet_2 \leftrightarrow \bullet_1 \rightarrow \bullet_2 \rangle$ with left application condition $ac_L = \#(\bullet_1 \bullet_2 \leftrightarrow \bullet_1 \rightarrow \bullet_2)$.

$$C(ac_L) = \forall(\emptyset \hookrightarrow \bullet_1 \bullet_2, \#(\bullet_1 \bullet_2 \leftrightarrow \bullet_1 \rightarrow \bullet_2)) \equiv \#(\bullet_1 \bullet_2, \exists(\bullet_1 \rightarrow \bullet_2))$$

meaning that no pair of nodes is connected by an edge. ◇

6.5 Expressing the applicability of a rule as a condition

Furthermore, we need a transformation that expresses the applicability of a rule. Basically, this amounts to checking whether the left-hand side of the rule has a match in the graph that does not violate the dangling condition. Since rules do not contain hyperedges, this construction is identical to the one presented in (Habel, Pennemann, and Rensink, 2006).

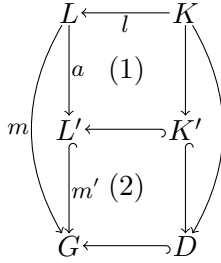
Lemma 6.5 (applicability of a rule).

There is a transformation Appl from rules into application conditions such that, for every rule ρ and every morphism $m: L \rightarrow G$, $m \models \text{Appl}(\rho) \Leftrightarrow \exists H.G \xRightarrow[\rho, m, m']{*} H$.

Construction (Appl). The construction is identical to the one for nested conditions in (Habel, Pennemann, and Rensink, 2006). For a plain rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$, let $\text{Appl}(p) = \bigwedge_{a \in A} \#a$, where A is the set of all graph morphisms $a: L \rightarrow L'$ such that $\langle l, a \rangle$ has no pushout complement and A is minimal, i.e. there is no decomposition $a = a'' \circ a'$ with $a'' \in \mathcal{M}$, $a'' \neq \text{id}$ such that $\langle l, a' \rangle$ has no pushout complement. For a rule $\rho = \langle p, \text{ac}_L, \text{ac}_R \rangle$, let $\text{Appl}(\rho) = \text{Appl}(p) \wedge \text{ac}_L \wedge \text{L}(\rho, \text{ac}_R)$. \spadesuit

Proof. Similar to the proof for nested constraints in (Habel, Pennemann, and Rensink, 2006, p. 30f). We first show that, for any plain rule q and morphism $m: L \rightarrow G$, $m \models \text{Appl}(q) \iff \exists H.G \xRightarrow{q, m, m'} H$.

“ \Leftarrow ”: Assume there is no direct derivation $G \xRightarrow{q, m, m'} H$. Then the pair $\langle l, m \rangle$ has no pushout complement, and there is a decomposition $a: L \rightarrow L'$ such that $\langle l, a \rangle$ has no pushout complement and $m \models \exists a$. This implies $m \not\models \text{Appl}(q)$. Since this contradicts the assumption, there is a direct derivation $G \xRightarrow{q, m, m'} H$.



“ \Rightarrow ”: Let $G \xRightarrow{q, m, m'} H$. Then, for every morphism $a: L \rightarrow L'$, $m \models \exists a$ iff there is as $m': L' \rightarrow G$ in \mathcal{M} such that $m = m' \circ a$. By the pushout-pullback decomposition, pushout (1) + (2) has a decomposition into two pushouts (1) and (2) and, in particular, $\langle l, a \rangle$ has a pushout complement. Thus, for every $a \in A$, $m \models \neg \exists a$, and $m \models \text{Appl}(q)$.

By the definition of L and the statement above, it follows for every rule $\rho = (q, \text{ac}_L, \text{ac}_R)$ and every morphism $m: L \rightarrow G$, $m \models \text{Appl}(\rho)$ iff $m \models \text{Appl}(q) \wedge m \models \text{ac}_L \wedge m \models \text{L}(\rho, \text{ac}_R)$ iff $\exists H.G \xRightarrow{q, m, m'} \bigwedge m \models \text{ac}_L \wedge m' \models \text{ac}_R$ iff $\exists H.G \xRightarrow{q, m, m'} H$. This completes the proof. \square

Example 6.10. For the plain rule $q = \langle \textcircled{a}_1 \leftarrow \emptyset \rightarrow \textcircled{c} \rangle$,

$$\text{Appl}(q) = \#(\textcircled{a}_1 \rightarrow \bullet) \wedge \#(\textcircled{a}_1 \leftarrow \bullet) \wedge \#(\textcircled{a}_1 \rightarrow \textcircled{a}_1)$$

meaning that node \textcircled{a}_1 has no incoming edge, no outgoing edge and no loop edge, i.e. no adjacent edge at all. Otherwise, deletion of the node would violate the dangling condition. \diamond

6.6 Correctness of graph programs

Following the approach taken in (Habel and Pennemann, 2009) for nested conditions, the constructions introduced in this chapter can be used to transform a postcondition

over a graph program into a weakest precondition. Weakest preconditions are useful in the context of *specifications*, i.e. a program P together with a precondition c and a postcondition d . A specification is *correct* if for every graph G satisfying the precondition c , any graph H resulting from execution of the program P on G satisfies the postcondition. As suggested by (Dijkstra, 1976), we can check the correctness of a specification by generating a weakest precondition from the program and the postcondition, and checking whether the original precondition implies the weakest precondition.

Definition 6.3 (weakest precondition). A condition c is a *liberal precondition* of a program P relative to condition d if for all graphs $G \models c$, $G \Rightarrow_P H$ implies $H \models d$ for all H . A liberal precondition c is a *weakest liberal precondition* of P relative to d if any liberal precondition of P relative to d implies c . A weakest liberal precondition c of P is a *weakest precondition* if there is at least one graph H such that $G \Rightarrow_P H$ and $G \models c$. Δ

The following theorem establishes that HR* conditions can be transformed over a program into a weakest precondition.

Theorem 6.6 (weakest precondition).

For every graph program P and condition d , there are conditions $\text{Wlp}(P, d)$ and $\text{Wp}(P, d)$ such that $\text{Wlp}(P, d)$ is a weakest liberal precondition of P relative to d and $\text{Wp}(P, d)$ is a weakest precondition of P relative to d .

Given a graph program P , a precondition c and a postcondition d , we can check whether the specification is correct by checking if the implication $c \implies \text{Wp}(P, d)$ holds.

Construction. For any rule ρ , programs P, P_1, \dots, P_n and condition d , let

1. $\text{Wlp}(\rho, d) = \text{C}(\text{Appl}(\rho) \Rightarrow \text{L}(\rho, \text{A}(\rho, d)))$
2. $\text{Wlp}(\{P_1, \dots, P_n\}, d) = \text{Wlp}(P_1, d) \vee \dots \vee \text{Wlp}(P_n, d)$
3. $\text{Wlp}(P_1; P_2, d) = \text{Wlp}(P_1, \text{Wlp}(P_2, d))$
4. $\text{Wlp}(P \downarrow, d) = \bigwedge_{i=0}^n \text{Wlp}(P^i, \text{Wlp}(P, \text{false}) \Rightarrow d)$
5. $\text{Wp}(P, d) = \text{Wlp}(P, d) \wedge \neg \text{Wlp}(P, \text{false})$.



Remark. Note that in general, the construction for $\text{Wlp}(P \downarrow, d)$ yields an infinite condition. This is ineffective for practical applications. One way to fix this is to use the approximation $w_k = \bigvee_{i=0}^k \text{Wlp}(P^i, d)$ for some $k \in \mathbb{N}$. If there is a k with $\text{Wlp}(P \downarrow, d) \Rightarrow w_k$, then this approximation is equivalent (since $w_k \Rightarrow \text{Wlp}(P \downarrow, d)$) and we have a finite weakest liberal precondition. Another approach might be counterexample-guided refinement, as done in (Pennemann, 2009, Chapter 5.4).

Proof (of Theorem 6.6). We first prove a small lemma (lem) stating that $G \models \neg \text{Wlp}(P, \text{false})$ iff $G \Rightarrow_P H$ for some graph H .

$G \models \neg \text{Wlp}(P, \text{false})$	Def. \models
$\Leftrightarrow \neg G \models \text{Wlp}(P, \text{false})$	part (1)
$\Leftrightarrow \neg \forall H. G \Rightarrow_P H \Rightarrow H \models \text{false}$	Logic Axioms
$\Leftrightarrow \neg \forall H. \neg(G \Rightarrow_P H \wedge H \models \text{true})$	Logic Axioms
$\Leftrightarrow \exists H. G \Rightarrow_P H \wedge H \models \text{true}$	Tautology
$\Leftrightarrow \exists H. G \Rightarrow_P H$	

1. For all graphs G , we have:

$G \models \text{Wlp}(\rho, d)$	Construction
$\Leftrightarrow G \models \text{C}(\text{Appl}(\rho) \Rightarrow \text{L}(\rho, \text{A}(\rho, d)))$	Def. 3.9
$\Leftrightarrow \forall m: L \rightarrow G. m \models \text{Appl}(\rho) \Rightarrow \text{L}(\rho, \text{A}(\rho, d))$	Logic
$\Leftrightarrow \forall m: L \rightarrow G. m \models \text{Appl}(\rho)$ implies $m \models \text{L}(\rho, \text{A}(\rho, d))$	Theorem 6.4
$\Leftrightarrow \forall m: L \rightarrow G. m \models \text{Appl}(\rho)$ implies $m \models \text{L}(\rho, \text{A}(\rho, d))$	Def. 3.9
$\Leftrightarrow \forall m: L \rightarrow G, m': R \rightarrow H. m \models \text{Appl}(\rho) \Rightarrow m' \models \text{A}(\rho, d)$	Theorem 6.2
$\Leftrightarrow \forall m: L \rightarrow G, m': R \rightarrow H. m \models \text{Appl}(\rho) \Rightarrow H \models d$	Lemma 6.5
$\Leftrightarrow \forall H. G \Rightarrow_P H \Rightarrow H \models d$	

Thus, $\text{Wlp}(\rho, d)$ is a weakest liberal precondition of ρ relative to d .

2. For all graphs G , we have

$G \models \text{Wlp}(\{P_1, \dots, P_n\}, d)$	Construction
$\Leftrightarrow G \models \text{Wlp}(P_1, d) \vee \dots \vee \text{Wlp}(P_n, d)$	part (1)
$\Leftrightarrow \forall H. G \Rightarrow_{P_1} H \Rightarrow H \models d \vee \dots \vee G \Rightarrow_{P_n} H \Rightarrow H \models d$	Def. 2.7
$\Leftrightarrow \forall H. G \Rightarrow_{\{P_1, \dots, P_n\}} H \Rightarrow H \models d.$	

3. For all graphs G , we have

$G \models \text{Wlp}(P_1; P_2, d)$	Construction
$\Leftrightarrow G \models \text{Wlp}(P_1, \text{Wlp}(P_2, d))$	part(1)
$\Leftrightarrow \forall G', H. G \Rightarrow_{P_1} G' \Rightarrow G' \models \text{Wlp}(P_2, d) \wedge G' \Rightarrow_P H \Rightarrow H \models d$	Def. 2.7
$\Leftrightarrow \forall H. G \Rightarrow_{P_1; P_2} H \Rightarrow H \models d.$	

4. Let $W_i = \text{Wlp}(P^{i-1}, \text{Wlp}(P, \text{Wlp}(P, \text{false}) \Rightarrow d))$. For all graphs G , we have:

$G \models \text{Wlp}(P \downarrow, d)$	Construction
$\Leftrightarrow G \models \bigwedge_{i=0}^n \text{Wlp}(P^i, \text{Wlp}(P, \text{false}) \Rightarrow d)$	Def. 2.7
$\Leftrightarrow G \models (\text{Wlp}(P, \text{false}) \Rightarrow d) \wedge$ $\quad \forall i \in \mathbb{N} - \{0\}. G \models W_i$	Logic
$\Leftrightarrow G \models (d \vee \neg \text{Wlp}(P, \text{false})) \wedge \forall i \in \mathbb{N} - \{0\}. G \models W_i$	Lemma (lem)
$\Leftrightarrow (G \models d \vee \forall H. G \Rightarrow_P H \Rightarrow H \models d) \wedge \forall i \in \mathbb{N} - \{0\}. G \models W_i.$	
By induction over i , we get	
$(G \models d \vee \forall H. G \Rightarrow_P H \Rightarrow H \models d) \wedge \forall i \in \mathbb{N} - \{0\}. G \models W_i$	Def. W_i
$\Leftrightarrow (G \models d \vee \forall H. G \Rightarrow_P H \Rightarrow H \models d) \wedge \forall i \in \mathbb{N} - \{0\}.$	
$\quad G \models \text{Wlp}(P^{i-1}, \text{Wlp}(P, \text{Wlp}(P, \text{false}) \Rightarrow d))$	induction
$\Leftrightarrow (G \models d \vee \forall H. G \Rightarrow_P H \Rightarrow H \models d) \wedge \forall i \in \mathbb{N} - \{0\} (G \models$	Logic
$\quad d \vee \forall H^i. G \Rightarrow_{P^i} H^i \Rightarrow H^i \models \text{Wlp}(P^{i-1}, d))$	
$\Leftrightarrow \forall H. G \Rightarrow_{P \downarrow} H \Rightarrow H \models d.$	

5. By construction, for any graph G , $G \models \text{Wp}(P, \text{false})$ iff $G \models \text{Wlp}(P, d) \wedge \neg \text{Wlp}(P, \text{false})$. By the semantics of HR* conditions, this equals $G \models \text{Wlp}(P, d) \wedge G \models \neg \text{Wlp}(P, \text{false})$. By part (1-4) of the proof, for any program P , we know that $G \models \text{Wlp}(P, d)$ iff $\text{Wlp}(P, d)$ is a weakest precondition of P relative to d . By part (0), $G \models \neg \text{Wlp}(P, \text{false})$ iff $G \Rightarrow_P H$ for some graph H . Thus, for every program P , $\text{Wp}(P, d) = \text{Wlp}(P, d) \wedge \neg \text{Wlp}(P, \text{false})$ is a weakest precondition of P relative to d .

□

For illustration, we return to the car platooning example from Chapter 3.4.

Example 6.11. We check whether the constraint “a follower always has an incoming path from a terminator and an outgoing path to a leader” is an invariant for the Merge rule for a free agent and a platoon:

$$\begin{aligned}
 c &= \forall (\text{grey}_2, \exists (\text{red}_1 \xrightarrow{+} \text{grey}_2 \xrightarrow{+} \text{green}_3)) \\
 \rho &= \text{blue}_a \text{ red}_b \Rightarrow \text{red}_a \text{ grey}_b \\
 \text{with } \text{grey}_1 \xrightarrow{+} \text{grey}_2 &::= \text{grey}_1 \rightarrow \text{grey}_2 \mid \text{grey}_1 \xrightarrow{+} \text{grey}_2 \xrightarrow{+} \text{grey}_3
 \end{aligned}$$

Note that the replacement rules can be applied to non-white cars because, as stated in Chapter 3.4, a node’s color is modeled by a loop edge labeled with the color.

We first construct a right application condition ac_R from our postcondition c by *shifting* the rule’s right-hand side into the condition.

$$\begin{aligned}
 ac_R &= \forall (\text{red}_a \rightarrow \text{grey}_b, \exists (\text{red}_a \rightarrow \text{grey}_b, \text{grey}_2, \text{red}_1 \xrightarrow{+} \text{grey}_2 \xrightarrow{+} \text{green}_3) \vee \exists (\text{red}_a \xrightarrow{+} \text{grey}_b, \text{grey}_2 \xrightarrow{+} \text{green}_3)) \wedge \\
 &\quad \forall (\text{red}_a \rightarrow \text{grey}_{b=2}, \exists (\text{red}_a \xrightarrow{+} \text{grey}_{b=2}, \text{red}_1 \xrightarrow{+} \text{grey}_2 \xrightarrow{+} \text{green}_3) \vee \exists (\text{red}_{a=1} \rightarrow \text{grey}_{b=2} \xrightarrow{+} \text{green}_3))
 \end{aligned}$$

Remark. There are more possibilities of identifying nodes a and b with nodes 1, 2 and 3 than those shown. However, only the two combinations above, i.e. unifying either $a = 1$ and $b = 2$ together or neither of them, do not contradict any of the constraints given for the respective car types. Since we assume that the rule is applied to a valid model, these invalid combination are left out.

We can now construct the left application condition ac_L from ac_R . Basically, this amounts to applying the rule backwards on ac_R . In this case, the colors of the cars change, and any part of the condition where this process would unify cars of different colors evaluates to **false** (for colors encoded as node labels; since we use loop edges here, an additional constraint forbidding nodes with more than one color is needed).

$$ac_L = \forall (\text{blue}_a \text{ red}_b, \exists (\text{blue}_a \text{ red}_b, \text{grey}_2, \text{red}_1 \xrightarrow{+} \text{grey}_2 \xrightarrow{+} \text{green}_3) \vee \text{false}) \wedge \neg \text{false}$$

To construct the weakest precondition, we need to make sure that the left application condition ac_L is satisfied for any possible match of the rule, under the condition that the rule is applicable:

$$\begin{aligned} \text{Appl}(\rho) = & \#(\boxed{\text{blue}_a} \rightarrow \boxed{\text{red}_b}) \wedge \#(\boxed{\text{blue}_a} \leftarrow \boxed{\text{red}_b}) \wedge \\ & \#(\boxed{\text{white}} \rightarrow \boxed{\text{blue}_a} \quad \boxed{\text{red}_b}) \wedge \#(\boxed{\text{white}} \leftarrow \boxed{\text{blue}_a} \quad \boxed{\text{red}_b}) \wedge \#(\boxed{\text{white}} \rightarrow \boxed{\text{blue}_a} \quad \boxed{\text{red}_b}) \wedge \\ & \#(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b} \rightarrow \boxed{\text{white}}) \wedge \#(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b} \leftarrow \boxed{\text{white}}) \wedge \#(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b} \rightarrow \boxed{\text{white}}) \end{aligned}$$

$$\text{Wlp}(\rho, c) = C(\text{Appl}(\rho) \Rightarrow ac_L) = \forall(L, \text{Appl}(\rho) \Rightarrow ac_L)$$

$$\forall(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \text{Appl}(\rho) \Rightarrow \forall(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \exists(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \exists(\boxed{\text{red}_1} \rightarrow \boxed{\text{grey}_2} \rightarrow \boxed{\text{green}_3}))))$$

Our original goal was to check whether condition c is an invariant for the **Merge** rule, i.e. whether for all graphs G satisfying c , all graphs H resulting from application of **Merge** to G also satisfy c . Following the approach from (Dijkstra, 1976), we computed a weakest precondition c_{wlp} and now have to check whether the original precondition c implies the weakest precondition c_{wlp} .

$$\begin{aligned} & \forall(\boxed{\text{grey}_2}, \exists(\boxed{\text{red}_1} \rightarrow \boxed{\text{grey}_2} \rightarrow \boxed{\text{green}_3})) \\ \Rightarrow & \\ & \forall(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \text{Appl}(\rho) \Rightarrow \forall(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \exists(\boxed{\text{blue}_a} \quad \boxed{\text{red}_b}, \exists(\boxed{\text{red}_1} \rightarrow \boxed{\text{grey}_2} \rightarrow \boxed{\text{green}_3})))) \quad \diamond \end{aligned}$$

In this case, the implication holds: if every follower car is on a path from a terminator to a leader, this is also true in the context of an additional free car a and a terminator b with no in- or outgoing edges.

The (generally undecidable) problem of proving whether an implication $\text{imp} := c \Rightarrow \text{Wlp}(P, d)$ holds for an HR^* precondition c , postcondition d and program P can be approached in different ways. One way would be to use the transformation SO from Chapter 5.3 to transform imp into a second-order graph formula and then use a higher-order theorem prover, e.g. CoQ (Bertot and Castéran, 2004) or Isabelle (Nipkow et al., 2002), to conduct the proof. Another way lies in the development of a dedicated calculus and theorem prover for HR^* conditions, similar to the ProCon prover for nested graph conditions (Pennemann, 2008b), which can be supplemented by a SAT solver similar to SeekSat (Pennemann, 2008a) looking for counterexamples. However, a calculus for HR^* conditions has to match graphs with variables and possibly different hyperedge replacement systems. For example, it would have to check whether the condition

$$\exists(\bullet \xrightarrow{+1} \bullet) \text{ with } \bullet \xrightarrow{+1} \bullet ::= \bullet \rightarrow \bullet \rightarrow \bullet \mid \bullet \rightarrow \bullet \xrightarrow{+1} \bullet$$

is equivalent to the condition

$$\exists(\bullet \rightarrow \bullet \xrightarrow{+2} \bullet) \text{ with } \bullet \xrightarrow{+2} \bullet ::= \bullet \rightarrow \bullet \mid \bullet \xrightarrow{+2} \bullet \xrightarrow{+2} \bullet,$$

which both express the property ‘‘There is a path over at least two edges from node 1 to node 2’’. This difficult problem is beyond the scope of this PhD thesis and left as future work.

Bibliographic notes

The idea of calculating weakest preconditions to prove the adherence of a program to given pre- and postconditions originates from (Dijkstra, 1976). (Heckel and Wagner, 1995) first described basic transformations over rules for consistency constraints (i.e. constraints of the form $\forall(P, \exists(P \rightarrow C))$). Basic transformations for nested conditions were first described in (Habel, Pennemann, and Rensink, 2006); an investigation for strongest postconditions is given in (Habel and Pennemann, 2009). (Pennemann, 2009) provides translation over *programs with interfaces*, a more fine-grained approach to graph programs. (Blume, 2014) provides verification for graph transformation systems over invariants specified by bounded graph automata, and gives counter-examples in the negative case.

Chapter 7

Application to meta-modeling

Contents

7.1	An overview of the Object Constraint Language	94
7.2	Graphs and conditions for OCL	101
7.3	Translating Essential OCL to graph conditions	110
7.4	Translating OCL constraints beyond first-order expressiveness	120
7.5	Integration of graph constraints into graph grammars	124

The following chapter is largely oriented at the joint paper (Radke et al., 2015) of the author with Thorsten Arendt, Jan Steffen Becker, Annegret Habel and Gabriele Taentzer.

The goal of this chapter is to apply graph transformation techniques to the problem of meta-model instance generation. Usually, meta-models are formalized in a two-fold way: Structure and typing of the meta-model are specified by a class diagram, and additional constraints can be specified in the OCL (Object Constraint Language). OCL is part of the industry-standard Unified Modeling Language (UML) used to graphically specify meta-models.

This is a purely declarative way of specification. However, some tasks, such as generating instances (K. Ehrig et al., 2009) or generating or recognizing edit operations (Kehrer et al., 2013), are better suited for a constructive approach. Graph grammars provide such a constructive approach and are a useful tool to design visual languages (Bardohl et al., 1999).

Our approach is shown in Figure 7.1. We transform the type graph into a graph grammar as in (Taentzer, 2012), and the OCL constraints into graph conditions. The constraints are then integrated into the graph grammar as application conditions. This grammar can then be used to generate instances of the meta-model that adhere to all the constraints given in the type graph and the OCL constraints.

The chapter begins with an overview on the OCL and its semantics in Chapter 7.1. We then introduce conditions over typed, attributed graphs with inheritance suitable for use

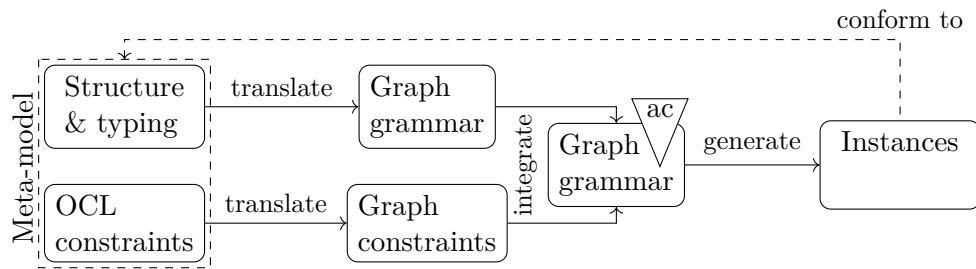


Figure 7.1: Overview of the approach for instance generation

in the context of OCL constraints in Chapter 7.2. Chapter 7.3 contains the transformation of a large part of Essential OCL into graph conditions. Special consideration for the OCL `iterate` operation is taken in Chapter 7.4. Chapter 7.5 explains the integration of the constraints into the grammar.

7.1 An overview of the Object Constraint Language

This subchapter gives a brief introduction and an overview of the syntax and semantics of the Object Constraint Language (OCL). Throughout this chapter, we use the version 2.2 of the OCL standard as specified in (Object Management Group, 2010). The syntax of OCL is covered in detail in chapters 7 and 8 of (Object Management Group, 2010). The semantics of OCL used here, though, is the one from (Richters, 2002), as it is more precise and less error-prone than the semantics presented in (Object Management Group, 2010). See (Brucker and Wolff, 2012) for a discussion of inconsistencies and contradictions in OCL 2.3.

OCL is designed as a pure specification language and thus free of side-effects. The part of OCL that this thesis aims to translate is Essential OCL as defined in Chapter 13 of (Object Management Group, 2010).

Types and operations

OCL is a typed language, so every OCL expression has a type. A set of OCL constraints is tied to a specific meta-model, usually specified by a UML class diagram. The constraints use the types of the meta-model and supplement the constraints given by the diagram.

Definition 7.1 (Object Model). Let $DSIG = (S, OP)$ be a *data signature* with $S = \{Int, Real, Bool, String\}$ and corresponding operation symbols OP . An *object model* over $DSIG$ is a structure $M = (CLS, ACLS, ENUM, attr, assoc, \prec)$ with

- a finite set CLS of classes with a subset $ACLS \subseteq CLS$ of abstract classes,
- a finite set $ENUM$ of enumerations, where each enumeration $E \in ENUM$ is a non-empty, finite set of literals,

- a function $attr : (CLS \times String) \rightarrow (S \cup ENUM)$ mapping from a class and an attribute name to an attribute value,
- a function $assoc : (CLS \times CLS) \rightarrow (String \times \mathbb{N} \times \mathbb{N})$ is a function $assoc(s, t) = (name, min, max)$ signifying an association named $name$ from s to t with multiplicity between min and max ¹, and
- a partial order \prec on CLS reflecting the generalization hierarchy.

△

Example meta-model. As a running example, we use a meta-model for Petri nets. Figure 7.2 shows a possible model for Petri nets, in the usual class diagram notation (Object Management Group, 2003). A Petri net consists of places and transitions, each of which have a name. Places are connected to Transitions (and vice versa) via weighted arcs. Places also have a number of tokens, represented as a class for easy extensibility.

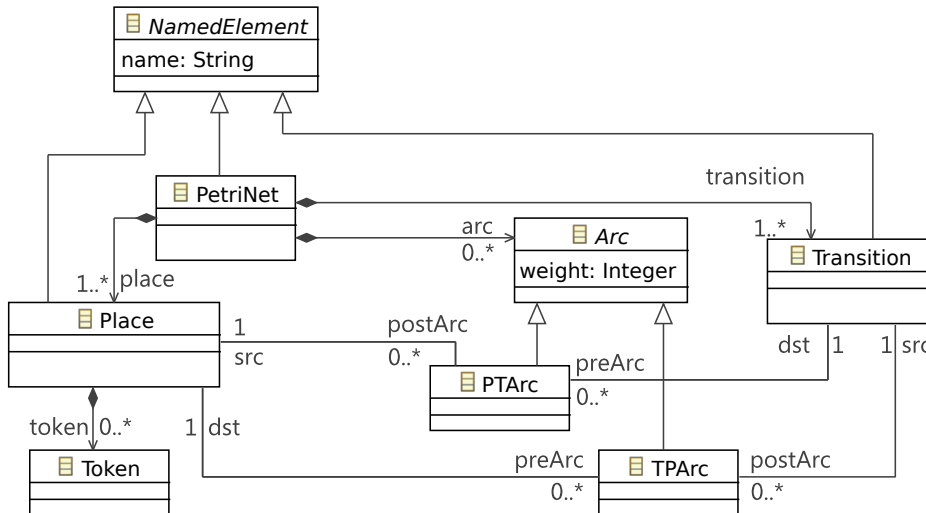


Figure 7.2: Meta-model for Petri nets (adapted from (Wachsmuth, 2007))

OCL invariants. The OCL can be used for a number of different purposes. All OCL expressions have some kind of *context* that specifies the scope for which the expression should be valid. In this work, we are primarily concerned with the purpose of formulating *invariants* of an object model. An OCL invariant has the form

context <Context> inv: <Formula>

¹ $min = max = 0$ signifies there is no association from s to t

Types in OCL. The type hierarchy of OCL consists of a few built-in types together with the types of the underlying model. OCL has the built-in basic types `Boolean` (also called `Bool` here), `Integer` (also called `Int` here), `Real` and `String`. For each of these types, the usual operations and infix operators are defined. OCL also has the type `Collection` and its subtypes `Set`, `OrderedSet`, `Bag` and `Sequence`, as well as named `Tuples`.

Properties and Navigation. One of the most important operations in OCL expressions is the navigation through the hierarchy of classes and their respective properties, i.e. attributes, methods and associations. Navigation to a property B of an object A is specified by the OCL expression $A.B$, and operations on the built-in OCL types (in most cases, collections) are denoted by $A \rightarrow B$. The special keyword `self` stands for an object of the context type of the expression. As an example, the OCL expression `context Place inv: self.token->size() < 3` specifies that, for any object of type `Place`, the number of tokens (i.e. associations of the kind `token`) should be less than three.

Formal semantics of OCL operations

In order to define and prove a formal transformation from Essential OCL to HR* conditions, we need a detailed semantics for Essential OCL. The following definitions follow those from (Richters, 2002).

The evaluation of an OCL constraint on an instance (also called *object model*) M is dependent on its *system state* at a given point in time. The system state consists of a set of class objects, functions assigning the attribute values to each object and links connecting the objects.

Definition 7.2 (system state). The *system state* of an object model M is a structure $\mathcal{S}(M) = (\mathcal{S}_{Cls}, \mathcal{S}_{Att}, \mathcal{S}_{Assoc})$, where

- \mathcal{S}_{Cls} is a function assigning a finite set of object identifiers $\{o_1 : C, o_2 : C, \dots\}$ to each class C . Note that $\mathcal{S}_{Cls}(C) = \emptyset$ for all abstract classes C .
- \mathcal{S}_{Att} is a function assigning a value to each owned and inherited attribute $attr$ of an object in \mathcal{S}_{Cls} .
- \mathcal{S}_{Assoc} is a finite set of directed links (o_1, o_2) between objects that respect the constraints on links and their multiplicities of the meta-model. For a pair of classes (C_1, C_2) , let $\mathcal{S}_{Assoc}((C_1, C_2)) \subset \mathcal{S}_{Assoc}$ be the set of all links from objects of type C_1 to objects of type C_2 .

△

We can now define the semantics for basic types and simple operations.

Definition 7.3 (Semantics of a signature). Let $\Sigma_M = (T_M, \leq_M, \Omega_M)$ be a signature over an object model M . The *semantics of M* is a structure $I(\Sigma_M) = (I(T_M), I(\leq_M), I(\Omega_M))$ where

7.1 An overview of the Object Constraint Language

- $I(T_M)$ assigns to each type $T \in T_M$ an interpretation $I(T)$, e.g., $I(\text{Real}) = \mathbb{R}$, $I(T) = \bigcup_{T \leq T'} \mathcal{S}_{Cls}(T')$, $I(\text{Set}(T)) = 2^{I(T)}$ where $2^{I(T)}$ is the set of all finite subsets of $I(T)$,
- $I(\leq_M)$ implies for all types $T, T' \in T_M$ that $I(T) \subset I(T')$ if $T \leq_M T'$.
- $I(\Omega_M)$ assigns to each operation $\omega : t_1 \times \dots \times t_n \rightarrow t \in \Omega_M$ a total function $I(\omega) = I(t_1) \times \dots \times I(t_n) \rightarrow I(t)$, e.g., $I(42) = 42$, $I(+_{Int})(i, j) = i + j$ for integers i and j , and $I(=_t)(v_1, v_2) = (v_1 = v_2)$ with values $v_1, v_2 \in I(t)$.

△

Definition 7.4 (semantics of Essential OCL expressions). Let T_M be the set of types of a given object model M . Let $Env = \{\tau \mid \tau = (\mathcal{S}, \beta)\}$ be a set of environments with system states \mathcal{S} and variable assignments β which map variable names to values of the corresponding type $I(T)$. The *semantics of an Essential OCL expression* e of type T is a function $I[\![e]\!]: Env \rightarrow I(T)$ and is defined inductively as follows for each $\tau = (\mathcal{S}, \beta) \in Env$.

Let $\tau\{v/x\}$ denote the substitution of all occurrences of v in τ by x .

- For a variable v , let $I[\![v]\!](\tau) := \beta(v)$.
- For an operation $\text{op}(e_1, \dots, e_n)$, let $I[\![\text{op}(e_1, \dots, e_n)]\!](\tau) := I(\text{op})(\tau)(I[\![e_1]\!](\tau), \dots, I[\![e_n]\!](\tau))$. The syntax and semantics of concrete operations in Essential OCL are listed in Tables 7.1, 7.2 and 7.3².
- For an expression **if** e_1 **then** e_2 **else** e_3 , let $I[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!](\tau) := I[\![e_2]\!](\tau)$ if $I[\![e_1]\!](\tau)$ and $I[\![e_3]\!](\tau)$ otherwise.
- For an expression **let** $v=e$ **in** e' , $I[\![\text{let } v=e \text{ in } e']\!](\tau) := I[\![e']\!](\mathcal{S}, \beta\{v/I[\![e]\!](\tau)\})$.

△

Remark. An invariant context $v:C \text{ inv:expr}$ can be expressed by the OCL constraint $C.\text{allInstances} \rightarrow \text{forall}(v|\text{expr})$. Therefore, the semantics of this invariant is equal to the semantics of the corresponding Essential OCL expression.

²For primitive types we present selected operations only.

Table 7.1: OCL operations on single nodes and attributes.

Syntax $e, e_1, e_2 \in Expr$	Semantics $I[[e]](\tau)$ with $\tau = (\mathcal{S}, \beta) \in Env$
$e_1:T = e_2:T \rightarrow Bool$	$I[[e_1]](\tau) = I[[e_2]](\tau)$
$e_1:T \lt;> e_2:T \rightarrow Bool$	$I[[e_1]](\tau) \neq I[[e_2]](\tau)$
$e_1 + e_2 \rightarrow Int$	$I[[e_1]](\tau) + I[[e_2]](\tau)$
$e_1 \leq e_2 \rightarrow Bool$	$I[[e_1]](\tau) \leq I[[e_2]](\tau)$
$e_1 \text{ and } e_2 \rightarrow Bool$	$I[[e_1]](\tau) \wedge I[[e_2]](\tau)$
'a string' $\rightarrow String$	'a string'
$e:C.allInstances() \rightarrow Set(C)$	$\mathcal{S}_{Cls}(C)$
$e:C.attr \rightarrow T$ with $attr \in \mathcal{S}_{Att}(C)$	$\mathcal{S}_{Att}(attr)(I[[e_c]](\tau))$
$e:C.nav \rightarrow C'$ with $(e, nav) \in \mathcal{S}_{Assoc}$	nav with $(I[[e]](\tau), nav) \in \mathcal{S}_{Assoc}$
$e:C.nav \rightarrow Set(C')$	$\{nav \mid (I[[e]](\tau), nav) \in \mathcal{S}_{Assoc}\}$
$e:T.oclIsTypeOf(T')$	$I[[e]](\tau) \in I(T') - \bigcup_{T'' \leq_M T'} I(T'')$
$e:T.oclIsKindOf(T')$	$I[[e]](\tau) \in I(T')$
$e:T.oclAsType(T')$	$I[[e]](\tau)$ if $I[[e]](\tau) \in I(T')$ and \emptyset otherwise.

7.1 An overview of the Object Constraint Language

Table 7.2: OCL operations on sets, yielding sets.

Syntax	Semantics
$e, e' \in Expr, S = \text{Set}(T)$	$I[e](\tau)$ with $\tau = (S, \beta) \in Env$
$Set(e_1, \dots, e_n) \rightarrow S$ with e_1, \dots, e_n of type T	$\{I[e_1](\tau), \dots, I[e_n](\tau)\}$
$e:S \rightarrow \text{union}(e':S) \rightarrow S$	$I[e](\tau) \cup I[e'](\tau)$
$e:S \rightarrow \text{intersection}(e':S) \rightarrow S$	$I[e](\tau) \cap I[e'](\tau)$
$e:S - e':S \rightarrow S$	$I[e](\tau) - I[e'](\tau)$
$e:S \rightarrow \text{symmetricDifference}(e':S) \rightarrow S$	$(I[e](\tau) \cup I[e'](\tau)) - (I[e](\tau) \cap I[e'](\tau))$
$e:S \rightarrow \text{including}(e':T) \rightarrow S$	$I[e](\tau) \cup \{I[e'](\tau)\}$
$e:S \rightarrow \text{excluding}(e':T) \rightarrow S$	$I[e](\tau) - \{I[e'](\tau)\}$
$e:S \rightarrow \text{select}(v:T e':\text{Bool}) \rightarrow S$	$\{x \mid x \in I[e](\tau) \wedge I[e'](\tau\{v/x\})\}$
$e:S \rightarrow \text{reject}(v:T e':\text{Bool}) \rightarrow S$	$\{x \mid x \in I[e](\tau) \wedge \neg I[e'](\tau\{v/x\})\}$
$e:S \rightarrow \text{collect}(v:T e':T') \rightarrow \text{Set}(T')$	$\{I[e'](\tau\{v/x\}) \mid x \in I[e](\tau)\}$
$e \rightarrow \text{iterate}(v:T; a=e' e'') \rightarrow S$ with $e:S$ and $e', e'':T'$	$I[e''](\tau\{a/a'\}\{v/x_n\})$ with $a' =$ $I[\{x_1, \dots, x_{n-1}\} \rightarrow \text{iterate}(v; a=e' e'')]$ if $I[e](\tau) = \{x_1, \dots, x_n\}$ and $I[e'](\tau)$ else.

Table 7.3: Further OCL operations on sets.

Syntax $e, e' \in Expr, S = \text{Set}(T)$	Semantics $I[\mathbf{e}](\tau)$ with $\tau = (S, \beta) \in Env$
$e:S \rightarrow \text{size}() \rightarrow \text{Int}$	$ I[\mathbf{e}](\tau) $
$e:S \rightarrow \text{one}(v:T e':\text{Bool}) \rightarrow \text{Bool}$	true if $ I[e \rightarrow \text{select}(v:T e')](\tau) = 1$ false otherwise
$e:S \rightarrow \text{isEmpty}() \rightarrow \text{Bool}$	$I[\mathbf{e}](\tau) = \emptyset$
$e:S \rightarrow \text{notEmpty}() \rightarrow \text{Bool}$	$I[\mathbf{e}](\tau) \neq \emptyset$
$e:S \rightarrow \text{exists}(v:T e':\text{Bool}) \rightarrow \text{Bool}$	$\bigvee_{1 \leq i \leq n} I[e'](\tau\{v/x_i\})$ if $I[\mathbf{e}](\tau) = \{x_1, \dots, x_n\}$, else false
$e:S \rightarrow \text{forall}(v:T e':\text{Bool}) \rightarrow \text{Bool}$	$\bigwedge_{1 \leq i \leq n} I[e'](\tau\{v/x_i\})$ if $I[\mathbf{e}](\tau) = \{x_1, \dots, x_n\}$, else true
$e:S \rightarrow \text{any}(v:T e':\text{Bool}) \rightarrow T$	x if $x \in I[\mathbf{e}](\tau) \wedge I[e'](\tau\{v/x\})$ and false otherwise
$e:S \rightarrow \text{includes}(e':T) \rightarrow \text{Bool}$	$I[\mathbf{e}'](\tau) \in I[\mathbf{e}](\tau)$
$e:S \rightarrow \text{excludes}(e':T) \rightarrow \text{Bool}$	$I[\mathbf{e}'](\tau) \notin I[\mathbf{e}](\tau)$
$e:S \rightarrow \text{includesAll}(e':S) \rightarrow \text{Bool}$	$I[\mathbf{e}'](\tau) \subseteq I[\mathbf{e}](\tau)$
$e:S \rightarrow \text{excludesAll}(e':S) \rightarrow \text{Bool}$	$I[\mathbf{e}](\tau) \cap I[\mathbf{e}'](\tau) = \emptyset$

7.2 Graphs and conditions for OCL

Representing meta-models and OCL constraints with graph conditions puts some special requirements on the conditions.

Typing and inheritance. Types and their relation to each other are a core concept of OCL. Thus, graph conditions also need to support typed nodes, including inheritance.

Attributes. Most objects will have one or more attributes, and OCL constraints often reference attributes. Therefore, it is essential that our conditions offer a way to reference attributes, too, and to compare them to other attributes or constants.

Notation. Nodes have to be annotated with their respective type and attributes. It is also useful to give names to the nodes. A node named u of type PTArc , with the attribute *weight* having a value of 2, is represented as $\boxed{\begin{array}{l} u:\text{PTArc} \\ \text{weight} = 2 \end{array}}$. Edges are annotated with a label representing the role name in the meta-model. Two edges in opposite directions between the same nodes are written as a bidirectional edge; to discern the two labels, they are positioned near the respective source node of the edge: Two nodes u and v of type \mathbb{T} , with an a -edge from u to v and a b -edge from v to u , are written as $\boxed{u:\mathbb{T}} \xrightarrow{a} \boxed{v:\mathbb{T}} \xrightarrow{b} \boxed{u:\mathbb{T}}$.

Compact conditions. Graph conditions have the tendency to become rather large and cumbersome when the nesting gets deeper. Take, for example, the following property: “There is a node $\boxed{u:\mathbb{T}}$, such that for any other node $\boxed{v:\mathbb{T}}$, there is an outgoing edge from $\boxed{v:\mathbb{T}}$ to some other node $\boxed{w:\mathbb{T}}$ ”. Note that $\boxed{u:\mathbb{T}}$ might be identified with $\boxed{w:\mathbb{T}}$. As a nested condition, this can be formulated as

$$\exists(\boxed{u:\mathbb{T}}, \forall(\boxed{u:\mathbb{T}} \boxed{v:\mathbb{T}}, \exists(\boxed{u:\mathbb{T}} \boxed{v:\mathbb{T}} \boxed{w:\mathbb{T}}, \exists(\boxed{u:\mathbb{T}} \boxed{v:\mathbb{T}} \rightarrow \boxed{w:\mathbb{T}})) \vee \exists(\boxed{v:\mathbb{T}} \rightarrow \boxed{u:\mathbb{T}}))$$

The *compact conditions* we propose in this subchapter shorten the above to

$$\exists(\boxed{u:\mathbb{T}}, \forall(\boxed{v:\mathbb{T}}, \exists(\boxed{v:\mathbb{T}} \rightarrow \boxed{w:\mathbb{T}}))).$$

Typed graphs with inheritance and attributes

The definition of attributed graphs used here is loosely based on the concept given in (H. Ehrig, K. Ehrig, Prange, et al., 2006). This concept is enhanced by allowing variables for attribute values and formulas over these variables, similar to the symbolic graphs in (Orejas, 2011). For an alternative concept for attributed graphs with inheritance, see e.g. (Löwe et al., 2013). The definitions use some basic concepts of algebraic specifications. For further lecture on this topic, consider e.g. (H. Ehrig, K. Ehrig, Prange, et al., 2006) or (H. Ehrig and Mahr, 1985; Loeckx et al., 1996).

We first define *A-graphs*, graphs equipped with node attributes. Attributes are represented by an attribute edge from the graph node to a data node.

Definition 7.5 (A-graphs). An *A-graph* is a tuple $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ consisting of sets G_V and G_D , called graph and data nodes (or vertices), respectively, G_E and G_A , called graph and node attribute edges, respectively, source and target functions: $src_G: G_E \rightarrow G_V, tgt_G: G_E \rightarrow G_V$ for graph edges and $src_A: G_A \rightarrow G_V, tgt_A: G_A \rightarrow G_D$ for node attribute edges (going from a graph to a data node). Given two A-graphs G^1 and G^2 , an *A-graph morphism* $f: G^1 \rightarrow G^2$ is a tuple of functions $f_V: G_V^1 \rightarrow G_V^2, f_D: G_D^1 \rightarrow G_D^2, f_E: G_E^1 \rightarrow G_E^2$ and $f_A: G_A^1 \rightarrow G_A^2$ such that f commutes with all source and target functions, e.g. $f_V \circ src_G^1 = src_G^2 \circ f_E$. An A-graph morphism f is *injective* (an *inclusion*) if the functions f_V, f_D, f_E , and f_A are injective.

$$\begin{array}{ccccccc}
 G_E^1 & \xrightarrow{src_G^1(tgt_G^1)} & G_V^1 & \xleftarrow{src_A^1} & G_A^1 & \xrightarrow{tgt_A^1} & G_D^1 \\
 \vdots \scriptstyle f_E & & \vdots \scriptstyle f_V & = & \vdots \scriptstyle f_A & = & \vdots \scriptstyle f_D \\
 G_E^2 & \xrightarrow{src_G^2(tgt_G^2)} & G_V^2 & \xleftarrow{src_A^2} & G_A^2 & \xrightarrow{tgt_A^2} & G_D^2
 \end{array}$$

△

The A-graphs are now enhanced with formulas over the attributes. Let $DSIG = (S, OP)$ be a data signature, $X = \{X_s\}_{s \in S}$ a family of variables, and $T_{DSIG}(X)$ the term algebra with respect to $DSIG$ and X , i.e. .

Definition 7.6 (attributed graphs). An *attributed graph* over $DSIG$ and X is a tuple $AG = (G, D, \Phi)$ where G is an A-graph, D is a $DSIG$ -algebra with $\sum_{s \in S} D_s = G_D$, and Φ is a finite set (i.e. a conjunction) of Boolean formulas over $T_{DSIG}(X)$. An attributed graph $AG = (G, D, \emptyset)$ with an empty set of formulas is *basic* and is shortly denoted by $AG = (G, D)$.

For attributed graphs AG^1 and AG^2 , an *attributed morphism* $f: AG^1 \rightarrow AG^2$ is a pair $f = (f_G, f_D)$ of an A-graph morphism $f_G: G^1 \rightarrow G^2$ and a $DSIG$ -homomorphism $f_D: D^1 \rightarrow D^2$ such that (1) commutes for all $s \in S$, $f_{G, G_D} = \sum_{s \in S} f_{D, s}$, and $\Phi^2 \Rightarrow f(\Phi^1)$ where $f(\Phi^1)$ is the set of formulas obtained when replacing in Φ^1 every variable x in G^1 by $f(x)$. An attributed morphism f is *injective* (an *inclusion*) if f_G and f_D are injective (inclusions).

$$\begin{array}{ccc}
 G_D^1 & \longleftarrow & D_s^1 \\
 f_{G, G_D} \downarrow & (1) & \downarrow f_{D, s} \\
 G_D^2 & \longleftarrow & D_s^2
 \end{array}$$

△

Example 7.1. For the Petri net model in Figure 7.2, the attributed graph

$u:PTArc$
$weight \geq 1$

 consists of an A-Graph with a single node u , a $DSIG$ -algebra for natural numbers and the formula set $\Phi = \{weight \geq 1\}$. ◇

Our attributed graphs correspond to the basic attributed graphs in Chapter 8 of (H. Ehrig, K. Ehrig, Prange, et al., 2006). The results from (H. Ehrig, K. Ehrig, Prange, et al., 2006) for basic attributed graphs can be generalized to our attributed graphs:

Fact 7.1 (properties of attributed graphs).

1. Attributed graphs and attributed morphisms form the category AGraphs.
2. The category has pushouts and \mathcal{E}' - \mathcal{M} pair factorization in the sense of (H. Ehrig, K. Ehrig, Prange, et al., 2006) with \mathcal{M} being the class of all injective morphisms $m : A \rightarrow B$ with $\Phi_B \Leftrightarrow m(\Phi_A)$ and \mathcal{E}' the class of all jointly surjective morphism pairs.

Proof. The proof follows more or less from (H. Ehrig, K. Ehrig, Prange, et al., 2006). The first statement is straightforward; the second one follows from the one in (H. Ehrig, K. Ehrig, Prange, et al., 2006): Let $r : K \rightarrow R$ and $d : K \rightarrow D$ be attributed morphisms on basic attributed graphs and Φ_K, Φ_R, Φ_D be the corresponding sets of formulas. By (H. Ehrig, K. Ehrig, Prange, et al., 2006), there are a basic attributed graph H and basic attributed morphisms $r' : R \rightarrow H$ and $h : D \rightarrow H$ such that the square (1) in the diagram below is a pushout. Let Φ_H be equivalent to $r'(\Phi_D) \cup h(\Phi_R)$. Then $\Phi_H \Rightarrow r'(\Phi_D)$ and $\Phi_H \Rightarrow h(\Phi_R)$, i.e. r' and h are attributed morphisms.

$$\begin{array}{ccc} K & \xrightarrow{r} & R \\ d \downarrow & (1) & \downarrow r' \\ D & \xrightarrow{h} & H \end{array}$$

For \mathcal{E}' - \mathcal{M} pair factorization, we perform the construction suggested by (H. Ehrig, K. Ehrig, Prange, et al., 2006, Remark 5.26): For morphisms $f_1 : A_1 \rightarrow C$ and $f_2 : A_2 \rightarrow C$, compute the coproduct injections $\iota_1 : A_1 \rightarrow A_1 + A_2$ and $\iota_2 : A_2 \rightarrow A_1 + A_2$ component-wise on the node, edge and attribute sets and $\Phi_{A_1+A_2} = \Phi_{A_1} \cup \Phi_{A_2}$ (see diagram below). Define $f : A_1 + A_2 \rightarrow C$ as $f(x) = \iota_j(x)$ for $x \in \iota_j$ and compute $K, e : A_1 + A_2 \rightarrow K$ and $m : K \rightarrow C$ as epi-mono factorization $f = m \circ e$, i.e. $K = f(A_1 + A_2)$, $\Phi_K = \Phi_C$ and m an injection. Finally, $(e_1, e_2) = (e \circ \iota_1, e \circ \iota_2)$.

$$\begin{array}{c} A_1 \xrightarrow{\iota_1} A_1 + A_2 \xleftarrow{\iota_2} A_2 \\ \begin{array}{ccc} e_1 \searrow & e \downarrow & e_2 \swarrow \\ & K & \\ f_1 \searrow & m \downarrow & f_2 \swarrow \\ & C & \end{array} \end{array}$$

□

The attributed graphs are now enhanced with types and inheritance.

Definition 7.7 (attributed type graph with inheritance, ATGI-graph). An *attributed type graph with inheritance* $\text{ATGI} = (TG, Z, I)$ consists of an A-graph TG , a final DSIG-algebra Z , and a simple³ inheritance graph I with $I_V = TG_V$. For each node $v \in I_V$, the *inheritance clan*, $\text{clan}_I(v)$, is the set of all nodes $v' \in I_V$ with an outgoing path to v of arbitrary length.

A typed attributed graph (AG, type) over ATGI , short *ATGI-graph*, consists of an attributed graph $AG = (G, D, \Phi)$ and a *clan morphism*, $\text{type}: AG \rightarrow \text{ATGI}$.

A clan morphism type consists of typing functions $\text{type}_V: G_V \rightarrow TG_V$, $\text{type}_D: G_D \rightarrow TG_D$ for nodes, $\text{type}_E: G_E \rightarrow TG_E$, $\text{type}_A: G_A \rightarrow TG_A$ for edges, and the unique final DSIG-homomorphism $\text{type}_{\text{DSIG}}: D \rightarrow Z$ such that $\text{type}_V \circ \text{src}_{GE} \preceq \text{src}_{TGE} \circ \text{type}_E$ ⁴, $\text{type}_V \circ \text{tgt}_{GE} \preceq \text{tgt}_{TGE} \circ \text{type}_E$, $\text{type}_V \circ \text{src}_{GA} \preceq \text{src}_{TGA} \circ \text{type}_A$, $\text{type}_D \circ \text{tgt}_{GA} = \text{tgt}_{TGA} \circ \text{type}_A$ and $\text{type}_{\text{DSIG},s} = \text{type}_{D|D_s}$ for all $s \in S$.

$$\begin{array}{ccccc}
 G_E & \xrightarrow{\text{src}_G(\text{tgt}_G)} & G_V & \xleftarrow{\text{src}_A} & G_A & \xrightarrow{\text{tgt}_A} & G_D \\
 \text{type}_E \downarrow & & \preceq \text{type}_V \downarrow & & \succeq & & \text{type}_A = \downarrow & \text{type}_D \downarrow \\
 TG_E & \xrightarrow{\text{src}_{TGE}(\text{tgt}_{TGE})} & TG_V & \xleftarrow{\text{src}_{TGA}} & TG_A & \xrightarrow{\text{tgt}_{TGA}} & TG_D
 \end{array}$$

A clan morphism type is *injective* (an *inclusion*) if type_V , type_E , and $\text{type}_{\text{DSIG}}$ are injective (inclusions).

Given two ATGI-graphs $AG_1 = (G_1, \text{type}_1)$ and $AG_2 = (G_2, \text{type}_2)$, an *ATGI-morphism* $f: AG_1 \rightarrow AG_2$ is an attributed morphism such that $\text{type}_2 \circ f \preceq \text{type}_1$. An ATGI-morphism is called *(type-)strict* if $\text{type}_2 \circ f = \text{type}_1$.

$$\begin{array}{ccc}
 AG_1 & \xrightarrow{f} & AG_2 \\
 \text{type}_1 \searrow & = & \swarrow \text{type}_2 \\
 & \text{ATGI} &
 \end{array}
 \quad \triangle$$

Remark. If I is a discrete graph (i.e. has no edges), there is no inheritance, so ATGI is an *attributed type graph*.

Example 7.2. The typed attributed graph below consists of four nodes with attributes, connected by three edges. It models a Petri net with one place, one transition and an edge from the place to the transition.

³A graph is *simple* if it has neither multiple edges nor loops.

⁴For functions $f: A \rightarrow B, g: A \rightarrow \text{clan}_I(B)$, $f \preceq g$ means $f(x) \in \text{clan}_I(g(x))$ for all $x \in A$ where $\text{clan}_I(B) = \{\text{clan}(v) \mid v \in B\}$.

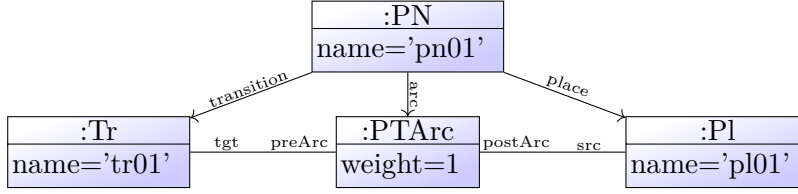


Figure 7.3: A simple attributed graph.

The corresponding type graph is shown in Figure 7.2. \diamond

Fact 7.2 (properties of typed attributed graphs). ATGI-graphs and ATGI-morphisms form the category $\text{AGraphs}_{\text{ATGI}}$. For \mathcal{M} being the class of all type-strict injective morphisms and \mathcal{E}' the class of all jointly surjective pairs, the category has pushouts along \mathcal{M} -morphisms and \mathcal{E}' - \mathcal{M} pair factorization.

Proof. The first statement is straightforward. The other statements follow with the help of Fact 7.1: For an injective morphism $d : K \rightarrow D$ and type-strict injective morphism $r : K \rightarrow R$ with typing-morphisms $\text{type}_K : K \rightarrow \text{ATGI}$, $\text{type}_D : D \rightarrow \text{ATGI}$ and $\text{type}_R : R \rightarrow \text{ATGI}$ compute the pushout (1), ignoring typing, in AGraphs as stated in Fact 7.1 and choose the typing-morphism $\text{type}_H : H \rightarrow \text{ATGI}$ as follows: For $y \in H$ with $x \in D$ and $h(x) = y$ let $\text{type}_H(y) = \text{type}_D(x)$. Let $\text{type}_H(y) = \text{type}_R(z)$ with $z \in R$ and $r'(z) = y$ otherwise. Morphisms h and r' are valid morphisms in $\text{AGraphs}_{\text{ATGI}}$ since for all $x \in K$, $\text{type}_H(h(d(x))) = \text{type}_D(d(x)) \preceq \text{type}_K(x) = \text{type}_R(r(x))$ ⁵ holds. Furthermore, (1) in the diagram below is a pushout in $\text{AGraphs}_{\text{ATGI}}$ since for any pair $(f : D \rightarrow G, g : R \rightarrow G)$ of morphisms with common co-domain G and $f \circ d = g \circ r$ there is a morphism $m : H \rightarrow G$ with $m \circ h = f$, $m \circ r' = g$ in AGraphs (since (1) is pushout in AGraphs). Since for all $y \in H$ there exists $x \in D$ with $h(x) = y$ and $\text{type}_H(x) = \text{type}_D(y) \preceq \text{type}_G(f(x)) = \text{type}_G(m(y))$ or $z \in R$ with $r'(z) = y$ and $\text{type}_R(z) = \text{type}_D(y) \preceq \text{type}_G(g(r)) = \text{type}_G(m(x))$, m is also a morphism in $\text{AGraphs}_{\text{ATGI}}$.

$$\begin{array}{ccc} K & \xrightarrow{r} & R \\ d \downarrow & (1) & \downarrow r' \\ D & \xrightarrow{h} & H \end{array}$$

\mathcal{E}' - \mathcal{M} pair factorization is similar: Let $f_i : A_i \rightarrow C$ ($i \in \{1, 2\}$) with typing $\text{type}_i : A_i \rightarrow \text{ATGI}$ and $\text{type}_C : C \rightarrow \text{ATGI}$. Compute the \mathcal{E}' - \mathcal{M} pair factorization, ignoring typing, in AGraphs as stated in Fact 7.1 and choose the typing-morphism $\text{type}_K : K \rightarrow \text{ATGI}$ such that $\text{type}_K = \text{type}_C \circ m$, i.e. $\text{type}_K(x) = \text{type}_C(m(x))$ for $x \in K$. Morphisms e_1 and e_2 respect typing since $\text{type}_K(e_i(x)) = \text{type}_C(f_i(x)) \preceq \text{type}_i(x)$ for all $x \in A_i$, $i \in \{1, 2\}$ (see diagram below).

⁵We write $x \preceq y$ iff $x \in \text{clan}_I(y)$.

$$\begin{array}{ccccc}
 A_1 & \xrightarrow{e_1} & K & \xleftarrow{e_2} & A_2 \\
 & \searrow & \downarrow m & \swarrow & \\
 & & C & & \\
 & \nearrow f_1 & & \nwarrow f_2 & \\
 & & & &
 \end{array}$$

□

Compact conditions

As explained in the introduction, graph conditions tend to become rather large and cumbersome with deeper nesting. This makes deeply nested graph conditions not only hard to read, but also makes it difficult to use graph conditions in some step-by-step construction which relies in later steps on the existence of some nodes added in earlier steps. To alleviate this, we suggest so-called *compact conditions*, an abbreviated form of nested conditions. We show that compact conditions are syntactic sugar and can be converted into nested conditions.

In the following, the graphs in consideration are equipped with a *name function* $n_G: Names \rightarrow V_G$ assigning a set of names to nodes in the graph. Moreover, morphisms are extended to these graphs as follows: A morphism $f: G \rightarrow H$ respects names if $n_H(a) = f_V(n_G(a))$ for all names a in $\text{Dom}(n_G)$.

Definition 7.8 (compact conditions). A *compact condition* on typed attributed graphs is of the form

1. **true**, or
2. $\exists(C, c)$ where C is a graph and c is a compact condition.
3. Boolean formulas over compact conditions yield compact conditions.

$\exists(C)$ abbreviates $\exists(C, \mathbf{true})$.

△

Example 7.3. $\exists(\boxed{u:T}, \exists(\boxed{v:T}, \exists(\boxed{u:T} \xrightarrow{\text{role}} \boxed{v:T})))$ is a compact condition, intuitively meaning that there exist two nodes u, v of type T and an edge of type **role** between them.

The semantics of compact conditions is defined by the semantics of conditions. For this purpose, we “complete” compact conditions to conditions. The construction yields an injective nested condition with injective satisfaction.

Construction (From compact conditions to nested conditions). For a graph P and a compact condition d , $\text{Uncomp}(P, d)$ denotes the condition over P , inductively defined as follows:

$$\begin{array}{lcl}
 \emptyset \xrightarrow{\quad} C & \triangleleft & \text{Uncomp}(P, \mathbf{true}) = \mathbf{true}. \\
 \vdots & & \\
 P \xrightarrow{a} C' & \triangleleft & \begin{array}{l} \text{Uncomp}(P, \exists(C, c)) = \bigvee_{(a,b) \in \mathcal{F}} \exists(P \xrightarrow{a} C', \text{Uncomp}(C', c)) \\ \text{where } \mathcal{F} = \{(a, b) \mid (a, b) \text{ jointly surjective, } a, b \text{ respect names.}\}. \\ \text{Uncomp}(P, \neg c) = \neg \text{Uncomp}(P, c). \\ \text{Uncomp}(P, c \wedge c') = \text{Uncomp}(P, c) \wedge \text{Uncomp}(P, c'). \end{array}
 \end{array}$$



Remark. The Uncomp and the Shift construction in (H. Ehrig et al., 2014) look very similar. While Shift is based on injective morphisms, Uncomp is restricted to name-respecting morphisms. Uncomp is based on empty morphisms and completes compact conditions $\exists(C, c)$ with empty morphism $\emptyset \rightarrow C$ with respect to a morphism $\emptyset \rightarrow P$ (displayed by dotted lines in the diagram above). Instead of the empty morphisms, we write the codomain of the morphisms.

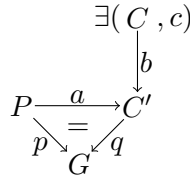
Example 7.4. The property “There is a node $\boxed{u:T}$ such that for any other node $\boxed{v:T}$, there is an outgoing edge from $\boxed{v:T}$ to some other node $\boxed{w:T}$ ” can be formulated as a compact condition $c = \exists(\boxed{u:T}, \forall(\boxed{u:T} \boxed{v:T}, \exists(\boxed{v:T} \rightarrow \boxed{w:T})))$. Using the Uncomp construction yields the nested condition

$$\begin{aligned} \text{Uncomp}(\emptyset, c) &= \\ \exists(\boxed{u:T}, \text{Uncomp}(\boxed{u:T}, \forall(\boxed{u:T} \boxed{v:T}, \exists(\boxed{v:T} \rightarrow \boxed{w:T})))) &= \\ \exists(\boxed{u:T}, \forall(\boxed{u:T} \boxed{v:T}, \text{Uncomp}(\boxed{u:T} \boxed{v:T}, \exists(\boxed{v:T} \rightarrow \boxed{w:T})))) &= \\ \exists(\boxed{u:T}, \forall(\boxed{u:T} \boxed{v:T}, \exists(\boxed{u:T} \boxed{v:T} \boxed{w:T}, \exists(\boxed{u:T} \boxed{v:T} \rightarrow \boxed{w:T})) \vee \exists(\boxed{v:T} \rightarrow \boxed{u=w:T}))) & \end{aligned}$$

with the same meaning. ◇

The semantics of compact conditions can also be defined directly without the Uncomp construction.

Definition 7.9 (alternative semantics of compact conditions). The *satisfaction* of a compact condition c by a morphism $p : P \rightarrow G$, denoted $p \models_{\text{cmp}} c$, is inductively defined as follows:



$p \models_{\text{cmp}} \mathbf{true}$.

$p \models_{\text{cmp}} \exists(C, c)$ iff there exists some C' and morphisms $q : C' \rightarrow G$, $a : P \rightarrow C'$ and $b : C \rightarrow C'$ such that (a, b) are jointly surjective, $q \circ a = b$ and $q \models_{\text{cmp}} c$.

$p \models_{\text{cmp}} \neg c$ iff not $p \models_{\text{cmp}} c$.

$p \models_{\text{cmp}} c \wedge c'$ iff $p \models_{\text{cmp}} c$ and $p \models_{\text{cmp}} c'$. △

The semantics and the alternative semantics of compact conditions are equal.

Fact 7.3 (equality of the semantics). The semantics according to Definition 7.1 and the alternative semantics according to Definition 7.9 are equal, i.e.,

$$p \models_{\text{cmp}} c \iff p \models \text{Uncomp}(\emptyset, c).$$

Proof. For $p : P \rightarrow G$, we show the statement by induction over the structure of compact conditions c : For $c = \mathbf{true}$, the fact is trivially true. For $c = \exists(C, c')$, we have

$p \models \text{Uncomp}(P, \exists(C, c'))$	Def. Uncomp
$\Leftrightarrow p \models \bigvee_{(a,b) \in \mathcal{F}} \exists(a, \text{Uncomp}(C', c'))$	Def. \models
$\Leftrightarrow \exists(a, b) \in \mathcal{F}. p \models \exists(a, \text{Uncomp}(C', c'))$	Def. \models
$\Leftrightarrow \exists(a, b) \in \mathcal{F}. \exists q: C' \rightarrow G. q \circ a = p \text{ and } q \models_{\text{cmp}} c'$	Ind. hyp.
$\Leftrightarrow p \models_{\text{cmp}} \exists(C, c).$	

The remaining cases are straightforward. □

Compact conditions can be further simplified by the following equivalences.

Fact 7.4 (equivalences for compact conditions). Let $C_1 \oplus_P C_2$ denote the pushout of C_1 and C_2 along P and let \mathcal{P} denote the set of all intersections of C_1 and C_2 . Two graphs C_1 and C_2 are *clan-disjoint* if the clans of the types of C_1 and C_2 are disjoint. $C_1 \uplus C_2$ denotes the disjoint union of C_1 and C_2 .

- (E1) a) $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2).$
 b) $\exists(C_1, \exists(C_2)) \equiv \exists(C_1 \uplus C_2)$ if C_1 and C_2 are clan-disjoint.
 c) $\exists(C_1, \exists(C_2)) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1.$
- (E2) a) $\exists(C_1, \exists(C_2) \wedge \exists(C_3)) \equiv \exists(C_1, \bigvee_{P \in \mathcal{P}} \exists(C_2 \oplus_P C_3)),$ if for all node names occurring in both C_2 and C_3 , a node with that name already exists in $C_1.$
 b) $\exists(C_1) \wedge \exists(C_2) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint and have disjoint sets of node names.
- (E3) $\exists(\underline{u:\mathbf{T}}, \exists(C) \wedge \exists(\underline{u=v:\mathbf{T}})) \equiv \exists(\underline{u:\mathbf{T}}, \exists(C[u=v]))$ provided that either u or v does not exist in C and $C[u=v]$ is the graph obtained from C by renaming u by $u=v.$

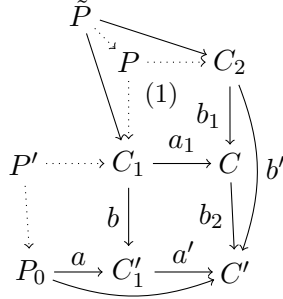
Proof. The proof of the equivalences makes use of the Pullback-Pushout-Lemma in (H. Ehrig and Kreowski, 1979): The pushout of the pullback of a pair $(b_1, b_2) \in \mathcal{F}$ leads to the pushout $C_1 \oplus_P C_2$ of C_1 and C_2 along the pullback $P.$ In the following, \mathcal{P} denotes the set of pairs (a_1, a_2) induced by the pairs $(b_1, b_2) \in \mathcal{F}.$ Let $p: P_0 \rightarrow G.$

$$\begin{array}{ccc}
 P & \xrightarrow{a_2} & C_2 \\
 a_1 \downarrow & (1) & \downarrow b_2 \\
 C_1 & \xrightarrow{b_1} & C
 \end{array}$$

(E1) (a) follows with the help of the definition of Uncomp:

$$\begin{aligned}
 & \exists(C_1, \exists(C_2)) \\
 \equiv & \text{Uncomp}(P_0, \exists(C_1, \exists(C_2))) \\
 \equiv & \bigvee_{(a,b) \in \mathcal{F}} \exists(a, \text{Uncomp}(C'_1, \exists(C_2, \text{true}))) \\
 \equiv & \bigvee_{(a,b) \in \mathcal{F}} \exists(a, \bigvee_{(a',b') \in \mathcal{F}'} \exists(a', \text{Uncomp}(C', \text{true}))) \\
 \equiv & \bigvee_{(a,b) \in \mathcal{F}} \exists(a, \bigvee_{(a',b') \in \mathcal{F}'} \exists(a', \text{true})) \\
 \equiv & \bigvee_{(a,b) \in \mathcal{F}} \bigvee_{(a',b') \in \mathcal{F}'} \exists(a' \circ a) \\
 \equiv & \bigvee_{(a,b) \in \mathcal{F}} \text{Uncomp}(C'_1, \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)) \\
 \equiv & \text{Uncomp}(P_0, \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)) \\
 \equiv & \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2).
 \end{aligned}$$

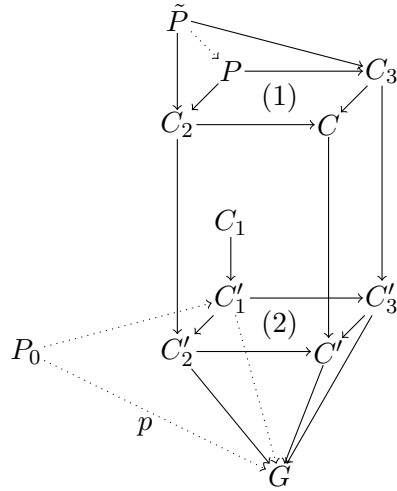
where $\mathcal{F} = \{(a, b)\}$, \mathcal{F}' is the set of pairs $a': C_1 \rightarrow C$, and $b': C_2 \rightarrow C$ such that (a', b') is jointly surjective and a', b' are injective, P is the pullback of (a', b') , and C is the pushout of C_1 and C_2 along P . \tilde{P} is the common part of C_1 and C_2 , i.e. every pair of injective and jointly surjective morphisms (a_1, b_1) such that (1) extended to \tilde{P} commutes. Given the morphisms (a', b') , some C exists due to \mathcal{E}' - \mathcal{M} pair factorization.



(b) If C_1 and C_2 are clan-disjoint, then $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2) \equiv \exists(C_1 + C_2)$ because \mathcal{F} consists of the pair $C_1 \rightarrow C_1 + C_2 \leftarrow C_2$, \mathcal{P} of the pair $C_1 \leftarrow \emptyset \rightarrow C_2$ and $C_1 \oplus_{\emptyset} C_2 = C_1 + C_2$.

(c) If $C_1 \subseteq C_2$, then C_1 is the pullback of C_1 and C_2 and C_2 is the pushout of C_1 and C_2 along C_1 . If $C_2 \subseteq C_1$, then C_2 is the pullback of C_1 and C_2 and C_1 is the pushout of C_1 and C_2 along C_2 . Thus, $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1$.

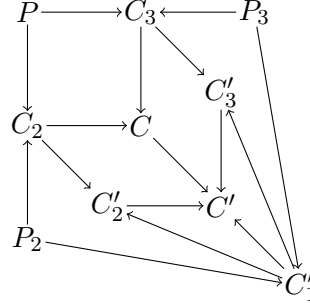
(E2) follow from the definition of Uncomp and \models . We show both directions separately. For “ \Rightarrow ” consider the commutative diagram below.



Assume $p \models \exists(C_1, \exists(C_2) \wedge \exists(C_3))$. By the definition of Uncomp, some C'_1 , C'_2 and C'_3 exist. Let \tilde{P} be the common part of (C_2, C_3) , i.e. in every co-span $C_2 \rightarrow C \leftarrow C_3$ of injective and jointly surjective morphisms such that (1) extended by \tilde{P} commutes, the morphisms are name-respecting. Because all node names that are common in C_2 and C_3 are also contained in C_1 , C'_1 is the common part of C'_2 and C'_3 . By \mathcal{E}' - \mathcal{M} pair factorization (consider (1)), some C' exists with $C' \rightarrow G$ injective. By \mathcal{E}' - \mathcal{M} pair factorization again

(consider (2) extended by \tilde{P}), some C exists with $C \rightarrow C'$ name-respecting. By definition of Uncomp, $p \models \exists(C_1, \bigvee_{P \in \mathcal{P}} \exists(C_2 \oplus_P C_3))$.

For the proof's other direction consider the commutative diagram



By definition of Uncomp, some $P \in \mathcal{P}$, C , C' and C'_1 with $C' \rightarrow G$ exist. Let P_2 and P_3 be the common part of C'_1 and C_2 , C_3 respectively. By \mathcal{E}' - \mathcal{M} pair factorization, C'_2 and C'_3 also exist and with the definition of \models , $p \models \exists(C_1, \exists(C_2) \wedge \exists(C_3))$.

In the case of clan-disjointness of C_1 and C_2 , $\exists(C_1) \wedge \exists(C_2) \equiv \exists(\emptyset, \exists(C_1 \wedge \exists(C_2))) \equiv \exists(\emptyset, \bigvee_{P \in \mathcal{P}} \exists(C_1 \oplus_P C_2)) \equiv \exists(\emptyset, \exists(C_1 + C_2)) \equiv \exists(C_1 + C_2)$ because \mathcal{F} consists of the pair $C_1 \rightarrow C_1 + C_2 \leftarrow C_2$, \mathcal{P} of the pair $C_1 \leftarrow \emptyset \rightarrow C_2$, and $C_1 \oplus_{\emptyset} C_2 = C_1 + C_2$.

(E3) is a special case of (E2)(a) since $C[u=v]^6 = C \oplus_P \underline{\mathbf{u}=\mathbf{v}}$. □

7.3 Translating Essential OCL to graph conditions

This chapter of the thesis gives the translation tr from Essential OCL constraints to graph constraints. Note that the translation does not support all of Essential OCL. A thorough overview of the limitations will be given in Chapter 7.4. The translation itself is quite large, since it has to be defined for the many different OCL operators. Before heading right into the translation, an explanation should help to understand the overall idea.

The goal of this chapter is the following theorem, stating that for any OCL invariant that holds for a system state, the corresponding graph constraint is fulfilled by the corresponding graph.

Theorem 7.1 (Correct Translation of Essential OCL invariants).

Given an object model M and its corresponding attributed type graph, for all Essential OCL invariants \mathbf{inv} and all environments (\mathcal{S}, β) ,

$$I[\mathbf{inv}](\mathcal{S}, \beta) = \mathbf{true} \text{ iff } G = \mathit{corr}_{State}(\mathcal{S}) \models \mathit{tr}_I(\mathbf{inv}).$$

Note that β only occurs on the left-hand side; since \mathbf{inv} does not contain free variables, $\text{Dom}(\beta) = \emptyset$, and the environment β is irrelevant.

⁶ $C[u = v]$ is the graph C with the nodes named u and v identified.

7.3 Translating Essential OCL to graph conditions

- The translation proceeds along the abstract syntax tree of the OCL constraint. For example, given $a \rightarrow \text{union}(b) \rightarrow \text{notEmpty}()$, we first translate notEmpty , followed by union and then its arguments a and b .
- The set operations are translated with the characteristic function in mind, e.g., the characteristic function of $a \rightarrow \text{union}(b)$ is the disjunction of the characteristic functions of a and b : $v \in A \cup B$ iff $v \in A \vee v \in B$; node v here serves as a representative of the set. Navigation expressions, which yield a single object, are treated like single-element sets.
- When translating an OCL operation which yields a set of objects (translation tr_S), we pass a single node as an extra parameter serving as representative of the set: $tr_S(a \rightarrow \text{union}(b), \boxed{v:T}) := tr_S(a, \boxed{v:T}) \vee tr_S(b, \boxed{v:T})$.

As an introductory example, regard OCL expressions of the form $a \rightarrow \text{exists}(v:T \mid b)$ as part of an invariant. We start at the outermost part, that is $\text{exists}(v:T \mid b)$. This is translated in a first step to $\exists(\boxed{v:T}, tr_E(b))$, where tr_E denotes the translation of a Boolean expression and depends solely on b . Now we have to formalize that $\boxed{v:T}$ comes from the set described by a . This is done by giving a predicate $tr_S(a, \boxed{v:T})$ that describes the set precisely. Because we need the predicate over $\boxed{v:T}$, we pass $\boxed{v:T}$ as a parameter to tr_S . So the translation of the whole expression $a \rightarrow \text{exists}(v:T \mid b)$ becomes $\exists(\boxed{v:T}, tr_E(b) \wedge tr_S(a, \boxed{v:T}))$, because $\boxed{v:T}$ has to fulfill both $tr_E(b)$ and $tr_S(\boxed{v:T}, a)$.

Definition 7.10 (constraint translation). Let DSIG be a data signature and $M = (CLS, ACLS, ENUM, attr, assoc, \prec)$ be an object model over DSIG. Let $t : Expr \rightarrow T$ be a typing function which returns the type of an OCL expression.

The *translation functions*

- tr_I for the translation of invariants,
- tr_E for OCL expressions yielding *Bool*,
- tr_N for navigation expressions yielding a single object, and
- tr_S for set expressions yielding a set of objects

are defined as follows:

Let $expr$, $expr1$ and $expr2$ be OCL expressions, u, v, v' names of nodes (i.e. variables), $T = t(v)$ denote the type of v and likewise $T' = t(v')$, $attr1$ and $attr2$ be attribute names, $op \in \{<, >, \leq, \geq, =, <>\}$ a comparison operator, and $role$ be a role of a class. Then

1. a) $tr_I(\text{context } C \text{ inv: } expr) := \forall(\boxed{\text{self}:C}, tr_E(expr))$
 b) $tr_I(\text{context } var:C \text{ inv: } expr) := \forall(\boxed{var:C}, tr_E(expr))$
2. a) $tr_E(\text{true}) := \text{true}$

- b) $tr_E(\text{not expr}) := \neg tr_E(\text{expr})$
 - c) $tr_E(\text{expr1 and expr2}) := tr_E(\text{expr1}) \wedge tr_E(\text{expr2})$
 - d) $tr_E(\text{expr1 or expr2}) := tr_E(\text{expr1}) \vee tr_E(\text{expr2})$
 - e) $tr_E(\text{expr1 implies expr2}) := \neg tr_E(\text{expr1}) \vee tr_E(\text{expr2})$
 - f) $tr_E(\text{if cond then expr1 else expr2}) :=$
 $((tr_E(\text{cond}) \wedge tr_E(\text{expr1})) \vee (\neg tr_E(\text{cond}) \wedge tr_E(\text{expr2})))$
3. a) $tr_E(\text{expr1} \rightarrow \text{exists}(v:T \mid \text{expr2})) :=$
 $\exists(\underline{v:T}, tr_S(\text{expr1}, \underline{v:T}) \wedge tr_E(\text{expr2}))$
- b) $tr_E(\text{expr1} \rightarrow \text{forall}(v:T \mid \text{expr2})) :=$
 $\forall(\underline{v:T}, tr_S(\text{expr1}, \underline{v:T}) \Rightarrow tr_E(\text{expr2}))$ ⁷
4. a) $tr_E(\text{expr1} \rightarrow \text{includesAll}(\text{expr2})) :=$
 $\forall(\underline{v:T}, tr_S(\text{expr2}, \underline{v:T}) \Rightarrow tr_S(\text{expr1}, \underline{v:T}))$
- b) $tr_E(\text{expr1} \rightarrow \text{excludesAll}(\text{expr2})) :=$
 $\forall(\underline{v:T}, tr_S(\text{expr2}, \underline{v:T}) \Rightarrow \neg tr_S(\text{expr1}, \underline{v:T}))$

where $t(\text{expr1}) = t(\text{expr2}) = \text{Set}(T)$.

5. $tr_E(\text{expr} \rightarrow \text{notEmpty}()) := \exists(\underline{v:T}, tr_S(\text{expr}, \underline{v:T}))$
6. $tr_E(\text{expr} \rightarrow \text{size}() \geq n) := \exists(\underline{v_1:T} \dots \underline{v_n:T}, \bigwedge_{i=1}^n tr_S(\text{expr}, \underline{v_i:T}))$
 where n is an integer constant ≥ 0 , $t(\text{expr}) = \text{Set}(T)$ and v_1, \dots, v_n are fresh variables of type T .
7. a) $tr_E(\text{expr1} = \text{expr2}) := \exists(\underline{v:T}, tr_N(\text{expr1}, \underline{v:T}) \wedge tr_N(\text{expr2}, \underline{v:T}))$
 if $t(\text{expr1}) = t(\text{expr2}) = T$ for some class T ,
- b) $tr_E(\text{expr1} = \text{expr2}) := \forall(\underline{v:T}, tr_S(\text{expr1}, \underline{v:T}) \Leftrightarrow tr_S(\text{expr2}, \underline{v:T}))$
 if $t(\text{expr1}) = t(\text{expr2}) = \text{Set}(T)$ for some class T .

8. $tr_E(\text{expr.attr1 op con}) := \exists(\underline{v:T}, tr_N(\text{expr}, \underline{v:T}) \wedge \exists(\frac{v:T}{\text{attr1 op con}}))$
 where con is a constant and $t(\text{expr}) = T$ for some class T .

9. $tr_E(\text{expr1.attr1 op expr2.attr2}) :=$
 $\exists(\underline{v:T}, tr_N(\text{expr1}, \frac{v:T}{\text{attr1 op x}}) \wedge tr_N(\text{expr2}, \frac{v:T}{\text{attr2} = x})) \vee$ ⁸
 $\exists(\underline{v:T} \ \underline{v':T'}, tr_N(\text{expr1}, \frac{v:T}{\text{attr1 op x}}) \wedge tr_N(\text{expr2}, \frac{v':t(v')}{\text{attr2} = x}))$

⁷We can express $\text{expr1} \rightarrow \text{exists}(v:T \mid \text{expr2})$ as “there exist objects v of type T , such that v is contained in the set described by expr1 and v satisfies expr2 ”, and $\text{expr1} \rightarrow \text{forall}(v:T \mid \text{expr2})$ as “for all nodes v of type T , if v is contained in the set described by expr1 then v also satisfies expr2 ”.

⁸The part before \vee is omitted if $\text{clan}(t(\text{expr1})) \cap \text{clan}(t(\text{expr2})) = \emptyset$, and the part after \vee is omitted if $\text{expr1} = \text{expr2}$.

7.3 Translating Essential OCL to graph conditions

where $t(\text{expr1}) = T$, $t(\text{expr2}) = T'$, $t(x) = t(\text{attr1}) = t(\text{attr2})$ and x , v and v' are fresh variables.

10. a) $tr_E(\text{expr.oclIsKindOf}(T)) := \exists(\overline{v:T'} \leftrightarrow \overline{v:T}, tr_N(\text{expr}, \overline{v:T'}))$
 b) $tr_E(\text{expr.oclIsTypeOf}(T)) :=$
 $\exists(\overline{v:T'} \leftrightarrow \overline{v:T}, \bigwedge_{T'' \in \xi} \neg \exists(\overline{v:T'} \leftrightarrow \overline{v:T''}) \wedge tr_N(\text{expr}, \overline{v:T'}))$
 where $T' = t(\text{expr})$, $T \in \text{clan}(T')$ and $\xi = \text{clan}(T) - \{T\}$.
11. $tr_N(\text{expr.oclAsType}(T), \overline{v:T}) := \exists(\overline{v:T'} \leftrightarrow \overline{v:T}, tr_N(\text{expr}, \overline{v:T'}))$
 where $T' = t(\text{expr})$ and $T \in \text{clan}(T')$
12. a) $tr_N(v, \overline{v:T}) := \exists(\overline{v=v':T})$ if v is a variable,
 b) If role has a multiplicity of 1, $tr_N(\text{expr.role}, \overline{v:T}) :=$
 $\exists(\overline{v':T'} \xrightarrow{\text{role}} \overline{v:T}, tr_N(\text{expr}, \overline{v':T'}))$ if $T' \notin \text{clan}(T)$ and
 $\exists(\overline{v':T'} \xrightarrow{\text{role}} \overline{v:T}, tr_N(\text{expr}, \overline{v':T'})) \vee \exists(\overline{v:T} \xrightarrow{\text{role}}, tr_N(\text{expr}, \overline{v:T}))$ else.
 c) If role has a multiplicity > 1 , $tr_S(\text{expr.role}, \overline{v:T}) :=$
 $\exists(\overline{v':T'} \xrightarrow{\text{role}} \overline{v:T}, tr_N(\text{expr}, \overline{v':T'}))$ if $T' \notin \text{clan}(T)$ and
 $\exists(\overline{v':T'} \xrightarrow{\text{role}} \overline{v:T}, tr_N(\text{expr}, \overline{v':T'})) \vee \exists(\overline{v:T} \xrightarrow{\text{role}}, tr_N(\text{expr}, \overline{v:T}))$ else,
 where v' is a fresh variable and $t(\text{expr}) = T'$ ⁹.

13. a) $tr_S(\text{expr1} \rightarrow \text{select}(v:T \mid \text{expr2}), \overline{v:T}) :=$
 $tr_S(\text{expr1}, \overline{v:T}) \wedge tr_E(\text{expr2})\{v/v'\}$ ¹⁰
 b) $tr_S(\text{expr1} \rightarrow \text{reject}(v:T \mid \text{expr2}), \overline{v:T}) :=$
 $tr_S(\text{expr1}, \overline{v:T}) \wedge \neg tr_E(\text{expr2})\{v/v'\}$

where $\text{expr2}\{v/v'\}$ means replacing v in expr2 with v' .

14. a) $tr_S(\text{expr1} \rightarrow \text{collect}(v:T \mid \text{expr2}), \overline{v:T'}) :=$
 $\exists(\overline{v:T}, tr_S(\text{expr1}, \overline{v:T}) \wedge tr_S(\text{expr2}, \overline{v:T'}))$ if expr2 yields a set, and
 b) $tr_S(\text{expr1} \rightarrow \text{collect}(v:T \mid \text{expr2}), \overline{v:T'}) :=$
 $\exists(\overline{v:T}, tr_S(\text{expr1}, \overline{v:T}) \wedge tr_N(\text{expr2}, \overline{v:T'}))$ if expr2 yields an object.¹¹
15. a) $tr_S(\text{expr1} \rightarrow \text{union}(\text{expr2}), \overline{v:T}) := tr_S(\text{expr1}, \overline{v:T}) \vee tr_S(\text{expr2}, \overline{v:T})$
 b) $tr_S(\text{expr1} \rightarrow \text{intersect}(\text{expr2}), \overline{v:T}) := tr_S(\text{expr1}, \overline{v:T}) \wedge tr_S(\text{expr2}, \overline{v:T})$
 c) $tr_S(\text{expr1} - \text{expr2}, \overline{v:T}) := tr_S(\text{expr1}, \overline{v:T}) \wedge \neg tr_S(\text{expr2}, \overline{v:T})$

⁹Case (a) presents the final step in a chain of navigations, while cases (b) and (c) present the navigation to single nodes and sets of nodes, respectively. Translations (b) and (c) are identical, since single nodes are treated as single-element sets.

¹⁰The idea for **select** (point 13 in the definition) is to restrict the set of nodes described by **expr1** by requiring that each node v' satisfying **expr1** also satisfies **expr2**. The construction for **reject** is analogous.

¹¹The translation $tr_S(\text{expr1} \rightarrow \text{collect}(v:T \mid \text{expr2}), \overline{v:T'})$ (point 14) is a condition over v' that is true iff there is a node v such that (a) v is contained in the set described by **expr1** (i.e. v satisfies $tr_S(\text{expr1}, \overline{v:T})$) and (b) the relation between v and v' given by **expr2** is satisfied. This is described by $tr_S(\text{expr2}, \overline{v:T'})$.

- d) $tr_S(\text{expr1} \rightarrow \text{symmetricDifference}(\text{expr2}), \boxed{v:T}) := tr_S(\text{expr1}, \boxed{v:T}) \vee tr_S(\text{expr2}, \boxed{v:T})$ ¹²
16. $tr_S(\text{T.allInstances}(), \boxed{v:T}) := \exists (\boxed{v:T})$ ¹³
17. $tr_S(\text{Set}\{\text{expr1}, \dots, \text{exprN}\}, \boxed{v:T}) := tr_N(\text{expr1}, \boxed{v:T}) \vee \dots \vee tr_N(\text{exprN}, \boxed{v:T})$
 where $\text{expr1}, \dots, \text{exprN}$ are OCL expressions of type T.

This concludes the transformation. △

Further translations of Essential OCL constraints can be derived from equivalences of OCL expressions. Most of these equivalences follow from basic set theory and logic axioms. See also (Richters, 2002, Tables 4.4 and 4.5 and page 73).

Definition 7.11 (further constraint translation).

1. $tr_E(\text{expr1} \rightarrow \text{includes}(\text{expr2})) := tr_E(\text{expr1} \rightarrow \text{includesAll}(\text{Set}\{\text{expr2}\}))$
 $tr_E(\text{expr1} \rightarrow \text{excludes}(\text{expr2})) := tr_E(\text{expr1} \rightarrow \text{excludesAll}(\text{Set}\{\text{expr2}\}))$
2. $tr_S(\text{expr1} \rightarrow \text{including}(\text{expr2}), \boxed{v:T}) := tr_S(\text{expr1} \rightarrow \text{union}(\text{Set}\{\text{expr2}\}), \boxed{v:T})$
 $tr_S(\text{expr1} \rightarrow \text{excluding}(\text{expr2}), \boxed{v:T}) := tr_S(\text{expr1} - \text{Set}\{\text{expr2}\}, \boxed{v:T})$
3. $tr_E(\text{expr1} \langle \rangle \text{expr2}) := tr_E(\text{not expr1} = \text{expr2})$
4. $tr_E(\text{expr1} \rightarrow \text{isEmpty}()) := tr_E(\text{not expr1} \rightarrow \text{notEmpty}())$
5. $tr_E(\text{expr} \rightarrow \text{size}() > n) := tr_E(\text{expr} \rightarrow \text{size}() \geq n+1)$
 $tr_E(\text{expr} \rightarrow \text{size}() = n) := tr_E(\text{expr} \rightarrow \text{size}() \geq n \text{ and not expr} \rightarrow \text{size}() \geq n+1)$
 $tr_E(\text{expr} \rightarrow \text{size}() \leq n) := tr_E(\text{not expr} \rightarrow \text{size}() > n)$
 $tr_E(\text{expr} \rightarrow \text{size}() < n) := tr_E(\text{not expr} \rightarrow \text{size}() \geq n)$
 $tr_E(\text{expr} \rightarrow \text{size}() \langle \rangle n) := tr_E(\text{not expr} \rightarrow \text{size}() = n)$
6. $tr_N(\text{expr1} \rightarrow \text{any}(v \mid \text{expr2}), \boxed{v:T}) := tr_S(\text{expr1} \rightarrow \text{select}(v \mid \text{expr2}), \boxed{v:T})$ ¹⁴
 $tr_E(\text{expr1} \rightarrow \text{one}(v \mid \text{expr2})) := tr_E(\text{expr1} \rightarrow \text{select}(v \mid \text{expr2}) \rightarrow \text{size}() = 1)$
7. $tr(\text{let } v=e \text{ in } e') := tr(e' [v/e])$

where expr , expr1 and expr2 are OCL expressions and n is an integer constant. △

Example 7.5. In the following example, an index above the = sign refers to the translation rule used; an index at the equivalence sign \equiv refers to the used equivalence rule of Proposition 7.4.

¹² $c \vee d$ denotes the exclusive disjunction operator.

¹³For $\text{T.allInstances}()$, the characteristic function is true for all nodes which are of type T.

¹⁴If expr1 is empty, the OCL expression would return `null`. Since we use two-valued logic, our translated constraint returns `false` in this case.

$$\begin{aligned}
 & tr_I(\text{context PetriNet inv:} \\
 & \quad \text{self.place} \rightarrow \text{select}(p:\text{Place} | p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}()) =^1 \\
 & \forall(\underline{\text{self:PN}}, tr_E(\text{self.place} \rightarrow \text{select}(p:\text{Place} | p.\text{token} \rightarrow \text{notEmpty}()) \rightarrow \text{notEmpty}())) =^5 \\
 & \forall(\underline{\text{self:PN}}, \exists(\underline{p:\text{Pl}}, tr_S(\text{self.place} \rightarrow \text{select}(p:\text{Place} | p.\text{token} \rightarrow \text{notEmpty}()), \underline{p:\text{Pl}}))) =^{13} \\
 & \forall(\underline{\text{self:PN}}, \exists(\underline{p:\text{Pl}}, tr_S(\text{self.place}, \underline{p:\text{Pl}}) \wedge tr_E(p.\text{token} \rightarrow \text{notEmpty}()))) =^5 \\
 & \forall(\underline{\text{self:PN}}, \exists(\underline{p:\text{Pl}}, tr_S(\text{self.place}, \underline{p:\text{Pl}}) \wedge \exists(\underline{t:\text{Tk}}, tr_S(p.\text{token}, \underline{t:\text{Tk}})))) =^{12} \\
 & \forall(\underline{\text{self:PN}}, \exists(\underline{p:\text{Pl}}, \exists(\underline{\text{self:PN}} \xrightarrow{\text{place}} \underline{p:\text{Pl}}) \wedge \exists(\underline{t:\text{Tk}}, \exists(\underline{p:\text{Pl}} \xrightarrow{\text{token}} \underline{t:\text{Tk}})))) \equiv^{E1, E2} \\
 & \forall(\underline{\text{self:PN}}, \exists(\underline{\text{self:PN}} \xrightarrow{\text{place}} \underline{p:\text{Pl}} \xrightarrow{\text{token}} \underline{t:\text{Tk}}))
 \end{aligned}$$

◇

To show the correctness of our translation, we also need to establish a correspondence relation between system states and typed attributed graphs.

Definition 7.12 (state correspondence). Let DSIG be a data signature and $M = (CLS, ACLS, ENUM, attr, assoc, \prec)$ be an object model over DSIG.

Let ATGI = (TG, Z, Inh) be an attributed type graph with inheritance and $I(s) = D_s$ for all sorts $s \in S' = S \cup ENUM$. Given a system state $\mathcal{S}(M) = (\mathcal{S}_{Cls}, \mathcal{S}_{Att}, \mathcal{S}_{Assoc})$, it *corresponds* to an attributed graph $AG = (G, type)$ with $G = (G_V, G_D, G_E, G_A, src_G, tgt_G, src_A, tgt_A)$ typed over ATGI by clan morphism *type* if there is a *state correspondence relation* from $corr_{State} = (c_{Cls}, c_{Att}, c_{Assoc})$ defined by the following bijective mappings

- $c_{Cls} : \mathcal{S}_{Cls} \rightarrow G_V$ such that $TG_V = CLASS$ and $type_{G_V}(c_{Cls}(o)) = c$ for $o \in \mathcal{S}_{Cls}(c)$ and for all $c_1, c_2 \in CLS$, $c_1 \prec c_2$ iff there is an edge from c_1 to c_2 in Inh .
- $c_{Att} : \mathcal{S}_{Att} \rightarrow G_A$ such that for every class c , object $o \in \mathcal{S}_{Cls}(c)$ and attribute $a \in \text{Dom}(\mathcal{S}_{Att})$ of a class in $clan(c)$, $src_A(c_{Att}(a)) = c_{Cls}(o)$ and $tgt_A(c_{Att}(a)) = \mathcal{S}_{Att}(a)$.
- $c_{Assoc} : \mathcal{S}_{Assoc} \rightarrow G_E$ such that for every association $A = (C_s, C_t)$ and every pair of objects $a = (o_s, o_t) \in \mathcal{S}_{Assoc}(A)$, $src_G(c_{Assoc}(a)) = c_{Cls}(o_s)$, $tgt_G(c_{Assoc}(a)) = c_{Cls}(o_t)$ and $type_{G_E}(c_{Assoc}(\mathcal{S}_{Assoc}(A))) = name$ for $assoc(A) = (name, min, max)$.

△

We can now prove Theorem 7.1: For a given model M , OCL invariant inv and environment (\mathcal{S}, β) ,

$$I[\text{inv}](\mathcal{S}, \beta) = \text{true} \text{ iff } G = corr_{State}(\mathcal{S}) \models tr_I(inv).$$

Proof (of Theorem 7.1). We prove, by induction over the structure of Essential OCL invariants, the more general statement

- (1) $I[\text{expr}](\mathcal{S}, \beta) = \text{true} \Leftrightarrow p \models tr_E(\text{expr})$,
- (2) $I[\text{expr}](\mathcal{S}, \beta) = v \Leftrightarrow p \oplus id_v \models tr_N(\text{expr}, \underline{v:\mathbb{T}})^{15}$,
- (3) $I[\text{expr}](\mathcal{S}, \beta) = \{v_1, \dots, v_n\} \Leftrightarrow \forall v \in \{v_1, \dots, v_n\}. p \oplus id_v \models tr_S(\text{expr}, \underline{v:\mathbb{T}})$.

¹⁵For morphisms $p: P \rightarrow G$, let function composition $p \oplus id_v$ be the morphism $p': P \oplus \underline{v:\mathbb{T}} \rightarrow G$, with $p'(v) = p(v)$ if $v \in \text{Dom}(p)$ and $p'(v) = v$ otherwise. Note that $P = \emptyset$ for constraints.

Basis. $I[\text{context } C \text{ inv: true}](\mathcal{S}, \beta) = \text{true} = \forall v \in \mathcal{S}_{Cls}(C). \text{true}$
 $= \forall (\overline{v:C}, \text{true}) = tr_I(\text{context } C \text{ inv: true}).$

Hypothesis. For subexpressions `expr`, objects v, v_1, \dots, v_n and morphisms $p: \{\overline{v:T} \in c_{Cls}(\beta(v)) \mid v \in \text{Dom}(\beta)\} \rightarrow corr_{state}(\mathcal{S})$, let statements (1), (2) and (3) be true.

Step.

(1) Let $t(\text{expr}) = T$.

$I[\text{context var:T inv: expr}](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow I[C.allInstances() \rightarrow \text{forall}(\text{expr})](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow p \models \forall (\overline{v:T}, tr_E(\text{expr}))$ Def. \models
 $\Leftrightarrow p \models tr_I(\text{context var:T inv: expr})$

Case `context C inv: expr` follows as a special case of the above with `var = self`.

(2) Since the Boolean operators of OCL have corresponding Boolean operators in graph conditions, the proofs are straightforward.

(3) Let $t(\text{expr1}) = T$.

$I[\text{expr1} \rightarrow \text{exists}(v:T \mid \text{expr2})](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow I[\text{expr1}](\mathcal{S}, \beta) = \{v_1, \dots, v_n\} \wedge \bigvee_{1 \leq i \leq n} I[\text{expr2}](\mathcal{S}, \beta\{v/v_i\})$ Set axioms
 $\Leftrightarrow I[\text{expr1}](\mathcal{S}, \beta) = \{v_1, \dots, v_n\} \wedge$
 $\quad \exists v_i \in \{v_1, \dots, v_n\}. I[\text{expr2}](\mathcal{S}, \beta\{v/v_i\})$ Set axioms
 $\Leftrightarrow \exists v \in \mathcal{S}_{Cls}(T). I[\text{expr1}](\mathcal{S}, \beta) \wedge I[\text{expr2}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow \exists (\overline{v:T} \in c_{Cls}(\mathcal{S}_{Cls}(T)). p \oplus id_v \models tr_S(\text{expr1}, \overline{v:T})$
 $\quad \wedge p \oplus id_v \models tr_E(\text{expr2}))$ Def. \models
 $\Leftrightarrow p \models \exists (\overline{v:T}, tr_S(\text{expr1}, \overline{v:T}) \wedge tr_E(\text{expr2}))$ Def. 7.10.3
 $\Leftrightarrow p \models tr_E(\text{expr1} \rightarrow \text{exists}(v:T \mid \text{expr2}))$

The proof of `forall` is analogous.

(4) Let $t(\text{expr1}) = t(\text{expr2}) = \text{Set}(T)$.

$I[\text{expr1} \rightarrow \text{includesAll}(\text{expr2})](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow I[\text{expr2}](\mathcal{S}, \beta) \subseteq I[\text{expr1}](\mathcal{S}, \beta)$ Set axioms
 $\Leftrightarrow \forall v \in \mathcal{S}(T). v \in I[\text{expr2}](\mathcal{S}, \beta) \text{ implies } v \in I[\text{expr1}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow \forall (\overline{v:T} \in c_{Cls}(\mathcal{S}(T)). p \oplus id_v \models tr_S(\text{expr2}, \overline{v:T})$
 $\quad \text{implies } p \oplus id_v \models tr_S(\text{expr1}, \overline{v:T})$ Def. \models
 $\Leftrightarrow p \models \forall (\overline{v:T}, tr_S(\text{expr2}, \overline{v:T}) \text{ implies } tr_S(\text{expr1}, \overline{v:T}))$ Def. 7.10.4
 $\Leftrightarrow p \models tr_E(\text{expr1} \rightarrow \text{includesAll}(\text{expr2}))$

The proof of `excludesall` is analogous.

(5)

$I[\text{expr} \rightarrow \text{notEmpty}()](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow I[\text{expr}](\mathcal{S}, \beta) \neq \emptyset$ Set axioms
 $\Leftrightarrow \exists v \in \mathcal{S}_{Cls}(T). v \in I[\text{expr}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow \exists (\overline{v:T} \in c_{Cls}(\mathcal{S}_{Cls}(T)). p \oplus id_v \models tr_S(\text{expr}, \overline{v:T})$ Def. \models
 $\Leftrightarrow p \models \exists (\overline{v:T}, tr_S(\text{expr}, \overline{v:T}))$ Def. 7.10.5
 $\Leftrightarrow p \models tr_E(\text{expr} \rightarrow \text{notEmpty}())$

(6)

7.3 Translating Essential OCL to graph conditions

$$\begin{aligned}
& I[\mathbf{expr} \rightarrow \mathbf{size}() \geq n](\mathcal{S}, \beta) && \text{Def. 7.4} \\
& \Leftrightarrow |\{v \mid I[\mathbf{expr}](\mathcal{S}, \beta)\}| \geq n && \text{Set axioms} \\
& \Leftrightarrow \exists v_1, \dots, v_n \in \mathcal{S}(\mathbf{T}). \bigwedge_{i,j=1, i \neq j}^n (v_i \neq v_j) \\
& \quad \wedge \bigwedge_{i=1}^n (v_i \in I[\mathbf{expr}](\mathcal{S}, \beta)) && \text{Ind. hyp.} \\
& \Leftrightarrow \exists \boxed{v_1:\mathbf{T}} \cdots \boxed{v_n:\mathbf{T}} \in c_{Cls}(\mathcal{S}(\mathbf{T})). \bigwedge_{i=1}^n p \oplus \text{id}_{v_i} \models \text{tr}_S(\mathbf{expr}, \boxed{v_i:\mathbf{T}}) && \text{Def. } \models \\
& \Leftrightarrow p \models \exists (\boxed{v_1:\mathbf{T}} \cdots \boxed{v_n:\mathbf{T}}, \bigwedge_{i=1}^n \text{tr}_S(\mathbf{expr}, \boxed{v_i:\mathbf{T}})) && \text{Def. 7.10.6} \\
& \Leftrightarrow p \models \text{tr}_E(\mathbf{expr} \rightarrow \mathbf{size}() \geq n)
\end{aligned}$$

(7a) For $t(\mathbf{expr1}) = t(\mathbf{expr2}) = \mathbf{T}$ for some class \mathbf{T} ,

$$\begin{aligned}
& I[\mathbf{expr1} = \mathbf{expr2}](\mathcal{S}, \beta) && \text{Def. 7.4} \\
& \Leftrightarrow I[\mathbf{expr1}](\mathcal{S}, \beta) = I[\mathbf{expr2}](\mathcal{S}, \beta) && \text{use variable} \\
& \Leftrightarrow \exists v \in \mathcal{S}_{Cls}(\mathbf{T}). v \in I[\mathbf{expr1}](\mathcal{S}, \beta) \wedge v \in I[\mathbf{expr2}](\mathcal{S}, \beta) && \text{Ind. hyp.} \\
& \Leftrightarrow \exists \boxed{v:\mathbf{T}} \in c_{Cls}(\mathcal{S}_{Cls}(\mathbf{T})). p \oplus \text{id}_v \models \text{tr}_N(\mathbf{expr1}, \boxed{v:\mathbf{T}}) \\
& \quad \wedge p \oplus \text{id}_v \models \text{tr}_N(\mathbf{expr2}, \boxed{v:\mathbf{T}}) && \text{Def. } \models \\
& \Leftrightarrow p \models \exists (\boxed{v:\mathbf{T}}, \text{tr}_N(\mathbf{expr1}, \boxed{v:\mathbf{T}}) \wedge \text{tr}_N(\mathbf{expr2}, \boxed{v:\mathbf{T}})) && \text{Def. 7.10.7a} \\
& \Leftrightarrow p \models \text{tr}_E(\mathbf{expr1} = \mathbf{expr2})
\end{aligned}$$

(7b) For $t(\mathbf{expr1}) = t(\mathbf{expr2}) = \text{Set}(\mathbf{T})$ for some class \mathbf{T} ,

$$\begin{aligned}
& I[\mathbf{expr1} = \mathbf{expr2}](\mathcal{S}, \beta) && \text{Def. 7.4} \\
& \Leftrightarrow I[\mathbf{expr1}](\mathcal{S}, \beta) = I[\mathbf{expr2}](\mathcal{S}, \beta) && \text{Set axioms} \\
& \Leftrightarrow \forall v \in \mathcal{S}_{Cls}(\mathbf{T}). v \in I[\mathbf{expr1}](\mathcal{S}, \beta) \text{ iff } v \in I[\mathbf{expr2}](\mathcal{S}, \beta) && \text{Ind. hyp.} \\
& \Leftrightarrow \forall \boxed{v:\mathbf{T}} \in c_{Cls}(\mathcal{S}(\mathbf{T})). p \oplus \text{id}_v \models \text{tr}_S(\mathbf{expr1}, \boxed{v:\mathbf{T}}) \text{ iff} \\
& \quad p \oplus \text{id}_v \models \text{tr}_S(\mathbf{expr2}, \boxed{v:\mathbf{T}}) && \text{Def. } \models \\
& \Leftrightarrow p \models \forall (\boxed{v:\mathbf{T}}, \text{tr}_S(\mathbf{expr1}, \boxed{v:\mathbf{T}}) \text{ iff } \text{tr}_S(\mathbf{expr2}, \boxed{v:\mathbf{T}})) && \text{Def. 7.10.7b} \\
& \Leftrightarrow p \models \text{tr}_E(\mathbf{expr1} = \mathbf{expr2})
\end{aligned}$$

(8) Let $t(v) = \mathbf{T}$.

$$\begin{aligned}
& I[v.\mathbf{attr} \text{ op } \mathbf{x}] && \text{Def. 7.4} \\
& \Leftrightarrow I(\text{op})(\mathcal{S}, \beta)(I[v.\mathbf{attr}](\mathcal{S}, \beta), I(\mathbf{x})(\mathcal{S}, \beta)) \\
& \Leftrightarrow I(\text{op})(\mathcal{S}, \beta)(\mathcal{S}_{Att}(\mathbf{attr})(\beta(v)), \mathbf{x}) && \text{Ind. hyp.} \\
& \Leftrightarrow \exists q: \boxed{\begin{array}{c} v:\mathbf{T} \\ \mathbf{attr} \text{ op } \mathbf{x} \end{array}} \hookrightarrow G.p = q \circ (\text{Dom}(p) \rightarrow \boxed{\begin{array}{c} v:\mathbf{T} \\ \mathbf{attr} \text{ op } \mathbf{x} \end{array}}) && \text{Def. } \models \\
& \Leftrightarrow p \models \exists \boxed{\begin{array}{c} v:\mathbf{T} \\ \mathbf{attr} \text{ op } \mathbf{x} \end{array}} && \text{Def. 7.10.9} \\
& \Leftrightarrow p \models \text{tr}_E(v.\mathbf{attr} \text{ op } \mathbf{x})
\end{aligned}$$

(9) Let $\mathbf{T} = t(\mathbf{expr1})$, $\mathbf{T}' = t(\mathbf{expr2})$ and $\text{att}(v:\mathbf{T}, \mathbf{attr}) = \mathcal{S}_{Att}(\mathbf{att})(I[v:\mathbf{T}](\mathcal{S}, \beta))$,
Let $p_v = p \oplus \text{id}_v$ and $p_{v'} = p \oplus \text{id}_{v'}$.

$I[\text{ex1.a1 op ex2.a2}](\mathcal{S}, \beta)$	Def. 7.4
$\Leftrightarrow \text{att}(\text{ex1, a1}) \text{ op } \text{att}(\text{ex2, a2})$	Def. 7.4
$\Leftrightarrow \exists v, v'. v = I[\text{ex1}](\mathcal{S}, \beta) \wedge v' = I[\text{ex2}](\mathcal{S}, \beta)$ $\quad \wedge \text{att}(v, \text{a1}) \text{ op } \text{att}(v', \text{a2})$	Ind. hyp.
$\Leftrightarrow \exists (\overline{v:\mathbf{T}}, \overline{v':\mathbf{T}'}). p_v \models (\text{tr}_N(\text{ex1}, \overline{v:\mathbf{T}}) \wedge \exists (\frac{v:\mathbf{T}}{\text{a1} = x}))$ $\quad \wedge p_{v'} \models \text{tr}_N(\text{ex2}, \overline{v':\mathbf{T}'}) \wedge \exists (\frac{v':\mathbf{T}'}{\text{a2} = x})$	Fact. 7.4
$\Leftrightarrow \exists (\overline{v=v':\mathbf{T}}). p_v \models \text{tr}_N(\text{ex1}, \frac{v=v':\mathbf{T}}{\text{a1 op x}}) \wedge p_{v'} \models \text{tr}_N(\text{ex2}, \frac{v=v':\mathbf{T}}{\text{a2} = x})$	
$\vee \exists (\overline{v:\mathbf{T}}, \overline{v':\mathbf{T}'}) . p_v \models \text{tr}_N(\text{ex1}, \frac{v:\mathbf{T}}{\text{a1 op x}}) \wedge p_{v'} \models \text{tr}_N(\text{ex2}, \frac{v':\mathbf{T}'}{\text{a2} = x})$	Def. \models
$\Leftrightarrow p \models \exists (\overline{v:\mathbf{T}}, \text{tr}_N(\text{ex1}, \frac{v:\mathbf{T}}{\text{a1 op x}}) \wedge \text{tr}_N(\text{ex2}, \frac{v:\mathbf{T}}{\text{a2} = x}))$ $\quad \vee \exists (\overline{v:\mathbf{T}}, \overline{v':\mathbf{T}'}) . \text{tr}_N(\text{ex1}, \frac{v:\mathbf{T}}{\text{a1 op x}}) \wedge \text{tr}_N(\text{ex2}, \frac{v':\mathbf{T}'}{\text{a2} = x})$	Def. 7.10.9
$\Leftrightarrow p \models \text{tr}_E(\text{ex1.a1 op ex2.a2})$	
(10a) Let $t(\text{expr}) = \mathbf{T}'$ and $\mathbf{T} \in \text{clan}(\mathbf{T}')$.	
$I[\text{expr.oc1IsTypeOf}(\mathbf{T})](\mathcal{S}, \beta)$	Def. 7.4
$\Leftrightarrow I[\text{expr}](\mathcal{S}, \beta) \in (I(\mathbf{T}) - \bigcup_{\mathbf{T}'' \leq_M \mathbf{T}}^{\mathbf{T}'' \neq \mathbf{T}} I(\mathbf{T}''))$	Set axioms
$\Leftrightarrow \exists v = I[\text{expr}](\mathcal{S}, \beta). v \in I(\mathbf{T}) \wedge \bigwedge_{\mathbf{T}'' \leq_M \mathbf{T}}^{\mathbf{T}'' \neq \mathbf{T}} . v \notin I(\mathbf{T}'')$	Def. 7.12, 7.2
$\Leftrightarrow \exists v = I[\text{expr}](\mathcal{S}, \beta). v \in \mathcal{S}_{Cls}^<(\mathbf{T}) \wedge \bigwedge_{\mathbf{T}'' \leq_M \mathbf{T}}^{\mathbf{T}'' \neq \mathbf{T}} . v \notin \mathcal{S}_{Cls}^<(\mathbf{T}'')$	Ind. hyp.
$\Leftrightarrow \exists (\overline{v:\mathbf{T}'}) . \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}}) \wedge \bigwedge_{\mathbf{T}'' \leq_M \mathbf{T}}^{\mathbf{T}'' \neq \mathbf{T}} . \neg \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}''})$	Def. \models
$\Leftrightarrow p \models \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}}, \bigwedge_{\mathbf{T}'' \in \text{clan}(\mathbf{T})}^{\mathbf{T}'' \neq \mathbf{T}} \neg \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}''}))$ $\quad \wedge \text{tr}_N(\text{expr}, \overline{v:\mathbf{T}'})$	Def. 7.10.10
$\Leftrightarrow p \models \text{tr}_E(\text{expr.oc1IsTypeOf}(\mathbf{T}))$	
(10b) The proof is analogous to the one for <code>oc1IsTypeOf</code> (without the \bigcup -part):	
Let $t(\text{expr}) = \mathbf{T}'$.	
$I[\text{expr.oc1IsKindOf}(\mathbf{T})](\mathcal{S}, \beta)$	Def. 7.4
$\Leftrightarrow I[\text{expr}](\mathcal{S}, \beta) \in I(\mathbf{T})$	Set axioms
$\Leftrightarrow \exists v = I[\text{expr}](\mathcal{S}, \beta). v \in I(\mathbf{T})$	Def. 7.12, 7.2
$\Leftrightarrow \exists v = I[\text{expr}](\mathcal{S}, \beta). v \in \mathcal{S}_{Cls}(\mathbf{T})$	Ind. hyp.
$\Leftrightarrow \exists (\overline{v:\mathbf{T}'}) . \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}})$	Def. \models
$\Leftrightarrow \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}}, \text{tr}_N(\text{expr}, \overline{v:\mathbf{T}'}))$	Def. 7.10.10
$\Leftrightarrow \text{tr}_E(\text{expr.oc1IsKindOf}(\mathbf{T}))$	
(11) Let $t(\text{expr}) = \mathbf{T}'$.	
$v = I[\text{expr.oc1AsType}(\mathbf{T})](\mathcal{S}, \beta)$	Def. 7.4
$\Leftrightarrow v = I[\text{expr}](\mathcal{S}, \beta) \wedge I[\text{expr}](\mathcal{S}, \beta) \in I(\mathbf{T})$	Def. 7.12, 7.2
$\Leftrightarrow v = I[\text{expr}](\mathcal{S}, \beta) \wedge v \in \mathcal{S}_{Cls}(\mathbf{T})$	Ind. hyp.
$\Leftrightarrow \exists \overline{v:\mathbf{T}'} \in \text{cCls}(\mathcal{S}(\mathbf{T})). \wedge p \oplus \text{id}_v \models \text{tr}_N(\text{expr}, \overline{v:\mathbf{T}'}) \wedge \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}})$	Def. \models
$\Leftrightarrow p \models \exists (\overline{v:\mathbf{T}'} \rightarrow \overline{v:\mathbf{T}}, \text{tr}_N(\text{expr}, \overline{v:\mathbf{T}'}))$	Def 7.10.11
$\Leftrightarrow p \models \text{tr}_N(\text{expr.oc1AsType}(\mathbf{T}), \overline{v:\mathbf{T}'})$	

7.3 Translating Essential OCL to graph conditions

(12a) Let $t(v) = t(v') = T$.
 $v' = I[\underline{v}](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow \exists v' \in \mathcal{S}(T). \beta(v) = v'$ Def. 7.12
 $\Leftrightarrow \exists \underline{v=v':T} \in c_{Cls}(\mathcal{S}(T))$ Def. \models
 $\Leftrightarrow p \models \exists(\underline{v=v':T})$ Def. tr_N
 $\Leftrightarrow p \models tr_N(v, \underline{v':T})$

(12b) First, assume $T \notin \text{clan}(T')$ and let $t(\text{expr}) = T', t(\text{expr.role}) = T$.
 $v = I[\text{expr.role}](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow (I[\text{expr}](\mathcal{S}, \beta), v) \in \mathcal{S}_{Assoc}(\text{role})$ Def. 7.12
 $\Leftrightarrow \exists v' = I[\text{expr}](\mathcal{S}, \beta) \wedge t(v') = T'$
 $\quad \wedge \underline{v':T'} \xrightarrow{\text{role}} \underline{v:T} \in c_{Assoc}(\mathcal{S}_{Assoc}(\text{role}))$ Ind. hyp.
 $\Leftrightarrow \exists(\underline{v':T'} \in c_{Cls}(\mathcal{S}(T')). p \oplus \text{id}_{v'} \models tr_N(\text{expr}, \underline{v':T'}) \wedge$
 $\quad p \oplus \text{id}_{v'} \oplus \text{id}_v \models \underline{v':T'} \xrightarrow{\text{role}} \underline{v:T})$ Def. \models
 $\Leftrightarrow p \models \exists(\underline{v':T'} \xrightarrow{\text{role}} \underline{v:T}, tr_N(\text{expr}, \underline{v':T'}))$ Def. 7.10.12
 $\Leftrightarrow p \models tr_N(\text{expr.role}, \underline{v':T})$

Now assume $T \in \text{clan}(T')$ and let $t(\text{expr}) = T', t(\text{expr.role}) = T$.

$v = I[\text{expr.role}](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow (I[\text{expr}](\mathcal{S}, \beta), v) \in \mathcal{S}_{Assoc}(\text{role})$ Def. 7.12
 $\Leftrightarrow \exists v' = I[\text{expr}](\mathcal{S}, \beta). \underline{v':T'} \xrightarrow{\text{role}} \underline{v:T} \in c_{Assoc}(\mathcal{S}_{Assoc}(\text{role}))$
 $\quad \vee \underline{v=v':T'} \curvearrowright_{\text{role}}$ Ind. hyp.
 $\Leftrightarrow \exists(\underline{v':T'}. p \oplus \text{id}_{v'} \models tr_N(\text{expr}, \underline{v':T'}) \wedge$
 $\quad (p \oplus \text{id}_{v'} \oplus \text{id}_v \models \underline{v':T'} \xrightarrow{\text{role}} \underline{v:T} \vee \underline{v=v':T'} \curvearrowright_{\text{role}}))$ Def. \models
 $\Leftrightarrow p \models \exists(\underline{v':T'} \xrightarrow{\text{role}} \underline{v:T}, tr_N(\text{expr}, \underline{v':T'}))$
 $\quad \vee \exists(\underline{v:T} \curvearrowright_{\text{role}}, tr_N(\text{expr}, \underline{v:T}))$ Def. 7.10.12
 $\Leftrightarrow p \models tr_N(\text{expr.role}, \underline{v':T})$

The proof of the tr_S cases is analogous to the tr_N cases.

(13) Let $t(\text{expr1}) = \text{Set}(T)$.
 $v \in I[\text{expr1} \rightarrow \text{select}(v:T \mid \text{expr2})](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow v \in \{v \mid v \in I[\text{expr1}](\mathcal{S}, \beta) \} \wedge I[\text{expr2}](\mathcal{S}, \beta)$ Set axioms
 $\Leftrightarrow \exists v \in \mathcal{S}(T). v \in I[\text{expr1}](\mathcal{S}, \beta) \wedge I[\text{expr2}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow p \oplus \text{id}_v \models tr_S(\text{expr1}, \underline{v:T}) \wedge p \oplus \text{id}_v \models tr_E(\text{expr2})$ Def. 7.10.13
 $\Leftrightarrow p \models tr_S(\text{expr1} \rightarrow \text{select}(v:T \mid \text{expr2}), \underline{v:T})$

The proof for reject is analogous.

(14) Let $t(\text{expr1}) = \text{Set}(T)$.
 $v \in I[\text{expr1} \rightarrow \text{collect}(v:T \mid \text{expr2})](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow v \in \{I[\text{expr2}](\mathcal{S}, \beta\{v/v'\}) \mid v' \in I[\text{expr1}](\mathcal{S}, \beta)\}$ Set axioms
 $\Leftrightarrow \exists v' \in I[\text{expr1}](\mathcal{S}, \beta). v \in I[\text{expr2}](\mathcal{S}, \beta\{v/v'\})$ Ind. hyp.
 $\Leftrightarrow \exists(\underline{v:T}, \underline{v':T'}). p \oplus \text{id}_{v'} \models tr_S(\text{expr1}, \underline{v':T'})$
 $\quad \wedge p \oplus \text{id}_v \models tr_S(\text{expr2}, \underline{v:T})$ Def. \models
 $\Leftrightarrow \exists(\underline{v:T}, p \models \exists(\underline{v':T'}, tr_S(\text{expr1}, \underline{v':T'}))$
 $\quad \wedge p \oplus \text{id}_v \oplus \text{id}_{v'} \models tr_S(\text{expr2}, \underline{v:T}))$ Def. \models
 $\Leftrightarrow p \models \exists(\underline{v:T}, tr_S(\text{expr1}, \underline{v:T}) \wedge tr_S(\text{expr2}, \underline{v':T'}))$ Def. 7.10.14
 $\Leftrightarrow p \models tr_S(\text{expr1} \rightarrow \text{collect}(v:T \mid \text{expr2}), \underline{v':T'})$

The proof for expr2 yielding an object is analogous.

(15) Let $t(\text{expr1}) = \text{Set}(\mathbb{T})$.
 $v \in I[\text{expr1} \rightarrow \text{union}(\text{expr2})](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow v \in \{v' \mid v' \in I[\text{expr1}](\mathcal{S}, \beta)\} \cup \{v' \mid v' \in I[\text{expr2}](\mathcal{S}, \beta)\}$ Set axioms
 $\Leftrightarrow v \in I[\text{expr1}](\mathcal{S}, \beta) \vee v \in I[\text{expr2}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow p \oplus \text{id}_v \models \text{tr}_S(\text{expr1}, \boxed{v:\mathbb{T}}) \vee p \oplus \text{id}_v \models \text{tr}_S(\text{expr2}, \boxed{v:\mathbb{T}})$ Def. \models
 $\Leftrightarrow p \models \text{tr}_S(\text{expr1}, \boxed{v:\mathbb{T}}) \vee \text{tr}_S(\text{expr2}, \boxed{v:\mathbb{T}})$ Def. 7.10.15
 $\Leftrightarrow p \models \text{tr}_S(\text{expr1} \rightarrow \text{union}(\text{expr2}), \boxed{v:\mathbb{T}})$

The proofs for `intersect`, `-` and `symmetricDifference` are analogous.

(16)
 $v \in I[\mathbb{T}.\text{allInstances}](\mathcal{S}, \beta) = \mathcal{S}_{Cls}(\mathbb{T})$ Def. 7.4
 $\Leftrightarrow v \in \mathcal{S}_{Cls}(\mathbb{T})$ Def. 7.12
 $\Leftrightarrow t(v) = \mathbb{T}$ Def. \models
 $\Leftrightarrow p \models \exists(\boxed{v:\mathbb{T}})$ Def. 7.10.16
 $\Leftrightarrow p \models \text{tr}_S(\mathbb{T}.\text{allInstances}(), \boxed{v:\mathbb{T}})$

(17) Let $t(\text{expr1}) = \dots = t(\text{exprN}) = \mathbb{T}$.
 $v \in I[\text{Set}\{\text{expr1}, \dots, \text{exprN}\}](\mathcal{S}, \beta)$ Def. 7.4
 $\Leftrightarrow v \in \{I[\text{expr1}](\mathcal{S}, \beta), \dots, I[\text{exprN}](\mathcal{S}, \beta)\}$ Set axioms
 $\Leftrightarrow v = I[\text{expr1}](\mathcal{S}, \beta) \vee \dots \vee v = I[\text{exprN}](\mathcal{S}, \beta)$ Ind. hyp.
 $\Leftrightarrow p \oplus \text{id}_v \models \text{tr}_N(\text{expr1}, \boxed{v:\mathbb{T}}) \vee \dots \vee p \oplus \text{id}_v \models \text{tr}_N(\text{exprN}, \boxed{v:\mathbb{T}})$ Def. \models
 $\Leftrightarrow p \models \text{tr}_N(\text{expr1}, \boxed{v:\mathbb{T}}) \vee \dots \vee \text{tr}_N(\text{exprN}, \boxed{v:\mathbb{T}})$ Def. 7.10.17
 $\Leftrightarrow p \models \text{tr}_S(\text{Set}\{\text{expr1}, \dots, \text{exprN}\}, \boxed{v:\mathbb{T}})$

This completes the induction proof.

We obtain Theorem 7.1 because for any OCL expression $\text{inv} = \text{context } \mathbb{C} \text{ inv: expr}$ and morphism $p: \emptyset \rightarrow G$, $G \models \text{tr}_I(\text{inv})$ iff $p \models \forall(\boxed{\text{self}:\mathbb{C}}, \text{tr}_E(\text{expr}))$. \square

Translation tr is defined along the structure of Essential OCL constraints. However, there are some parts of Essential OCL which go beyond the power of the translation. The next part will deal with such cases.

7.4 Translating OCL constraints beyond first-order expressiveness

With the above translation, most Essential OCL constraints can be translated into nested conditions. However, the expressive power of Essential OCL goes beyond first-order logic. Indeed, it is possible to express arbitrary primitive-recursive functions using the `iterate` operation, as shown in (Mandel and Cengarle, 1999). With `iterate`, it is also possible to formulate properties involving arbitrarily long paths.

This part extends the joint work from (Radke et al., 2015) by providing a translation of `iterate` into HR* conditions (as usual, with \mathcal{M} -satisfaction).

Example 7.6. The following constraint checks whether a Petri net contains a cycle, i.e. there is a path from a place to itself. For a place `p1`, let `p1.postPlace` abbreviate the expression `p1.postArc.dst.postArc.dst` of places which have incoming edges (via some transition) from `p1`. The `iterate` body accumulates, for all `Place` objects `p1` in the

set `acc`, all places `pl.postPlace`. The set `acc` is primed with all places `acc.postPlace` reachable from `p` via a direct transition. The Petri net contains a cycle (of length ≥ 3) if the set of places resulting from the `iterate` operation includes the starting place `self`:

```
context Place inv: Place.allInstances()->iterate(p:Place; acc=
  self.postPlace | acc->union(acc.postPlace))->includes(self)
```

◇

The role of non-determinism in iterate expressions. Generally, the result of an `iterate` expression might depend on the order in which the members of the collection are processed. (Richters, 2002) writes on page 94 that “For operations where evaluation order indeed makes a difference [...], a non-ordered collection first has to be transformed into a sequence”. Concerning the operation `asSequence: Set(t) → Sequence(t)` (Richters, 2002) notes on page 71 that “the semantics of the operation `asSequence` is non-deterministic. Any sequence containing only the elements of the source set (in arbitrary order) satisfies the operation specification in OCL”. In our case, we restrict ourselves to `iterate` operations over flat sets, and the non-determinism of the `iterate` operation fits well with the non-determinism of graph conditions.

Definition 7.13 (set-constructive iterate expression). Let `expr` and S_0 be OCL expressions yielding a set of type T' and T , respectively. Let $build_{S,x}$ be an OCL expression of either the form $S \rightarrow \text{union}(nav_x)$ with some navigation expression nav_x from $x = v_1$ to a set of objects v_n of type T , or of the form $S \rightarrow \text{including}(nav_x)$ with a navigation expression nav_x from $x = v_1$ to a single object v_n of type T . Let

$$cond_x = \bigvee_{i \in I} \left(\bigwedge_{j \in J_i} (ex_{(i,j)}) \right)$$

be an OCL expression of type `Bool` in DNF (disjoint normal form), where each literal $ex_{(i,j)}$ is an attribute comparison of either the form $v_i.att1 \text{ op } c$ for some constant c or $v_i.att1 \text{ op } v_j.att2$ for attributes `att1`, `att2` and nodes v_i, v_j occurring in nav_x .

An OCL `iterate` expression `expr->iterate(...)` is called *set-constructive* if it has the form `expr->iterate(x:T'; S:Set(T)=S0 | if (condx) then buildS,x else S)`. △

The translation works by representing the set S resulting from the iteration by a hyperedge \boxed{S} of rank 0. At the same time, a hyperedge $\boxed{v_1:T_1^1} \text{---} \boxed{P} \text{---} \boxed{v_n:T_n}$ of rank 2 represents the navigation from an object v_1 to an object v_n . The goals of the navigation, the nodes v_n , are required to be in the set of nodes generated from hyperedge \boxed{S} . The step-by-step generation of the set by the replacement rules of hyperedge P simulates the iteration process. The replacement rules rely on the DNF structure of $cond_x = \bigvee_{i \in I} (cond_i)$ and the navigational expression nav_x inside of $build_{S,x}$ to construct a set of graphs $G(nav_x, cond_i)$ that represent the structure and constraints expressed by nav_x and $cond_x$.

Construction. Because of the structure of set-constructive iteration expressions, nodes v_1 and v_n have the same type $T = T_1 = T_n$. Let

$\text{iter} = \text{expr} \rightarrow \text{iterate}(x:T'; S:\text{Set}(T)=S_0 \mid \text{if } (cond_x) \text{ then } build_{S,x} \text{ else } S)$

be a set-constructive OCL iteration expression. For every conjunction of expressions $cond_i = (\bigwedge_{j \in J_i}(ex_{(i,j)}))$ in $cond_x$, let $cond_{i,v_j}$ be the set of all literals in $cond_i$ over¹⁶ v_j , and let $T_i = t(v_i)$ be the type of OCL variable v_i . For a navigation expression $nav_{v_1} = v_1.\text{role1} \dots \text{roleN}$ and a conjunction $cond_i$ of literals, let

$$G(nav_x, cond_i) = \frac{v_1:T_1}{cond_{i,v_1}} \xrightarrow{\text{role1}} \frac{v_2:T_2}{cond_{i,v_2}} \dots \frac{v_n:T_n}{cond_{i,v_n}}.$$

We define $tr_S(\text{iter}, \overline{v:T}) :=$

$$\begin{aligned} & \exists(\overline{v:T}, tr_S(\text{expr}, \overline{v:T}), tr_S(S_0, \overline{v_1:T}) \wedge \exists(\overline{v_1:T} \perp \overline{P} \text{ } \overline{v_n:T}), \\ & \exists(\overline{v_1:T} \perp \overline{P} \text{ } \overline{v_n:T} \sqsupseteq \overline{v_1:T} \perp \overline{P} \text{ } \overline{v:T} \perp \overline{P} \text{ } \overline{v_n:T}) \\ & \vee \exists(\overline{v=v_1:T}) \vee \exists(\overline{v=v_n:T})) \end{aligned}$$

with

$$\begin{aligned} \mathcal{R} = \bigcup_{i \in I} \{ & \frac{\overline{v_1:T_1} \perp \overline{P} \text{ } \overline{v_n:T_n}}{\frac{v_1:T_1}{cond_{i,v_1}} \dots \frac{v_n:T_n}{cond_{i,v_n}}} / \\ & \frac{\overline{v_1:T_1} \perp \overline{P} \text{ } \overline{v'_n:T_n}}{\underbrace{\frac{v_1:T_1}{cond_{i,v_1}} \dots \frac{v_n:T_n}{cond_{i,v_n}}}_{G(nav_x, cond_i)} \perp \overline{P} \text{ } \overline{v'_n:T_n}} \} \end{aligned}$$



Example 7.7. Let

$$\overline{p1:Pl} \xrightarrow{\text{postPlace}} \overline{p2:Pl} \text{ abbreviate } \overline{p1:Pl} \xrightarrow{\text{postArc}} \overline{ArcPT} \xrightarrow{\text{dst}} \overline{Tr} \xrightarrow{\text{postArc}} \overline{ArcTP} \xrightarrow{\text{dst}} \overline{p2:Pl}.$$

The cycle constraint

```
context Place inv:Place.allInstances()->iterate(p:Place;acc=
  self.postPlace | acc->union(p.postPlace))->includes(self)
is first translated into the equivalent OCL constraint
context Place inv:Place.allInstances()->iterate(p:Place;acc=
  self.postPlace | if true then acc->union(p.postPlace) else acc)
->includes(self)
```

and then into the HR* condition

$$\begin{aligned} & \forall(\overline{self:Pl}, \exists(\overline{p:Pl}, \exists(\overline{self:Pl} \xrightarrow{\text{postPlace}} \overline{p1:Pl}) \wedge \exists(\overline{p1:Pl} \perp \overline{P} \text{ } \overline{p2:Pl}), \\ & \exists(\overline{p1:Pl} \perp \overline{P} \text{ } \overline{p2:Pl} \sqsupseteq \overline{p1:Pl} \perp \overline{P} \text{ } \overline{p:Pl} \perp \overline{P} \text{ } \overline{p2:Pl}) \\ & \vee \exists(\overline{p=p1:Pl}) \vee \exists(\overline{p=p2:Pl})) \wedge \exists(\overline{p=self:Pl})) \end{aligned}$$

¹⁶A literal is over some object v if it has the form $v.\text{att1 op ex}$ for some expression ex , e.g. $v.\text{name}=v'.$ name is a literal over v .

with

$$\mathcal{R} = \{ \boxed{v_1:\text{Pl}} \xrightarrow{1} \boxed{P} \xrightarrow{2} \boxed{v_2:\text{Pl}} \quad ::= \quad \boxed{v_1:\text{Pl}} \xrightarrow{\text{postPlace}} \boxed{v_2:\text{Pl}} \mid \\ \boxed{v_1:\text{Pl}} \xrightarrow{\text{postPlace}} \boxed{v_2:\text{Pl}} \xrightarrow{1} \boxed{P} \xrightarrow{2} \boxed{v'_2:\text{Pl}} \quad \}$$

This condition can be simplified to

$$\forall (\boxed{\text{self}:\text{Pl}}, \exists (\boxed{\text{self}:\text{Pl}} \xrightarrow{\text{postPlace}} \boxed{v_1:\text{Pl}}) \wedge \exists (\boxed{v_1:\text{Pl}} \xrightarrow{1} \boxed{P} \xrightarrow{2} \boxed{v_2:\text{Pl}}, \\ \exists (\boxed{v_1:\text{Pl}} \xrightarrow{1} \boxed{P} \xrightarrow{2} \boxed{v_2:\text{Pl}} \sqsupset \boxed{\text{self}:\text{Pl}})) \vee \exists (\boxed{\text{self}:\text{Pl}} \xrightarrow{\text{postPlace}} \boxed{\text{self}:\text{Pl}}))$$

The meaning of the above formula can be explained as “There is a node `self` and a node `v1` with a `postPlace` edge from `self`, and there is a path (built with edges of type `postArc`) from `v1` to a node `v2`, such that `self` is a node on this path or a node reached from `self` by a `postPlace` edge”. Put more simply, this means that a path of `postPlace` edges beginning from `self` leads back to `self`, i.e. a cycle. \diamond

Theorem 7.2 (Translating iterate expressions).

For every set-constructive OCL expression $expr$ of type T , all environments $(\mathcal{S}, \beta) \in Env$ and every morphism $p: v:T \rightarrow G$,

$$v:T \in I[\![expr]\!](\tau) \text{ iff } p \models_{\mathcal{A}}^r tr_{\mathcal{S}}(expr, \boxed{v:T}).$$

Proof. We proceed by induction over the size of the set $expr$ in $expr \rightarrow \text{iterate}(\dots)$. Let $p: \boxed{v:T} \rightarrow G$ be a morphism.

Basis. For $I[\![expr]\!](\tau) = \emptyset$,

$$\begin{aligned} u \in I[\![expr \rightarrow \text{iterate}(x; \text{acc}=\mathcal{S}_0 \mid \text{expr2})]\!](\tau) & \quad \text{Def. 7.4} \\ \Leftrightarrow v \in I[\![\mathcal{S}_0]\!](\tau) & \quad \text{Construction} \\ \Leftrightarrow p \models_{\mathcal{A}}^r tr_{\mathcal{S}}(\mathcal{S}_0, \boxed{v:T}) & \quad \text{Def. 7.4} \\ \Leftrightarrow p \models_{\mathcal{A}}^r tr_{\mathcal{S}}(expr \rightarrow \text{iterate}(x; \text{acc}=\mathcal{S}_0 \mid \text{expr2}), \boxed{v:T}) \end{aligned}$$

Hypothesis. Assume $v:T \in I[\![expr \rightarrow \text{iterate}(x; \text{acc}=\mathcal{S}_0 \mid \text{expr2})]\!](\tau)$

$$\Leftrightarrow p \models_{\mathcal{A}}^r tr_{\mathcal{S}}(expr \rightarrow \text{iterate}(x; \text{acc}=\mathcal{S}_0 \mid \text{expr2})).$$

Step. Let $\tau' = \tau \{ \text{acc} / I[\![\{x_1, \dots, x_{n-1}\} \rightarrow \text{iterate}(x; \text{acc}=\mathcal{S}_0 \mid \text{expr2})]\!] \} \{ x/x_n \}$ and $I[\![expr]\!](\tau) = \{x_1, \dots, x_n\}$. Without loss of generality, $expr2$ is of the form `if (condx) then acc \rightarrow union(navx) else acc`. Otherwise, since the iteration expression is set-constructive, $expr2$ is of the form `if (condx) then acc \rightarrow including(navx) else S`, and we can use Definition 7.11 to convert $expr2$ into the above form. By construction and Definition 7.4,

$v \in I[\text{expr} \rightarrow \text{iterate}(x; \text{acc}=\text{S}_0 \mid \text{expr2})](\tau)$	Construction
$\Leftrightarrow v \in I[\text{expr2}](\tau')$	Construction
$\Leftrightarrow I[\text{cond}_x](\tau') \wedge v \in I[\text{acc} \rightarrow \text{union}(nav_x)](\tau') \vee \neg I[\text{cond}_x](\tau') \wedge v \in I[\text{acc}](\tau')$	Def. 7.10
$\Leftrightarrow (I[\text{cond}_x](\tau') \wedge (v \in I[\text{acc}](\tau') \vee v \in I[nav_x](\tau'))) \vee (\neg I[\text{cond}_x](\tau') \wedge (v \in I[\text{acc}](\tau')))$	Logic axioms
$\Leftrightarrow v \in I[\text{acc}](\tau') \vee (I[\text{cond}_x](\tau') \wedge v \in I[nav_x](\tau'))$.	
We distinguish two cases $v \in I[\text{acc}](\tau')$ and $I[\text{cond}_x](\tau')$.	
For $v \in I[\text{acc}](\tau')$,	
$p \models_{\mathcal{A}}^r tr_S(\{x_1, \dots, x_{n-1}\} \rightarrow \text{iterate}(x; \text{acc}=\text{S}_0 \mid \text{expr2}))$.	Ind. hyp.
For $v \in I[nav_x](\tau') \wedge I[\text{cond}_x](\tau')$,	Def. $G(_, _)$
$v \in G(nav_x, cond_x) \wedge G(nav_x, cond_x) \in G$.	
The rules in \mathcal{R} ensure that $G(nav_x, cond_x) \in \boxed{v_1:\mathbb{T}}^1 \text{---} \boxed{P} \text{---} \boxed{v_n:\mathbb{T}}^\sigma$, so	
$\Leftrightarrow v \in \boxed{v_1:\mathbb{T}}^1 \text{---} \boxed{P} \text{---} \boxed{v_n:\mathbb{T}}^\sigma \in G$	Construction
$\Leftrightarrow \exists \sigma. p \models_{\mathcal{A}}^r tr_S(\text{expr} \rightarrow \text{iterate}(x; \text{acc}=\text{S}_0 \mid \text{expr2}), \boxed{v:\mathbb{T}})^\sigma$	Def. 7.4
$\Leftrightarrow p \models_{\mathcal{A}}^r tr_S(\text{expr} \rightarrow \text{iterate}(x; \text{acc}=\text{S}_0 \mid \text{expr2}), \boxed{v:\mathbb{T}})$.	

□

Limitations

The translation given on the last pages does not handle all of Essential OCL. As the goal is to connect meta-modeling with graph transformation systems, the focus is on the translation of OCL **invariants only**.

Concerning the OCL collection types, we restrict ourselves to **flat sets of objects**. Operations like `isUnique`, which is only useful for bags and sequences, or `sum`, which is only defined for sets of integers, are not considered. Since the sets are flat, the `flatten` operation also becomes superfluous. (Kuhlmann and Gogolla, 2012) argue that sets of objects are sufficient for language definition where order of objects and distinction of duplicates are not crucial.

Furthermore, the results of Boolean expressions are restricted to **two-valued logic**, as is the case for HR^* conditions. The values `void` and `invalid` are not considered, as well as the `oclIsUndefined` operation. As argued in (Schürr, 2001), the use of three- or four-valued logic often leads to rather unexpected results; also, existing meta-model specifications have shown that two-valued logic covers the substantial part of well-formedness rules specified in OCL.

Finally, the translation of the `iterate` operator is restricted to **set-constructive iterate expressions** as per Definition 7.13.

7.5 Integration of graph constraints into graph grammars

The previous pages described how to build a graph grammar from a UML class diagram and how to transform OCL constraints into graph constraints. This yields a method to generate instances that satisfy the model: First, we use the grammar to generate an instance that respects the relations given in the class diagram. If this instance satisfies

all the graph constraints generated from the OCL constraints, we keep it; otherwise, we generate a new instance and recheck the constraints.

However, this generate-and-test approach is very inefficient and negates the advantages gained by using grammars to generate instances. Many instances are generated in vain, only to be rejected because they do not satisfy all the constraints.

In this chapter, we thus propose another approach: To integrate the graph constraints into the grammar. In this way, we can check satisfaction of the constraints at every step of the generation process, ensuring that only valid instances are generated.

The general idea is to use the transformations from Chapter 6. First, we use the Shift* construction from Chapter 6.1 to transform the conjunction of all graph constraints into right application conditions, and then use L to transform the right into left application conditions. In this form, it is possible to check the constraints directly before the application of a rule of the grammar.

However, there is one pitfall: Depending on the rules of the graph grammar, integration of constraints might lead to a left application condition **false**.

Example 7.8. The rule $\rho = \langle \boxed{\text{pn:PN}} \leftrightarrow \boxed{\text{pn:PN}} \xrightarrow{\text{place}} \boxed{\text{:Pl}} \rangle$ adds a place to a given Petri net, and $c = \forall(\boxed{\text{p:Pl}}, \exists(\boxed{\text{p:Pl}}^{\text{postarc}} \boxed{\text{:PTArc}}^{\text{dst}} \boxed{\text{:Tr}}))$ is a graph condition stating that every place has an outgoing edge to a transition. Translating c into a left application condition for ρ yields **false**, since the rule generates a new place that is not yet connected to a transition. \diamond

This is an unfortunate situation, since a rule with **false** as left application condition cannot be applied at all. This situation can be remedied in two ways.

One idea would be to introduce “composite” rules which combine several rules into one rule that immediately satisfies all constraints. In the above example, such a rule could simultaneously insert a place and a transition, connected by a **PTArc**. This idea has two disadvantages. Firstly, depending on the constraints, the generation rules might become overly large, slowing the generation process and making an interactive use difficult. Secondly, finding these composite rules can be difficult for a large and complex set of constraints.

As an alternative, we could allow rules to not satisfy all the constraints in each derivation step, by transforming only a subset of the constraints into an application condition. In the above example, condition c would not be part of the left application condition for rule ρ , but for another rule which connects a new transition to a place. This means, however, that not every derivation step leads to a valid model. In particular, a derivation step can turn a valid model (e.g. a Petri net with all Places connected to at least one Transition) into an invalid one (e.g. said Petri net with an new, isolated place). The derivation process thus needs to be controlled in some way. A rough idea would be to start with only a few constraints and enforce that a derivation step does not invalidate a constraint already satisfied, but might satisfy a previously unsatisfied constraint. Controlling the derivation process with a finite automaton also seems promising. However, a detailed solution for this problem is beyond the scope of this thesis.

Bibliographic notes

The idea of translating a class diagram into a graph grammar with the goal to create instances has been discussed in several works, e.g. (K. Ehrig et al., 2009) or (Besova et al., 2015).

Several approaches exist in the literature for the translation of OCL into a formal framework. Most of them translate the constraints into logical formulas. Typical motivations for such a translation include the definition of a clearer semantics for OCL, formal verification of models or, as is this case, the generation of instances.

Logic-oriented approaches Logic-oriented approaches have in common that they translate class models with OCL constraints into a textual representation with little or no visual components. For such languages, many powerful tools are available.

(Beckert et al., 2002) present a translation of a UML meta-model with accompanying OCL constraints into first-order logic and provide an implementation in the KeY system. (Cabot et al., 2007) translate class models with OCL constraints into logical formulas. The formal approach taken by the Alloy system (Jackson, 2006) can be used for instance generation: After translating a class diagram to Alloy, an instance can be generated or it can be shown that no instances exist. This generation relies on the use of SAT solvers and can also enumerate an arbitrary number of instances. A similar approach is taken by the Kodkod tool (Kuhlmann and Gogolla, 2012).

Another application for these techniques is the generation or recognition of edit operations. (Kehrer et al., 2013) lifts model change recognition and patching to recognizing and packaging edit operations to patches. The approach taken here could assist this with the automated generation of edit operations.

The EER/GRAL language in (Ebert et al., 1996) uses a combination of extended entity-relationship diagrams and a *Z*-like specification language, in order to specify classes of graphs that represent certain meta-model instances. Repositories of these graphs can be queried using the textual, first-order query language GReQL.

Graph-based approaches Graph-based approaches translate OCL constraints into a more visual formalism, e.g. graph patterns or graph constraints. Following this line, models and meta-models (without OCL constraints) are translated to instance and type graphs. Graph-based approaches keep the graph structure of models as units of abstraction, hence, graph axioms are satisfied by default. (Pennemann, 2009) shows that using a specialized theorem prover for graph conditions is more efficient than applying general theorem provers to graph conditions. (Bardohl et al., 1999) arguments that graph grammars are a suitable and natural way to specify visual languages in a constructive way.

The hybrid query language HQL presented in (Andries and Engels, 1996) offers both an SQL-like textual and a graphical syntax to specify properties of *Extended Entity-Relationship* diagrams. In (Schürr, 2001), OCL constraints are expressed with the *path expression language* of the PROGRES system, and extensions to OCL for functional abstraction and transitive closure (which has since been added to OCL with the `closure`

operation) are suggested. Furthermore, the paper suggests a visual representation of OCL, *collaboration constraint diagrams*, similar to (Bottoni et al., 2000). In (Amelunxen et al., 2007), OCL constraints are translated into SDM diagrams as supported by Fujaba/MOFLON, a combination of graph transformation rules, Java code and constraints on nodes and their attributes. (Winkelmann et al., 2008) shows the translation of a subset of OCL constraints, restricted to navigation expressions, size and attribute constraints and Boolean combinations thereof, into constraints for graph grammars.

In (Arendt, Habel, et al., 2014), we translate the Core subset of OCL into nested graph constraints. This work was extended to a larger subset of OCL, including set operations, in (Radke et al., 2015). (Bergmann, 2014) has implemented a translator of OCL constraints to INCQUERY graph patterns. The covered subset of OCL is similar to the one presented here. The focus of that work, however, is not a formal translation showing correctness and completeness, but an efficient implementation of constraint checking, using the incremental VIATRA engine. (Richa, Borde, Pautet, et al., 2014) have a similar idea of representing OCL with graph conditions, but can only translate to positive application conditions without nesting. Thus this approach is not as expressive. This was improved on in (Richa, Borde, and Pautet, 2015), which extends the approach by a representation for sets and a limited representation for ordered sets. This thesis, does not go into great detail on process of transforming a UML class diagram into a graph grammar. One such approach using layered graph grammars can be found in (Taentzer, 2012).

Chapter 8

Conclusion

Contents

8.1 Summary	129
8.2 Open problems and future work	130

In this final chapter, we summarize the results of the thesis and discuss several directions in which the results of this work may be extended.

8.1 Summary

We motivated the use of graph conditions and showed the limits of nested conditions: they cannot express non-local properties, such as the existence of paths or cycles of arbitrary length.

New concept of HR* conditions. We generalized the well-known concept of nested graph conditions to so-called HR* conditions. These conditions can also express non-local conditions, such as paths of arbitrary length, cycle-freeness or the existence of an even number of nodes. We introduced several variants of HR* conditions, discussed their respective advantages and disadvantages and provided constructions to transform these variants into one another.

Expressiveness of HR* conditions. We explored the expressiveness of HR* conditions, and established that HR* conditions can express every counting monadic second-order (CMSO) formula, and every HR* condition can be expressed by a second-order formula.

Correctness. The correctness of a graph program with respect to HR* pre- and post-conditions can be checked by calculating a weakest precondition and checking whether the original precondition implies the weakest precondition. We presented a construction for weakest preconditions for graph programs with respect to HR* conditions. The construction uses several basic transformations of HR* conditions over rules. Apart from

their use in the construction of weakest preconditions, they can be used to transform constraints into application conditions and vice versa, and to transform right into left application conditions (and vice versa). The use of HR* conditions in verification was demonstrated with a car platooning example. In contrast to other formalizations of that example, HR* conditions allow the modeling of an actual chain of cars linked to one another, instead of each car being linked directly to the platoon's leader.

Application to meta-modeling. We used HR* conditions together with graph grammars to generate instances of a meta-model with Essential OCL constraints and presented a transformation from Essential OCL constraints into graph conditions. To facilitate the transformation, we introduced compact conditions as a useful way to reduce redundancy and complexity. For a part of Essential OCL which cannot be expressed with nested conditions because it is beyond first-order, we provide a construction to express them with HR* conditions.

8.2 Open problems and future work

Results on expressiveness. At present, we can only say that any property expressible in counting MSO logic is expressible with HR* conditions and any property beyond SO logic is not. It would be interesting to have an exact characterization of the expressive power of HR* conditions. In particular, an in-depth comparison between HR* conditions and the μ -conditions of (Flick, 2016) would be interesting. Furthermore, one could look at the expressiveness of HR* conditions using a different replacement language than hyperedge replacement. As long as the replacement language is monotone, this should not change the decidability of a condition's validity, as shown in Theorem 3.1. An interesting candidate language would be contextual hyperedge replacement (Drewes and Hoffmann, 2015), which allows limited context in the replacement rules.

Automated correctness proofs. Similar to the correctness of graph programs with respect to nested conditions in (Pennemann, 2009), one could develop a theorem prover and / or a SAT solver trying to prove or refute whether a given HR* condition is a tautology. The transformations of HR* conditions over rules and programs needed for this are presented in this thesis. Developing a theorem prover for HR* conditions is complicated by the HR systems. In order to check whether one literal implies another, one has to check not only the graphs, but also the HR system used. This amounts to the problem whether the language generated by one HR grammar is a subset of the language generated by another HR grammar.

Implementation of HR* conditions. There is currently ongoing work to implement HR* conditions in several frameworks. The ENFORCe framework (Azab et al., 2007) for verifying graphical program specifications with respect to nested conditions was recently extended by a component for hyperedge replacement. It would be interesting to extend this to a full implementation of the algorithms and transformations of HR* condition

presented in this thesis. Support for nested conditions was also recently added to the HENSHIN framework (Arendt, Biermann, et al., 2010; Richa, Borde, and Pautet, 2015); this could also be extended to HR* conditions in order to support the HR* translations of `iterate` OCL constraints suggested in this thesis.

Translation of Essential OCL. In (Radke et al., 2015), we translated a substantial part of Essential OCL into HR* conditions. This thesis extended the translation by a translation of a subset of the `iterate` operation. As outlined in Chapter 7.4, the translation still has several limits. With further research, the restriction on invariants could be lifted to include query operations, and (Richa, Borde, and Pautet, 2015) outlined a way to support ordered sets. It would also be desirable to support the `iterate` operation without restrictions.

Efficient algorithms for instance generation. The application part of this thesis focuses on the translation of Essential OCL constraints into HR* conditions. However, we showed that naively translating the HR* conditions into left application conditions can evaluate to `false`, rendering the rule obsolete. Chapter 7.5 gives some ideas on how to remedy this; a thorough survey of this topic is beyond the scope of this work. For an efficient way to generate valid instances using graph grammars, more research on this topic has to be conducted.

Bibliography

- Adámek, Jiří, Horst Herrlich, and George E. Strecker (2004). *Abstract and concrete Categories*. Dover Publications.
- Amelunxen, Carsten, Elodie Legros, Andy Schürr, and Ingo Stürmer (2007). „Checking and Enforcement of Modeling Guidelines with Graph Transformations“. In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Vol. 5088. LNCS, pp. 313–328.
- Andries, Marc and Gregor Engels (1996). „A Hybrid Query Language for an Extended Entity-Relationship Model“. In: *Journal of Visual Languages and Computing* 7.3, pp. 321–352.
- Angelaccio, Michele, Tiziana Catarci, and Giuseppe Santucci (1990). „QBD*: A Graphical Query Language with Recursion“. In: *IEEE Trans. Software Eng.* 16.10, pp. 1150–1163.
- Arendt, Thorsten, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer (2010). „Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation“. In: *Proceedings MoDELS 2010*. Vol. 6394. LNCS, pp. 121–135.
- Arendt, Thorsten, Annegret Habel, Hendrik Radke, and Gabriele Taentzer (2014). „From Core OCL Invariants to Nested Graph Constraints“. In: *Int. Conf. on Graph Transformations (ICGT)*. Vol. 8571. LNCS, pp. 97–112.
- Azab, Karl, Annegret Habel, Karl-Heinz Pennemann, and Christian Zuckschwerdt (2007). „ENFORCE: A System for Ensuring Formal Correctness of High-level Programs“. In: *Proc. of the Third Int. Workshop on Graph Based Tools (GraBaTs'06)*. Vol. 1. Electronic Communications of the EASST, pp. 82–93.
- Baldan, Paolo, Andrea Corradini, Barbara König, and Bernhard König (2004). „Verifying a Behavioural Logic for Graph Transformation Systems“. In: *Electronic Notes in Theoretical Computer Science* 104, pp. 5–24.
- Bardohl, Roswitha, Mark Minas, Andy Schürr, and Gabriele Taentzer (1999). „Application of Graph Transformation to Visual Languages“. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 2. World Scientific, pp. 105–180.
- Bauer, Jörg (2006). „Analysis of Communication Topologies by Partner Abstraction“. PhD thesis. Universität des Saarlandes.

Bibliography

- Beckert, Bernhard, Uwe Keller, and Peter H. Schmitt (2002). „Translating the Object Constraint Language into First-Order Predicate Logic“. In: *Verification Workshop VERIFY, Copenhagen*. Ed. by Serge Autexier and Heiko Mantel. DIKU technical reports, pp. 113–123.
- Bergmann, Gábor (2014). „Translating OCL to Graph Patterns“. In: *Proc. MoDELS*. Vol. 8767. LNCS, pp. 670–686.
- Bertot, Yves and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag Berlin. ISBN: 9783642058806.
- Besova, Galina, Dominik Steenken, and Heike Wehrheim (2015). „Grammar-based model transformations: Definition, execution, and quality properties“. In: *Computer Languages, Systems & Structures* 43, pp. 116–138.
- Blume, Christoph (2014). „Graph Automata and Their Application to the Verification of Dynamic Systems“. PhD thesis. Universität Duisburg-Essen.
- Bottoni, Paolo, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer (2000). „Consistency Checking and Visualization of OCL Constraints“. In: *Proc. UML 2000 - The Unified Modeling Language*. Vol. 1939. LNCS, pp. 294–308.
- Brucker, Achim D. and Burkhart Wolff (2012). „Featherweight OCL: a study for the consistent semantics of OCL 2.3 in HOL“. In: *Proc. Workshop on OCL and Textual Modelling*. ACM, pp. 19–24.
- Bruggink, H. J. Sander, Mathias Hülsbusch, and Barbara König (2012). „Towards Alternating Automata for Graph Languages“. In: *Electronic Communications of the EASST* 47.
- Bruggink, H. J. Sander and Barbara König (2010). „A Logic on Subobjects and Recognizability“. In: *Theoretical Computer Science*, pp. 197–212.
- Cabot, Jordi, Robert Clarisó, and Daniel Riera (2007). „UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming“. In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, pp. 547–548.
- Courcelle, Bruno (1990). „Graph Rewriting: An Algebraic and Logical Approach“. In: *Handbook of Theoretical Computer Science*. Ed. by J. van Leeuwen. Vol. B. Amsterdam: North Holland Publ. Comp., pp. 192–242.
- Courcelle, Bruno (1994). „Monadic Second-Order Definable Graph Transductions: A Survey“. In: *Theoretical Computer Science* 126, pp. 53–75.

- Courcelle, Bruno (1996). „On the Expression of Graph Properties in some Fragments of Monadic Second-Order Logic“. In: *Descriptive Complexity and Finite Models*. Vol. 31. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, pp. 33–62.
- Courcelle, Bruno (1997). „The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic“. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, pp. 313–400.
- Courcelle, Bruno and Joost Engelfriet (2012). *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Vol. 138. Encyclopedia of Mathematics and its applications. Cambridge University Press.
- Cruz, Isabel F., Alberto O. Mendelzon, and Peter T. Wood (1987). „A Graphical Query Language Supporting Recursion“. In: *Proc. ACM Group on Management of Data (SIGMOD)*. Vol. 33, pp. 323–330.
- Dalen, Dirk van (2004). *Logic and Structure*. 4th edition. Springer-Verlag Berlin.
- Dijkstra, Edsger Wybe (1976). *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall.
- Dodds, Mike and Detlef Plump (2009). „From Hyperedge Replacement to Separation Logic and back“. In: *Proc. Doctoral Symposium at the International Conference on Graph Transformation*. Vol. 16. Electronic Communications of the EASST.
- Drewes, Frank, Annegret Habel, and Hans-Jörg Kreowski (1997). „Hyperedge Replacement Graph Grammars“. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 1. World Scientific, pp. 95–162.
- Drewes, Frank and Berthold Hoffmann (2015). „Contextual Hyperedge Replacement“. In: *Acta Informatica* 52.6, pp. 497–524.
- Ebert, Jürgen, Andreas Winter, Peter Dahm, Angelika Franzke, and Roger Süttenbach (1996). „Graph Based Modeling and Implementation with EER / GRAL“. In: *Int. Conf. on Conceptual Modeling - ER'96*. Vol. 1157. LNCS, pp. 163–178.
- Ehrig, Hartmut (1979). „Introduction to the Algebraic Theory of Graph Grammars“. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Vol. 73, pp. 1–69.
- Ehrig, Hartmut, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann (2006). „Theory of Constraints and Application Conditions: From Graphs to High-Level Structures“. In: *Fundamenta Informaticae* 74(1), pp. 135–166.

Bibliography

- Ehrig, Hartmut, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer (2006). *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer-Verlag Berlin.
- Ehrig, Hartmut, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas (2012). „ \mathcal{M} -Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence“. In: *Fundamenta Informaticae* 118, pp. 35–63.
- Ehrig, Hartmut, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas (2014). „ \mathcal{M} -Adhesive Transformation Systems with Nested Application Conditions. Part 1: Parallelism, Concurrency and Amalgamation“. In: *Mathematical Structures in Computer Science* 24(4).
- Ehrig, Hartmut and Annegret Habel (1986). „Graph Grammars with Application Conditions“. In: *The Book of L*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Springer-Verlag Berlin, pp. 87–100.
- Ehrig, Hartmut and Hans-Jörg Kreowski (1979). „Pushout-Properties: An Analysis of Gluing Constructions for Graphs“. In: *Mathematische Nachrichten* 91, pp. 135–149.
- Ehrig, Hartmut and Bernd Mahr (1985). *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Vol. 6. EATCS Monographs on Theoretical Computer Science. Springer-Verlag Berlin.
- Ehrig, Karsten, Jochen Malte Küster, and Gabriele Taentzer (2009). „Generating instance models from meta models“. In: *Software and System Modeling* 8.4, pp. 479–500.
- Flick, Nils Erik (2016). „Proving Correctness of Graph Programs Relative to Recursively Nested Conditions“. In: *Electronic Communications of the EASST* 73.
- Gadducci, Fabio, Alberto Lluch-Lafuente, and Andrea Vandin (2012). „Exploiting Over- and Under-Approximations for Infinite-State Counterpart Models“. In: *Graph Transformations (ICGT)*. Vol. 7562. LNCS, pp. 51–65.
- Gaifman, Haim (1982). „On Local and Non-Local Properties“. In: *Proc. of the Herbrand Symposium: Logic Colloquium '81*. North Holland Pub. Co., pp. 105–135.
- Habel, Annegret (1992). *Hyperedge replacement: grammars and languages*. Vol. 643. LNCS.
- Habel, Annegret, Reiko Heckel, and Gabriele Taentzer (1996). „Graph Grammars with Negative Application Conditions“. In: *Fundamenta Informaticae* 26.3/4, pp. 287–313.
- Habel, Annegret, Jürgen Müller, and Detlef Plump (2001). „Double-Pushout Graph Transformation Revisited“. In: *Mathematical Structures in Computer Science* 11.5, pp. 637–688.

- Habel, Annegret and Karl-Heinz Pennemann (2009). „Correctness of High-Level Transformation Systems Relative to Nested Conditions“. In: *Mathematical Structures in Computer Science* 19, pp. 245–296.
- Habel, Annegret, Karl-Heinz Pennemann, and Arend Rensink (2006). „Weakest Preconditions for High-Level Programs“. In: *Int. Conf. on Graph Transformations (2006)*. Vol. 4178. LNCS, pp. 445–460.
- Habel, Annegret and Detlef Plump (2001). „Computational Completeness of Programming Languages Based on Graph Transformation“. In: *Foundations of Software Science and Computation Structures (FOSSACS)*. Vol. 2030. LNCS, pp. 230–245.
- Habel, Annegret and Detlef Plump (2012). „ \mathcal{M}, \mathcal{N} -adhesive transformation systems“. In: *Proc. International Conference on Graph Transformation (ICGT 2012)*. Vol. 7562. LNCS, pp. 218–233.
- Habel, Annegret and Hendrik Radke (2010). „Expressiveness of Graph Conditions with Variables“. In: *Electronic Communications of the EASST* 30.
- Heckel, Reiko and Annika Wagner (1995). „Ensuring consistency of conditional graph rewriting - a constructive approach“. In: *Electronic Notes in Theoretical Computer Science* 2, pp. 118–126.
- Hermann, Frank, Mathias Hülsbusch, and Barbara König (2010). „Specification and Verification of Model Transformations“. In: *Electronic Communications of the EASST* 30.
- Hildebrandt, Stephan, Leen Lambers, Basil Becker, and Holger Giese (2012). „Integration of Triple Graph Grammars and Constraints“. In: *Electronic Communications of the EASST* 54.
- Hsu, Ann, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya (1991). *The Design of Platoon Maneuver Protocols for IVHS*. Tech. Report. Institute of Transportation Studies, University of California at Berkeley.
- Huth, Michael and Mark Ryan (2004). *Logic in Computer Science: Modelling and Reasoning about Systems*. Vol. 73. Cambridge University Press, pp. 45–85.
- Jackson, Daniel (2006). *Software Abstractions - Logic, Language, and Analysis*. MIT Press.
- Kastenberg, Harmen and Arend Rensink (2006). „Model Checking Dynamic States in GROOVE“. In: *Model Checking Software (SPIN)*. Vol. 3925. LNCS, pp. 299–305.
- Kehrer, Timo, Udo Kelter, and Gabriele Taentzer (2013). „Consistency-preserving edit scripts in model versioning“. In: *Proc. 28th IEEE/ACM Int. Conf. on Automated*

Bibliography

- Software Engineering (ASE)*. Ed. by Ewen Denney, Tevfik Bultan, and Andreas Zeller. IEEE, pp. 191–201.
- Klar, Felix, Alexander Königs, and Andy Schürr (2007). „Model transformation in the large“. In: *ACM Symposium on Foundations of Software Engineering (SIGSOFT)*. ACM, pp. 285–294.
- Koch, Manuel, Luigi V. Mancini, and Francesco Parisi-Presicce (2005). „Graph-based Specification of Access Control Policies“. In: *Journal of Computer and System Sciences* 71, pp. 1–33.
- König, Barbara and Javier Esparza (2010). „Verification of Graph Transformation Systems with Context-Free Specifications“. In: *Graph Transformations (ICGT)*. Vol. 6372. LNCS, pp. 107–122.
- König, Barbara and Vitali Kozioura (2008a). „Augur 2 – A New Version of a Tool for the Analysis of Graph Transformation Systems“. In: *Electronic Notes in Theoretical Computer Science* 211, pp. 201–210.
- König, Barbara and Vitali Kozioura (2008b). „Towards the Verification of Attributed Graph Transformation Systems“. In: *Graph Transformations (ICGT)*. Vol. 5214. LNCS, pp. 305–320.
- Kuhlmann, Mirco and Martin Gogolla (2012). „From UML and OCL to Relational Logic and Back“. In: *Model Driven Engineering Languages and Systems (MoDELS)*. Vol. 7590. LNCS, pp. 415–431.
- Kutz, Oliver, Janna Hastings, and Till Mossakowski (2012). „Modelling Highly Symmetrical Molecules: Linking Ontologies and Graphs“. In: *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*. Vol. 7557. LNCS, pp. 103–111.
- Lambers, Leen (2010). *Certifying rule-based models using graph transformation*. Südwestdeutscher Verlag für Hochschulschriften.
- Lara, Juan de and Hans Vangheluwe (2004). „Defining visual notations and their manipulation through meta-modelling and graph transformation“. In: *Journal of visual Languages and Computing* 15.3-4, pp. 309–330.
- Libkin, Leonid (2004). *Elements of Finite Model Theory*. Springer-Verlag Berlin.
- Linker, Sven (2015). „Proofs for traffic safety - combining diagrams and logic“. PhD thesis. Universität Oldenburg. URL: <http://oops.uni-oldenburg.de/2337/>.
- Loeckx, Jacques, Hans-Dieter Ehrich, and Markus Wolf (1996). *Specification of abstract data types*. Wiley.

- Löwe, Michael, Harald König, Christoph Schulz, and Marius Schultchen (2013). „Algebraic Graph Transformations with Inheritance“. In: *Formal Methods: Foundations and Applications*. Vol. 8195. LNCS. Springer-Verlag Berlin, pp. 211–226.
- Mandel, Luis and María Victoria Cengarle (1999). „On the Expressive Power of OCL“. In: *Formal Methods 1999, Toulouse, France, September 20-24, 1999*. Vol. 1708. LNCS, pp. 854–874.
- Manzano, María (2005). *Extensions of first order logic*. Cambridge University Press.
- Navarro, Marisa, Fernando Orejas, Elvira Pino, and Leen Lambers (2016). „A Logic of Graph Conditions Extended with Paths“. In: *Graph Computation Models (GCM 2016)*. To appear.
- Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer-Verlag Berlin. ISBN: 3540433767.
- Object Management Group (2003). *UML 2.0 superstructure final adopted specification*. OMG document pts/03-08-02.
- Object Management Group (2010). *Object Constraint Language, Version 2.2, OCL*.
- Orejas, Fernando (2011). „Symbolic Graphs for Attributed Graph Constraints“. In: *Journal of Symbolic Computing* 46.3, pp. 294–315.
- Pennemann, Karl-Heinz (2004). „Generalized Constraints and Application Conditions for Graph Transformation Systems“. MA thesis. Dept. für Informatik, Universität Oldenburg. URL: <http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/Penn04-Dipl.pdf>.
- Pennemann, Karl-Heinz (2008a). „An Algorithm for Approximating the Satisfiability Problem of High-level Conditions“. In: *Proc. Int. Workshop on Graph Transformation for Verification and Concurrency (GT-VC'07)*. Vol. 213. Electronic Notes in Theoretical Computer Science, pp. 75–94.
- Pennemann, Karl-Heinz (2008b). „Resolution-like theorem proving for high-level conditions“. In: *Int. Conf. on Graph Transformations (ICGT)*. Vol. 5214. LNCS, pp. 289–304.
- Pennemann, Karl-Heinz (2009). „Development of Correct Graph Transformation Systems“. PhD thesis. Universität Oldenburg. URL: <http://oops.uni-oldenburg.de/884>.
- Plump, Detlef and Annegret Habel (1996). „Graph Unification and Matching“. In: *Graph Grammars and Their Application to Computer Science*. Vol. 1073. LNCS, pp. 75–89.

Bibliography

- Poskitt, Christopher M. and Detlef Plump (2013). „Verifying Total Correctness of Graph Programs“. In: *Electronic Communications of the EASST* 61.
- Poskitt, Christopher M. and Detlef Plump (2014). „Verifying Monadic Second-Order Properties of Graph Programs“. In: *Int. Conf. on Graph Transformations (ICGT)*. Vol. 8571. LNCS, pp. 33–48.
- Radke, Hendrik (2013). „HR* Graph Conditions Between Counting Monadic Second-Order and Second-Order Graph Formulas“. In: *Electronic Communications of the EASST* 61.
- Radke, Hendrik, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer (2015). „Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations“. In: *Graph Transformations (ICGT 2015)*. Vol. 9151. LNCS, pp. 155–170.
- Rensink, Arend (2003). „Towards Model Checking Graph Grammars“. In: *Workshop on Automated Verification of Critical Systems (AVoCS)*. Tech. Report DSSE-TR-2003-2. University of Southampton, pp. 150–160.
- Richa, Elie, Etienne Borde, and Laurent Pautet (2015). „Translating ATL Model Transformations to Algebraic Graph Transformations“. In: *Proc. Theory and Practice of Model Transformations (ICMT)*. Vol. 9151. LNCS, pp. 183–198.
- Richa, Elie, Etienne Borde, Laurent Pautet, Matteo Bordin, and José F. Ruiz (2014). „Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation“. In: *Proc. MoDELS 2014*. Vol. 1277. CEUR Workshop Proceedings, pp. 34–43.
- Richters, Mark (2002). „A precise approach to validating UML models and OCL constraints“. PhD thesis. Universität Bremen, Logos Verlag, Berlin.
- Schürr, Andy (2001). „Adding Graph Transformation Concepts to UML’s Constraint Language OCL“. In: *Electronic Notes in Theoretical Computer Science* 44.4, pp. 93–106.
- Simmons, H. (2011). *An Introduction to Category Theory*. Cambridge University Press.
- Strecker, Martin (2008). „Modeling and Verifying Graph Transformations in Proof Assistants“. In: *Electronic Notes in Theoretical Computer Science* 203.1, pp. 135–148.
- Strecker, Martin (2011). „Locality in Reasoning about Graph Transformations“. In: *Applications of Graph Transformation With Industrial Relevance (AGTIVE)*. Vol. 7233. LNCS, pp. 169–181.
- Taentzer, Gabriele (2012). „Instance Generation from Type Graphs with Arbitrary Multiplicities“. In: *Electronic Communications of the EASST* 47.

- Varró, Daniel (2003). „Towards symbolic analysis of visual modeling languages“. In: *Electronic Notes in Theoretical Computer Science* 72.3, pp. 51–64.
- Varró, Dániel and András Balogh (2007). „The model transformation language of the VIATRA2 framework“. In: *Science of Computer Programming* 68.3, pp. 214–234.
- Wachsmuth, Guido (2007). „Metamodel Adaptation and Model Co-adaptation“. In: *21st European Conference on Object-Oriented Programming (ECOOP'07)*. Vol. 4609. LNCS, pp. 600–624.
- Winkelmann, Jessica, Gabriele Taentzer, Karsten Ehrig, and Jochen Malte Küster (2008). „Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars“. In: *Electronic Notes in Theoretical Computer Science* 211, pp. 159–170.

Symbol Glossary

2^M	The powerset of M	50
$ACLS$	A set of abstract classes	95
Appl	HR* condition ensuring a rule's applicability	87
$\Rightarrow_{x/R,y}$	direct derivation of hyperedge y along rule x/R	20
$\Rightarrow_{\mathcal{R}}^*$	derivation sequence along rules in rule set \mathcal{R}	20
$assoc$	Association function for classes	95
att_G	Hyperedge attachment function for graph G	16
$att(y)_i$	Designates the i^{th} tentacle of hyperedge y	16
$attr$	Attribution function	95
c	Denotes a graph condition	9
$h \circ g$	Composition of graph morphisms g and h	8, 17
CLS	A set of classes	95
Uncomp	Transforms a compact condition into a nested condition	106
Compress	Transformation for compacting HR* conditions	32
$Cond_{\mathcal{A}}$	Translate from \mathcal{M} - to \mathcal{A} -satisfiable HR* condition	41
$Cond_{\mathcal{M}}$	Translate from \mathcal{A} - to \mathcal{M} -satisfiable HR* condition	38
Cond	Translate node-counting MSO formulas to HR* conditions	53
$G \cong H$	Graphs G and H are isomorphic	8, 17
C	Transform a left application condition into a precondition	86
d	Denotes a graph condition	9
Decomp	Transformation for expanding HR* conditions	33
DSIG	A data signature for attributed graphs	102
D_G	Set of all items (i.e. nodes, edges and hyperedges) of graph G	16, 48
\mathcal{E}	Set of all surjective morphisms	8, 17
$\mathcal{E}'(a,b)$	Set of all jointly surjective morphisms for span (a,b)	38
$\mathcal{E}(P)$	Set of all surjective morphisms for an object P	38
$edge(e)$	formula for “ e is an edge”	49
$edg(x,y)$	formula for “there is an edge from x to y ”	49
\emptyset	Depending on context, an empty set, or the empty graph	7, 16
$ENUM$	A set of enumerations	95
Env	The environment of a meta-model instance	97
\mathcal{E}'	Set of all jointly surjective morphisms	18

Symbol Glossary

\doteq	$x \doteq y$ is the formula for “x equals y”	48
E_G	Set of directed edges of graph G	7, 16
$\exists F$	Existential closure of formula F	48
expr	An OCL expression	111
F	a logical formula over graphs	48
$\forall F$	Universal closure of formula F	48
$\text{Free}(F)$	Denotes that formula F is free	48
G	Denotes a graph	7, 16
\mathcal{G}	Set of all graphs without variables	7
\mathcal{G}_{Var}	Set of all graphs with variables over alphabet Var	16
GG	A graph grammar	12
$-$	Difference between two graphs	7, 16
$+$	Disjoint union of two graphs	7, 16
g^*	Symbol-wise extension for a graph morphism g	17
H	Denotes a graph	7, 16
id_G	Identity morphism for graph G	8, 17
$\text{inc}(e, x, y)$	formula for “ e is an edge from x to y ”	48
Integrate_e	Integrates edges into HR^* conditions	79
Integrate_n	Integrates nodes into an HR^* condition	75
$I[\mathbf{ex}]$	Interpretation of OCL expression \mathbf{ex}	97
$I(\mathbf{op})$	The semantics of an object model M	97
le_G	Labeling function for edges in graph G	7, 16
lab_b	$\text{lab}_b(x)$ is the formula for “ x has label b ”	48
lv_G	Labeling function for nodes in graph G	7, 16
ly_G	Labeling function for hyperedges in graph G	16
L	Transforms a right application condition into a left one	84
\mathbb{L}	An alphabet of node and edge labels	48
\mathbb{L}_E	Edge label alphabet for a graph	7, 16
\mathbb{L}_V	Node label alphabet for a graph	7, 16
\mathcal{M}	Set of all injective morphisms	8, 17
\models	$G \models f$ denotes that G satisfies (condition, formula) f	10, 23
$\models_{\mathcal{A}}$	Arbitrary satisfaction for HR^* conditions	34, 35
$\models_{\mathcal{A}}^r$	Arbitrary satisfaction with replacement for HR^* conditions	44
$\models_{\mathcal{A}}^s$	Arbitrary satisfaction with substitution for HR^* conditions	44
$\models_{\mathcal{M}}$	Injective satisfaction for HR^* conditions	35
\models_{cmp}	Satisfaction for compact conditions	107
\hookrightarrow	An injective morphism	8, 18
\mathbb{N}	The set of natural numbers, including 0	16
n_G	Naming function for nodes	106

$\text{node}(v)$	formula for “ v is a node”	49
$P \downarrow$	Execute program P as long as possible	13
Φ	A set of formulas attached to an attributed graph	102
$\text{Pin}(G)$	Set of pinpoints in a pointed graph	18
pin_G	Sequence of pinpoints in a pointed graph	18
x^\bullet	An x -labeled hyperedge and its attachment points	18
$\langle x \rangle$	The attachment points of an x -labeled hyperedge	18
\prec	Inheritance relation between classes	95
\preceq	The clan relation for classes	104, 105
$P; Q$	Sequential execution of programs	13
Pure	Removes containment operators in a nested condition	14
rank	The rank of an edge variable or a hyperedge	16
$\text{rank } G$	Number of pinpoints in a pointed graph	18
\mathcal{R}	Denotes a replacement system	19
$\mathcal{R}(x)$	set of all graphs derivable from x^\bullet by \mathcal{R}	20
Rep2Sub	Transform a condition from replacement to substitution	43
ρ	A graph transformation rule	10
$\llbracket F \rrbracket(\theta)$	The semantics of formula F under assignment θ	49–51
Shift*	Shifts a path-like HR* condition over a morphism	70
Shift	Shifts a nested condition over a morphism	13
Shift _{e}	Helper construction for Integrate _{e}	78
Shift _{n}	Helper construction for Integrate _{n}	75
Shiva	Construction that adds tentacles to a hyperedge	74
\mathcal{S}_{Assoc}	Represents links between objects in a meta-model	96
\mathcal{S}_{Att}	Assigns values to attributes of an object	96
\mathcal{S}_{Cls}	Assigns a set of object identifiers to a class	96
Σ	set of all substitutions induced by a set of rules	20
σ	A substitution	20
$\sigma(G)$	Application of substitution σ to graph G	20
G^σ	Application of substitution σ to graph G	20
x/R	Denotes a replacement pair	19
SOedg	represent a graph edge as a second-order formula	58
SOgra	represent a graph as a second-order formula	58
SOhyp	represent a graph hyperedge as a second-order formula	58
SONod	represent a graph node as a second-order formula	58
SORule	Represent a graph transformation rule as an SO formula	59
SOset	Represent a set of graph items as a second-order formula	59
SOsys	Represent a graph transformation system as an SO formula	59
sg	Maps an edge in graph G to its source node	7, 16

Symbol Glossary

$G \subseteq H$	Graph G is included in graph H	8, 17
Sub2Rep	Translate an HR* condition from substitution to replacement	44
\sqsubseteq	Containment operator for conditions	22
t_G	Maps an edge in graph G to its target node	7, 16
$\tau\{v/x\}$	Substitute all occurrences of v in β by x	97
tr	Translation from OCL constraints into graph conditions	111
\uplus	Disjoint union of two sets	7
\mathcal{V}_0	The set of individual variables in graph formulas	48
\mathcal{V}_1	The set of set variables in graph formulas	48
\mathcal{V}_2	The set of second-order variables in graph formulas	48
V_G	Set of nodes of a graph G	7, 16
Wlp	Constructs a weakest liberal precondition	88
Wp	Constructs a weakest precondition	88
Var	Set of variable names for hyperedges	16
Y_G	Set of hyperedges of graph G	16

Index

A

A-graph, 101, 102
A-graph morphism, 102
application (of an HR rule), 19
application condition, 10
 \mathcal{A} -satisfaction, 34
ATGI, 104
ATGI-graph, 104
ATGI-morphism, 104
attachment function, 16
attributed graph, 102
attributed morphism, 102
attributed type graph, 104
automorphism, 51

B

basic attributed graph, 102

C

car platooning, 25
clan morphism, 104
clan-disjoint, 108
closed formula, 48
CMSO formula, 50
codomain, 8, 17
comatch, 10
compact condition, 106
composition, 17
composition (of morphisms), 8
compressed normal form, 32
condition, 21
consistency conditions, 3, 14
constraint, 9, 22
containment operator, 21
contextual hyperedge replacement, 29
correctness, 88

cospan, 18

D

dangling condition, 11
data signature, 94
decompressed normal form, 33
derivation, 10, 20
direct derivation, 10, 20
discrete graph, 35, 53
domain, 8, 17
DSIG, 102

E

E-conditions, 3, 29
 \mathcal{E}' - \mathcal{M} pair factorization, 18
empty graph, 7
epimorphism, 8, 17
existential closure, 48

F

first-order formula, 48
FO formula, 48
full-containment normal form, 82

G

graph, 7, 16
graph constraint, 22
graph grammar, 12
graph morphism, 8, 17
graph program, 12
graph transformation rule, 10
graph transformation system, 12
graph with variables, 16

H

HR system, 15, 19

INDEX

HR* condition, 15, 21
HR conditions, 24
hyperedge replacement system, 19

I

identity morphism, 8, 17
inclusion, 8, 17, 102, 104
index (of a tentacle), 16
inheritance clan, 104
injective, 8, 17, 102, 104
invariant, OCL, 95
isomorphic, 8, 17
isomorphism, 8, 17

J

jointly surjective, 18

L

language, 12
liberal precondition, 88
logic on subobjects, 3, 29

M

match, 10
M-conditions, 3, 29
monadic second-order formula, 49
monomorphism, 8, 17
monotone, 24
morphism, 17
 \mathcal{M} -satisfaction, 23
MSO formula, 49
 μ -conditions, 3, 29

N

name function, 106
name-respecting, 106
negative application conditions, 3, 14
nested graph condition, 8, 14
Node-CMSO formula, 51
Node-MSO formula, 50

O

object model, 96
one-tentacle normal form, 58

P

partial morphism, 18
path-like, 67

pinpoints, 18
plain rule, 10
pointed graph, 18
programs with interfaces, 92

R

rank, 16, 48
replacement semantics, 43
rule, 10

S

satisfaction
 \mathcal{A} -satisfaction, 34
 by replacement, 44
 by substitution, 44
 decidability, 24
 \mathcal{M} -satisfaction, 23, 34
 of compact conditions, 107
 of HR* conditions, 23
 of nested conditions, 9
second-order formula, 51
Set condition, 35
set rule, 35
set-constructive, 121
shifting, 90
simple graph, 50, 104
SO formula, 51
span, 18
specification, 88
state correspondence relation, 115
strict, 104
substitution, 20
surjective, 8, 17
system state, 96

T

tentacle, 15
translation functions, 111
type-strict, 104

U

undirected induced graph, 67
universal closure, 48

W

weakest liberal precondition, 88
weakest precondition, 88

Curriculum Vitæ

- 18.03.2016 Defense of the dissertation
- 2012–2015 Research assistant in DFG project “Meta modeling and graph grammars”
at the Carl von Ossietzky University Oldenburg
- 2009–2011 Member of the Research Training Group “TrustSoft”
at the Carl von Ossietzky University Oldenburg
- 2008 Defense of Diploma thesis “COPPINRAD – An inductive Theorem Prover
based on the Connection Method”
- 2002–2008 Studies “Diplom Informatik”
at the University of Potsdam
- 2002 Alternative civilian service
- 2001 Abitur
- 05.11.1981 Born in Ludwigsfelde, Germany

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst und ausschließlich die angegebenen Hilfsmittel benutzt habe. Des weiteren erkläre ich, dass mir die Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg bekannt sind und von mir beim Anfertigen dieser Dissertation befolgt wurden. Ich versichere, dass ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste (Promotionsberatung) in Anspruch genommen habe.

I hereby confirm that I completed the work independently and used only the indicated resources. Furthermore, I confirm that I am aware of the guidelines of good scientific practice of the Carl von Ossietzky University Oldenburg and that I observed them while writing this dissertation. I did not use any commercial dispatching or counseling services concerning the dissertation.

Oldenburg, June 10, 2016

Ort, Datum

Unterschrift