UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR ELEKTROTECHNIK , INFORMATIK UND MATHEMATIK
INSTITUT FÜR INFORMATIK
FACHGEBIET SPEZIFIKATION UND
MODELLIERUNG VON SOFTWARESYSTEMEN
WARBURGER STRAßE 100
33098 PADERBORN

Verification of Infinite-State Graph Transformation Systems via Abstraction

DISSERTATION

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Dipl. Inform. Dipl. Math.Dominik Steenken

Paderborn, im December 2014

Betreuer: Professor Dr. Heike Wehrheim Dominik Steenken

Verification of Infinite-State Graph Transformation Systems via Abstraction

Abstract

In the field of model-driven software development (MDSD), the use of visual modeling languages is abundant. One particular application of such modeling languages is the specification of structurally dynamic systems, i.e. systems that are able to dynamically change their structural configuration at run-time. To be useful, such systems must adhere to certain quality constraints, such as the inability to produce configurations that are deemed dangerous.

If MDSD is to be a viable alternative to traditional software development, it must be firmly grounded in formal methods, so as to support the automatic checking of such quality constraints at the model level. The natural formalization of structurally dynamic systems is the mathematical notion of graph transformation systems. The formal verification of such systems poses a challenge, since they often exhibit infinite state spaces, i.e. they are able to generate an unbounded variety of different configurations.

In this thesis, we develop a new way to check infinite-state graph transformation systems for their capacity to produce potentially dangerous configurations using finite abstractions. In order to do so, we adapt established methods from other fields of verification to the graph transformation domain. To create the abstraction itself, we use three-valued abstractions common in shape analysis methods to produce abstract graph transformation systems that can be refined using first-order logic. The construction of the state space is handled in a lazy fashion, as has been done in many model checking tools, where we only look at those parts of the state space in detail that have a chance of containing the error. We then employ the concepts of counterexample-guided abstraction refinement and interpolation, commonly used to great effect in the verification of sequential programs, to semi-automatically construct abstraction refinements where needed. This is based in a complete transformation of the graph concepts used in this thesis into first-order logic with uninterpreted functions, which enables us to take advantage of the recent impressive progress in the field of SMT solving.

The result is a method that allows for a highly flexible analysis of infinite-state graph transformation systems and thus represents a contribution to the verification of complex, structurally dynamic systems in the context of MDSD. We present a proof-of-concept implementation and report on first experimental results.

Betreuer: Professor Dr. Heike Wehrheim Dominik Steenken

Verification of Infinite-State Graph Transformation Systems via Abstraction

Zusammenfassung

Im Bereich der modellgetriebenen Softwareentwicklung ist der Gebrauch von visuellen Modellierungssprachen weit verbreitet. Eine Anwendung solcher Modellierungssprachen ist die Spezifikation von strukturell dynamischen Systemen, also solchen Systemen die in der Lage sind, ihre strukturelle Konfiguration zur Laufzeit zu ändern. Um nützlich zu sein, müssen solche Systeme bestimmten Qualitätsanforderungen genügen, wie etwa die Eigenschaft, dass potenziell gefährliche Konfigurationen von dem System nicht generiert werden können.

Wenn modellgetriebene Softwareentwicklung eine gangbare Alternative zur klassischen Softwareentwicklung darstellen soll, muss sie mit formalen Methoden untermauert werden, um eine automatische Überprüfung solcher Qualitätsanforderungen auf der Modellebene zu ermöglichen. Die naheliegendste Formalisierung von strukturell dynamischen Systemen ist das mathematische Konzept der Graphtransformationssysteme. Die formale Verifikation solcher Systeme stellt eine Herausforderung dar, da sie häufig unendlich große Zustandsräume aufweisen, d.h. sie sind in der Lage, eine unendliche Vielfalt unterschiedlicher Konfigurationen zu erzeugen.

In dieser Arbeit entwickeln wir eine neue Methode um Graphtransformationssysteme mit unendlichem Zustandsraum mittels einer endlichen Abstraktion auf ihre Fähigkeit, potenziell gefährliche Konfigurationen zu erzeugen, zu überprüfen. Um dies zu erreichen, adaptieren wir etablierte Methoden aus anderen Bereichen der Verifikation für die Graphtransformationsdomäne. Um die Abstraktion selbst zu realisieren, verwenden wir dreiwertige Abstraktionen, die in Shapeanalysemethoden häufig verwendet werden, um ein abstraktes Graphtransformationssystem zu erstellen welches wir mittels Prädikatenlogik erster Ordnung verfeinern können. Der Aufbau des Zustandsraum wird nach dem "Lazy" Paradigma gehandhabt, wie es in vielen Modelchecking Werkzeugen umgesetzt wird. Auf diese Weise können wir es vermeiden, Teile des Zustandsraums, die keine Fehler enthalten, detailliert betrachten zu müssen. Abschließend verwenden wir das Konzept der Gegenbeispiel-geleiteten Abstraktionsverfeinerung mittels Interpolation, häufig erfolgreich eingesetzt bei der Verifikation sequenzieller Programme, um halbautomatisch Abstraktionsverfeinerungen erzeugen zu können, wo sie benötigt werden.

Betreuer: Professor Dr. Heike Wehrheim Dominik Steenken

Das Ergebnis ist eine Methode die eine hoch flexible Analyse von Graphtransformationssystemen mit unendlichem Zustandsraum erlaubt. Sie stellt daher einen Beitrag zum Gebiet der Verifikation komplexer, strukturell dynamischer Systeme im Kontext der modellgetriebenen Softwareentwicklung dar. Wir stellen eine prototypische Implementation, sowie erste experimentelle Ergebnisse vor.

Acknowledgments

First of all, i would like to thank my supervisor, Prof. Dr. Heike Wehrheim, for the support and guidance she has given me and for the opportunity to work in an extraordinary research group. Further, i would like to thank Prof. Dr. Holger Giese for taking the time to review my thesis, and Prof. Dr. Wilhelm Schäfer, Prof. Dr. Eyke Hüllermeier, and Dr. Theo Lettmann for attending my exam.

Special thanks go to Galina Besova, for many valuable exchanges about our respective research, and to Daniel Wonisch and Tobias Isenberg, who directly contributed to my research. I further want to give thanks to the remaining current and former members of our research group, Dr. Björn Metzler, Dr. Thomas Ruroth, Dr. Nils Timm, Steffen Ziegert, Sven Walther, Oleg Travkin, and Marie-Christin Jakobs for providing a collegial and open environment. I would also like to thank my student assistants, Manuel Töws and Patrick Schleiter, who supported me in developing the early versions of what became the implementation for this thesis.

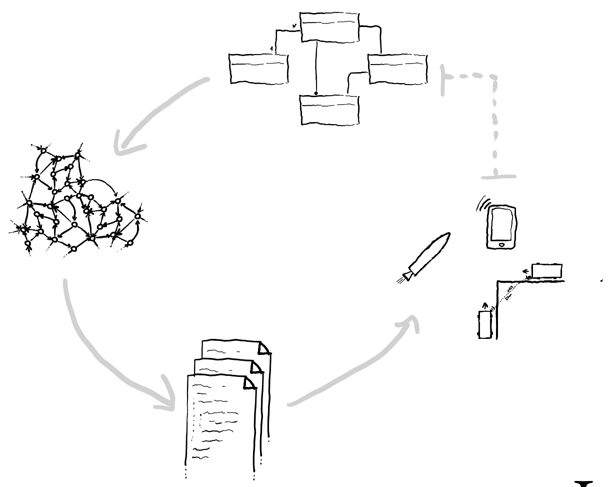
I thank Claudia Priesterjahn, Christian Heinzemann, Kathrin Flaßkamp, Martin Krüger, and Oliver Sudmann for a good and interesting collaboration within the Collaborative Research Center 614. I also thank all other members of the CRC for providing me the opportunity to work in a very interdisciplinary environment. The discussions i had with my colleagues in mechanical and control engineering helped me appreciate the value of collaborating outside of one's field.

Additionally, i thank Dr. Thomas Wahl, Dr. Philipp Rümmer, and Prof. Dr. Daniel Kröning for a very instructional research stay at Oxford University and for introducing me to new fields of study.

Finally, i thank my parents for providing me with the opportunity to study Computer Science and eventually write this dissertation. Thank you for having the patience to let me explore both my interests in computer science and math.

Contents

Listing of figures



1 Introduction

The ever growing complexity and interconnectedness of the software that increasingly runs in and on every aspect of our world poses a challenge to the discipline of software engineering. It means that ever more complex software will, over time, have to adhere to ever more stringent quality standards. This problem is compounded by the insight that this complexity growth will always out-pace the classical approach of creating software more or less directly from a textual specification and only analyzing the final product. The process of solving these problems led to a complete rethinking of the software development process, culminating in the idea of *Model Driven Software Development (MDSD)*[5, 31, 43], an approach that represents a paradigm shift in the way software is created.

In this approach, software is formally specified using *modeling languages*, often geared specifically to the problem domain for which the software is intended. These software models can then be verified to exhibit whatever quality criteria are appropriate for the particular use cases the software is being designed for. This is an improvement over the status quo since, as highly complex, Turing complete formalisms, regular full-scale programming languages produce programs that are very hard to verify. In contrast, models created with custom-built modeling languages, which are often appropriately restricted in scope, can be much more amenable to verification. These easier-to-verify models are then transformed into programs in classical programming languages, using fully-automatic transformations, yielding fully realized software with guaranteed quality characteristics – provided the transformations are correct.

This promise of model driven software development can only be delivered upon through the use of formal methods. Formal methods in the context of software engineering are all methods and concepts that strive to provide a solid, rigorous mathematical foundation for the specification, design, and analysis of software. They are not limited to software – formal methods have been very successfully applied to the field of hardware[82], and even completely outside the field of information technology, such as in mechanical engineering, or electrical engineering[87].

For software, formal methods come in three broad groups: formal specification, formal modeling, and formal verification[158]. First, formal methods for specification enable the mathematically rigorous expression of the *intent* of the software, i.e. they encode (part of) its purpose in terms of restrictions on its outputs and inputs and can express functional (or even some non-functional) properties that the software must have. Next, formal methods for the modeling of software focus on providing mathematical formalisms that can express certain aspects of the system itself, like its structural composition, or its dynamic behavior, or even unify all relevant aspects of the system in one model. Finally, formal methods for the verification of software combine the first two to start from a formal specification of the properties of a system and a formal model of that system, to use a mathematically sound process to prove or disprove that the given software satisfies the given specification.

If model driven software development is to be established as a serious alternative to the classical approach, it needs to leverage these formal methods as much as possible in order to provide a true advantage. The benefits are clear – in a fully realized MDSD process, software systems are easier to create, due to more domain-specific models and automated support based on the mathematical foundation of the modeling language, and exhibit a much higher quality, because most errors can be caught in the modeling phase by automated processes that understand the semantics of the model, not just the syntax.

This last aspect, the employment of formal verification to reduce errors, is central. Examining the literature, we can find many different formal methods to realize this. One approach is *program derivation*[72, 138, 153], which aims to directly create (a model of) a system from its specification, thus providing correctness by construction. There are also methods that focus on issues external to the development process itself, such as providing the establishment of trust between different providers of software, supported by *proof carrying code*[119], where a supplier can augment program code with correctness proofs which can then be verified by the recipient. But the most prevalent approaches, and the ones most closely associated with the term "formal verification", are *deductive verification*[13] and *model checking*[53]. Deductive verification means the use of logical inference rules to derive from a model of a system its adherence to the specification. Model checking, on the other hand, is a technique that uses a formal representation of a model to essentially simulate all of its possible executions, and then checks the result against the formal specification.

All these techniques are predicated on the use of formal mathematical models for the representation of systems. Usually, this is achieved by creating a textual or visual *modeling language* and defining its semantics by mapping the models it creates to certain mathematical objects. Mathematical concepts that are frequently used for this purpose include *automata*, both in their basic finite form and in extended forms, such as *timed automata*[11] or *hybrid automata*[88], and also *Petri nets*[116], *process calculi*[137], and *(well-structured) transition systems*[7]. There are many modeling languages that are based on this basic mathematical framework, such as Promela[28] or CSP[92], hardware description languages like VHDL[118], or even high-level functional programming languages, such as Haskell[95][*], as well as consistent groups of modeling languages, such as the UML[1] that mix formal models with languages whose semantics are specified in natural language, i.e. informally.

This abundance of modeling languages and the supposition that domain specific modeling languages will aid in the description of systems lead to a problem – all these languages have to be learnt by the designers, which, given their prospective numbers can incur large costs. One way to combat this is to use modeling languages that are as intuitive and clear as possible, which, given that most of them are limited in scope, is a more realistic proposition for these languages than for, e.g., regular programming languages. *Graphical* or *visual modeling languages* go a long way towards this goal, exploiting human spatial intelligence and pattern recognition to present the content of the model in a more concise, understandable way[79]. Such languages consist of visual abstractions of the systems to be modeled in order to make it easier for a human designer to grasp the "big picture" described by a model. They tend to be very intuitively understandable and usually require comparatively little effort to learn. They are particularly suited to model aspects of systems that can be interpreted as consisting of discrete entities and relationships between those entities.

Such graphical modeling languages are at the core of the current effort to establish model driven software development as an alternative to the traditional software development framework, and thus formal methods supporting their use are of great importance. Examples of such modeling languages include, e.g., the UML[1], SysML[2], MechatronicUML[26, 43] or Matlab/Simulink[12]. Of particular interest are those languages that are designed to model systems that exhibit autonomous structural reconfiguration, such as self-healing[73, 122] or self-optimizing[4] systems. Such systems often exhibit complex behavior at runtime that is not easily predictable at design time, even though they are in many cases deployed in highly safety-critical environments[149].

[*]Full programming languages are not usually grouped under modeling languages, but since the mathematical foundations of functional programming languages are particularly rigorous, they were included here.

As argued above, the application of formal methods to the use of these languages goes beyond simple syntax checking and specifically includes automatic verification and validation of models. Automation, in turn, requires precise mathematical definitions of the syntax and semantics of the visual modeling languages used. Such definitions make extensive use of the mathematical concept of *graphs*, and, more specifically, *graph transformations*. The verification of properties of such systems is the subject of this thesis.

Multiple approaches have been proposed to approach this topic (see Sec. 8.1). So far, none of them has emerged as a complete solution, and every approach provides a different set of benefits using a different trade-off in expressiveness, decidability, or time and space complexity. With this thesis, we hope to present a new approach to the problem that could provide new impulses toward a solution.

1.1 Problem Definition

Graphical modeling languages are such languages that, in their most reduced form, can be expressed as graphs, mathematical objects consisting of *nodes* and labeled *edges* between the nodes. In a graphical model, nodes are usually used to represent *entities* in a system. This could, for example, be a class or an object in a UML diagram, a logic gate in a circuit layout, a table in a database schema diagram, or a host in a network diagram. The edges in a graphical model are then taken to represent some kind of *relationship* between two (or sometimes more) entities. Examples include generalization arrows in UML class diagrams, connections in circuit layouts, relations in database schema diagrams, and network links in network diagrams.

In order to fix the problem definition for this thesis, we need to differentiate between the various uses that graphs are put to. The most common use of graphs is in *static structural models*, i.e. models that represent either fixed structural relationships between entities, such as class diagrams[3], entity-relationship models[45], or meta-models[5], or momentary snapshots of fluid relationships, such as object diagrams[3]. While such models can also exhibit properties that must be formally verified[†], they are not the focus of this thesis.

There a two main fields in which graphs are also used in dynamic settings – model transformations and behavior modeling.

Model transformation[5] is the transformation of models belonging to a well-defined modeling language into other models of a different (or possibly the same) modeling language. The importance of such transformations for model driven software development cannot be overstated. A consequence of the application of more domain-specific and thus less generic modeling languages is the fact that such languages are

[†]such as the feasibility of OCL-enhanced class diagrams[154]

often employed in contexts where their scopes overlap, i.e. where aspects of one model are relevant in a different model. Also, the same modeled system might be analyzed or otherwise acted upon in different contexts in the development process, requiring different models and thus transformations that map between them. Applications in the domain of model transformations often use relatively complex graph transformation formalisms (such as Triple Graph Grammars[80, 142]) and employ extensions of the graph formalism itself, such as attributes. While the field of model transformations can also benefit from the verification of graph transformation systems, we want to focus on a different application in this thesis.

Behavior modeling refers to the activity of using (graphical) modeling languages to define how a given system can evolve over time. The resulting models can focus on the evolution of the internal state of a system, or the exchange of messages between systems over time, such as in activity diagrams and state charts[3], or in (timed) automata networks[29]. They can also model the *structural evolution* of a system over time[71], by describing how the relationships between the entities that the graphs concerned are made of can change. Graph transformation systems are the ideal mathematical formalism to model this, and since it has been shown that models like activity diagrams and state charts can also be modeled by them, we consider them a general-purpose formalism for behavior modeling. Once a behavior model has been created, it must be verified against the requirements implied by its later use.

At the highest level of abstraction, a graph transformation system consists of a single graph indicating the initial state of the system, as well as a (finite) number of graph transformation rules, that describe what kind of actions the system can perform that affect its structural composition, i.e. the graph that models its current state. A rule consists of a left-hand side graph and a right-hand side graph. These describe the effect of the rule by providing quite literally a before and an after picture of the affected part of the graph. One applies a rule by finding a part of the graph modeling the current state of the system that looks like the left-hand side of the rule, and replaces that part with the right-hand side of the rule.

This basic concept of a graph transformation system has no notion of execution control – rules can be (and are) applied whenever it is possible to apply them. There are extensions of this formalism that allow for prioritization among the rules[75], or even complete control flow specifications[71]. However, for this thesis we will focus on the basic formalism. In this formalism, the semantics of a graph transformation system are given by all graphs that can possibly be created from the initial graph by applying sequences of graph transformation rules. This set of graphs then represents, in a sense, the set of all possible configurations that the modeled system can have. The question then becomes whether all of these configurations, or sequences of configurations, are intended by the designer, or if some of them lie outside the specification. This question is often exceedingly difficult to answer since the number of possible configurations is usually extremely large, and in many important cases even infinitely large, removing all hope to actually enumerate all possible configurations.

There are many kinds of properties that could be verified for graph transformation systems (as for any other kind of behavior model). A very broad categorization[143] splits them into *safety properties* (making sure that bad things never happen) and *liveness properties* (making sure that, eventually, good things happen). In this thesis we will focus on safety properties, more specifically, on the so-called *coverage problem*. This kind of property identifies a particular kind of sub-configuration of a system to be dangerous, and then sets out to prove that the system can never produce a configuration that contains, i.e. "covers", that dangerous sub-configuration.

1.2 Contributions

For generic graph transformation systems, being a Turing-complete formalism[‡], the coverage problem is undecidable. This is related to the aforementioned fact that many graph transformation systems have the ability to generate arbitrarily large sets of non-isomorphic graphs, and thus induce infinite state spaces. Nevertheless, methods can be designed to provide solutions for it in special cases (see Sec. 8.1). In this thesis, we seek to develop a new method of solving the coverage problem for graph transformation systems.

The main contribution to the field of model driven software development thus lies in the exploration of a new verification technique for visual modeling languages describing dynamic behavior. As the chief obstacle in verifying models in such languages usually consists of the infinite variety of structurally-distinct states they generate, we will provide a verification technique that is able to handle infinite state spaces. This is achieved by a novel approach to the verification of infinite-state graph transformation systems.

In order to create this new approach, this thesis draws on successful techniques that are already being employed in other subfields of software verification, and applies them in the graph transformation context. In the following, we will name the individual main contributions of this thesis and describe them briefly. The necessary correctness proofs are assumed to be implied by each of the contribution descriptions.

Firstly, we provide an abstract graph formalism based on three-valued logic inspired by the work of Sagiv et.al.[134] on shape analysis. This results in the ability to "generalize" graphs by summarizing subgraphs using so-called "summary nodes". This basic principle is illustrated in Fig. 1.1. We further create a way of finely tuning this abstraction using first-order logical formulas[144]. These two contributions together enable the construction of sound abstractions of the state spaces induced by behavior models

[‡]since a GTS can easily simulate a Turing machine

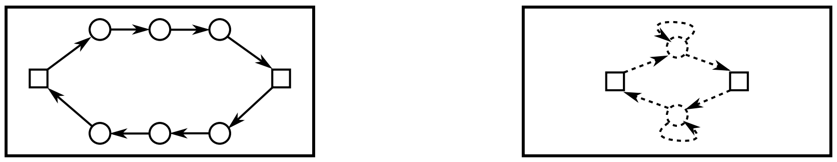


Figure 1.1: Abstraction Illustration

in visual languages. Any verification results for the coverage problem obtained for the abstraction automatically hold for the original system as well. The use of first-order logic for abstraction refinement further means that a wide variety of domain knowledge can be utilized to refine the abstraction if necessary. While the graph abstraction itself is nearly identical to the original shape analysis, the application of regular transformation rules to the abstract graphs, as well as the automatic maintenance of the abstraction refinements goes significantly beyond that.

Secondly, we provide a way to facilitate the detection of abstraction errors, i.e. instances where the abstraction glosses over important aspects of the system, as well as analytical tools to detect unsound abstractions and superfluous abstract states. We achieve this by encoding graphs, the membership in the represented graph set of an abstract graph, as well as sequences of transformation rule applications guided by abstract graph sequences in first-order logic. These contributions are based on prior work on bounded model checking[97], but are modified and extended to cover a much wider range of applications. We also provide specific implementation details to enable the implementation of these encodings in the well-established language SMTLib to facilitate the employment of state-of-the-art solvers. This will allow any verification technique integrated into an MDSD process that is based on the techniques presented in this thesis in the short term to rely on readily available, off-the-shelf software for its implementation, and in the mid- and long-term to benefit from the rapid growth in efficiency currently present in the field of SMT solving.

Thirdly, we provide a fully automatic state space construction algorithm that is amenable to lazy refinement strategies, and a semi-automatic abstraction refinement algorithm. These two algorithms make heavy use of the encodings mentioned above. By adopting the lazy abstraction paradigm, we ensure that the construction of the state space is only explored in detail where absolutely necessary, leaving safe portions of the state space at as high an abstraction level as possible. By using a semi-automatic refinement scheme we combine the convenience and easy applicability of an automatic approach for the common case with the analytical power of a human designer for special cases where the automation is not sufficient.

Finally, we describe a prototypical implementation of these algorithms and report on some experimental results. The implementation is modeling language-agnostic and operates on a purely conceptual level. This serves to better illustrate the results of the thesis and also underscores the wide range of applicability of the general approach – it can be applied to any behavioral modeling language that models structural adaptation.

1.3 Thesis Outline

The remainder of this thesis retraces the important steps in developing our approach. This leads to an outline where each successive chapter adds to and builds on the concepts introduced in the chapter before, while after each chapter the approach as described so far is closed in itself, if incomplete – it can be used to solve some problems, but relies on the remaining chapters to make it useful.

We begin by introducing our modeling formalism, i.e. basic graphs and graph transformation systems in Chapter 2, and describe its semantics that we will base our formal analysis on. There we also describe how the model checking approach to verification can be applied to graph transformation systems, by creating graph transition systems.

We build on this to then introduce our abstraction approach in Chapter 3. This allows us to tackle the problem of infinite state spaces by providing a way to construct finite abstractions of the state space in such a way that it represents a sound approximation with respect to the coverage problem. Working from a template provided by shape analysis for pointer structures[134], we create an abstract graph transformation formalism and use it to define shape transition systems in analogy to the graph transition systems encountered before. We also provide an abstraction refinement scheme based on first-order logic that can be used to tune the abstraction to include or exclude certain pieces of information.

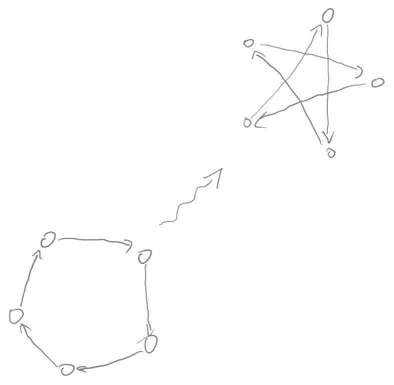
The abstractions introduced in this chapter solve the problem of infinite state spaces, but introduce new ones, such as the possibility of false error reports (known as *spurious counterexamples*), and vacuous states, i.e. abstract states that do not represent any actual states. Motivated by this, in Chapter 4 we provide analysis tools that can detect these problems and be utilized to alleviate them. Specifically, we create encodings of graphs and graph transformations into first-order logic that can be used to identify spurious counterexamples produced by the analysis, as well as contradictory refinements.

Chapter 5 then introduces a new state space construction algorithm that is designed to overcome some of the weaknesses of the approach from Chapter 3 and provide a framework specifically designed to take advantage of the encodings developed in Chapter 4. The algorithm described here is complete with the exception of abstraction refinement, and can thus verify relevant examples where the initial abstraction is suitable to prove the safety property.

An algorithm for abstraction refinement that fits into this state space construction technique is then introduced in Chapter 6. It is a semi-automatic approach, containing an interpolation-based algorithm, an automatic backup algorithm, and finally a framework for interaction with a human designer in case both of these attempts fail.

Chapter 7 then describes the prototypical implementation, called SGA, that was created as part of this thesis. This implementation uses a pure graph formalism and is thus not bound to any particular modeling language. This serves its purpose as a demonstration of the abstract concepts described in this thesis and makes it easy to transfer it to any modeling language that models structural changes in graphical models based on the structure of those models. SGA is then applied to example problems in order to show state space construction and abstraction refinement in action, as well as to point out the remaining weaknesses of the approach.

Chapter 8 then concludes with an overview over related work, as well as an exploration of possibilities for future work on our approach.



2 Basic Definitions for Verifying Graph Transformation Systems

Graphs and graph transformations are central to graphical modeling languages. They are necessary to provide a proper mathematical background upon which automated tools can be built. In this chapter, we will introduce the notions of graphs and graph transformations that we will use as the basis for this thesis. We will also describe the basic approach to verifying graphical models and show why it is insufficient.

Graphical models generally model two different aspects of a system - structural aspects and behavioral aspects. Structural models describe the static structure of a system. They can provide a high-level overview of relationships between elements or element types in the model (e.g. a UML[3] class or component diagram) or a snapshot of a state of the model in its runtime evolution (e.g. a UML object diagram). Since what is expressed in these models are typed relationships between distinct elements, graphs are the ideal mathematical concept to define their semantics.

Behavioral models, on the other hand, describe how the modeled system evolves over time. This can mean the change of system states (e.g. in a UML statechart), or structural change in a system (e.g. in a MechatronicUML[26] story diagram). When structures are modeled using graphs, then modeling structural changes requires a mathematical concept capable of expressing changes in graphs.

There are many approaches to mathematically define the transformation of graphs into other graphs. Examples of such approaches include Node Label Controlled (NLC)[70] transformation, Hyperedge Replacement Grammars (HRG)[64], and Triple Graph Grammars (TGG)[142].

In this thesis, we will focus on the algebraic approach to graph transformation[67] and its use for the modeling and verification of systems that undergo structural transformations as part of their evolution. Modeling based on algebraic graph transformation is already part of some graphical modeling languages (such as MechatronicUML), and some verification techniques for it have already been developed. However, as we will see at the end of this chapter, the straightforward approach to verify such models quickly runs into problems.

This chapter provides the basic definitions on which the approach presented in this thesis is built. The definitions closely follow or derive from the standard definitions of graphs and transformation as defined by Rozenberg et.al.[133]. The verification process described at the end of this chapter amounts to the basic technique employed by the GROOVE-tool[75].

We begin with the most basic notions, i.e. graphs and relationships between graphs.

2.1 Graphs

The graphs we use in this thesis are *labeled*, meaning their edges are annotated with symbols taken from a finite set fixed at the beginning of modeling. Any kind of set can be used as a label set, but it must be partitioned into *unary labels* and *binary labels*.

Definition 1 (Label Set). *A* label set $L = (U_L, B_L)$ *is a pair of two sets of symbols. They are called the set of* unary symbols U_L*, and the set of* binary symbols B_L*. If L is clear from context, we use the short forms U and B.*

Having established the label set, we can now define our notion of labeled graphs. It is the basic notion of a simple graph, with the addition of unary edges, i.e. edges that are only connected to one node. The definition makes the expected restriction that unary edges must have unary labels and binary edges must have binary labels. This extra modeling effort is made in order to be able to distinguish between a binary edge that only happens to connect a node to itself, and a unary edge that is meant to only apply to one node. This will become important in Chapter 3.

Definition 2 (Labeled Graph). *A* labeled graph G *over a label set L is a triple* $G = \big(N, E^1, E^2\big)$*, where*

$$\begin{aligned} &N_G && \textit{is the } \text{set of nodes} \\ &E^1_G \subseteq U_L \times N && \textit{is the set of } \text{unary edges} \end{aligned}$$

$\mathrm{L}=(\{\mathrm{u}_1,\ \mathrm{u}_2\},\ \{\mathrm{b}_1,\ \mathrm{b}_2\})$

$$\begin{aligned}
&\mathrm{sample\_graph} = \{\\
&\quad \mathrm{N} = \{\mathrm{n}_1,\ \mathrm{n}_2,\ \mathrm{n}_3\}\\
&\quad \mathrm{E}^1 = \{(\mathrm{u}_1,\mathrm{n}_1),(\mathrm{u}_1,\mathrm{n}_2),(\mathrm{u}_2,\mathrm{n}_3)\}\\
&\quad \mathrm{E}^2 = \{(\mathrm{n}_1,\mathrm{b}_2,\mathrm{n}_2),\ (\mathrm{n}_2,\mathrm{b}_1,\mathrm{n}_3),\\
&\qquad\qquad (\mathrm{n}_3,\mathrm{b}_1,\mathrm{n}_1),(\mathrm{n}_1,\mathrm{b}_2,\mathrm{n}_3)\}\\
&\}
\end{aligned}$$

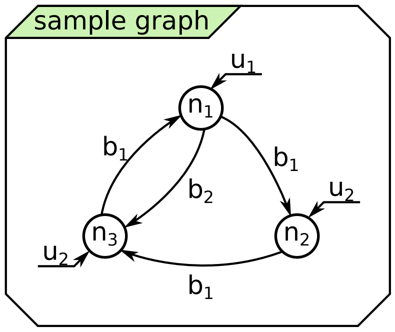


Figure 2.1: A sample graph and its visual representation

$$E_G^2 \subseteq N \times B_L \times N \qquad \textit{is the set of } \text{binary edges}$$

If the graph is clear from context, we refer to these parts as N, E^1 and E^2 respectively. Where appropriate, individual unary edges are denoted e^1, binary edges are denoted e^2, while e denotes an edge without a specified arity.

In the following, we will use “graph” as a shorthand for “labeled graph”. The purpose of a graph is to define relationships between its nodes and edges. These relationships are called *adjacency* and *incidence*.

Definition 3 (Adjacency and Incidence). *Let $G = (N, E^1, E^2)$ be a graph. Two nodes $n_1, n_2 \in N$ that are connected by an edge $(n_1, b, n_2) = e^2 \in E^2$, labeled b, are said to be* b-adjacent*, or just adjacent if the label is irrelevant. Any edge e is said to be* incident *to its node(s). This is denoted $n \in e$ for each node n in e. Two edges that connect to the same node are said to be incident to each other.*

Every graph has a visual representation. This representation shows each node as a circle annotated with its name (e.g. n_1). It further presents each unary edge (u, n) as a u-labeled arrow originating close to and terminating at the circle representing n. Finally, it draws a b-labeled arrow for each binary edge (n_1, b, n_2), originating at the circle representing n_1 and terminating at the circle representing n_2. Figure 2.1 shows a sample graph and its visual representation.

It is sometimes necessary to look at only a part of a given graph, instead of the whole graph. The following definitions thus introduce two slight variations on this concept.

Definition 4 (Induced Graph). *Given a graph G, every subset $N \subseteq N_G$ of its node set* induces *a new graph. The graph induced by G and N is defined $G{\downarrow}_N = (N,\, E_G^1{\downarrow}_N\,,\, E_G^2{\downarrow}_N)$,* *where*

$$E^1_G\!\downarrow_N := \left\{(u,n) \mid (u,n) \in E^1_G \wedge n \in N\right\}$$
$$E^2_G\!\downarrow_N := \left\{(n_1,b,n_2) \mid (n_1,b,n_2) \in E^2_G \wedge n_1,n_2 \in N\right\}$$

An induced graph is thus defined by a subset of its nodes and all edges incident to (only) those nodes. In many cases, one also wants to restrict the edge set. For this reason, the notion of a subgraph is useful.

Definition 5 (Subgraph). *A graph G is said to be a* subgraph *of a graph H, written $G \leq H$, iff $N_G \subseteq N_H$, $E^1_G \subseteq E^1_H$, and $E^2_G \subseteq E^2_H$.*

Having defined graphs and ways to identify parts of them, we can now begin to define how a graph can be transformed into another. The basic idea of transforming graphs can be intuitively understood to involve the finding of parts of a graph that are "similar" to a given graph, and then swapping out that part for a different graph. Thus, we need a more formal definition of *similarity* between (parts of) graphs.

This is provided by a notion from category theory, called a *morphism*. In this thesis, we will abstract from most aspects of category theory that make up the mathematical foundation of algebraic graph transformation. The notion of a morphism, however, is so central that it cannot be omitted.

Definition 6 (Morphism). *Given two mathematical objects X and Y, a* morphism *$f : X \to Y$ is a* structure-preserving *mapping from X to Y. The precise meaning of "structure-preserving" is dependent on the domain of X and Y. The set of all morphisms between X and Y is denoted $\mathcal{M}(X,Y)$.*

The general idea of a morphism carries over to graphs (and the category of graphs) in the obvious way.

Definition 7 (Graph Morphism). *Given two graphs G and H over a common label set L, a* graph morphism *$f : G \to H$ is a total function between the node and edge sets of G and H, such that*

$$\forall (u,n) \in E^1_G : f((u,n)) = (u, f(n)) \qquad \textit{and}$$
$$\forall (n_1,b,n_2) \in E^2_G : f((n_1,b,n_2)) = (f(n_1), b, f(n_2))$$

G is called the source graph *of the morphism, while H is called the* target graph.

Thus, the structure preserved by a graph morphism is the adjacency between nodes defined by the edge sets. If two nodes are connected by an edge with a certain label in the source graph, their images in the target graph must be connected by the image of that edge. These conditions can be interpreted constructively, as the following definition shows.

Definition 8 (Induced Graph Morphism). *Let G and H be graphs, and let $f : N_G \to N_H$ be a total function.* f induces the graph morphism $\hat{f} : G \to H$ *by setting*

$$\begin{aligned} \hat{f}(n) &:= f(n) && \forall n \in N_G \\ \hat{f}((u,n)) &:= \left(u, \hat{f}(n)\right) && \forall (u,n) \in E^1_G \\ \hat{f}((n_1,b,n_2)) &:= \left(\hat{f}(n_1), b, \hat{f}(n_2)\right) && \forall (n_1,b,n_2) \in E^2_G \end{aligned}$$

if $\hat{f}\left(E^1_G\right) \subseteq E^1_H$ and $\hat{f}\left(E^2_G\right) \subseteq E^2_H$.

The existence of a graph morphism between two graphs G and H intuitively states that G is "similar" to (a part of) H. A direct corollary from this definition is the concept of a graph isomorphism.

Definition 9 (Graph Isomorphism). *Given two graphs G and H, a function $f : G \to H$ is a* graph isomorphism *between G and H iff f is bijective, f is a morphism, and f^{-1} is a morphism as well.*

This is written $G \equiv_f H$, or simply $G \equiv H$ if f is of no consequence. G and H are then said to be isomorphic.

While two isomorphic graphs are usually not literally the same graph, their only difference lies in the identity of their nodes. None of the concepts presented here depend in any way on that identity, so treating isomorphic graphs as if they were the same graph is sound.

Having defined graphs and relations of similarity between them, we now move on to define graph transformations, i.e. processes that transform a given graph into a different graph.

2.2 Graph Transformation Systems

In order to transform a graph, one essentially needs two bits of information: which parts of the graph need to be changed, and how they need to be changed. In algebraic graph transformation (specifically the single- and double pushout approaches), this is accomplished by specifying a single graph transformation as a tuple of two graphs. The first describes a subgraph that the transformation is applied to, and the second describes how that subgraph should look after the transformation has taken place.

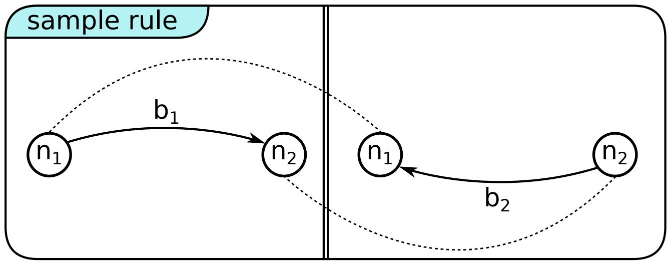


Figure 2.2: A sample rule reversing and relabeling an edge (p indicated by dashed lines)

This kind of transformation definition is captured in the concept of a *graph rule*.

Definition 10 (Graph Rule). *A* graph rule $P = (L, p, R)$ *consists of two graphs* L *and* R*, as well as a graph isomorphism* $p : L_c \to R_c$ *between subgraphs* $L_c \leq L$ *and* $R_c \leq R$*.*

It is worth noting, that the specific definition of a graph rule varies quite a bit between various applications of algebraic graph transformation. It is, for example, equivalent to the above definition to replace the isomorphism with the condition that both graphs must have a non-empty intersection. This makes certain aspects of application easier, but also makes it harder to state the algebraic application definition on which the approach is based. It is also possible (and common) to make the rule definition less restrictive, and only require a partial graph morphism (a graph morphism on L_c) instead of an isomorphism, allowing for the possibility of a transformation merging nodes. We choose the above definition as a compromise between conceptual clarity and simplicity of application.

The graph L in this definition is referred to as the *left hand side* of P, whereas R is referred to as the *right hand side* of P. We can now concretize the intuitive idea of graph transformation, now called rule application, thusly:

Idea. *Find an occurrence of the left hand side in a given graph, and replace this occurrence with the right hand side, preserving the subgraph indicated by* p*.*

Figure 2.2 shows an example of a graph rule. It expresses that an occurrence of a b_1 edge should be replaced by a b_2 edge in the opposite direction. The isomorphism p is indicated by dashed lines. Note that neither edge is touched by p.

The processes of finding an occurrence, and of replacing it, are called *matching* and *application*, respectively.

Definition 11 (Match). *Given a graph* G*, and a graph rule* $P = (L, p, R)$*, a* match *for* P *in* G *is an injective graph morphism* $m : L \to G$*.*

In general, it is not necessary to require matches to be injective. We exclude non-injective matches here in order to avoid a long list of problems with our verification approach. This is not a very significant constraint, since many interesting systems can be adequately modeled using injective matching, and many models that use non-injective matching can be rewritten to work with injective matching.

Using Def. 11, a match expresses that a subgraph of G is *similar* to the left hand side in the sense of graph morphisms. We can now transform the graph G by applying the rule to that subgraph, which, for simplicity, we will also call the match of the rule.

Applying a graph rule in the algebraic approach can be done in one of two ways, called single pushout and double pushout[133]. These ways differ in their underlying categorical formalism. We will describe both of them here, distinguishing them not by their categorical definition, but by the implications of those definitions on the application process itself.

First, we need to define the condition under which a rule is applicable to a graph. This always takes the form of some kind of condition on the match at which the rule is to be applied. If the rule is applicable for a given match, we call that match *valid*.

Definition 12 (Rule Applicability (SPO)). *Given a graph G and a rule $P = (L, p, R)$, any match $m : L \to G$ is* valid*.*

Thus, in SPO, the mere existence of an injective morphism from L to G suffices to obtain rule applicability. Given a valid match, a rule can be *applied* to a graph in the following way.

Definition 13 (Rule Application (SPO)). *Let G be a graph, $P = (L, p, R)$ be a rule, $m : L \to G$ be a valid match, and G' be the result of* applying P to G at m using the single pushout (SPO) method. *We define the match on the nodes of the right hand side as $f := p \circ (m \mid N_{L_c}) \cup id_{N_{R_c}}$. The graph G' is then defined as follows:*

$$
\begin{aligned}
N_{G'} &:= N_G \setminus m\left(N_L \setminus N_{L_c}\right) \cup \left(N_R \setminus N_{R_c}\right)\\
E^1_{G'} &:= \left[E^1_G \setminus m\left(E^1_L \setminus E^1_{L_c}\right) \cup \hat{f}\left(E^1_R \setminus E^1_{R_c}\right)\right]\Big\downarrow_{N_{G'}}\\
E^2_{G'} &:= \left[E^2_G \setminus m\left(E^2_L \setminus E^2_{L_c}\right) \cup \hat{f}\left(E^2_R \setminus E^2_{R_c}\right)\right]\Big\downarrow_{N_{G'}}
\end{aligned}
$$

As defined above, a rule application intuitively does the following:

- It begins with the node set N_G, then removes the matches of the nodes that occur exclusively in the left hand side and adds the nodes that occur exclusively in the right hand side.

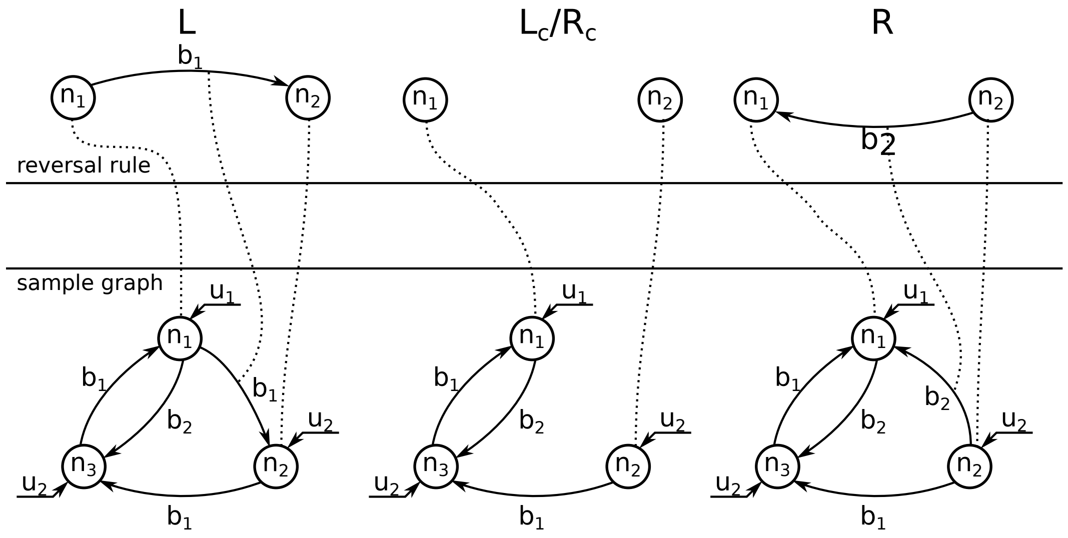


Figure 2.3: Applying the reversal rule to the sample graph

- For $i \in \{1, 2\}$, it begins with the edge set E^i_G, then removes the matches of the edges that occur exclusively in the left hand side and adds the edges that occur exclusively in the right hand side. It then restricts this new edge set to the nodes of G' in order to remove possible *dangling edges* created by removing a node without explicitly removing all its incident edges.

Essentially, the isomorphic part of the rule ($L_c \equiv_p R_c$) is kept, the parts exclusive to the left hand side are deleted, and the parts exclusive to the right hand side are added.

Note that the actual nodes of the rule are added to the graph itself, possibly creating problems when applying the same rule to the graph once more. These problems can be easily circumvented by creating a new instance of the rule each time it is applied and using consistent renaming to avoid node identity conflicts.

Figure 2.3 shows the application of the reversal rule to the sample graph at one particular match. In the figure, the remove and add steps are separated for clarity.

In its effects and in the context of the definitions made in this chapter, the double pushout approach to rule application does not vary a lot from the single pushout approach. Essentially, it adds an additional application condition intended to exclude the possibility of dangling edges.

Definition 14 (Rule Applicability (DPO)). *Given a graph G and a rule $P = (L, p, R)$, a match $m : L \to G$ is* valid *iff* $\forall n \in N_L \setminus N_{L_c}$ *we have*

$$\forall e^1 \in E^1_G : n \in e^1 \to \exists e^1_d \in \left(E^1_L \setminus E^1_{L_c}\right) : m\left(e^1_d\right) = e^1$$

$$\forall e^2 \in E_G^1 : n \in e^2 \rightarrow \exists e_d^2 \in \left(E_L^2 \setminus E_{L_c}^2\right) : m\left(e_d^2\right) = e^2$$

Intuitively speaking, in DPO rule application, a rule must *explicitly* delete all incident edges of every node it deletes. This means that, should the application of a rule imply the creation of dangling edges, then the rule is deemed not applicable. Designing models using DPO is thus more difficult, since node deletion is only possible if the entire context of a node at runtime is known a priori. However, this more restrictive view on rule application has many theoretical benefits as we will see in Sec. 3.6, as well as in the following definition.

Definition 15 (Rule Application (DPO)). *Given a graph G, a rule $P = (L, p, R)$ and a valid match $m : L \rightarrow G$, the result G' of* applying P to G at m*, using the double pushout (DPO) method, is defined as follows:*

$$\begin{aligned} N_{G'} &:= N_G \setminus m\left(N_L \setminus N_{L_c}\right) \cup \left(N_R \setminus N_{R_c}\right) \\ E_{G'}^1 &:= E_G^1 \setminus m\left(E_L^1 \setminus E_{L_c}^1\right) \cup \hat{f}\left(E_R^1 \setminus E_{R_c}^1\right) \\ E_{G'}^2 &:= E_G^2 \setminus m\left(E_L^2 \setminus E_{L_c}^2\right) \cup \hat{f}\left(E_R^2 \setminus E_{R_c}^2\right) \end{aligned}$$

where f is defined as in Def. 13.

As expected, inducing the resulting graph on its node set is no longer necessary, since the DPO application condition already makes sure that all remaining edges are only incident to preserved nodes.

Having defined graph rules and their application to graphs, we can now transform graphs into new graphs using rules. This forms the basis for the modeling of structural changes in systems described by graphs. There are many ways to actually define a model of such a system, depending on the needs of the designer. In this thesis, we will focus on the most basic way to model a structurally dynamic system : the *graph transformation system*.

A graph transformation system defines an initial graph and a set of rules that can be applied to this graph. The operational semantics of the system are obtained by exhaustively applying the rules to the start graph until no new graphs can be created.

Definition 16 (Graph Transformation System (GTS)). *A* Graph Transformation System (GTS) *is a tuple $\mathcal{G} = (I, \mathcal{R})$, where I is a graph and $\mathcal{R}$ is a set of rules.*

Applying all applicable rules in $\mathcal{R}$ to I, we get a new set of graphs. From each isomorphic subset of these graphs, we keep only one representative. Applying all applicable rules to the remaining graphs yields yet more graphs, and so on. If the graph transformation system is finite (and it may very well not be, as we will see in the next section), then this process of creating new graphs will eventually terminate and yield the *reach set* of the GTS.

Definition 17 (Transition, Reach Set). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, G be a graph, let $P \in \mathcal{R}$ be a rule applicable to G at some match m, and let G' be the result of applying P to G at m. This relationship between G, P, m and G' constitutes a* transition *and is denoted $G \xrightarrow{P,m} G'$. When P or m are inconsequential, this can be written $G \xrightarrow{P} G'$ or $G \to G'$, respectively. We write $G \to^* G'$ if there is a sequence of applications of rules in $\mathcal{R}$ that transforms G into G'. The* reach set *of $\mathcal{G}$ is defined as* $\mathrm{reach}\left(\mathcal{G}\right) := \{G \mid I \to^* G\} /_{\equiv}$.

The reach set of a graph transformation system itself has no structure. In order to obtain an overview of not just what kinds of graphs a GTS can produce, but also *how* they are produced, i.e. by what rule sequences, we need a *graph transition system*.

Definition 18 (Graph Transition System). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system, and let* $\mathrm{reach}\left(\mathcal{G}\right)$ *be its reach set. The* graph transition system *of $\mathcal{G}$, denoted* $\mathrm{trans}\left(\mathcal{G}\right) := \left(N, E^1, E^2\right)$*, is a graph over the label set $\mathcal{R} \times \mu$, defined as follows:*

$$\begin{aligned} N &:= \mathrm{reach}\left(\mathcal{G}\right) \\ E^1 &:= \{(G, (P, m)) \mid m \textit{ is a valid match for } P \textit{ in } G \wedge L_P = R_P\} \\ E^2 &:= \left\{\left(G, (P, m), G'\right) \mid G \xrightarrow{P,m} G' \wedge L_P \neq R_P\right\} \end{aligned}$$

where μ is the set of all matches of rules in $\mathcal{P}$ to graphs in $\mathrm{reach}\left(\mathcal{G}\right)$.

Note that the definition of E^2 excludes rules that perform no action, whereas E^1 allows only such rules that perform no action. This is a restriction that becomes relevant in the next section.

This fully defines the operational semantics of a graph transformation system. Figure 2.4 shows the graph transition system for the graph transformation system consisting of the sample graph in Fig. 2.1 and the reversal rule in Fig. 2.2.

2.3 Model Checking Graph Transformation Systems

A graph transformation system like this could, for example, model a UML object diagram (the initial state), together with the effects of all methods in the system that affect the object structure. The graph transition system would then show which object configurations are reachable with the operations provided, and how these configurations are reached.

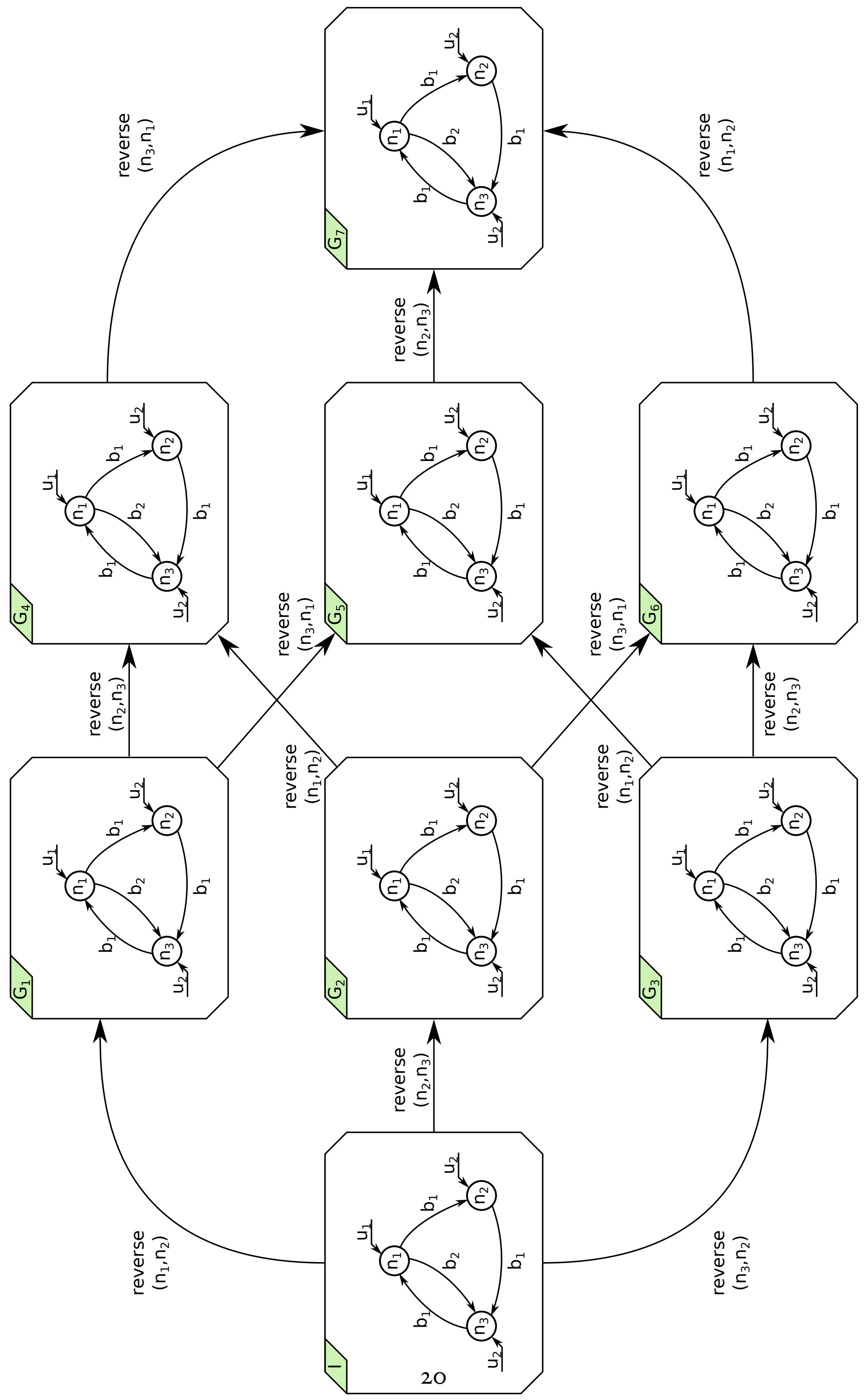


Figure 2.4: The graph transition system induced by the sample graph and the sample rule

A natural question to ask, then, would be whether the behavior as modeled conforms to the desired behavior of the system. This question motivates the *model checking*[52] of graph transformation systems. Given a model of a system, in this case a graph transformation system, does it conform to a given specification?

The easiest way to answer this question in the context of model checking, is to transform our graph transition system into a Kripke Structure, the basic model on which all (explicit) model checking techniques are based. In order to achieve this, two things are necessary: all paths in the graph transition systems must be made infinite, and the states (the graphs) of the graph transition system must be annotated with atomic propositions which the specification will refer to.

Making all paths infinite is easy by simply adding a no-action self-edge to every graph that does not have an outgoing edge. In order to add useful atomic propositions, we need a way to express *properties* of graphs. As in the work by Rensink et.al.[75], we achieve this by way of graph patterns.

Definition 19 (Graph Pattern). *A* graph pattern *is a graph rule $P = (L, p, R)$ with $L = R$ and $p = id_L$.*

Thus, a graph pattern is simply a rule that performs no action. Using patterns, we can now define the Kripke Structure for any given GTS.

Definition 20 (Kripke Structure for a GTS). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system,* $\text{trans}(\mathcal{G}) = \left(N, E^1, E^2\right)$ *be its graph transition system, and let $\mathcal{P} \subseteq \mathcal{R}$ be a set of patterns. The Kripke Structure $\mathcal{K}(\mathcal{G}) = (S, S_0, R, L)$ for $\mathcal{G}$ is defined as follows:*

$$\begin{aligned} S &= N, \\ S_0 &= I, \\ R &= \left\{\left(G, G'\right) \in S \times S \mid \exists P, m : \left(G, (P, m), G'\right) \in E^2\right\} \\ &\quad \cup \left\{(G, G) \in S \times S \mid \neg\exists G', P, m : \left(G, (P, m), G'\right) \in E^2\right\}, \\ L &= N \mapsto \left\{P \in \mathcal{P} \mid \exists m : ((P, m), N) \in E^1\right\}. \end{aligned}$$

Thus, the patterns of the GTS become unary state predicates (atomic propositions) in the Kripke Structure. In this way, all standard notions of model checking are translatable to graph transformation systems. This includes full LTL/CTL model checking.

An implementation of this is explained in great detail in the works of Rensink et.al.[75] and demonstrated in the GROOVE[*] tool.

In this thesis, we focus on a particular kind of model checking problem – the covering problem. Given a single pattern (modeling an "error"), and a graph transformation

[*]Graphs for Object Oriented Verification

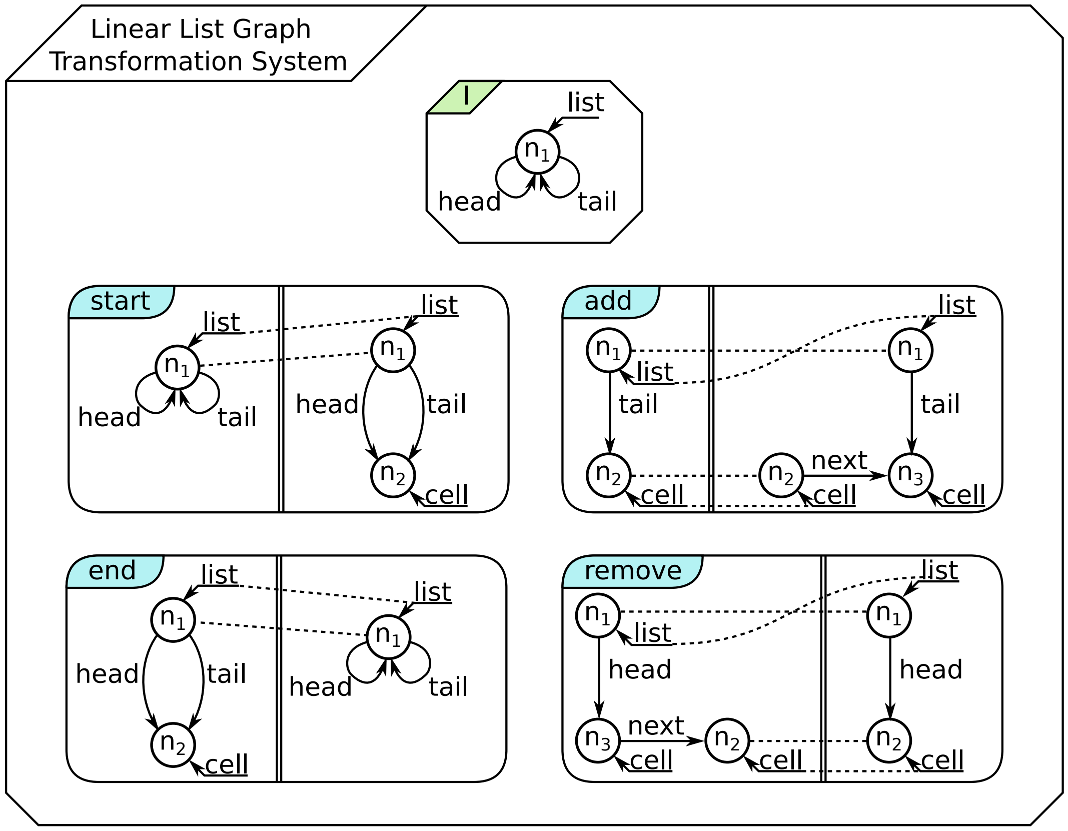


Figure 2.5: A graph transformation system modeling a linear list

system, is it possible to construct a graph that matches the pattern? This is known as the covering problem and amounts to LTL model checking the specification $\mathbf{G}\neg error$ for a given system.

Definition 21 (Covering Problem). *A* covering problem *is a tuple* $(\mathcal{G}, P)$ *of a graph transformation system and a pattern.* $\mathcal{G}$ *is said to* cover P *iff there exists a* $G \in \text{reach}(\mathcal{G})$ *and a match* m *for* P *in* G*.*

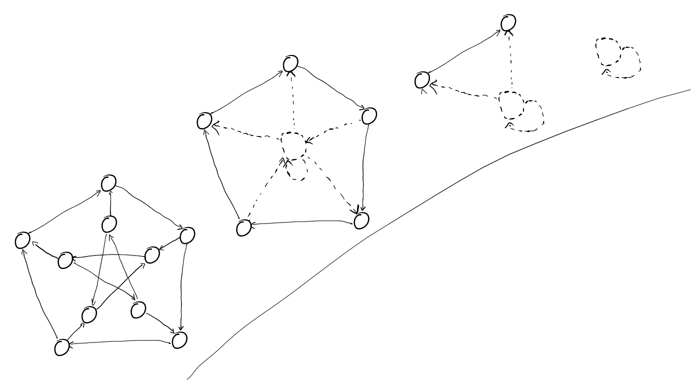
Checking whether a given graph transformation system covers a given pattern is very straightforward – simply construct the transition system (or even just the reach set) and check whether a graph matching the pattern exists. However, it is easy to see that this approach runs into problems when the reach set grows very large, or even infinitely large.

This can happen in relevant application scenarios very easily. Consider, for example, the graph transformation system depicted in Fig. 2.5. It models a simple linear list data structure, with actions to create a list, delete a list, and to add or remove objects.

While the data structure and the operations on it are very simple, it is also very clear that its reach set is infinitely large, since nothing prohibits arbitrarily many add operations from being applied. Clearly, any process that relies on the construction of the entire reach set of such a graph transformation system will fail to prove any properties about it.

One way to deal with this issue is *bounded model checking*, i.e. only constructing a finite prefix of the state space in the hope that, if errors are present, the rule sequences leading to them will be relatively short. The other way to approach the problem is *abstraction*. By constructing a finite and sound abstraction of the infinite graph transition system, it is possible to derive properties of the original system. This approach is taken by a number of verification frameworks for graph transformation systems, as well as this thesis.

In the next chapter, we will describe our approach to graph abstraction and compare it to other approaches with similar goals.



3 Abstracting Graph Transformation Systems using Three-Valued Logic

State space explosion is a widespread problem in all facets of verification and model checking. It occurs when small, incremental increases in the size of the model result in huge increases in the size of the state space, making state enumeration intractable. In models describing dynamically reconfigurable systems, state space explosion comes about via the combination and interaction of the rules that govern the creation of new configurations. Each new rule creates not just all the new combinations of rule execution sequences with the new rule, but can also lead to entirely new kinds of configurations, on which the old rules also generate new configurations.

Since dynamically reconfigurable models are often, as described in the previous chapter, modeled using graphs and graph transformation systems, this problem is mirrored in the model checking of graph transformation systems. The tendency of GTSs to produce large or even infinite numbers of states makes it impossible to verify any of their properties using explicit state space generation. A standard approach to solving this problem is to abandon the direct checking of the state space itself, and to work instead on a finite and small overapproximation of it. This is an instance of a general technique called *abstract interpretation*.

The basic idea is to create a notion of abstraction wherein each abstract state represents a (possibly infinite) set of concrete states. The properties of each abstract state represent an overapproximation of the properties of the concrete states it represents. Transitions between abstract states mirror transitions between the concrete states that they represent and thereby form an *abstract transition system*.

An abstraction is called *sound*, if all parts of the original transition system are *covered* by some part of the abstract transition system. It is easy to see that when a sound abstract transition system does not contain a given type of error, this result is transferable to the original system.

The effort required to find a good abstraction and compute the resulting abstract transition system is usually substantial. Thus, this approach is only feasible when the original problem is sufficiently intractable (which in the case of an infinite state space is certainly the case) and enough information to base the design of the abstraction on is available.

In this chapter, we will present our approach to tackling these issues for graph transformation systems. It is based on the three-valued logic approach to verifying heap-manipulating programs by Sagiv et.al.[134].

3.1 Motivation

There are many approaches to graph abstraction that already exist. Each of them is built on a different theoretical framework. Some approaches focus on *patterns* instead of graphs, using a pattern to represent all graphs that contain it, such as the work by Daniela Schilling[139]. By starting from the error pattern and applying rules backwards, the absence of the pattern can be proven or disproved to be an inductive invariant of the system. Other approaches use more explicit abstraction mechanisms. Examples for this are, e.g., the work by Baldan, Corradini and König[16], where the GTS is fused with a Petri-net like structure to obtain a workable abstraction, and the work by Rensink et.al.[36], in which nodes with similar neighborhoods are summarized.

Each of these approaches has its own strengths and weaknesses. A new abstraction system for graph transformation systems should therefore attempt to address existing weaknesses so as not to be redundant. One weakness common to these approaches is that abstraction refinement is rather coarse-grained, if it is supported at all, and unable to adapt to the specifics of a given GTS. Our goal is thus to create an abstraction system for graph transformation systems that is highly adaptable, both manually and automatically, to a given GTS and even to specific covering problems for that GTS.

If the goal is maximum adaptability, then it makes sense to base the abstraction on a highly expressive foundation. Outside of the domain of graph transformations, there already is an abstraction system exhibiting some of these properties – shape analysis for heap-manipulating programs. This approach was developed by Sagiv, Reps and Wilhelm[134]. It represents the state of the heap in heap-manipulating programs using *logical structures*, a concept that is very close to graphs. In the following sections, we will present the logical foundations of shape analysis and adapt it to graph abstraction.

A Short Discussion of Shape Analysis

Before we move on to the minute details of the particular shape analysis approach that we base our approach on, we will give the reader a quick overview over shape analysis in general and explain how it fits into the overall landscape of formal verification.

Shape analysis techniques are an instance of a much more general concept called *abstract interpretation*. Pioneered and formalized in the late 1970s by Patrick and Radhia Cousot[57–59] it provided a very general framework by which concrete systems could be formally related to abstractions of these systems. Cousot and Cousot described a mathematical theory of abstraction functions (relating concrete models to their abstractions) that allowed for a standardized relating of concrete semantics to abstract semantics. Most formalisms in the area of formal verification that use abstraction are either outright formulated in terms of abstract interpretation or can be interpreted in such a way that the viewpoint provided by abstract interpretation is applicable.

The term shape analysis describes a category of abstraction approaches that apply the basic insights of abstract interpretation to systems whose state is defined predominantly by some kind of structural model. By far the most prevalent structure that is considered by shape analysis approaches is the pointer structure on the heap induced by imperative, heap-manipulating programs. These approaches chiefly seek to prove pointer safety of those programs, i.e. the absence of memory leaks and null-pointer dereferences, but can also express and prove structural properties of the data structures represented in the heap. They use a variety of different abstraction and abstraction refinement approaches, from three-valued logic as done by Sagiv et.al.[134] and Thomas Wies[155], over separation logic, as done by Distefano et.al.[61, 160], to more ad-hoc approaches as done by Ghiya and Hendren[77]. The application of these approaches has yielded many relevant results, even for industrially-relevant case studies (see, e.g., Yang et.al.[160]).

With the structure in the center of shape analysis – the pointer structure in the heap – being so closely related to graphs, it is no wonder that shape analysis has inspired many abstraction approaches for graph transformation systems. However, shape analysis is primarily a tool for the analysis of sequential programs. Thus, while the abstraction of the states themselves are very similar in graph abstraction and shape analysis, the operational semantics of the systems are very different. They also tend to focus on particular kinds of structures that often occur in heaps, such as linked lists, trees, or cycles, while in graph transformation systems, such “standard” structures are much rarer.

Graph abstraction approaches (our approach included) thus tend to bear only a superficial resemblance to shape analysis approaches. We will discuss other graph abstraction approaches in detail in Sec. 8.1.

3.2 Logical Encoding of Graphs

Shape analysis uses first-order predicate logic to describe heap states and their properties. At the core of predicate logic lies the concept of a universe. A universe contains all the objects that the logic will later enable us to express the properties of.

Definition 22 (Universe/Domain). *A* Universe *or* Domain *is a set of objects. The purpose of calling the set a domain or universe is to facilitate the creation of logical formulas describing subsets of (k-powers of) the set.*

Building on this concept, the notion of a *predicate* allows for the identification of arbitrary subsets of (k-powers of) the universe. Each predicate is denoted by a *predicate symbol.*

Definition 23 (Predicate Symbol). *A* Predicate Symbol *is a symbol (like a, b, φ, ψ, ...) together with an integer k, called its* arity. *Thus, a predicate symbol is a pair (s, k), where s is a symbol and $k \in \mathbb{N}_0$. s is said to* have arity k *or that it* is a k-ary predicate. *If the arity is clear from context, we omit k and simply write s.*

A predicate symbol of arity k denotes a predicate that defines a subset of the kth power of the universe. Sets of predicate symbols provide a vocabulary for building first-order formulas that describe properties of and relationships between the predicates referenced by the symbols in the set. All sets of predicate symbols will contain at least the equality symbol $(=, 2)$.

In order to build first-order formulas using predicate symbols, we need syntactical atoms that reference these symbols. This is provided by the *predicate literal*, expressing the membership of a given k-tuple of universe elements in the subset defined by the predicate represented by a given predicate symbol.

Definition 24 (Predicate Literal). *Let (p, k) be a predicate symbol and let U be a universe. A* Predicate Literal *of (p, k) is written $p\,(u_1, \ldots, u_k)$, where $u_1, \ldots, u_k$ are variables ranging over U.*

For the equality predicate, we write $(u_1 = u_2)$ instead of $= \ (u_1, u_2)$. From predicate literals, as well as boolean constants, boolean connectives and quantifiers, first-order formulas can be constructed.

$\wedge$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$\mathbf{1}$	$\mathbf{0}$	$\mathbf{1}$

$\vee$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

	$\neg$
$\mathbf{0}$	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{0}$

Table 3.1: Truth Tables for Basic Operators in Boolean Logic

Definition 25 (First-Order Formula). *Let $\mathcal{F}$ be the set of first-order formulas over a set of predicate symbols $\mathcal{P}$ and a set of variables $\mathcal{V}$. Let furthermore* $\mathrm{F} : \mathcal{F} \to 2^{\mathcal{V}}$ *identify the set of free variables of any given formula. $\mathcal{F}$ and* F *are inductively defined as follows:*

$$\begin{aligned}
\mathbf{1}, \mathbf{0} &\in \mathcal{F} & \mathrm{F}(\mathbf{1}) = \mathrm{F}(\mathbf{0}) &= \emptyset \\
p(u_1, \ldots, u_k) &\in \mathcal{F} \text{ iff } u_i \in \mathcal{V} \wedge (p, k) \in \mathcal{P} & \mathrm{F}(p(u_1, \ldots, u_k)) &:= \{u_1, \ldots, u_k\} \\
\neg(\varphi) &\in \mathcal{F} \text{ iff } \varphi \in \mathcal{F} & \mathrm{F}(\neg\varphi) &:= \mathrm{F}(\varphi) \\
(\varphi) \wedge (\psi) &\in \mathcal{F} \text{ iff } \varphi, \psi \in \mathcal{F} & \mathrm{F}(\varphi \wedge \psi) &:= \mathrm{F}(\varphi) \cup \mathrm{F}(\psi) \\
(\varphi) \vee (\psi) &\in \mathcal{F} \text{ iff } \varphi, \psi \in \mathcal{F} & \mathrm{F}(\varphi \vee \psi) &:= \mathrm{F}(\varphi) \cup \mathrm{F}(\psi) \\
\forall u : (\varphi) &\in \mathcal{F} \text{ iff } \varphi \in \mathcal{F} \wedge u \in \mathrm{F}(\varphi) & \mathrm{F}(\forall u : (\varphi)) &:= \mathrm{F}(\varphi) \setminus \{u\} \\
\exists u : (\varphi) &\in \mathcal{F} \text{ iff } \varphi \in \mathcal{F} \wedge u \in \mathrm{F}(\varphi) & \mathrm{F}(\exists u : (\varphi)) &:= \mathrm{F}(\varphi) \setminus \{u\}
\end{aligned}$$

Syntactically correct formulas by themselves have no direct meaning. This is because the predicate symbols they use have no inherent meaning. All that a formula does is to define relationships between the predicates represented by the predicate symbols it contains. In order to assign a truth value (either true or false) to a formula, we need to connect it to boolean logic first.

Definition 26 (Boolean Logic). Boolean Logic *consists of the set $\mathcal{B} := \{\mathbf{1}, \mathbf{0}\}$, together with the operators $\wedge, \vee : \{\mathbf{1}, \mathbf{0}\} \times \{\mathbf{1}, \mathbf{0}\} \to \{\mathbf{1}, \mathbf{0}\}$ and $\neg : \{\mathbf{1}, \mathbf{0}\} \to \{\mathbf{1}, \mathbf{0}\}$. These operators are defined as shown in Table 3.1.*

We can further relate the elements of $\mathcal{B}$ to each other in a *logical order*, formalizing the intuitive idea that $\mathbf{1}$ is "more true" than $\mathbf{0}$.

Definition 27 (Logical Order). *The* Logical Order $\leq$ *on $\mathcal{B}$ is a total order on $\mathcal{B}$ defined as* $\{(\mathbf{0}, \mathbf{0}), (\mathbf{0}, \mathbf{1}), (\mathbf{1}, \mathbf{1})\} \subseteq \mathcal{B}^2$.

Boolean Logic gives us a way to actually define predicates, not just predicate symbols, by treating them as characteristic functions of the subsets of the universe that they represent.

Definition 28 (Predicate). *A* Predicate of arity k *over a universe U is a total function $P : U^k \to \mathcal{B}$.*

The way to assign a truth value to a formula, then, is to assign to each predicate symbol a predicate that it represents. This is captured in a so-called *interpretation*.

Definition 29 (Interpretation). *Let U be a universe, and let $\mathcal{P}$ be a set of predicate symbols. An* interpretation ι of $\mathcal{P}$ over U *is a function that maps the elements of $\mathcal{P}$ to predicates, i.e.*

$$\iota_k : \quad \mathcal{P}_k \to \left(U^k \to \mathcal{B}\right)$$
$$\iota := \bigcup_{k\in\mathbb{N}} \iota_k$$

where $\mathcal{P}_k$ designates the subset of $\mathcal{P}$ restricted to k-ary predicates. The set of all possible interpretations for $\mathcal{P}$ and U is denoted $\mathcal{I}(\mathcal{P}, U)$. An interpretation is complete *if ι_k is a total function for all k.*

We assume that every interpretation automatically imbues the equality predicate symbol $=$ with its natural meaning, i.e. $u_1 = u_2$ is only set to true if u_1 and u_2 are variables referencing the same universe element.

Given a universe, a set of predicate symbols, and an interpretation, the truth value of any formula with respect to that context depends solely on the assignment of its free variables (if any) to specific universe elements. That context, which fixes the meaning of formulas, is called a *logical structure*.

Definition 30 (Logical Structure). *A* (2-valued) Logical Structure *is a triple $L = (U, \mathcal{P}, \iota)$, where U is a universe, $\mathcal{P}$ a set of predicate symbols and $\iota \in \mathcal{I}(\mathcal{P}, U)$ a* complete *interpretation. Any formula φ over $\mathcal{P}$ with $\mathrm{F}(\varphi) = \emptyset$ can be evaluated to a truth value in the context given by L. If $\mathrm{F}(\varphi) \neq \emptyset$, then the truth value of φ depends on the assignments to the free variables. Given a logical structure L, we refer to its constituent parts as U_L, $\mathcal{P}_L$ and ι_L.*

Since a formula might have free variables, it cannot ultimately be evaluated, until these free variables are *assigned* to universe elements.

Definition 31 (Assignment). *Given a set of Variables $\mathcal{V}$ and a universe U, an* assignment *is a function $m : \mathcal{V} \to U$. An assignment is called* complete *if it is a total function. Given an assignment m, a variable v and a universe element u, we define*

$$m[v \mapsto u] := x \mapsto \begin{cases} u & \textit{if } x = v \\ m(x) & \textit{else} \end{cases}$$

Given predicate symbols, predicates, interpretations, logical structures, and assignments, the final missing piece is a definition of what "evaluating a formula" actually means. This essentially defines the *meaning* or the *semantics* of a formula with respect to a given logical structure.

Definition 32 (Formula Semantics). *Let $S = (U, \mathcal{P}, \iota)$ be a logical structure and let φ be a first order formula over $\mathcal{P}$ and a set of variables $\mathcal{V}$ ranging over U. Let $m : \mathrm{F}(\varphi) \to U$ be an assignment of all free variables of φ. Then the* value *or the* semantics *of φ are denoted $[\![\varphi]\!]_S^m$ and are defined inductively in the following way:*

For all $c \in \mathcal{B}$:

$$[\![c]\!]_S^m := c$$

For all predicate literals with free variables $v_1, \ldots, v_k$:

$$[\![p(v_1, \ldots, v_k)]\!]_S^m := \iota(p)(m(v_1), \ldots, m(v_k))$$

For formulas φ, ψ:

$$\begin{aligned} [\![\varphi \wedge \psi]\!]_S^m &:= [\![\varphi]\!]_S^m \wedge [\![\psi]\!]_S^m \\ [\![\varphi \vee \psi]\!]_S^m &:= [\![\varphi]\!]_S^m \vee [\![\psi]\!]_S^m \end{aligned}$$

For a formula φ

$$[\![\neg\varphi]\!]_S^m := \neg[\![\varphi]\!]_S^m$$

For a formula φ with a free variable v

$$\begin{aligned} [\![\forall v : \varphi]\!]_S^m &:= \textstyle\bigwedge_{u \in U} [\![\varphi]\!]_S^{m[v \mapsto u]} \\ [\![\exists v : \varphi]\!]_S^m &:= \textstyle\bigvee_{u \in U} [\![\varphi]\!]_S^{m[v \mapsto u]} \end{aligned}$$

Should $\mathrm{F}(\varphi) = \emptyset$, the assignment is unnecessary. The semantics are then denoted $[\![\varphi]\!]_S$. We define two sets:

$$\begin{aligned} \sigma_S(\varphi) &:= \{m : free(\varphi) \to U \mid [\![\varphi]\!]_S^m = \mathbf{1}\} \\ \nu_S(\varphi) &:= \{m : free(\varphi) \to U \mid [\![\varphi]\!]_S^m = \mathbf{0}\} \end{aligned}$$

If $\sigma_S(\varphi) \neq \emptyset$, we call φ satisfiable *in S, or SAT for short.*
If $\sigma_S(\varphi) = \emptyset$, we call φ unsatisfiable *in S, or UNSAT for short.*

In summary, first-order formulas are made up of predicate literals, defined over a set of predicate symbols $\mathcal{P}$, boolean constants, boolean connectives, and quantifiers. They are given meaning by logical structures, consisting of a universe U and interpretations for the predicate symbols used, as well as assignments for their free variables. We can conceive of a logical structure as a logical context in which formulas, that is, specifications of properties of and interrelationships between predicates, can be evaluated.

Now, what does this have to do with graphs? In their papers[134], Sagiv et.al. use logical structures as representations of heap states. Heap states consist in essence of two types of things: memory cells, i.e. places on the heap where data is stored, and pointers, i.e. references to memory cells, which are themselves stored in memory cells as well. Thus, we have objects (memory cells), connected by binary relationships (pointers). It is easy to see that, viewed this way, a heap state can be modeled (or understood) as a graph.

We adopt this viewpoint in the following definitions.

Definition 33 (Logical Structure of a Graph). *Let* $G = \left(N, E^1, E^2\right)$ *be a graph over a label set* $L = (U_L, B_L)$*. The* logical structure of G*, denoted* $\mathrm{ls}(G) = (U, \mathcal{P}, \iota)$*, is defined as follows.*

$$\begin{aligned}
U &:= N \\
\mathcal{P}_L &:= \{(u,1) \mid u \in U_L\} \cup \{(b,2) \mid b \in B_L\} \\
\iota &:= \iota^1 \cup \iota^2 \\
\iota^1\left(u\right)\left(x\right) &:= (u,x) \in E^1 && \forall\left(u,1\right) \in \mathcal{P} \forall x \in U \\
\iota^2\left(b\right)\left(x,y\right) &:= (x,b,y) \in E^2 && \forall\left(b,2\right) \in \mathcal{P} \forall x,y \in U
\end{aligned}$$

Thus, the node set of a graph is used as the universe, the edge labels become predicate symbols, and their interpretations, that is, the actual predicates, are derived from the edge sets. This also works vice-versa, provided the predicate set does not contain predicates of arity other than 1 and 2.

Definition 34 (Graph of a Logical Structure). *Let* $ls = (U, \mathcal{P}, \iota)$ *be a logical structure. The* graph of ls*, denoted by* $G\left(ls\right)$*, is defined as follows:*

$$\begin{aligned}
L &:= \left(U_L := \{u \mid (u,1) \in \mathcal{P}\}, B_L := \{b \mid (b,2) \in \mathcal{P}\}\right) \\
N &:= U \\
E^1 &:= \{(u,n) \mid \iota\left(u\right)\left(n\right)\} \\
E^2 &:= \{(n_1,b,n_2) \mid \iota\left(b\right)\left(n_1,n_2\right)\}
\end{aligned}$$

	list	cell
n1	1	0
n2	0	1
n3	0	1
n4	0	1

head	n1	n2	n3	n4
n1	0	0	0	1
n2	0	0	0	0
n3	0	0	0	0
n4	0	0	0	0

next	n1	n2	n3	n4
n1	0	0	0	0
n2	0	0	0	0
n3	0	1	0	0
n4	0	0	1	0

tail	n1	n2	n3	n4
n1	0	1	0	0
n2	0	0	0	0
n3	0	0	0	0
n4	0	0	0	0

$U=\{n_1, n_2, n_3, n_4\}$
$P_L=\{(list,1),(cell,1),(head,2),(tail,2),(next,2)\}$

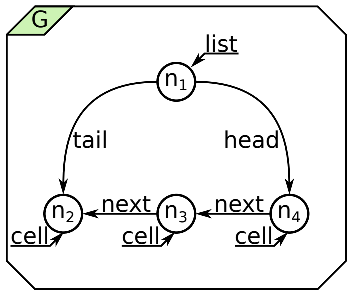


Figure 3.1: A graph and its corresponding Logical Structure

The above definitions mean that, at least in the absence of higher-arity (and 0-ary) predicates, we can use the terms *graph* and *logical structure* interchangeably. If we were to lift the restriction to simple graphs and allowed hypergraphs, this correspondence would even extend to logical structures over predicate sets with k-ary predicates. As an example of the correspondence between logical structures and graphs, consider the example shown in Fig. 3.1. The graph (representing a small linear list, such as those produced by the GTS in Fig. 2.5) and the logical structure are exactly equivalent.

The utility of identifying graphs with logical structures is twofold. On the one hand, we can reuse many of the definitions and results that were achieved by Sagiv et.al.[134] for abstracting heap states. This makes up the foundation of our abstraction method. On the other hand, the representation of graphs as logical structures creates a direct link between graphs and predicate logic with uninterpreted predicates, which enables the SMT encodings used in Chap. 4 for counterexample analysis and abstraction refinement.

As an immediate benefit, though, note that since a graph is a logical structure, and logical structures lend meaning to formulas, we can now express conditions on graphs as first-order formulas. For example, consider the following definition, which expresses the existence of a match of a given rule to any graph.

Definition 35 (Matching Formula). *Let $P = (L, p, R)$ be a graph rule. The* matching formula *for P is defined as*

$$\varphi_P = \underbrace{\bigwedge_{(n,b,n')\in E_L^2} b\left(n, n'\right)}_{\textit{binary edges}} \wedge \underbrace{\bigwedge_{(u,n)\in E_L^1} u\left(n\right)}_{\textit{unary edges}} \wedge \underbrace{\bigwedge_{\substack{n_1,n_2\in N_L\\ n_1\neq n_2}} \neg\left(n_1 = n_2\right)}_{\textit{injectivity}}$$

This formula can, given an assignment to its variables (sourced from the nodes of its left hand side), be evaluated in any logical structure (read: graph) that shares the same predicate set. In terms of Def. 32, we can express the matching condition of SPO using the matching formula: for any rule P and graph G, if φ_P is satisfiable in G, then P *is applicable to* G, and every assignment in $\sigma_S(\varphi_P)$ yields a valid match[*]. Similarly, if φ_P is unsatisfiable in G, then the rule is not applicable. A similar matching formula can be defined for the DPO case.

All concepts defined on graphs easily carry over to logical structures. As an example, we give the definition of a (graph) morphism between logical structures.

Definition 36 (Logical Structure Morphism). *Let L, L' be logical structures. A* logical structure morphism *between L and L' is a total function $f : U_L \to U_{L'}$, such that*

$$\begin{aligned}\iota_L(u)(n) &\leq \iota_{L'}(u)(f(n)) & \forall (u,1) \in \mathcal{P}_L \forall n \in U_L\\ \iota_L(b)(n_1,n_2) &\leq \iota_{L'}(b)(f(n_1), f(n_2)) & \forall (b,2) \in \mathcal{P}_L \forall n_1, n_2 \in U_L\end{aligned}$$

This directly corresponds to the concept of an induced graph morphism. Corollary concepts, like isomorphisms on logical structures, follow analogously.

Now that we can define graphs using first-order logic, the next step is to utilize the abstraction approach by Sagiv et.al.[134] to create the notion of an abstract graph, called a *shape graph*. In order to do this, though, we first need to introduce the logic that this abstraction approach is based on: Kleene's Three-Valued Logic.

3.3 Three-Valued Logic

Our approach to graph abstraction is based on the representation of graphs as logical structures, and the subsequent "blurring" of those structures using Kleene's Three-Valued Logic. The basic idea of a 3-valued logic is to introduce a third truth value, representing a kind of middle ground between $\mathbf{1}$ and $\mathbf{0}$, called ½. This third truth value is meant to express uncertainty about the actual truth value, i.e. to express that it could be either true or false. This third truth value interacts with boolean connectives as one might expect, as the following definition shows.

Definition 37 (Kleene Logic). *Kleene logic is an extension of boolean logic over the truth value set $\mathcal{K} = \{\mathbf{0}, ½, \mathbf{1}\}$. Table 3.2 shows the meaning of the standard operators in Kleene Logic.*

[*]in the sense of SPO

$\wedge$	$\mathbf{0}$	½	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
½	$\mathbf{0}$	½	½
$\mathbf{1}$	$\mathbf{0}$	½	$\mathbf{1}$

$\vee$	$\mathbf{0}$	½	$\mathbf{1}$
$\mathbf{0}$	$\mathbf{0}$	½	$\mathbf{1}$
½	½	½	$\mathbf{1}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

	$\neg$
$\mathbf{0}$	$\mathbf{1}$
½	½
$\mathbf{1}$	$\mathbf{0}$

Table 3.2: Truth Tables for Basic Operators in Kleene Logic

All definitions of the previous section are still valid if one uses the truth value set $\mathcal{K}$ instead of $\mathcal{B}$. A logical structure based on $\mathcal{K}$ rather than $\mathcal{B}$ is called a 3-*valued logical structure*. The only actual modification necessary regards the formula semantics definition, which is amended in the following way.

Definition 38. *Let $S = (U, \mathcal{P}, \iota)$ be a 3-valued logical structure and let φ be a first order formula over a set of predicate symbols $\mathcal{P}$ and a set of variables $\mathcal{V}$ ranging over U. Let $m : \mathrm{F}(\varphi) \to U$ be an assignment to all free variables of φ. Then $[\![\varphi]\!]_S^m$ and $[\![\varphi]\!]_S$, respectively, are defined as in Def. 32. We define three sets:*

$$\begin{aligned} \sigma_S(\varphi) &:= \{m : free(\varphi) \to U \mid [\![\varphi]\!]_S^m = \mathbf{1}\} \\ \pi(\varphi) &:= \{m : free(\varphi) \to U \mid [\![\varphi]\!]_S^m = ½\} \\ \nu_S(\varphi) &:= \{m : free(\varphi) \to U \mid [\![\varphi]\!]_S^m = \mathbf{0}\} \end{aligned}$$

If $\sigma_S(\varphi) \neq \emptyset$, we call φ satisfiable *ins S, or SAT for short.*
If $\sigma(\varphi) = \emptyset$ and $\pi(\varphi) \neq \emptyset$ we call φ possibly satisfiable*, or ½SAT for short.*
If $\sigma_S(\varphi) = \emptyset$ and $\pi(\varphi) = \emptyset$, we call φ unsatisfiable *in S, or UNSAT for short.*

Much like $\mathcal{B}$, $\mathcal{K}$ is ordered by a logical order.

Definition 39 (Logical Order on $\mathcal{K}$). *The total order*

$$\leq := \{(\mathbf{0},\mathbf{0}), (½,½), (\mathbf{1},\mathbf{1}), (\mathbf{0},½), (\mathbf{0},\mathbf{1}), (½,\mathbf{1})\}$$

is called the Logical Order *or* Truth Order *on $\mathcal{K}$.*

This defines the logical order one might expect. The truth value $\mathbf{1}$ is, in this sense, "more true" than ½, which, in turn is still "more true" than $\mathbf{0}$.

However, $\mathcal{K}$ has another order, born out of the intuition that the truth value ½ contains inherently "less information" than the other two values. Thus, ½ is, in a sense, "more abstract" than $\mathbf{1}$ or $\mathbf{0}$.

Definition 40 (Information Order on $\mathcal{K}$). *The partial order*

$$\sqsubseteq := \{(\mathbf{0},\mathbf{0}), (½,½), (\mathbf{1},\mathbf{1}), (\mathbf{0},½), (\mathbf{1},½)\}$$

is called the Information Order *or* Abstraction Order *on* $\mathcal{K}$.

There is also a join operator for the information order, which, as expected, maps everything except agreeing definite truth values to ½.

Definition 41 (Join Operator). *The operator* join $\sqcup : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$ *is defined as follows:*

$$x \sqcup y = \begin{cases} x & \textit{iff } x = y \\ ½ & \textit{else} \end{cases}$$

Figure 3.2 shows a combined diagram illustrating both orders on $\mathcal{K}$ in a Hasse diagram. The basic idea behind the abstraction approach employed by us and Sagiv et.al. are already visible here. When encoding graphs (or heap states) as logical structures, a truth value that is more abstract than $\mathbf{1}$ or $\mathbf{0}$ enables us to not specify whether a given edge exists or not. The resulting logical structure can be thought of as an abstract structure, representing both the case where the edge exists, and the case where it does not. With a little extra effort, this uncertainty can be extended to the existence of nodes, as well. The next section describes in detail how the information order defined here can be lifted from individual truth values to entire logical structures and how this constitutes a valid abstraction mechanism.

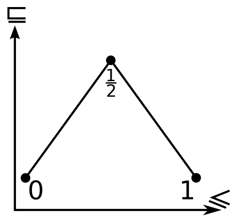


Figure 3.2: Truth and Information order on $\mathcal{K}$

3.4 Shapes and Embedding

In the previous section we have shown a direct link between graphs and 2-valued logical structures. We have also introduced a 3-valued logic, containing a third truth value that represents, in a sense, both $\mathbf{1}$ and $\mathbf{0}$. Simply by using logical structures that are not 2-valued, but 3-valued, we can thus obtain a notion of abstraction that allows us to mark certain edges as ½-edges, edges that may or may not actually exist. It is easy to see that such a 3-valued logical structure containing n ½-edges would "represent", in a sense, a total of 2^n 2-valued logical structures.

However, in order to build an effective abstraction mechanism, it is insufficient to merely abstract from the existence of edges alone. We need to also be able to abstract from the existence, and thereby, the number, of nodes in a graph. The following definition, equivalent to the corresponding definition by Sagiv et.al.[134], captures this.

Definition 42 (Shape). *A* shape graph *or* shape S *is a* 3*-valued logical structure with* $(\mathrm{sm}, 1) \in \mathcal{P}_S$. *The special predicate* sm *is called the* summary predicate. *In a shape, for any given node* v, $\iota\,(\mathrm{sm})\,(v)$ *is always automatically defined by* $\neg\,(v = v)$ *(i.e. it can be valued ½, but never* $\mathbf{1}$*).*

	sm	list	cell
n_1	0	1	0
n_2	0	0	1
n_3	$\frac{1}{2}$	0	1
n_4	0	0	1

head	n_1	n_2	n_3	n_4
n_1	0	0	0	1
n_2	0	0	0	0
n_3	0	0	0	0
n_4	0	0	0	0

next	n_1	n_2	n_3	n_4
n_1	0	0	0	0
n_2	0	0	0	0
n_3	0	$\frac{1}{2}$	$\frac{1}{2}$	0
n_4	0	0	$\frac{1}{2}$	0

tail	n_1	n_2	n_3	n_4
n_1	0	1	0	0
n_2	0	0	0	0
n_3	0	0	0	0
n_4	0	0	0	0

$U=\{n_1, n_2, n_3, n_4\}$
$P_L=\{(\text{list},1),(\text{cell},1),(\text{head},2),(\text{tail},2),(\text{next},2)\}$

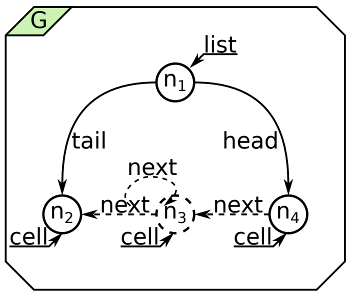


Figure 3.3: A shape representing an arbitrarily long linear list

It is easy to see that any graph, that is, any 2-valued logical structure, can be seen as a shape, i.e. a 3-valued logical structure, where the truth value ½ is simply not used. Thus, a graph is merely a special form of a shape.

The intention of the introduction of the summary predicate (and its link to the equality predicate) is to enable the abstraction of node sets. Note that the term $\neg(v = v)$ should be interpreted to **0** for all nodes v in a 2-valued logical structure, i.e. a concrete graph. In a 3-valued logical structure, however, we are able to interpret it as ½ for selected nodes. For these nodes v, we also have $\iota(\mathrm{sm})(v) = ½$, labeling them *summary nodes*.

The graphical representation of a shape is similar to that of a graph. Edges valued **1** in the interpretation are drawn as solid arrows, while edges valued ½ are drawn as dashed arrows. In addition, summary nodes are drawn as dashed circles. The summary predicate is omitted in the graphical representation. Figure 3.3 shows an example shape, meant to represent an arbitrarily long linear list.

Summary nodes are intended to represent whole sets of nodes. This means that graphs represented by the shape are allowed to "replace" summary nodes with an arbitrary number of concrete nodes, provided these nodes share the properties of the summary node. The following definition, also analogous to Sagiv et.al.[134], concretizes this notion of shapes representing other shapes and, ultimately, graphs.

Definition 43 (Embedding / Meaning of a Shape). *A shape S is said to be* embedded into *or* represented by *another shape S' over the same predicate set, iff there exists a function $f : U_S \to U_{S'}$ with the following three properties:*

(i) f *is surjective*

(ii) $\forall(p,k) \in \mathcal{P} \forall u_1, \ldots, u_k \in U : \iota_S(p)(u_1, \ldots, u_k) \sqsubseteq \iota_{S'}(p)(f(u_1), \ldots, f(u_k))$

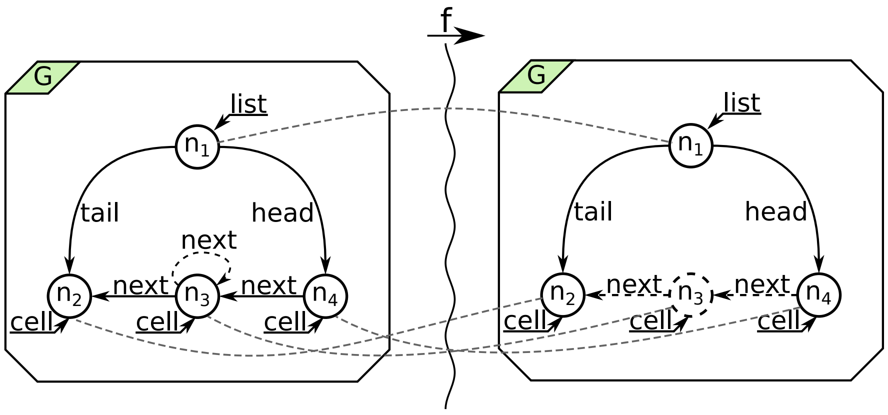


Figure 3.4: An example embedding

(iii) $\left(\left|f^{-1}(v)\right| > 1\right) \sqsubseteq \iota_{S'}(sm)(v)$

The function f is called an embedding. *This relationship between S and S' is denoted $S \sqsubseteq_f S'$ or $S \sqsubseteq S'$ if f is clear from context or inconsequential. Note that S and S' need to have the same set of predicates $\mathcal{P}$.*

The three embedding conditions defined above mirror closely the natural intuition on what kinds of shapes a given shape S should represent.

Firstly, the embedding must be surjective, which means that any embedded shape must have at least as many nodes as S and must map at least one of its nodes to each of the nodes of S.

Secondly, the edges touching a node in S must represent an overapproximation of the edges that touch the nodes that are mapped to it. This means that if, for a given label and node pair, the corresponding edge in S is a ½-edge, then there is no restriction for the such labeled edges between the nodes that are mapped to that pair (analogous for unary edges). However, when S contains a definite edge (either $\mathbf{1}$ or $\mathbf{0}$), this must be mirrored exactly in all represented shapes.

Thirdly, the value of sm for any given node v in S should be an abstraction of whether more than one node is mapped to v. This means that an embedding can only map more than one node to a node in S if that node is a summary node.

As an example, consider the embedding shown in Figure 3.4. It shows how the graph shown in Fig 3.1 can be embedded into the shape shown in Fig. 3.3.

The notion of an embedding imposes a partial order on shapes, that is, the relation $\sqsubseteq$ on the set of possible shapes over a predicate set $\mathcal{P}$ is reflexive, transitive, and anti-symmetric. The reflexivity property is easy to see. Each shape is clearly embedded into itself by the identity map on its universe. For transitivity, we present the following lemma:

Lemma 1. *Let $S \sqsubseteq_g S' \sqsubseteq_f S''$ be two embeddings. Then the concatenation $f' := f \circ g$ of f and g is also an embedding.*

Proof. Three conditions have to be shown:

- f' is *surjective* because it is the concatenation of two surjective maps.
- Let $(p, k) \in \mathcal{P}$ and $(u_1, \ldots, u_k) \in U_S$. Since g is an embedding, it follows that

  $$\iota_S(p)(u_1, \ldots, u_k) \sqsubseteq \iota_{S'}(p)(g(u_1), \ldots, g(u_k)).$$

  Since f is also an embedding, it follows that

  $$\iota_{S'}(p)(g(u_1), \ldots, g(u_k)) \sqsubseteq \iota_{S''}(p)(f(g(u_1)), \ldots, f(g(u_k))).$$

  And thus, by the transitivity of $\sqsubseteq$ on $\mathcal{K}$, we have

  $$\iota_S(p)(u_1, \ldots, u_k) \sqsubseteq \iota_{S''}(p)\left(f'(u_1), \ldots, f'(u_k)\right).$$

- Let $u \in U_{S'}$. Since g is an embedding, we know that $\left(\left|g^{-1}(u)\right| > 1\right) \sqsubseteq \iota_{S'}(sm)(u)$ holds. Since f is also an embedding, we know that, for any $v \in U_{S''}$, we have $\left(\left|f^{-1}(v)\right| > 1\right) \sqsubseteq \iota_{S''}(sm)(v)$. And thus, by the transitivity of '$\sqsubseteq$ on $\mathcal{K}$, we have $\left(\left|g^{-1}\left(f^{-1}(u)\right)\right| > 1\right) \sqsubseteq \iota_{S''}(sm)(v)$ for any $v \in U_{S''}$.

Thus, f' is an embedding. □

The antisymmetry condition in the context of shapes means that if two shapes are embedded into each other, they must be isomorphic.

Lemma 2. *Let S and S' be shapes. If $S \sqsubseteq S'$ and $S' \sqsubseteq S$, then $S \equiv S'$.*

Proof. See Appendix A, page 241 □

The partial order $\sqsubseteq$ on the set of shapes over a predicate set $\mathcal{P}$ is thus established. It allows us to define exactly which shapes (and ultimately graphs) are represented by a given shape S – namely all those shapes that are less than S in $\sqsubseteq$. This leads to the idea that a shape has a *meaning*, defined by the set of graphs that it represents.

Definition 44 (Shape Set / Graph Set). *Let S be a shape. The* shape set $\mathcal{S}(S)$ *of S is the set of all shapes that are represented by S:*

$$\mathcal{S}(S) := \left\{S' \mid S' \sqsubseteq S\right\}$$

The graph set $\mathcal{G}(S)$ *of* S *is the set of all graphs that are represented by* S*:*

$$\mathcal{G}(S) := \{G \mid G \sqsubseteq S \land G \text{ is 2-valued}\}$$

These graph and shape sets exhibit the intuitively expected properties.

Corollary 1. *Let* S *be a shape. Since every graph is also a shape,* $\mathcal{G}(S) \subseteq \mathcal{S}(S)$ *holds. Furthermore,let* S' *be a shape such that* $S \sqsubseteq S'$*. We now have* $\mathcal{S}(S) \subseteq \mathcal{S}(S')$*, as well as* $\mathcal{G}(S) \subseteq \mathcal{G}(S')$ *due to the transitivity of* $\sqsubseteq$*.*

Proof. trivial. □

The final missing piece in the abstraction of single graphs and shapes is the interplay between properties of shapes, i.e. formulas evaluated in them, and the embedding relation. This leads to the so-called *embedding theorem*, taken from Sagiv et.al.[134].

Theorem 1 (Embedding Theorem). *Let* S, S' *be shapes such that* $S \sqsubseteq_f S'$ *and let* φ *be a formula over* $\mathcal{P}_S$*. Then, for any assignment* m *of the free variables of* φ *(if necessary),*

$$[\![\varphi]\!]_S^m \sqsubseteq [\![\varphi]\!]_{S'}^{f \circ m}$$

holds.

Proof. See Sagiv et.al.[134]. □

Therefore, the values of formulas on a shape S are overapproximations of the values that this formula takes on all shapes represented by S. This theorem thus forms the basis of the argument why the verification technique detailed in the next section is sound.

3.5 Abstract Transformation and State Space Construction

Having defined abstractions of singular graphs, the next question is whether and how graph transformations can be lifted to the abstract level as well. Here we begin to digress from the original work by Sagiv et.al.[134], since their approach dealt with heap-manipulating programs. Such programs can only perform a limited set of operations on the heap, defined a priori. In contrast, each new graph transformation system brings with it a new set of operations, i.e. rules, that can affect graphs in unique ways. Nevertheless, our approach to abstract transformation follows the same concepts as theirs, and exhibits many of the same properties.

The first question one ought to ask is what it actually means to apply a graph rule to a shape graph. The most intuitive view on this would be that applying a rule P to

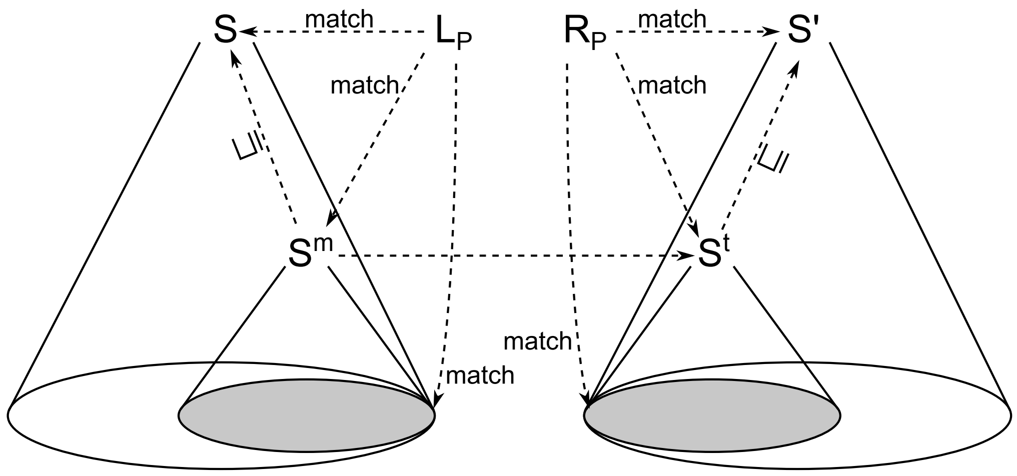


Figure 3.5: A schematic view on soundly applying a rule to a shape

a shape graph S at a match m simply means applying P to every graph in $\mathcal{G}(S)$ at matches m' compatible with m via their embedding into S. However, this idea is only viable if the rule is actually applicable at all these concrete matches m'. In fact, nothing keeps a shape from representing both graphs to which the rule is applicable, and graphs to which it is not.

We thus refine the naïve idea above to this: applying P to S at an (abstract) match m means creating a new shape covering all graphs that result from applying P to all graphs in $\mathcal{G}(S)$ for which matches m' for P exist that are compatible with m via their embedding. This idea of a sound rule application to a shape is shown schematically in Fig. 3.5. First, one identifies the set of graphs in $\mathcal{G}(S)$ that have matches for P compatible with the match to S. This set is represented by a new shape S^m. Then this shape, and thereby the graphs in $\mathcal{G}(S^m)$ are transformed using P into a set of graphs represented by a shape S^t. Optionally, this resulting shape can be further abstracted in order to normalize the abstraction.

In the following, we will address the key questions that arise from this idea: how to tell if a rule even matches a shape, how to identify those graphs represented by the shape that also match the rule, and how to construct the resulting shape covering the resulting graphs.

Step 1 – Matching When looking at a shape S and a rule P, with the intention of using P to transform S, the first question that arises is the question of applicability. Can P actually be applied to S? In the concrete case, i.e. if S were 2-valued, this reduces to the question of the existence of nodes and edges with certain labels in certain configurations, as defined by L_P. However, due to the presence of ½-edges and summary nodes, the existence of certain edges, and even of some nodes, is now no longer clear.

In an attempt to solve this, we first reduce the question to the simplest possible case: the case where the left hand side of the rule matches a concrete subgraph of S, i.e. a part of S that contains neither ½-edges nor summary nodes. In order to check the existence of such a match, we slightly extend the matching formula of P to exclude summary nodes.

Definition 45 (Amended Matching Formula). *Let $P = (L, p, R)$ be a graph rule. The* amended matching formula *for P is defined as*

$$\varphi_P = \underbrace{\bigwedge_{(n,b,n')\in E_L^2} b\left(n, n'\right)}_{\textit{binary edges}} \wedge \underbrace{\bigwedge_{(u,n)\in E_L^1} u\left(n\right)}_{\textit{unary edges}} \wedge \underbrace{\bigwedge_{\substack{n_1,n_2\in N_L\\ n_1\neq n_2}} \neg\left(n_1 = n_2\right)}_{\textit{injectivity}} \wedge \underbrace{\bigwedge_{n\in N_L} \neg\,\mathrm{sm}\left(n\right)}_{\textit{non-summarization}}$$

Any further references to matching formulas or φ_P will refer to this amended version, not the original formula as defined in Def. 35.

Clearly, if this formula is SAT in S, then any assignment $m \in \sigma\left(\varphi_P\right)$ is a match of the left hand side into S and the subgraph $m\left(L_P\right) \leq S$ is a 2-valued logical structure[†]. By the embedding theorem (Thm. 1), that means that the rule is applicable to all graphs represented by the shape at the corresponding concrete matches. It would thus be sound to just classically apply the rule at the shape level, since only the concrete part of the shape, i.e. that part mirrored by all embedded graphs, is affected. Conversely, should the matching formula be UNSAT in S, then by the embedding theorem there can be no graph in $\mathcal{G}\left(S\right)$ to which the rule is applicable, and thus, declaring the rule to not be applicable to S is sound. The really interesting case occurs when φ_P is ½SAT in S.

In such a case, the intuition is that the shape S is too abstract to definitely establish applicability. There are no definitely satisfying assignments in $\sigma\left(\varphi_P\right)$ (otherwise φ_P would be SAT), but there are assignments in $\pi\left(\varphi_P\right)$, i.e. assignments that make φ_P evaluate to ½. This might be because one of the edge terms is set to ½ in S, or because the match contains a summary node. In such a case, we call P *potentially applicable* to S. Each assignment in $\pi\left(\varphi_P\right)$ is called a *potential match*.

The existence of a potential match m of P to S means that there are some represented graphs $G \sqsubseteq_f S$ and definite matches m' of P into G, such that $m = f \circ m'$,

[†]if $m\left(L_P\right)$ were 3-valued, one of the literals in φ_P would be valued ½, leading to $[\![\varphi_P]\!]_S^m \leq$ ½.

but there are also represented graphs for which such m' do not exist. In order to actually apply the rule, we need to separate the former set of graphs from the latter set of graphs. This is done by *materialization*.

Step 2 – Materialization The idea of materialization is, given S, P and m, to construct a shape or set of shapes such that these shapes cover *exactly* those graphs in $\mathcal{G}(S)$ that have a match for P that is compatible with m.

The process is based on the idea that every graph that matches P at the same "place" that S does, must have a concrete expression of the left hand side of P at m. Thus, all ½-edges that are required by the rule in the match of m are set to $\mathbf{1}$. Furthermore, as many nodes as are matched to any given summary node by m are "materialized" (hence the name) out of that summary node, inheriting all their edges from it. For any such summary node, the shape does not specify whether the summary node represented exactly as many nodes as were needed, or more than that. These two cases represent different materializations, since, in both cases the rule would be applicable, yet, in the former case, the summary node would have to be deleted, whereas in the latter case, it would need to be preserved.

We thus gain a separate materialization for any subset of summary nodes in the match of the rule that we choose to preserve. The following definition describes this process.

Definition 46 (Materialization). *Let $S = (U, \mathcal{P}, \iota)$ be a shape, $P = (L, p, R)$ be a rule, and $m : L \to S$ be a match such that ½ $\leq [\![\varphi_P]\!]^m_S$. Let $U^s \subseteq m(N_L)$ be the set of nodes in the match that are summary nodes. Let $I \subseteq U^s$ be a subset of those summary nodes. Then the* materialization $S^m_I = (U^m, \mathcal{P}, \iota^m)$ of S according to m and I *is defined as follows. The node set (universe) of S^m_I is defined by*

$$U^m := U \setminus (m(N_L) \setminus I) \cup N_L$$

whereas ι^m is defined, for $(sm, 1) \neq (u, 1), (b, 2) \in \mathcal{P}$ by

$$\iota^m(u)(n) := \begin{cases} \mathbf{1} & \textit{if } n \in N_L \wedge (u, n) \in E^1_L \\ \iota(u)(f(n)) \end{cases}$$

$$\iota^m(b)(n_1, n_2) := \begin{cases} \mathbf{1} & \textit{if } n_1, n_2 \in N_L \wedge (n_1, b, n_2) \in E^2_L \\ \iota(b)(f(n_1), f(n_2)) \end{cases}$$

$$\iota^m(sm)(n) := \begin{cases} \mathbf{0} & \textit{if } n \in N_L \\ \iota(s)m(f(n)) \end{cases}$$

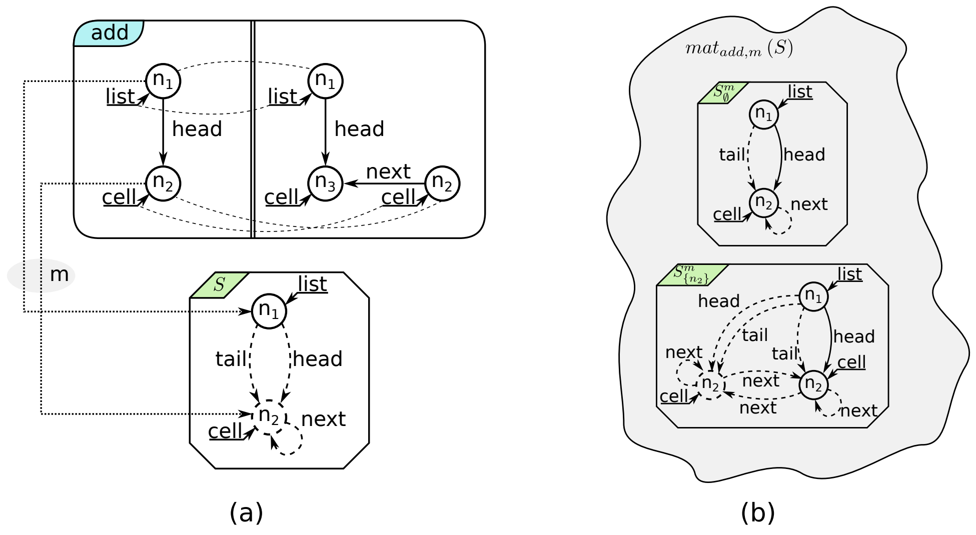


Figure 3.6: Materialization Example

where $f := m \cup id_U|_{U^m}$.

The set of materializations *of* S *with respect to* P *and* m *is then defined as*

$$\text{mat}_{P,m}(S) := \{S^m_I \mid I \subseteq U^s\}$$

Note that, if $[\![\varphi_P]\!]^m_S = \mathbf{1}$, then the corresponding materialization is $S^m = S$, i.e. materialization of definite matches is unnecessary and produces the same shape anyway.

As an example, consider the shape, rule, and match shown in Fig. 3.6(a). The shape represents (among other things) an arbitrarily long, but non-empty linear list. The rule represents the adding of a node to a list, which certainly matches any non-empty list using the match shown.

Figure 3.6(b) shows the set of materializations for this scenario. Since the match included one summary node, two materializations are produced, representing the cases where the list was exactly one entry long, and the case where it was at least two entries long.

A materialization, in a sense, *concretizes* a shape to only represent those graphs to which the rule is actually applicable. This has the effect that the resulting match of the rule into the materialization (which is the identity map on the left hand side of the rule) is always a concrete one, i.e. the image of the left hand side of the rule is a concrete subgraph of the materialization.

Lemma 3. *Let* S *be a shape, let* P *be a graph rule, and let* m *be a match for* P *in* S

such that $m \in \pi(\varphi_P)$*. Furthermore, let* $S^m \in \mathrm{mat}_{P,m}(S)$ *be a materialization of* P *in* S*. Then we have* $[\![\varphi_P]\!]^{\mathrm{id}}_{S^m} = \mathbf{1}$.

Proof. By construction, the match of P into S^m is $id_{N_{L_P}}$, and the definition of ι^m ensures that any edge in $E^1_{L_P}, E^2_{L_P}$ is definitely present in S^m, as well as that the summary predicate is valued **0** for all nodes in the match. Thus, φ_P must evaluate to true in any materialization S^m of P. □

Materializing a rule out of a shape using a match yields a set of shapes that definitely match the rule. Since the part of the shape that the rule affects has been "concretized", rule application itself can then commence as normal on concrete graphs.

The question remains, whether materialization covers all of the graphs embedded in a shape to which a rule could be applied at a given match. If the match is a definite match, this follows directly from the embedding theorem. However, if the match is a potential match, we have so far only shown that the rule is, in fact definitely applicable to the resulting materialization, not whether that materialization actually covers all the shapes it is supposed to cover. The following lemmata answer this question.

Lemma 4. *Let* S*,* $P = (L, p, R)$*,* m*,* S^m *and* f *be as in Def. 46. Then we have* $S^m \sqsubseteq_f S$

Proof. We show each of the three embedding properties separately.

(i) $f = m \cup id_U|_{U^m}$ is surjective, since $m(N_L)$ covers the match of the rule, whereas $id_U|_{U^m} = id_{U \setminus (m(N_L) \setminus I)}$ covers the remainder of U.

(ii) For any $(sm, 1) \neq (u, 1) \in \mathcal{P}$ and $n \in U^m$, we have two cases: If $n \in N_L \wedge (u, n) \in E^1_L$, then, from the structure of φ_P we know that $\iota_S(u)(f(n))$ must have been either $\mathbf{1}$ or ½, which means that $\mathbf{1} = \iota_{S^m}(u)(n) \sqsubseteq \iota_S(u)(f(n))$ holds. If $n \notin N_L \vee (u, n) \notin E^1_L$ then by construction we have $\iota_{S^m}(u)(n) = \iota_S(u)(f(n))$ and thus $\iota_{S^m}(u)(n) \sqsubseteq \iota_S(u)(f(n))$. The binary case follows analogously. For $(sm, 1) \in \mathcal{P}$ and $n \in U^m$, we have either $n \in N_L$, in which case $\iota_S(sm)(f(n))$ is either **0** or ½ (no other values are possible), and thus $\mathbf{0} = \iota_{S^m}(sm)(n) \sqsubseteq \iota_S(sm)(f(n))$ holds. In the other case, i.e. $n \notin N_L$, we have $\iota_{S^m}(sm)(n) = \iota_S(sm)(f(n))$ and thus $\iota_{S^m}(sm)(n) \sqsubseteq \iota_S(sm)(f(n))$.

(iii) Outside of $N_L \subseteq U^m$, f is defined as the identity function on U. Thus, any node in U to which more than one node in U^m is mapped must lie in the image $m(N_L)$. Any node n such that $|m^{-1}(n)| > 1$ must be a summary node, due to the injectivity condition in φ_P. If there is a node n such that $|m^{-1}(n)| = 1$, but $|f^{-1}(n)| > 1$, then there must be $n_1, n_2 \in U^m$ such that $id_U(n_1) = m(n_2)$. Since only summary nodes in the image of m are copied into U^m, n must be a summary node as well. Thus, in all cases we have $|f^{-1}(n)| > 1 \sqsubseteq \iota_S(sm)(n)$.

□

So, a materialization S^m is embedded into the shape S from which it was constructed, which means that it represents a subset of the graphs that S represents. The remaining issue is whether this graph set contains all the graphs that it should contain.

Lemma 5. *Let S, $P = (L, p, R)$, m, S^m and f be as in Def. 46. Let further $G \sqsubseteq_g S$ be a graph and $m' : L \to G$ be a match for P in G such that $m = g|_{m'(N_L)} \circ m'$. Then there exists an f' such that $G \sqsubseteq_{f'} S^m$.*

Proof. See Appendix A page 242. □

A materialization for a triple of shape, rule and match, therefore, represents graphs that were represented by the original shape (Lemma 4) and have a match for the rule that is compatible with the match to the original shape (Lemma 3). Furthermore, there are no graphs that are represented by the original shape, have a compatible match for the rule, but are not represented by the materialization (Lemma 5).

Step 3 – Application Once a match for a rule has been found, and the hosting shape materialized via that match, the conventional applicability of the rule to the shape is assured. The application process itself is performed exactly as for the concrete case. This is sound because the part of the shape that is affected by the rule application is concrete and thus mirrored exactly in all graphs embedded into the materialized shape, i.e. the effect of the rule on the graphs is the same as the effect of the rule on the materialized shape. This is expressed in the following lemmata.

Lemma 6 (Soundness of Shape Transformation (SPO)). *Let S be a shape, let $P = (L, R)$ be a rule, and let m be a definite match of P to S, i.e. $[\![\varphi_P]\!]_S^m = \mathbf{1}$. Let further S' be the result of applying P to S at m. Then, for every $G \in \mathcal{G}(S)$, there exist an $m' : L \to G$ and a $G' \in \mathcal{G}(S')$, such that $G \xrightarrow{P,m} G'$.*

Proof. The embedding theorem (Theorem 1) states that $[\![\varphi_P]\!]_S^m = \mathbf{1}$ is an overapproximation of the value of $[\![\varphi_P]\!]_G^{m'}$, where $G \sqsubseteq_f S$ and $m = f \circ m'$. Thus, P is applicable to G and produces a resulting graph G' and, since no summary nodes are part of the match, f is a bijective function on $m'(L)$. The locality of graph rule application ensures that P only changes G and S at its respective matches to those structures. Since the exact same changes are performed on G as well as on S, f, extended in the obvious way to any nodes added by the rule, remains a valid embedding for G' into S'. □

Thus, the resulting shape covers all the graphs that would result from applying the rule to the represented graphs themselves (at a compatible match). If a graph was covered by the (possibly materialized) shape before application, its result is covered by the result of the application.

In the absence of any kind of further restrictions on the graph sets embedded into the shapes, the converse is also true.

Lemma 7 (Completeness of Shape Transformation (SPO)). *Let S be a shape, let $P = (L, R)$ be a rule, and let m be a definite match of P to S, i.e. $[\![\varphi_P]\!]_S^m = \mathbf{1}$. Let further S' be the result of applying P to S at m. Then, for every $G' \in \mathcal{G}(S')$, there exist an $m' : L \to G$ and a $G \in \mathcal{G}(S)$, such that $G \xrightarrow{P,m} G'$.*

Proof. Let $G' = \left(N', E^{1'}, E^{2'}\right)$, let m' be the match of R in G, and let $G = \left(N, E^1, E^2\right)$ be given by

$$\begin{aligned} N &:= N' \setminus m'\left(N_R \setminus N_{R_c}\right) \cup N_L \setminus \left(N_{L_c}\right) \\ E^1 &:= \left[E'^1 \setminus m'\left(E_R^1 \setminus E_{R_c}^1\right) \cup E_L^1 \setminus E_{L_c}^1\right] \\ E^2 &:= \left[E'^2 \setminus m'\left(E_R^2 \setminus E_{R_c}^2\right) \cup E_L^2 \setminus E_{L_c}^2\right], \end{aligned}$$

i.e. by the reverse application of P to G', ignoring dangling edges. Since G has the same structure as G' outside the match of the rule, and the same holds for S and S', G is embedded into S outside the match. Further, since $[\![\varphi_P]\!]_S^m = \mathbf{1}$ we know that within the match, an instance of the left hand side of the rule exists in G and S. Thus, the only way that $G \sqsubseteq S$ could be violated is through a dangling edge present in S but absent (since no dangling edges were constructed) in G. Simply adding those edges to G will then construct a graph $\hat{G}$ that satisfies the condition. □

Note that, in the context of shape transformation, this notion of soundness applies to the subset of the represented graphs to which the rule is actually applicable with a match compatible to a given shape-level match (see Fig. 3.5). It is not the case that all graphs represented by a shape find a corresponding transformation result in the graph set of one of the resulting shapes, nor should it be, since the rule will not even be applicable to most of them.

Summary In concrete graphs, matching and application proceeds as defined in Chap. 2. Each match of a rule to a concrete graph yields one resulting graph.

On shapes, some matches are open to interpretation, which leads to uncertainty in the result of the transformation. Thus, for each potential match to a shape, rule application may yield a set of resulting shapes, via materialization. It is helpful to think of this non-determinism as a kind of case switch over the number of nodes available in the graphs represented by the shape. Each summary node is a node reservoir of uncertain size, so use of one by a match introduces a switch between the case where the reservoir was exhausted by the application of the rule and the case where it was not.

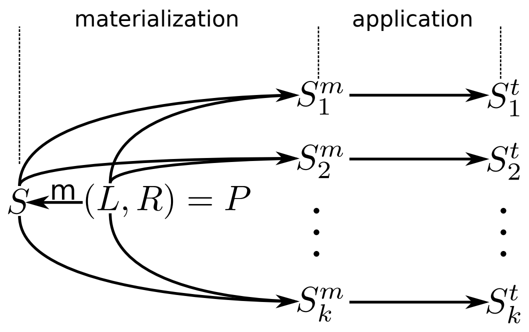


Figure 3.7: Application of a rule to a shape at a potential match (schematically)

Figure 3.7 shows a schematic of the whole process. First, a rule is (potentially) matched to a shape. Then, using that rule and match, materializations are constructed from the shape. Finally, these materializations are (classically) transformed, yielding a resulting shape set.

Having defined rule application on the shape level, we can now introduce the notion of an abstract graph transformation system, called a *shape transformation system*.

Definition 47 (Shape Transformation System (STS)). *A* Shape Transformation System (STS) *is a tuple $\mathcal{S} = (I, \mathcal{R})$, where I is a shape and $\mathcal{R}$ is a set of rules.*

Based on this notion, the concept of abstract transitions, an abstract reach set and the resulting notion of an abstract transition system follow. A shape transition is defined analogously to a graph transition. The only significant difference is the necessity to specify which subset of summary nodes was preserved in the transition.

Definition 48 (Shape Transition). *Let $\mathcal{S} = (I, \mathcal{R})$ be an STS, S be a shape, let $P \in \mathcal{R}$ be a rule applicable to S at some (potential or definite) match m, let $S^m \in \mathrm{mat}_{P,\mathrm{m}}(S)$, and let S' be the result of applying P to S at m using S^m. This relationship between S, P, m, S^m, and S' constitutes a* transition *and is denoted $S \xrightarrow{P,\mathrm{m},S^m} S'$. When P, m, or S^m are inconsequential, this can be written $S \xrightarrow{P,\mathrm{m}} S'$, $S \xrightarrow{P} S'$ or $S \rightarrow S'$, respectively. When there is a sequence of rules, matches and materializations that transforms S into S', we write $S \rightarrow^* S'$.*

Note that each transition incorporates a materialization. This necessarily leads to a loss of abstraction in each transition, since while a materialization concretizes the source shape, the target shape is not "re-abstracted" afterwards. While this may sometimes be the intended effect, it is often desirable to maintain a constant "level of abstraction" across shape transitions. For this, an abstraction scheme is required.

Definition 49 (Shape Abstraction Scheme). *Let $\mathcal{P}$ be a set of predicates, and let $\mathcal{S}(\mathcal{P})$ be the set of all shapes over $\mathcal{P}$. An* Abstraction Scheme over $\mathcal{P}$ *is a function* $\mathcal{A} : \mathcal{S}(\mathcal{P}) \to \mathcal{S}(\mathcal{P})$ *such that*

$$\begin{aligned} &\forall S \in \mathcal{S}(\mathcal{P}) : S \sqsubseteq \mathcal{A}(S) && \textit{shapes are embedded in their abstraction} \\ &\mathcal{A}^2(\mathcal{S}(P)) = \mathcal{A}(\mathcal{S}(P)) && \textit{the abstraction scheme is idempotent} \end{aligned}$$

hold. Each abstraction scheme $\mathcal{A}$ induces an equivalence relation

$$\approx_{\mathcal{A}} := \left\{ \left(S, S'\right) \mid \mathcal{A}(S) = \mathcal{A}\left(S'\right) \right\} \subseteq \mathcal{S}(\mathcal{P})^2$$

An abstraction scheme is called a Finite Abstraction Scheme *iff $\mathcal{S}(\mathcal{P}) /_{\approx_{\mathcal{A}}}$ is finite. An abstraction scheme is called a* Monotonic Abstraction Scheme, *if it also satisfies the condition*

$$\forall S, S' \in \mathcal{S}(\mathcal{P}) : S \sqsubseteq S' \to \mathcal{A}(S) \sqsubseteq \mathcal{A}\left(S'\right) \quad \textit{abstraction preserves embedding relations}$$

An abstraction scheme thus partitions the set of all shapes over a given predicate set $\mathcal{P}$ into regions, each of which is the shape set of a particular representative shape given by the abstraction scheme. Applying an abstraction scheme to a shape is called *blurring*. The design space for possible abstraction schemes is huge, ranging from the trivial, such as the identity map on $\mathcal{S}(\mathcal{P})$, to highly complex algorithms meant to ensure certain properties for the representative shapes. For now, we will use a very basic abstraction scheme, inspired by, and named after, the canonical abstraction given by Sagiv et.al. in their paper on parametric shape analysis[134]. We will call this scheme the canonical shape abstraction.

Definition 50 (Canonical Shape Abstraction). *Let $\mathcal{P}$ be a predicate set, and let $\mathcal{U} \subseteq \mathcal{P}$ be its subset of unary predicates. The* Canonical Shape Abstraction Scheme $\mathcal{C}$ *is given by*

$$\begin{aligned} \mathcal{C} : \mathcal{S}(\mathcal{P}) &\to \mathcal{S}(\mathcal{P}) \\ S &\mapsto \left(U/_{\simeq}, \mathcal{P}, \iota'\right) && \textit{where} \\ \simeq &:= \left\{(u_1, u_2) \mid \forall p \in \mathcal{U} : \iota(p)(u_1) = \iota(p)(u_2)\right\} && \textit{and} \\ \iota'(p)(u) &:= \iota(p)(u) && \forall p \in \mathcal{U} \\ \iota'(p)(u_1, u_2) &:= \bigsqcup_{\substack{u \in [u_1]_{\simeq} \\ v \in [u_2]_{\simeq}}} \iota(p)(u, v) && \forall p \in \mathcal{P} \setminus \mathcal{U} \end{aligned}$$

It is easy to see that the function specified by this definition is indeed a finite abstraction scheme. Each shape is embedded into its image under $\mathcal{C}$ via the function associating each node with its equivalence class under $\simeq$. Furthermore, it is obviously

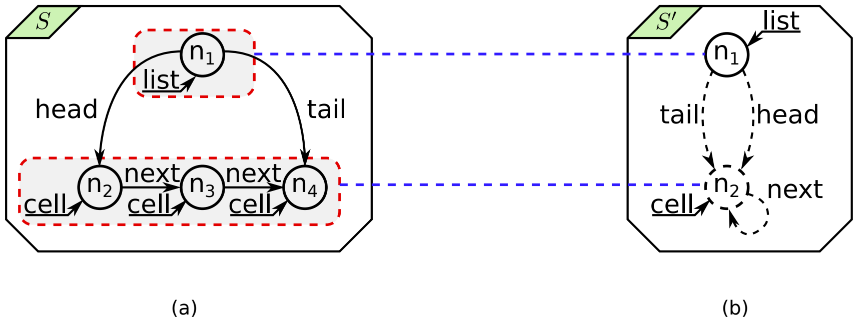


Figure 3.8: Example for a canonical abstraction of a graph

idempotent, since taking the quotient set of a quotient set for the same equivalence relation will yield the same quotient set as a result. Finally, it is finite, since $\mathcal{C}$ cannot produce shapes in which two nodes have the same values for all unary predicates. Since the number of unary predicates is finite, the number of different value combinations for these predicates are also finite, meaning that the number of different node sets that shapes in $\mathcal{S}\left(\mathcal{P}\right)_{\approx_{\mathcal{C}}}$ can have is limited. For each fixed node set, only a finite number of different shapes are possible, and thus we can conclude $\left|\mathcal{S}\left(\mathcal{P}\right)_{\approx_{\mathcal{C}}}\right| \in \mathbb{N}$.

It is worth noting that this abstraction scheme is not monotonic. Two nodes in a shape might be equivalent w.r.t. $\simeq$, but embedded into two nodes that are not equivalent w.r.t. $\simeq$. Thus, the abstraction would merge the two nodes in the embedded shape, but not the two nodes in the target shape, breaking the embedding relation.

As an example of how the canonical abstraction works, consider the shape shown in Fig. 3.8a. The equivalence classes with respect to $\simeq$ are the three cell nodes and the list node. This leads to the canonical abstraction shown in Fig. 3.8b.

In general, the function $\mathcal{A}$ that defines an abstraction scheme can be used to "re-abstract the result of a shape transformation before a transition is recorded, thus ensuring a constant level of abstraction. This is referred to as a *normalized shape transition*.

Definition 51 (Normalized Shape Transition). *Let $\mathcal{S} = (I, \mathcal{R})$ be a shape transition system over a predicate set $\mathcal{P}$, and let $\mathcal{A}$ be an abstraction scheme over $\mathcal{P}$. Let further S be a shape and $P \in \mathcal{R}$ be a rule, m be a match, S^m and be its materialization. A* Normalized Shape Transition $S \xrightarrow{P,m,S^m}_{\mathcal{A}} S''$ *between S and another shape S'' exists iff there is a shape S' such that*

$$S \xrightarrow{S,m,S^m} S' \qquad \textit{and}$$
$$S'' = \mathcal{A}\left(S'\right)$$

The same shorthands as for regular shape transitions apply.

Note that normalized shape transitions are a true generalization of regular shape transitions, since a regular transition is just a normalized shape transition using id as its abstraction scheme. In the remainder of this thesis, we will assume that all shape transitions are normalized using the canonical abstraction. In the rare cases where a regular shape transition is used, we will denote it as a normalized transition using id.

With transitions defined, the next step is to define the reach set of an STS. Other than with a GTS, we can relax the isomorphism condition here, making use of embeddings instead. If a transition produces a shape that is embedded into, or *covered* by a shape that was already produced, there is not much sense in recording this new shape. Before we can use this insight to define the reach set of an STS, we first need the following auxiliary definition.

Definition 52 (Non-dominated Set). *Let* $(X, \leq)$ *be a partially ordered set. For any set* $A \subseteq X$*, the* non-dominated subset *of* A *with respect to* $\leq$*, written* $A/\leq$*, is defined by*

$$A/_{\leq} := \{x \in A \mid \nexists y \in A : x \leq y\}$$

Thus, a non-dominated set, with respect to a given partial order on (a superset of) that set, contains only elements that are not *dominated*, or *covered*, i.e. less than, some other element of the set. This now allows us to succinctly define the reach set of a shape transformation system.

Definition 53 (Reach Set of an STS). *Let* $\mathcal{S} = (I, \mathcal{R})$ *be an STS, and let* $\mathcal{A}$ *be some abstraction scheme. The* reach set *of* $\mathcal{S}$*, denoted* $\mathrm{reach}(\mathcal{S})$ *is defined as*

$$\mathrm{reach}(\mathcal{S}) := \{S \mid I \to^{*}_{\mathcal{A}} S\} /_{\sqsubseteq}$$

Note that, since $\mathcal{C}$ is a finite abstraction scheme, any algorithm that computes $reach(\mathcal{S})$ with the canonical abstraction is guaranteed to terminate, provided that the process of creating the individual shapes is guaranteed to terminate. One such algorithm is shown in Listing 3.1. The reach set is computed by alternating between two tasks: firstly, computing all shapes that can be produced from the current shape set (starting with $\{I\}$), and secondly removing from the resulting new set all covered shapes.

This makes the definition of the resulting transition system slightly more complicated, since it removes, for many transitions, the actual target shapes. These transitions must then be redirected to the shapes that cover the discarded actual target shapes. The following definition formalizes this.

Listing 3.1: The algorithm for computing the reach set of an STS

```
input $\mathcal{S} = (I, \mathcal{S}), \mathcal{A}$
$\mathcal{O}$ := $I$
$reach(\mathcal{S})$ := $\mathcal{O}$
while $\mathcal{O} \neq \emptyset$ do
  $\mathcal{N}$ := $\emptyset$
  for each $S$ in $\mathcal{O}$ do
    $\mathcal{O}$ := $\mathcal{O} \setminus \{S\}$
    for each $(P, m, S^m)$ such that $[\![\varphi_P]\!]_S^m \geq ½$ do
      $S'$ := apply $P$ to $S^m$ at id
      $S''$ := $\mathcal{A}(S')$
      $\mathcal{N}$ := $\mathcal{N} \cup \{S''\}$
    od
  od
  $reach(\mathcal{S})$ := $reach(\mathcal{S}) \cup \mathcal{N}$
  $reach(\mathcal{S})$ := nondom($reach(\mathcal{S})$)
  $\mathcal{O}$ := $reach(\mathcal{S}) \cap \mathcal{N}$
od
```

Definition 54 (Shape Transition System). *Let $\mathcal{S} = (I, \mathcal{R})$ be an STS, and let $\mathcal{C}$ be the canonical abstraction scheme. The* shape transition system *of $\mathcal{S}$, denoted* $\text{trans}(\mathcal{S}) := (N, E^1, E^2)$*, is a graph over the label set $\mathcal{R} \times \mu$, defined as follows:*

$$N := \text{reach}(\mathcal{S})$$
$$E^1 := \{(S, (P, m, S^m)) \mid S^m \textit{ is a valid materialization of } P \textit{ from } G \textit{ using } m\}$$
$$E^2 := \left\{(S, (P, m, S^m), S') \mid S \xrightarrow{P,m,S^m}_{\mathcal{C}} S' \wedge L_P \neq R_P\right\}$$

where μ is the set of all tuples of rule, match and materialization that are applicable to shapes in $\text{reach}(\mathcal{S})$.

The shape transition system thus forms the operational semantics of the shape transformation system. A shape transformation system can be seen as a generalization of a graph transformation system, created by "abstracting" the initial graph into an initial shape, using the canonical abstraction scheme. This is beneficial since, by way of the ability of shapes to represent infinite sets of graphs, a finite shape transition system might be able to represent an infinite graph transition system.

The central question now becomes whether the soundness properties obtained for individual shape transformations will transfer to the entire transition system.

Lemma 8 (Soundness of Abstract Reach Set). *Let $\mathcal{S} = (I, \mathcal{R})$ be a shape transformation system, and let $\mathcal{G} = (G_I, \mathcal{R})$ be a graph transformation system with $I = \mathcal{A}(G_I)$. Let $\mathcal{A}$ be an abstraction scheme. Then, for every graph $G \in \text{reach}(\mathcal{G})$, there is an $S \in \text{reach}(\mathcal{S})$ w.r.t. $\mathcal{A}$, such that $G \sqsubseteq_g S$ for some g.*

Proof. Let H be a graph and H' be a shape, such that $H \sqsubseteq_g H'$. Let P be a rule such that $[\![\varphi_P]\!]_H^m = \mathbf{1}$ for some m, and $H \xrightarrow{P,m} H''$. Then, by Theorem 1, $[\![\varphi_P]\!]_{H'}^{g \circ m} \geq$ ½ follows, i.e. P is at least potentially applicable to H'. If $[\![\varphi_P]\!]_{H'}^{g \circ m} = \mathbf{1}$, then by Lemma 6, there is an H''' such that $H' \xrightarrow{P,m,H'}_C H'''$ and $H'' \sqsubseteq H'''$. If $[\![\varphi_P]\!]_{H'}^{g \circ m} =$ ½, then by Lemma 5, the same follows for one of the materializations of H', i.e. there is an H''' and an $H'^{g \circ m}$ such that $H' \xrightarrow{P,m,H'^{g \circ m}}_C H'''$ and $H'' \sqsubseteq H'''$. Thus, individual transitions are sound.

Let $A := \{S \mid I \to^* S\}$ be the set of all reachable shapes, regardless of coverage status. If $G \in \text{reach}(\mathcal{G})$, then there is a path $G_0 \xrightarrow{P_0,m_0} G_1 \xrightarrow{P_1,m_1} \cdots \xrightarrow{P_k,m_k} G$ of rule applications from the initial graph G_0 to G. By the above argument, we can now inductively construct a path over $I \xrightarrow{P_0,m'_0} S_1 \xrightarrow{P_1,m'_1} \cdots \xrightarrow{P_k,m'_k} S$ over A such that $G_0 \sqsubseteq_f I, G \sqsubseteq S$ and $G_i \sqsubseteq S_i$ for all $1 \leq i \leq k$.

Therefore, for every $G \in \text{reach}(\mathcal{G})$, there is an $S \in A$ such that $G \sqsubseteq S$. By definition, for every $S \in A$ there is an $S' \in \text{reach}(\mathcal{S})$ such that $S \sqsubseteq S'$ and thus, by the transitivity of $\sqsubseteq$, we obtain the original claim. □

So, the abstract reach set is an overapproximation of the actual reach set. Thus, for simple verification tasks like the covering problem, we can use the abstract reach set in place of the original one and obtain transferable (negative) results, i.e. if a pattern cannot be found in the abstract reach set it is also absent from the concrete reach set. The converse will in general not hold. A similar result can be transferred to the abstract transition system.

Theorem 2 (Soundness of Shape Transition Systems). *Let $\mathcal{S} = (I, \mathcal{R})$ be a shape transformation system, and let $\mathcal{G} = (G_I, \mathcal{R})$ be a graph transformation system with $\mathcal{G}_I \sqsubseteq_f I$. Let $G_0 \xrightarrow{P_0,m_0} G_1 \xrightarrow{P_1,m_1} \cdots \xrightarrow{P_k,m_k} G$ be a path in $\text{trans}(\mathcal{G})$. Then there exists a path in $\text{trans}(\mathcal{S})$ such that*

$$\begin{tikzcd}
I \arrow[r, "P_0, f \circ m_0"] & S_1 \arrow[r, "P_1, g_1 \circ m_1"] & \cdots & S_k \arrow[r, "P_k, g_k \circ m_k"] & S \\
\sqcup\!|_f & \sqcup\!|_{g_1} & & \sqcup\!|_{g_k} & \sqcup\!|_{f'} \\
G_0 \arrow[r, "P_0, m_0"] & G_1 \arrow[r, "P_1, m_1"] & \cdots & G_k \arrow[r, "P_k, m_k"] & G
\end{tikzcd}$$

Proof Sketch. The argument is the same here as in the proof for Lemma 8. The existence of the path through A rather than $\mathrm{trans}(\mathcal{S})$ is assured by the soundness of shape transformation. If $G \xrightarrow{P,m} G'$ is part of the concrete path, then $S \xrightarrow{P,m,X} S'$ such that $G \sqsubseteq S$ and $G' \sqsubseteq S'$ must be part of the abstract transition system for some materialization X. Rule applicability is over-approximated in the abstraction, this guarantees the source of the transition is valid. The targets of the transitions are redirected according to the embedding between the various shapes that are produced, which guarantees the target of the transition. Note, however, that S and S' might be the same shape. □

So, if there is a path in a graph transition system that leads from the initial graph to the error, then there is also a path in any shape transition system abstracted from that graph transition system that leads to a shape that at least potentially matches the error. The converse, however, does not hold. Thus, if there is a path in a shape transition system leading to a shape that potentially (or even definitely) matches an error pattern, there is no guarantee that any given concrete graph transition system covered by the shape transition system will have a path that corresponds to their abstract error path. This is an inherent property of over-approximating abstraction approaches.

For example, when one graph transitions to another, and both of these graphs are represented by the same shape in the abstraction, then the corresponding shape transition system will have a self-loop where none existed in the graph transition system. The shape transition system thus contains arbitrarily long paths along that loop that cannot be replicated in the graph transition system.

When such a situation occurs, the abstraction itself needs to be adjusted in order to rule out the erroneous path, but still remain an overapproximation of the original system. This is the subject of the next section.

3.6 Refinement Using Shape Constraints

The abstraction scheme introduced in the previous sections is quite powerful. The introduction of ½-edges exponentially increases the number of actual graphs a shape can stand for, while the addition of a summary node even adds infinitely many interpretations. Thus, even simple shapes quickly come to represent huge sets of graphs. It is no surprise, then, that usually, the set of graphs that a shape represents contains not only the graphs it is intended to contain, but a whole lot more.

Consider, for example, the shape shown in Fig. 3.9(a). It is the linear list example used previously, representing an arbitrarily long, non-empty list, as shown by the example embeddings in Fig. 3.9(b). However, those embeddings are by no means the only ones.

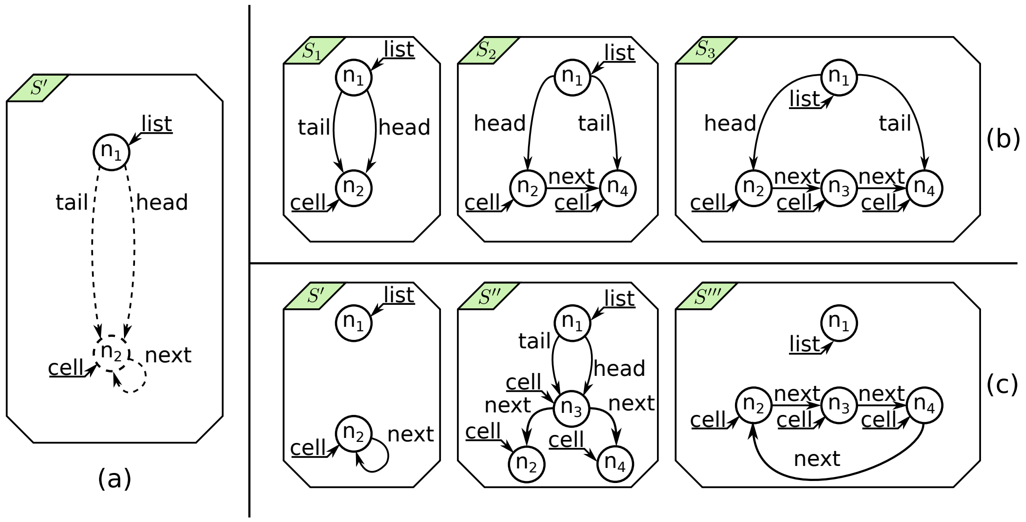


Figure 3.9: A shape and a few example embeddings

The *head-* and *tail*-edges, signifying the beginning and end of the list, illustrate this nicely. While any valid list must have exactly one pair of these edges, because they are ½-edges in the shape, their existence is not mandatory in any represented graph. Even worse, the fact that the target of these edges is a summary node means that nothing prevents them from having multiple instances, pointing anywhere within the list, not just at the beginning or end.

While these are problems that are soluble by simply redesigning the shape (see Fig. 3.3), there are other issues that are not. Such issues are caused by ½-edges connecting summary nodes, in particular, a ½-self-edge on a summary node, as seen on the *cell*-node in Fig. 3.9(a). Since there is no bound on the number of nodes that can be represented by a summary node, the ½-self-edge gives it the ability to represent nodes that are arbitrarily connected to each other by *next*-edges. Thus this one summary node can literally represent *any arbitrary graph* made up of *cell*-nodes and *next*-edges. Figure 3.9(c) shows a few such unwanted embeddings.

In the context of solving the covering problem for shape transformation systems, this is not always a big problem. After all, the objective is merely to show the absence of a given pattern from the system. If the unintended meanings of the shapes also do not contain the pattern, they do no harm to the analysis.

However, one usually finds that basic shapes are just too coarse an abstraction for any actual application. Thus, we need a way to exclude certain graphs from the graph set of a shape, while preserving others. We achieve this by using the work of Daniel Wonisch[144, 159]. We add first-order formulas to the shapes, together with assignments for their free variables. This new construct, a formula together with an assignment of its free variables to the nodes of a shape, is called a *shape constraint*[‡]. The intention of this is that any graph embedded into the shape has to also satisfy the attached formulas using all assignments that are compatible with the assignment to the shape via the embedding used.

As a side note, this solution is in many ways similar to the notion of *instrumentation predicates* introduced by Sagiv et.al. in their paper on shape analysis[134] and their follow-up paper on automatic updates for instrumentation predicates[129]. However, it differs in several key aspects, most notably in the way shape constraints are maintained across rule applications, which provides several benefits over instrumentation predicates, such as completeness under certain circumstances[159]. Shape constraints are formally defined as follows.

Definition 55 (Shape Constraint). *A* shape constraint (α, m) *for a shape* $S = (U, \mathcal{P}, \iota)$ *is a first-order formula* α *over* $\mathcal{P} \setminus \{sm\}$ *with free variables* $\mathrm{F}(\alpha)$*, together with a complete assignment* $m : \mathrm{F}(\alpha) \to U$.

Using shape constraints, the meaning of a given shape can be *refined*. Shape constraints are not (necessarily) mutually exclusive, i.e. a shape can have any number of shape constraints attached to it, tuning its represented graph set to a very specific set of graphs. This idea is formalized in the notion of the *constrained shape*.

Definition 56 (Constrained Shape). *A* Constrained Shape *is a tuple* (S, Λ)*, where* S *is a shape, and* Λ *is a set of shape constraints for* S.

The visual representation of a constrained shape shows the shape, the formulas of its constraints, and, for every single constraint, a hyperedge, labeled with the formula of the constraint and expressing the assignment using its tentacles labeled with the variable names.

Having defined what shape constraints are and what they look like, we must now define just what precisely it means to attach such a constraint to a shape. The basic idea behind the meaning of shape constraints is that any graph or shape to which a shape constraint is attached, has to *satisfy* that constraint, i.e. the formula, together with the given assignment, must not evaluate to $\mathbf{0}$ in the shape it is attached to.

For graphs, this is a binary proposition. Since all literals in the formula will be interpreted with definite values, the constraint formula either evaluates to $\mathbf{1}$, or it evaluates to $\mathbf{0}$. If it evaluates to $\mathbf{0}$, then the graph is invalid and can be discarded or ignored. If it evaluates to $\mathbf{1}$, then the graph is valid and the information contained in the constraint was already contained in the graph to begin with.

[‡]Daniel Wonisch called them*deductive constraints*

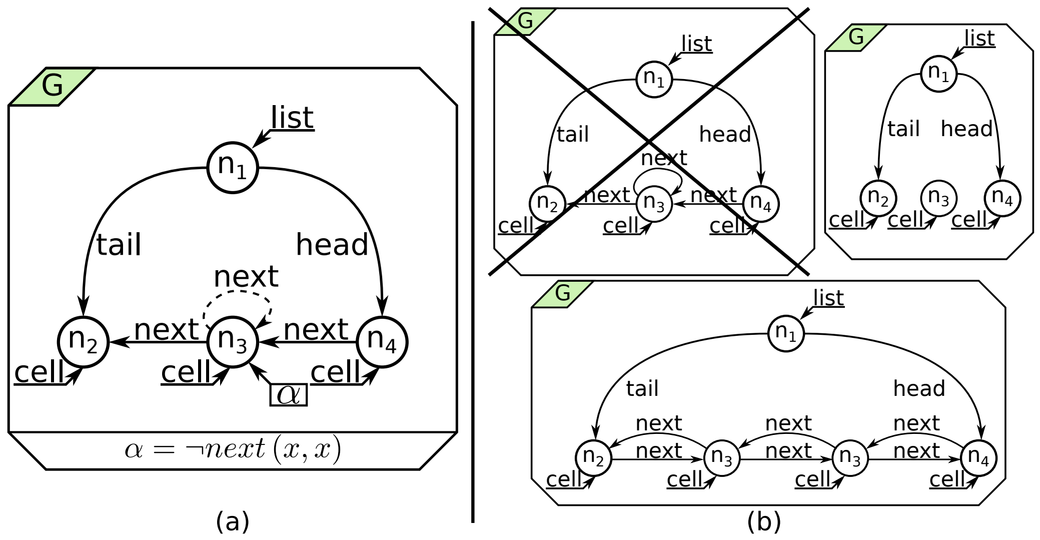


Figure 3.10: A constrained shape with valid and invalid represented graphs

For shapes, however, the third option, ½, would indicate that some of the represented graphs are valid with respect to the constraint, whereas some are not. This then represents a true *gain in precision* over an unconstrained shape.

As an example. consider the constrained shape shown in Fig. 3.10(a). It enhances the unconstrained shape from Fig. 3.3 with a constraint that disallows self-edges in the represented graphs. Note that the constraint uses only a single variable, ensuring that the constraint behaves just like a unary edge under embedding – Any node of a graph embedded into the constrained shape, that is mapped onto the cell node of the shape, will have to satisfy its own version of the constraint. Figure 3.10(b) shows valid embeddings into the shape, only some of which are still valid for the constrained shape.

In order to properly define the graph and shape sets represented by a constrained shape, we need a proper mathematical definition of constraint semantics. These are based on the idea introduced above – a shape or graph for which a constraint evaluates to $\mathbf{0}$ is considered invalid and not part of the actual graph or shape set.

Definition 57 (Constraint Satisfaction). *Let S be a shape, and let (α, m) be a constraint for S. If $[\![\alpha]\!]_S^m \geq$ ½, S is said to* satisfy α. *Let Λ be a set of constraints. If S satisfies every constraint in Λ, S is said to* satisfy Λ, *written $S \models \Lambda$.*

Note that in the case of graphs, the above condition becomes $[\![\alpha]\!]_S^m = \mathbf{1}$, meaning that a constraint is satisfied on a graph, if it evaluates to true. Note also, that, when a constraint evaluates to $\mathbf{1}$ in a shape, then the shape already contains all the information the constraint provides, rendering the constraint useless. On the other hand, if a constraint evaluated to $\mathbf{0}$ on a shape, then no embedded graph or shape could possibly satisfy it, meaning that the shape no longer represents any graphs. Any useful constrained shape therefore will have to satisfy all of its constraints, leading to the concept of *validity*.

Definition 58 (Constrained Shape Validity). *Let* (S, Λ) *be a constrained shape.* (S, Λ) *is said to be* valid*, iff* $S \models \Lambda$.

Given a valid constrained shape, we can now move on to the central question: which shapes and graphs are actually *represented*, or *covered* by it?

Definition 59 (Graph and Shape Sets of a Constrained Shape). *Let* (S, Λ) *be a constrained shape. The* set $\mathcal{G}(S, \Lambda)$ of represented graphs *is defined as follows:*

$$\begin{aligned} \mathcal{G}(S, \Lambda) &:= \{G \mid G \sqsubseteq_f S \land G \models \mathrm{concr}_f(\Lambda) \land G \text{ 2-}\textit{valued}\} \\ \mathcal{S}(S, \Lambda) &:= \{S' \mid \mathcal{G}(S') \subseteq \mathcal{G}(S)\} \qquad\qquad \textit{, where} \\ \mathrm{concr}_f(\Lambda) &:= \{(\alpha, m') \mid (\alpha, m) \in \Lambda \land m = f \circ m'\} \end{aligned}$$

This definition, specifically the set $\mathrm{concr}_f(\Lambda)$, is reminiscent of the definitions for matching and materialization, generalized from matching formulas to arbitrary formulas. It "projected", in a sense, the constraints of the shape down onto a graph along a classical embedding, where they can then be evaluated.

Note that, for both the shape and the graph sets, the requirement is that all constraints evaluate to $\mathbf{1}$. Given our previous definition of validity, this may seem surprising, especially for the shape set. However, this condition follows directly from our intuition about embeddings, which state that embeddings should imply subset relationships on the graph and shape sets. Thus, if an unconstrained shape is to be embedded in a constrained shape, then its own graph and shape sets must not have to be modified in order to make the embedding work. This means that the constraints projected down onto the shape must not restrict the shape set. For the unconstrained case, we can now define embeddings using our definition of the graph and shape sets above.

Definition 60 (Embedding of Unconstrained Shapes into Constrained Shapes). *Let* (S, Λ) *be a constrained shape. An unconstrained shape* S *is* embedded into (S, Λ) *iff* $S \in \mathcal{S}(S, \Lambda)$. *A graph* G *is* embedded into (S, Λ)*, iff* $G \in \mathcal{G}(S, \Lambda)$.

As an example, let $G = (U, \mathcal{P}\iota)$ be the top left graph in Fig. 3.10(b), and let

$$\left(S = \left(U', \mathcal{P}, \iota'\right), \Lambda = \left\{\left(\alpha, m' = \left[x \mapsto n_3'\right]\right)\right\}\right)$$

be the constrained shape in Fig. 3.10(a). The function

$$f := \left\{\left(n_1 \mapsto n_1'\right), \left(n_2 \mapsto n_2'\right), \left(n_3 \mapsto n_3'\right), \left(n_4 \mapsto n_4'\right)\right\}$$

is the only possible embedding of G into S, without taking constraints into account. The only assignment $m : \mathrm{F}(\alpha) \to U$ such that $f \circ m = m'$ is $[x \mapsto n_3]$. Thus, $\mathrm{concr}_f(\Lambda) = \{(\alpha, [x \mapsto n_3])\}$. Evaluating this, we see that $[\![\alpha]\!]_G^m = \mathbf{0}$ and thus $G \not\models \mathrm{concr}_f(\Lambda)$, i.e. $G \notin \mathcal{G}(S, \Lambda)$, even though clearly $G \in \mathcal{G}(S)$.

Thus, in this case, the set of graphs represented by the constrained shape is a true subset of the set of graphs represented by the original shape. This is not always the case when a constraint evaluates to ½ in a shape, since the constraint may be implied by other constraints already present. Adding a constraint will, however, never *increase* the represented graph and shape sets.

We now turn to embeddings between constrained shapes. Naturally, we want such embeddings to resemble as closely as possible embeddings on unconstrained shapes. This means, in particular, that embeddings should also reflect subset relations between the graph sets of the respective shapes, i.e. if a shape (S, Λ) is embedded into a shape (S', Λ'), then we must have $\mathcal{G}(S, \Lambda) \subseteq \mathcal{G}(S', \Lambda')$. As in the case of an unconstrained shape embedded into a constrained one, this means that the constraints that are projected onto S by S' via the embedding must be *redundant*, i.e. not restrict the graph and shape sets of (S, Λ) any further. Unlike before, we no longer have access to a simple test to determine whether this is the case. The projected constraints may well evaluate to ½ and still be redundant, because their meaning is implied by the constraints already in place. Thus, in order to check for an embedding relation between two constrained shapes, we would need to check whether the projected constraints are implied by the combined meaning of the (potentially) embedded shape and its constraints. For now, we will define this axiomatically. Chapter 4 will provide the tools which we will use in Chapter 5 to create a more workable definition.

Definition 61 (Embedding into a Constrained Shape – Constrained). *A constrained shape* (S', Λ') *is said to be* embedded into (S, Λ)*, written* $(S', \Lambda') \sqsubseteq_f (S, \Lambda)$*, iff*

$$\mathcal{G}(S', \Lambda') \subseteq \mathcal{G}(S, \Lambda)$$

As an example, consider the two constrained shapes in Fig. 3.11. The left shape describes a linear list where no *cell* node has a *next* self-edge. The right shape describes a linear list where no *cell* node may be the target of a *head* edge, and a *tail* edge at the same time. It is obvious that any valid linear list with more than one cell will be embedded in both of these shapes. However, neither graph set is a subset of the other. A linear list with only one *cell* node is embedded into the left shape, but not the right one. A linear list with *next* self-edges is allowed by the right shape, but not the left one. Therefore, no embedding relationship exists between these two shapes.

Having defined shape constraints and their effects on shapes, we can now refine the

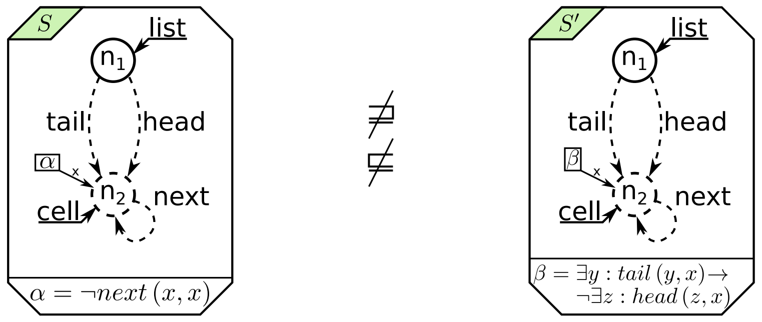


Figure 3.11: Two compatible, but mutually non-embedded constrained shapes

meaning of singular shapes. This is fine for refining abstractions of structural models, but this thesis is concerned with behavioral models, i.e. graph and shape transformations. Combining shape constraints with transformations poses a problem, however – the graph rules were defined in the absence of constraints and thus contain no information about their effect on them.

Clearly, this is a problem, since a constraint could, for example, require the non-existence of a certain substructure in a graph. When a graph rule then constructs just that substructure, the shape constraint cannot be left constant before and after the transformation, because the resulting shape will become invalid.

Therefore, we need a way to *update* shape constraints, i.e. to enable them to incorporate new situations created by rule applications. The basic idea of how to do this is to keep the meaning, rather than the formulas or assignments, of shape constraints *constant*.

The reasoning behind this idea is the following: Let's assume we have a constrained shape (S, Λ) and a rule P that matches (S, Λ) at a match m. Without loss of generality, we assume that $\Lambda = \{(\alpha, g)\}$ is a single constraint, that m is a definite match, and that S' is the result of classically applying P to S. Then all graphs in $\mathcal{G}(S, \Lambda)$ match P at a match compatible to m. Applying the rule will now change these graphs in some way. If the change does not affect the parts of the graphs that Λ makes assertions about, then Λ still holds on S'. If, however, the edge and node changes introduced by the rule change the interpretation of literals in α or the assignment g, then the meaning, i.e. the value, of the constraint will change due to the rule application.

Usually, however, a rule does not change *everything* that a constraint makes assertions about. For example, when a constraint asserts certain properties of three nodes in the shape, and a rule eliminates one of them, the properties of the other two should still be satisfied, i.e., the constraint should still evaluate to the same value as before, only now stripped of any reference to the node that was deleted. This motivates the idea that, when applying the rule P to (S, Λ), α' and g' should be constructed such that

$$[\![\alpha]\!]_S^g = [\![\alpha']\!]_{S'}^{g'}$$

The constrained shape $(S', \Lambda' = \{\alpha', g'\})$ would then be the result of applying P to (S, Λ) at m.

Of course, one could just set each shape constraint on a shape to a constant of the appropriate value, and set the assignments to empty maps to achieve this effect. However, this would obviously destroy any meaning inherent in the constraints. Rather, we try to find formulas α' that are as similar as possible to the original α, but exhibit the above property.

Now, what does this mean in concrete terms? To illustrate this, we look at a simple example before examining the (quite extensive) formal definition.

Example (Edge Deletion). *Assume that we have a constraint that asserts the existence or non-existence, respectively, of certain edges in a shape S. This means that the constraint is a long conjunction of literals, with the free variables in the literals bound to certain nodes in the shape by the assignment, such as this one:*

$$\begin{aligned}(p_1(x_1, x_2) \land \neg p_2(x_2) \land \neg p_3(x_4, x_4) \land p_3(x_4, x_5)\,,\\ [x_1 \mapsto n_1, x_2, x_3 \mapsto n_2, x_4 \mapsto n_3, x_5 \mapsto n_4])\end{aligned}$$

If a rule application now deletes the p_1-edge from n_1 to n_2, then the above constraint will evaluate to false, even though the information about the remaining edges is still valid. To combat this, we simply replace the term that carried the value of the deleted edge with a constant containing its former value ($\mathbf{1}$), leading to:

$$\begin{aligned}&(\iota_S(p_1)(n_1, n_2) \land \neg p_2(x_2) \land \neg p_3(x_4, x_4) \land p_3(x_4, x_5)\,,\\ &\qquad [x_2, x_3 \mapsto n_2, x_4 \mapsto n_3, x_5 \mapsto n_4])\\ \equiv\, &(\mathbf{1} \land \neg p_2(x_2) \land \neg p_3(x_4, x_4) \land p_3(x_4, x_5)\,,\\ &\qquad [x_2, x_3 \mapsto n_2, x_4 \mapsto n_3, x_5 \mapsto n_4])\\ \equiv\, &(\neg p_2(x_2) \land \neg p_3(x_4, x_4) \land p_3(x_4, x_5)\,, [x_2, x_3 \mapsto n_2, x_4 \mapsto n_3, x_5 \mapsto n_4])\end{aligned}$$

The resulting constraint preserves the information about the remaining edges without conflicting with the result of the rule application.

It is useful, as a metaphor, to think of the transformed constraint as expressing as much of the original constraint as possible in the transformed shape.

Of course, most rule applications do not just delete a single edge. But as we will see in the formal definition, all effects of rules can be dealt with in a similar matter. Deleted nodes can simply be removed from a formula by replacing all literals that contain it. Added nodes do not touch any constraint, unless the constraint includes a quantifier, in which case the constraint can be transformed by adding an exception for the new node to the formula affected by the quantifier. Combinations of such effects do not collide with each other, since they all involve replacing literals with constants for their interpretation in the original shape, as well as addition of new subformulas.

The following definitions, simplified from Daniel Wonisch's master thesis[159], formalize the intuitive ideas described above.

Definition 62 (Transformation of Constrained Shapes). *Let (S, Λ) be a constrained shape, $P = (L, R)$ be a rule and let $m : L \to S$ be a match such that $[\![\varphi_P]\!]_S^m = \mathbf{1}$. Let further be S' such that $S \xrightarrow{P,m} S'$. Then the* result of applying P to (S, Λ) at m *is defined as*

$$\begin{aligned}
&(S', \Lambda') && \text{, where}\\
\Lambda' &:= \left\{\left(c', m'\right) \mid (c, m) \in \Lambda\right\} && \text{and each}\\
m' &:= \left(m \cup id_{U_{S'}}\right) \cap \left(\mathrm{F}\left(c'\right) \times U_{S'}\right)\\
c' &:= replace\left(c, \left(m \cap \left(\mathrm{F}\left(c\right) \times \left(U_S \setminus U_{S'}\right)\right)\right)\right)
\end{aligned}$$

Thus, for each shape constraint, the formula is transformed using the $replace$-algorithm. The assignment is then cleared of any assignments from removed variables or to removed nodes, as well as extended by the identity map of any nodes added. The $replace$ algorithm is the subject of the following definition.

Definition 63 ($replace$-Algorithm). *The replace algorithm operates on first-order formulas and is defined inductively as follows:*

Literals *For $l \in \mathcal{K}$, we define $replace\,(l, \bar{m}) := l$. Now, let $(p, k) \in \mathcal{P}$ be a predicate, and $v_1, \ldots, v_k$ be variables:*

- *If $k = 0$, i.e., if p is a constant, then*

$$replace\,(p(), \bar{m}) = p()$$

- *If $k = 1$ and $\bar{m}\,(v) \notin U_{S'}$, then*

$$replace\,(p\,(v)\,, \bar{m}) = \iota_S\,(p)\,(\bar{m}\,(v))$$

- *If $k = 2$ and $\bar{m}\,(v_1)\,, \bar{m}\,(v_2) \notin U_{S'}$, then*

$$replace\,(p\,(v_1, v_2)\,, \bar{m}) = \iota_S\,(p)\,(\bar{m}\,(v_1)\,, \bar{m}\,(v_2))$$

- *If* $k = 2$ *and* either $\bar{m}(v_1) \notin U_{S'}$ *or* $\bar{m}(v_2) \notin U_{S'}$, *let w.l.o.g.* $\bar{m}(v_1) \notin U_{S'}$ *and:*

$$replace(p(v_1, v_2), \bar{m}) := \left(\bigwedge_{u \in m(N_L \cap N_R)} ((v_1 = u) \to \iota_s(p)(u, \bar{m}(v_2))) \right) \wedge \left(\bigvee_{u \in m(N_L \cap N_R)} (v_1 = u) \right)$$

- *If* $k = 1$, *let* $H^1 := \{u \mid u \in U_S \cap U_{S'} \wedge \iota_S(p)(u) \neq \iota_{S'}(p)(u)\}$ *be the set of individuals that are neither added nor removed by the rule and for which the interpretation of* p *changes when the rule is applied. Then:*

$$replace(p(v), \bar{m}) := \left(\bigwedge_{u \in H^1} (v = u) \to \iota_S(p)(u) \right) \wedge \left(\left(\bigwedge_{u \in H^1} \neg(v = u) \right) \to p(v) \right)$$

- *If* $k = 2$, *let* $H^2 := \{(u_1, u_2) \mid u_1, u_2 \in U_S \cap U_{S'} \wedge \iota_S(p)(u_1, u_2) \neq \iota_{S'}(p)(u_1, u_2)\}$ *be the set of individual pairs that are neither added nor removed by the rule and for which the interpretation of* p *changes when the rule is applied. Then:*

$$replace(p(v_1, v_2), \bar{m}) := \left(\bigwedge_{(u_1, u_2) \in H^2} ((v_1 = u_1) \wedge (v_2 = u_2)) \to \iota_S(p)(u_1, u_2) \right) \wedge \left(\left(\bigwedge_{(u_1, u_2) \in H^2} \neg((v_1 = u_1) \wedge (v_2 = u_2)) \right) \to p(v_1, v_2) \right)$$

- *If* $(p, k) = (=, 2)$, *then*

$$replace((v_1, v_2), \bar{m}) := \begin{cases} \mathbf{1} & \textit{if } v_1, v_2 \in dom(\bar{m}) \wedge \bar{m}(v_1) = \bar{m}(v_2) \\ \mathbf{0} & \textit{else, if } v_1 \in dom(\bar{m}) \vee v_2 \in dom(\bar{m}) \\ (v_1 = v_2) & \textit{otherwise} \end{cases}$$

Logical Connectives *For formulas φ_1 and φ_2:*

$$\begin{aligned} replace\left(\varphi_1 \vee \varphi_2, \bar{m}\right) &:= replace\left(\varphi_1, \bar{m}\right) \vee replace\left(\varphi_2, \bar{m}\right) \\ replace\left(\varphi_1 \wedge \varphi_2, \bar{m}\right) &:= replace\left(\varphi_1, \bar{m}\right) \wedge replace\left(\varphi_2, \bar{m}\right) \\ replace\left(\neg\varphi_1, \bar{m}\right) &:= \neg replace\left(\varphi_1, \bar{m}\right) \end{aligned}$$

Quantifiers *For a formula φ and a variable $v \notin dom\left(\bar{m}\right)$ (accomplished by consistent renaming):*

$$\begin{aligned} replace\left(\forall v : \varphi, \bar{m}\right) :=& \left(\forall v : \left(\bigwedge_{u \in U_{S'} \setminus U_S} \neg\left(v = u\right)\right) \rightarrow replace\left(\varphi, \bar{m}\right)\right) \\ & \wedge \left(\bigwedge_{u \in U_S \setminus U_{S'}} replace\left(\varphi, \bar{m}\left[v \mapsto u\right]\right)\right) \\ replace\left(\exists v : \varphi, \bar{m}\right) :=& \left(\exists v : \left(\bigwedge_{u \in U_{S'} \setminus U_S} \neg\left(v = u\right)\right) \rightarrow replace\left(\varphi, \bar{m}\right)\right) \\ & \vee \left(\bigvee_{u \in U_S \setminus U_{S'}} replace\left(\varphi, \bar{m}\left[v \mapsto u\right]\right)\right) \end{aligned}$$

A shape constraint, updated using the above definition, always preserves as much information of the original constraint as possible, while maintaining a stable evaluation across rule applications.

In the original thesis[159], shape constraints (then called *deductive constraints*, built on top of other abstraction refinement methods) are proven to update *precisely*, under certain conditions. This means that if rule application is performed in DPO, and certain additional properties are satisfied, e.g. that rules may never create edges that are already present, rule application is *complete*. Each graph in the represented graph set of the resulting constrained shape is the result of applying the rule to some graph in the original represented graph set.

In this thesis, SPO is used, and some of the conditions imposed in Wonisch's work are impractical this context. Thus, here, a weaker version of the correctness theorems is used.

Theorem 3 (Soundness of Constrained Shape Transformation). *Let (S, Λ) be a constrained shape, and let (S', Λ') be the result of applying a rule P at a definite match m. Then, for every $G \sqsubseteq_f S, \Lambda$, and every $m' : L_P \rightarrow G$ such that $m = f \circ m'$, there*

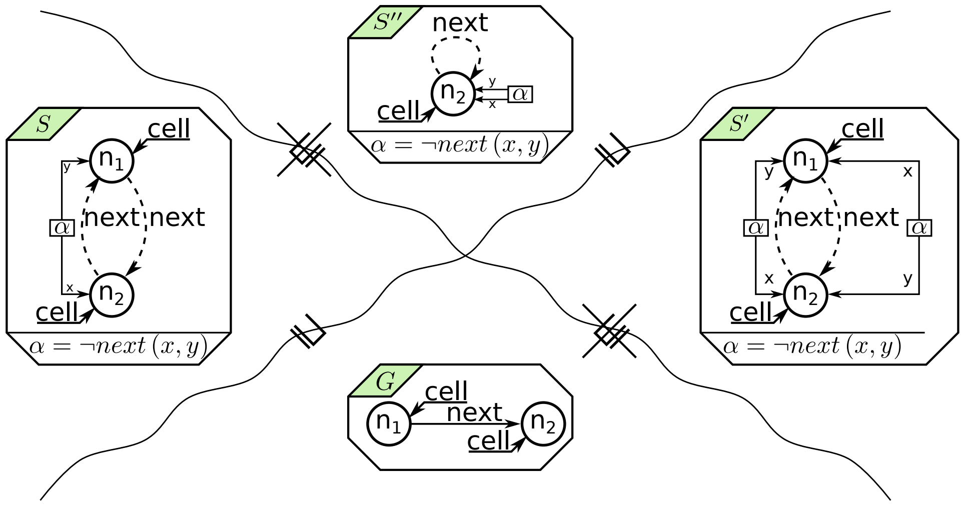


Figure 3.12: The graph G is excluded by a canonical abstraction that disregards the shape constraint α

is a unique graph G' such that

$$G \xrightarrow{P,m'} G' \qquad \textit{and}$$
$$G' \sqsubseteq (S', \Lambda')$$

i.e., applying P to (S, Λ') as per Def. 62 is sound.

Proof. See Daniel Wonisch's master thesis[159]. $\square$

Shape constraints are a tool for refinement, i.e. the restriction of shapes to smaller and smaller sets of represented shapes and graphs. It should come as no surprise then, that the very concept of shape constraints is at odds with the concept of an abstraction scheme, concerned with *enlarging* the graph and shape sets of shapes. Specifically, when an abstraction scheme merges nodes together to obtain more general graphs, the question arises how constraints that are attached to (some of) those nodes affect the outcome.

As an example, consider the canonical abstraction scheme presented in Def. 50. The naïve method of dealing with shape constraints in an abstraction scheme would be to simply apply the scheme as before, redirecting the assignments of the constraints as dictated by the merging of nodes. However, due to the definition of embedding in the presence of shape constraints (see Def. 60), this greatly expands the meaning of the constraint in almost all cases, resulting in a *restriction* of the graph set of the shape, rather than an *expansion*.

Consider the shape (S, Λ) shown in Fig. 3.12. It represents two *cell* nodes, possibly connected with *next*-edges. It also has a constraint attached to it, forbidding one of the two possible directions for the *next*-edge.

If we were to abstract this shape using the canonical abstraction scheme, we would arrive at the shape (S', Λ'). Here, both *cell* nodes have been merged, since they did not differ in the values of the unary predicates (*cell* and *list*). The potential *next*-edge is now a self-edge, and the constraint now assigns both its variables to the merged node.

The problem now is this: in order for the original shape (and all the graphs in its graph set) to still be embedded into the shape $\mathcal{C}(S, \Lambda)$, a requirement for a valid abstraction scheme in Def. 49, it must satisfy not just its original constraint, but also all constraints in $concr_f(\Lambda')$, where f is the embedding function induced by $\mathcal{C}$. As we can see in Fig. 3.12, this would mean that the resulting constraints rule out the existence of *any next*-edge, excluding the graph G from the graph set, even though it was embedded in the original shape (S, Λ).

Thus, abstraction schemes, if they are to be employed in the context of constrained shapes, must take shape constraints into account. In the remainder of this thesis, we will use the following modified canonical abstraction scheme. Given that there are an infinite number of possible constraints that can be attached to a shape, this scheme is no longer finite.

Definition 64 (Constraint-safe Abstraction). *Let $\mathcal{P}$ be a predicate set, and let $\mathcal{U} \subseteq \mathcal{P}$ be its subset of unary predicates. Let $\mathcal{F}$ be the set of all first-order formulas over $\mathcal{P}$ and a variable set $\mathcal{V}$. Let (S, Λ) be a shape and let $\mathcal{C}(S)$ be its unconstrained canonical abstraction. Let further $(X := \{S' \mid S \sqsubseteq S' \wedge S' \sqsubseteq \mathcal{C}(S)\}, \sqsubseteq)$ be the partially ordered set of shapes "between" S and its canonical abstraction. Then the* Constraint-safe canonical Abstraction *of (S, Λ) is given by a maximal (w.r.t. $\sqsubseteq$) shape $\overline{\mathcal{C}}(S, \Lambda) := (S', \Lambda')$ such that*

$$\begin{aligned} S \sqsubseteq_f S' \sqsubseteq_g \mathcal{C}(S) & \qquad \textit{and thus } S' \in X \\ \Lambda' = \{(\alpha, m) \mid \exists m' : (\alpha, m') \in \Lambda \wedge m = f \circ m'\} & \\ \mathcal{G}(S, \Lambda) = \mathcal{G}(S, \Lambda \cup \operatorname{concr}_f(\Lambda')) & \end{aligned}$$

With this definition, constraint-safe abstraction is derived from the classical canonical abstraction scheme with the additional property that no nodes may be merged such that the thusly implied additional constraints to the original shape further restrict its meaning. There are several things to note here. First, this is a declarative definition rather than a constructive one, and multiple implementations of it are possible. This is because, since we had to abandon the goal of a finite abstraction scheme, the particular implementation or construction of the abstraction is of lesser importance. Second, the canonical abstraction of a shape, under this definition, is not unique. Uniqueness of the abstraction is not part of the definition of an abstraction scheme and is really only relevant when a finite target set of abstractions is the goal of the scheme, or when multiple implementations of the abstraction scheme are required to reliably generate the same abstractions. This is not the case for us. We only require that the abstraction create an abstract shape into which the original shape is still embedded – which is true by definition – and that the scheme is idempotent, which it is by virtue of the idempotence of the classical canonical abstraction and the requirement of (local) maximality of the shape.

The above definitions now enable us to refine our analysis using first-order formulas. This provides a highly expressive tool to tune the analysis to precisely the GTS and error that are given, without giving up too much abstraction.

However, it is still unclear, where these first-order formulas should come from. Requiring them to be constructed by hand hardly seems feasible in any practical scenario.

Furthermore, we have not yet solved the problem of spurious errors. If our analysis proves an STS to be error-free, we obtain a result that is transferable to the original GTS. However, if an error is found, we do not have a proper way of figuring out whether there is an error in the GTS that corresponds to it. And if it doesn't, what does this imply about our abstraction?

These questions will be answered in the next chapter, where we will utilize the power of state-of-the art SMT solvers to test error paths for feasibility and derive new constraints from them.

4 SMT Encoding of Graph Embeddings and Traces

The expressive power of first-order logic is the key to the effectiveness of the GTS abstraction scheme presented in the previous chapter. It enables a very precise fine-tuning of the abstraction, that allows for very complex information to be incorporated. Such information could, for example, be provided by the error pattern, or the rules of the GTS. Even domain knowledge, i.e. knowledge about what the graphs are supposed to *represent* can be incorporated.

It is not surprising, then, that such extensive abilities come with a price. The price of using unrestricted first-order logic is two-fold: undecidability and the curse of dimensionality.

The first aspect, undecidability, is well-known. Unrestricted first-order logic is undecidable[*], meaning that there is no decision procedure that can decide the satisfiability of a given first-order formula in all cases in finite time. This directly implies that, for example, it is impossible to construct an algorithm that can decide for any given constrained shape S whether it is feasible, i.e., if $\mathcal{S}\left(S\right) \neq \emptyset$ holds. This problem is not limited to feasibility. Deciding, for example, whether a given abstract error trace has any concrete counterparts, is also undecidable in general.

[*]This was proved by Alan Turing and Alonzo Church in the 1930s

This is not surprising, though, since the covering problem itself is already undecidable for infinite-state GTSs[30]. The goal for an approach to solving the covering problem for infinite-state systems therefore should not be decidability, but a reasonable expectation of solvability. This means that the approach should work for as many systems as possible, as efficiently as possible. In the case of our approach, which uses first-order logic, the natural approach to achieving this is to utilize the incredible progress in the field of SMT solving that has been made in recent years, and continues to be made today.

The second aspect, the curse of dimensionality, meaning an explosion in the size of the design space, is a direct implication of the expressive power of first-order logic. There are abstraction refinement schemes, such as the one proposed by Rensink et.al.[36] (see Sec. 8.1), that only provide two dimensions for refinement. In this case, they are the radius of the neighborhood, i.e. the context of nodes used to decide which nodes are "similar", and the upper bound for precise node counting, given by integers k and ω. If the abstraction is too coarse, there is really only one thing to do: increase k or ω. This is easy enough to automate, but severely limits the ability of the approach to actually adapt to the problem at hand.

When using unrestricted first-order logic for shape refinement the problem is the reverse. The infinite number of possibilities for refinement make it extremely difficult to design refinement formulas (shape constraints) without the help of a human designer. The techniques provided in this chapter form the basis of our approach to this problem. The approach itself is given in Chapters 5 and 6.

4.1 Motivation

The close connection between graphs and logic made in the first two chapters of this thesis allows us to frame nearly all tasks that make up the state space construction algorithm for STS as satisfiability problems. Some of these tasks are not easily performed in the domain of graphs. The central example for this in this thesis is the problem of finding a concrete error trace to correspond to a given abstract trace.

Let $\mathcal{S} = (S_0, \mathcal{R})$ be an STS, E be an error pattern, and

$$S_0 \xrightarrow{P_1, m_1, S_1^m} S_1 \xrightarrow{P_2, m_2, S_2^m} S_2 \xrightarrow{P_3, m_3, S_3^m} \cdots \xrightarrow{P_n, m_n, S_n^m} S_n$$

be a path in $\mathcal{S}$, such that we have $[\![\varphi_E]\!]_{S_i}^m = \mathbf{0}$ for all m and $0 \leq i \leq n-1$, as well as $[\![\varphi_E]\!]_{S_n}^{m_e} \geq$ ½ for some potential match m_e. Let further $\mathcal{G} = (I, \mathcal{R})$ be a GTS over the same rule set, such that $G \sqsubseteq S_0$.

If we want to know whether the error we have just found on the abstract level, i.e. in the STS, is also present on the concrete level, i.e. in the GTS, we need to check for the

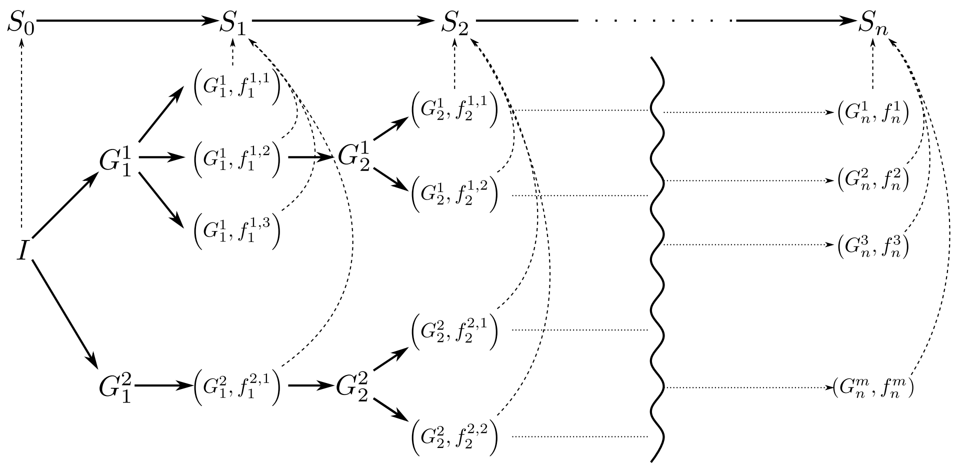


Figure 4.1: The naïve process of counterexample validation

existence of a path

$$I \xrightarrow{P_1, m'_1} G_1 \xrightarrow{P_2, m'_2} G_2 \xrightarrow{P_3, m'_3} \cdots \xrightarrow{P_n, m'_n} G_n$$

in the GTS such that for all $1 \leq i \leq n$ we have

$$G_i \sqsubseteq_{f_i} S_i \qquad \text{as well as}$$
$$m_i = f_{i-1} \circ m'_i \qquad \text{and}$$
$$\exists m'_e : m'_e = f_n \circ m_e \wedge [\![\varphi_E]\!]^{m'_e}_{G_n} = \mathbf{1}$$

meaning, every i-th graph in the concrete path is embedded into the i-th shape of the abstract path, and each match (including the *positive* error match) used in the concrete path is compatible via the appropriate embedding with the corresponding abstract match.

The direct, naïve approach to finding such a path would be to start with I, compute all matches that are compatible with m_1, and list, for every resulting graph, all embeddings with which they might be embedded into S_1. Then, for each pair of such a resulting graph and a possible embedding, repeat the process, constructing, in essence, a tree of paths through the transition system of the GTS, in which every level corresponds to a single shape in the abstract path. Figure 4.1 shows a schematic of this process.

The tree constructed in this manner is potentially huge, especially when the counterexample is particularly long and actually valid. The prospect of having to construct these huge trees for every abstract error trace that is found motivates the search for alternatives.

The problem of having to validate counterexamples is, of course, not limited to abstract graph transformation systems, but abstractions of dynamic systems in general. Established methods in this domain, such as predicate abstraction[81], already have proven ways of dealing with counterexample validation.

In predicate abstraction, an abstraction technique used to verify imperative programs, each counterexample is transcoded into a so-called *path formula*. This formula encodes the possible program states along the path using consistent renaming of the state variables (static-single-assignment, SSA). The advantage of this is that the states need not be explicitly constructed. Instead, the formula is given to an appropriate solver, which attempts to find a satisfying interpretation, i.e. a "model". Any model found by this solver represents an actual program path corresponding to the abstract error path. The solver used is usually a so-called "SMT solver", i.e. a program capable of finding satisfying assignments for first-order logic formulas with certain pre-interpreted predicates and functions, called a "theory".

In this chapter, I develop encodings of graph transformation systems into SMT formulas. This will provide a way to encode all relevant checks that come up during the construction of a shape transition system, including counterexample validation, in SMT. Performing these checks via an SMT solver provides a much more efficient and effective way to decide feasibility for shapes and traces (see Chapter 5). It can also serve as an information source for abstraction refinement (see Chapter 6).

A Short Survey of Logical Encodings of Graphs and Their Properties

Given the conceptual similarity between logical structures and graphs, it is not surprising that the idea of encoding graphs in logic is not new, and in fact, many techniques exist that in some way, as their central goal or as a side-effect or supporting factor, use various logics to encode graphs and their properties in logic. For the purposes of this short survey, we will distinguish between two categories of such techniques – on the one hand, techniques that are used to assert or express information *about* graphs, and on the other hand, techniques that use logic to express the graphs themselves.

A clear example of the first category of techniques is the OCL[50, 120] (Object Constraint Language). This specialized constraint language is part of the UML[1] family of languages and covers a large variety of applications within it. One of these applications is to restrict, e.g., the meaning of class diagrams, for example by disallowing the coexistence of certain types of associations (edges) or requiring that certain instances of classes (nodes) must always exist together, even when they are not connected by associations. In the intention of this particular instance, the OCL comes relatively close to the concept of shape constraints presented in the previous chapter. However, while shape constraints are simply first-order logic formulae related to graphs via assignments into corresponding logical structures, the formal semantics of OCL, i.e. the mathematical underpinnings are much wider, as they need to incorporate other modeling aspects, such as attributes, basic data types or operations on non-graphical aspects of a model. There is an official formal semantics of the OCL[6], but its development is ongoing and inconsistencies linger in the official specification[37].

As a more formal example of the use of logic to express properties of graphs (and graph transformation systems), we consider the wide-spread use of monadic second-order logic. It is commonly understood that monadic second-order logic is uniquely suited for expressing properties of graphs and graph transformations[133]. Some techniques for various applications use it to this effect[37, 56, 102]. However, greater expressive power always comes at the cost of feasibility – since monadic second order logic constitutes an extension of first-order logic, decision procedures for it must be at least as complex as decision procedures for first-order logic. That, and the comparative dearth of tools capable of deciding monadic second order logic, let alone interpolation support (see Chapter 6) led to our decision to stick with first-order logic.

In the second category we find several approaches that encode entire graphs in logic, usually to utilize formal logic tools to derive some kind of insight about them. Most related to our approach presented in this chapter is the work by Kreowski, Kuske and Wille[105]. In this approach, the authors use logical encodings of graph transformation systems to find derivations, i.e. sequences of transformation rule applications, of bounded length. Unlike in our approach, they do not use predicate logic, restricting themselves to basic propositional logic, which, while allowing them to use SAT solvers instead of SMT solvers, which in general are yet more efficient and fast, also prevents them from expressing higher order concepts in their encodings, such as matches. In their application they are able to solve this by simply enumerating all possible matches, but for our use case where we will, for example, explicitly encode embedding functions, thiis solution is intractable.

Another technique that is often used in the second category is the use of the well-known formal modeling laguage Alloy[99] and its associated tools. Most related to our technique in this regard is the work by Baresi and Spoletini[19]. Another approach at bounded model checking, this work translates graphs and graph transformations into the language of Alloy, which, underneath, is essentially first-order logic, like in our approach. The main drawback of this approach is that the use of Alloy effectively rules out the possibility of encodings with unbounded models, as well as the absence of more advanced logical analysis tools, like interpolation (see Chapter 6). For the application of encoding graph transformation sequences of bounded length, these drawbacks are irrelevant. For our purposes though, a more powerful formalism is required. There are other approaches that roughly belong to the same category[109, 148], but all of them have similar restrictions.

The following section gives a brief overview and introduction to SMT, providing context for the sections thereafter.

4.2 Satisfiability Modulo Theories

Given a first-order logical formula φ, the *satisfiability problem* is the problem of finding a logical structure S, and an assignment $m : \mathrm{F}(\varphi) \to U_S$ of its free variables, such that $[\![\varphi]\!]_S^m = \mathbf{1}$. This is assuming that the predicate symbols used in the formula are entirely without inherent meaning and can be arbitrarily interpreted. However, that is not the case in almost any practical application where first-order formulas are being used.

Usually, first-order formulas are being used in tandem with, or on top of, another formalism. For example, such a formalism might be linear integer arithmetic (LIA). Terms in linear integer arithmetic represent statements about subsets of (k-powers of) the natural numbers, using symbols like $<$, $>$, $+$, or $\cdot$. With first order logic, one might now want to express the statement that all natural numbers that satisfy one LIA term, also satisfy another LIA term. This fusion between pure first-order logic, and pre-interpreted symbols belonging to a coherent formalism, is the basis of *satisfiability modulo theories (SMT).*

SMT Overview SMT is similar to the basic satisfiability problem, in that it is about finding logical structures and assignments (called *models*) in which a given formula is true. The difference is that in SMT, a certain subset of the predicate symbols used in the formula are pre-interpreted, assigned to functions and relations over an also predetermined universe. These interpreted predicates, and the universe on which they operate, are packaged together into a so-called *theory*. A first-order formula, together with a theory fixing its universe and providing a number of interpretations for (some of) its predicates, constitutes an SMT problem. Such problems are solved, i.e. models are constructed for them, by programs called *SMT solvers*.

When attempting to find a model, an SMT solver will alternate between two submodules. One module is a regular SAT-solver, treating literals containing predicates as regular boolean variables. The other is a so-called theory solver. This is a program encapsulating the interpretations of the functions and predicates that make up the theory, as well as proof rules about these functions and predicates. Essentially, whenever the SAT-solver derives that a certain set of theory literals (literals of predicate symbols that are part of the theory) should be true at the same time, it queries the theory solver to figure out whether this is possible, and, if not, why.

As an example, consider the following LIA-formula:

$$(x = 3) \rightarrow ((y > x) \wedge (y < 0))$$

Purely from a logical standpoint, there are two types of models in which this formula evaluates to true: Any model S in which $m(x) = 3$, $[\![(y > x)]\!]^m_S = \mathbf{1}$, and $[\![(y < 0)]\!]^m_S = \mathbf{1}$, as well as any model in which $m(x) \neq 3$. The theory solver would now check $(x = 3) \wedge (y > x) \wedge (y < 0)$ and find that there are no integers x and y that satisfy that term. Thus, any model containing this interpretation would be discarded, and the model ultimately produced would contain an assignment of x to some value other than 3, as well as an arbitrary assignment for y.

There are many theories, including LIA, but also LRA (linear real arithmetic), BV (bit vectors), ArraysEx (array with extensionality) and several more. SMT solvers have been successfully employed in many fields, most notably (for this thesis) in counterexample analysis in various abstraction-based verification schemes[115].

SMT in practice: SMTLib and Solvers Over the last 10 years, a de-facto standard for describing SMT theories and problems has emerged: SMTLib. This common standard has allowed for the emergence of a competitive environment[21] for the development of SMT solvers and the proliferation of common benchmarks derived from real-world problems.

The complexity of SMTLib is considerable. While all encodings presented in this thesis are written in SMTLib format, only a fraction of the actual language is used. I will now present this fraction of the language in order to facilitate understanding of the following sections. This introduction is based partly on a lecture by Roberto Bruttomesso[39].

The basic unit of SMTLib is the *Script*. A script describes an SMT problem and all its contextual definitions. It further describes exactly what problems should be solved, and what output about the solutions should be generated.

Each script begins with a command to set the theory (called a *logic*) used in the script.

```
(set-logic QF_UF)
```

A logic is essentially just a theory, plus a number of restrictions about how formulas can be built. For use with graphs, a very basic logic will suffice. We will use two logics in this chapter: UF and QF_UF. These represent, respectively, first-order logic with uninterpreted functions, and quantifier-free first-order logic with uninterpreted functions.

SMTLib allows the universe over which formulas are evaluated to by *sorted*, i.e. *typed*, essentially partitioning the universe into a number of sub-universes. Such typing is not a part of basic first-order logic, but can easily be emulated in it, using unary predicates and constraints on functions and predicates enforcing typing rules. Most logics contain pre-defined types. For example, the LIA logic defines the type `Int`, for integers. However, scripts are allowed to define their own types, called *Sorts*.

```
(declare-sort Node 0)
```

In general, sorts can have arities and be parameter-dependent. For example, the sort `BV` for bit vectors has the length of the bit vectors as a parameter. For this thesis, a simple, nullary sort for nodes will suffice[†].

This concludes the typical preface of the script, dealing with the context of the problem described. What follows is a description of the problem itself.

This consists of a series of symbol declarations. Simple constants of either predefined types (like boolean) or user-defined types (like `Node` above) are introduced as nullary, uninterpreted functions / predicates.

```
(declare-fun boolVar () Bool)
(declare-fun nodeVar () Node)
```

When executing the script, any attempt to *solve* the given problem will involve finding singular elements of the respective type to assign to these nullary functions.

Higher-level predicates and functions are declared in a similar way. Naturally, predicates should have the return type `Bool`, while functions can have any return type. The list of parameter types determines the arity of the function / predicate.

```
(declare-fun binaryEdge (Node Node) Bool)
(declare-fun match (Node) Node)
```

For our encoding, we will mostly restrict ourselves to unary and binary predicates.

SMTLib contains keywords for all the standard operators in first-order logic. They follow a very intuitive naming scheme (see Listing B.1). All of these operators (like all symbols in SMTLib) are written in prefix form.

The actual formulas are built with `define-fun`-statements.

```
(define-fun someFormula () Bool
  (and (binaryEdge nodeVar nodeVar) (not boolVar))
)
```

[†]for now, additional sorts will be introduced later

The above code introduces the new symbol someFormula and interprets it as a nullary predicate (no parameter types, return type Bool), with the interpretation defined by the given formula.

Once formulas have been written, the actual SMT problem has to be defined. This is done by way of the assert statement.

```
(assert (and someFormula someOtherFormula))
```

One can have multiple assert statements. Each of these statement adds its formula parameter to the set of formulas that must be fulfilled for the problem described in the script to be considered SAT. In essence, the formula represented by the script is a conjunction of the formulas that are asserted in it.

In case one uses many assert-statements, it can make sense for formulas (other than those defined by define-fun) to be named.

```
(assert (!
  (and (binaryEdge nodeVar nodeVar) (not boolVar))
  :named someFormula
))
```

Of course, named formulas can also be used outside of asserts, for example to construct large formulas from named subformulas, to increase readability.

Once the formulas have been specified and asserted, the actual check must be performed. This is done by a simple call to check-sat.

```
(check-sat)
```

Reading the check-sat command will prompt the program interpreting the script to attempt to find a model within the given logic that satisfies all asserted formulas. If a model is found, SAT is returned, if none exists, UNSAT is returned. Since the combination of asserted formulas might be undecidable, a third result is also possible: UNKNOWN is returned when neither SAT nor UNSAT could be determined after a specified amount of time.

As stated earlier, SMTLib is massively more complex than these simple commands indicate. However, they suffice for most applications in this thesis.

There are many programs that are capable of reading SMTLib input (either in Version 1.2 or the more recent 2.0), such as CVC4[20], MathSat[40], OpenSMT[41], SMTInterpol[46], Yices[65], Z3[114], and several more. These programs are collectively referred to as *SMT Solvers*. Due to the standardization of SMTLib, any encoding scheme producing SMTLib-compatible scripts can utilize any of these solvers. However, the solvers themselves are not necessarily feature equivalent, and certain features (like interpolation support) are rather rare. For these reasons, we will restrict ourselves to two solvers for this thesis: SMTInterpol (for interpolation-based checks) and Z3 (for all others).

4.3 Encoding of a Graph

As an introductory example, we will first look at how one can encode a single graph as an SMT problem. The first question one ought to ask is this: what does it even mean for an SMT problem to "encode" a graph? In this context, what we mean by this is the following:

Idea. *An SMT problem "encodes" a given mathematical object, iff there is a bijective mapping between the object and the set of all valid models for the SMT problem.*

This means that, for example, if we want to "encode" a single graph, we need to create an SMT problem whose only model is exactly (modulo differences in formalism/syntax) the logical structure that corresponds to the graph. With this in mind, we set about to encode a given graph G by creating an SMT script describing an SMT problem whose only model will give us the logical structure for that graph.

All scripts that that encode a single graph begin with the same prelude. First, the logic is set, then the `Node` sort is introduced. Since quantifiers tend to greatly decrease the performance of solvers, we will avoid using them wherever possible. Thus, we will always use the `QF_UF` logic, unless explicitly stated otherwise.

Code Template 1 (SMT Prelude for Single Graphs). *Each smt script encoding a single graph begins with the same prelude:*

```
    (set-logic QF_UF)
    (declare-sort Node 0)
```

Now we can begin encoding a graph. Let $G = \left(V, E^1, E^2\right)$ be a graph with V a set and E^1, E^2 sets of unary and binary edges, labeled over a separate set $L = (U_L, B_L)$ of unary and binary labels. The first objective is to make sure that the universe of the generated model will contain exactly the nodes that make up V. Thus, we will introduce a nullary `Node` constant for each element in V.

Code Template 2 (Universe Constants (Single Graph)). *The universe of G is encoded in node constants $v_1, \ldots, v_k$.*

```
    (declare-fun $v_1$ () Node)
    ⋮
    (declare-fun $v_k$ () Node)
```

Note that this alone does not ensure that the universe will be *exactly* the node set, only that there will be a `Node` constant for every node in the set. Nothing prevents the SMT solver from instantiating more nodes than are necessary to interpret these node constants. However, since all following formulas will explicitly reference only these `Node` constants and use no quantifiers or variables that range over the `Node` sort as a whole, there is no reason for the solver to instantiate more than k `Node` objects in the universe.

On the other hand, there is also no explicit provision that states that these nullary constants cannot have the same value, leading to a singleton universe. In order to prevent this, we need to introduce a formula to make sure that at least k `Node` constants will be instantiated. We do this by explicitly stating the inequality of every possible node pair.

Code Template 3 (Pairwise Inequality (Single Graph)). *The formula* `distinctNodes` *is given by the following code template.*

```
(define-fun distinctNodes () Bool
  (and (not (= v1 v2)) (not (= v1 v3)) ··· (not (= v1 vk))
                       (not (= v2 v3)) ··· (not (= v2 vk))
                                                 ⋮
                                           (not (= vk-1 vk)))
)
```

It represents the first-order formula

$$\bigwedge_{i=1}^{k} \bigwedge_{j=i+1}^{k} \neg (v_i = v_j)$$

We believe the property of this encoding that all its models represent k distinct instances of the node sort to be obvious enough as to not require a proof.

Having established the universe, i.e. the node set, we now move on to the edges. This is based very much on the same idea as the encoding as a logical structure (see Sec. 3.2). Each binary edge is represented by a binary predicate, while each unary edge is represented by a unary predicate.

Thus, for each label $l_1 = (s_1, k_1), \ldots, l_n = (s_n, k_n) \in L$, we add an uninterpreted predicate of the appropriate arity. For the sake of presentation, we assume we have $|U_L| = l$ and $|L| = n$.

Code Template 4 (Label / Predicate Set (Single Graph)). *Each unary / binary label in the graph G is represented by a unary / binary predicate in the SMT encoding.*

```
  ; first the unary predicates
```

```
(declare-fun $s_1$ (Node) Bool)
⋮
(declare-fun $s_l$ (Node) Bool)
; then the binary predicates
(declare-fun $s_{l+1}$ (Node Node) Bool)
⋮
(declare-fun $s_n$ (Node Node) Bool)
```

Once the predicate set is established, the actual edges can be encoded. The unary edges on each node will be encoded in a separate (interpreted) function. In this function, we assert for each node the existence or non-existence of each type of unary edge explicitly.

Code Template 5 (Unary Edges (Single Graph)). *For nodes $v_1, \ldots, v_k$ and unary labels $s_1, \ldots, s_l$ the function* `unaryG` *is given by:*

```
(define-fun unaryG () Bool (and
  $B(s_1, v_1)$ ⋯ $B(s_l, v_1)$
  ⋮
  $B(s_1, v_k)$ ⋯ $B(s_l, v_k)$
))
```

where each block $B(s, v)$ stands for either `(s v)` *or* `(not (s v))`*, depending on whether the edge (s, v) exists in the graph. Thus, it effectively represents the logical formula*

$$\bigwedge_{v \in V} \left[\bigwedge_{((s,1),v) \in E^1} s(v) \wedge \bigwedge_{((s,1),v) \notin E^1} \neg s(v) \right]$$

The binary edges on each node pair will be encoded similarly, in another separate (interpreted) function. The procedure is essentially the same as for unary edges. For each binary label, we assert the existence or non-existence of that type of binary edge on each possible node pair explicitly.

Code Template 6 (Binary Edges (Single Graph)). *For binary labels $s_{l+1}, \ldots, s_n$ and node pairs*

$$(v_1, v_1), \ldots, (v_1, v_k), (v_2, v_1), \ldots, (v_k, v_{k-1}), (v_k, v_k)$$

we obtain the following function:

```
(define-fun binaryG () Bool (and
```

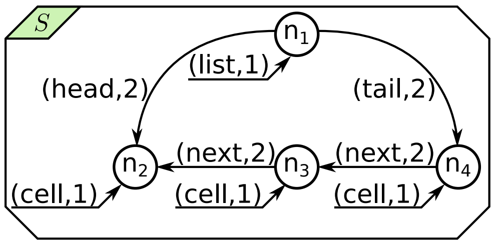


Figure 4.2: An example graph (see also Fig. 3.1)

```
    ; we begin with the first binary label $s_{l+1}$
    ; each B-block encodes the (non-)existence of an edge
    ; labeled $s_{l+1}$ on the given node pair
    B($s_{l+1}$,$v_1$,$v_1$) ⋯ B($s_{l+1}$,$v_1$,$v_k$) B($s_{l+1}$,$v_2$,$v_1$) ⋯ B($s_{l+1}$,$v_k$,$v_k$)
    ⋮
    B($s_n$,$v_1$,$v_1$) ⋯ B($s_n$,$v_1$,$v_k$) B($s_n$,$v_2$,$v_1$) ⋯ B($s_n$,$v_k$,$v_k$)
    ))
```

where each `B(x, y, z)` *block is either* `(x y z)` *or* `(not (x y z))`*, depending on whether the corresponding edge exists in the graph or not. Thus, the function* `binaryG` *effectively represents the logical formula*

$$\bigwedge_{v_1 \in V} \left(\bigwedge_{v_2 \in V} \left[\bigwedge_{(v_1,(s,2),v_2) \in E^2} s(v_1, v_2) \wedge \bigwedge_{(v_1,(s,2),v_2) \notin E^2} \neg s(v_1, v_2) \right] \right)$$

These three functions together, in the SMT context established so far, are already sufficient to encode the graph. All that remains is to assert them and check for satisfiability.

Code Template 7 (Graph Assertion). *Given an SMT code containing instances of the code templates 2, 3, 4, 5, and 6, the entire graph can then be asserted using*

```
    (assert (and distinctNodes unaryG binaryG))
```

As an example, consider the graph shown in Fig. 4.2. It represents a linear list with three cells. A graph such as this can easily be encoded using the formulas defined above. The encoding will be such that the resulting SMT problem is SAT, with exactly one satisfying model – the logical structure of the graph itself.

Lemma 9 (Satisfiability of Single Graph Encoding). *Let $G = \left(N, E^1, E^2\right)$ be a graph over a label set L, and let Sc be an SMT script built from G according to the code templates 1, 2, 3, 4, 5, 6, and 7. Then there exists exactly one satisfying model for Sc (modulo unused universe elements). This model represents exactly the logical structure $ls\,(G)$.*

Proof. See Appendix A, page 243 □

The encoding for this particular graph is shown in Listing 4.1. It has been formatted for maximum readability. It is easy to see that, especially for binary edges, the size of the formula can grow very quickly. Even for such a simple graph, the listing barely fits on one page. However, the power of modern SMT solvers is such that models of this size pose no serious problem, and even much larger problems (i.e. larger, more complex graphs) are usually solved in a matter of seconds or minutes at most[21].

This simple encoding is intended to show the basic idea of how graphs can be expressed in first-order logic. The actual encodings that we will use are based not only on the encoding of singular graphs, like here, but on the encoding of embeddings between graphs and shapes, and the encoding of rule applications as well. In the following, we will go through these encodings and prove that they do in fact encode what they are supposed to encode. The encodings themselves will tend to grow very large. For this reason, any further listings for example encodings will be grouped in Appendix B.

4.4 Encoding of a Graph Embedded Into a Shape

As mentioned in Section 4.1, the first application of SMT encodings to the verification of shape transformation systems is the validation of abstract traces. Figure 4.1 shows that the notion of an embedding between a graph and a (possibly) constrained shape is central to this process.

Thus, as a first step, we will now develop an encoding that will enable us to tell whether a given graph G can be embedded into a shape (S, Λ). While this problem, at least in the unconstrained case, is more easily solved in the domain of graphs, its SMT encoding will provide the basis for much more complex encodings later on. We will build upon the simple graph encoding above, and successively add an encoding of the embedding function, and of the various aspects of the shape.

We will begin by considering the embedding function itself. As stated in Def. 43, an embedding function $f : U_G \rightarrow U_S$ is a surjective function between the universes of two shapes. It must obey the following conditions:

Edge Abstraction For any node tuple t in U_G and any predicate p the value of p on $f(t)$ in S must be equal to or more abstract than the value of p on t in G.

Listing 4.1: SMT Encoding of the graph shown in Fig. 4.2

```
(set-option :print-success false)
(set-logic QF_UF)
; Sort for graph nodes
(declare-sort Node 0)
; four node individuals (node set / universe)
(declare-fun v_1 () Node)
(declare-fun v_2 () Node)
(declare-fun v_3 () Node)
(declare-fun v_4 () Node)
; assert distinctiveness of the individuals
(define-fun distinctNodes () Bool
  (and (not (= v_1 v_2)) (not (= v_1 v_3)) (not (= v_1 v_4)))
  (and                   (not (= v_2 v_3)) (not (= v_2 v_4)))
  (and                                     (not (= v_3 v_4)))
)
; labels used in the graph become predicates
(declare-fun list (Node) Bool)
(declare-fun cell (Node) Bool)
(declare-fun tail (Node Node) Bool)
(declare-fun head (Node Node) Bool)
(declare-fun next (Node Node) Bool)
; defines the unary edges of the graph
(define-fun unaryG () Bool (and
       (list v_1)  (not (cell v_1))
  (not (list v_2))      (cell v_2)
  (not (list v_3))      (cell v_3)
  (not (list v_4))      (cell v_4)
))
; defines the binary edges of the graph
(define-fun binaryG () Bool (and
  (not (tail v_1 v_1))      (tail v_1 v_2)  (not (tail v_1 v_3)) (not (tail v_1 v_4))
  (not (tail v_2 v_1)) (not (tail v_2 v_2)) (not (tail v_2 v_3)) (not (tail v_2 v_4))
  (not (tail v_3 v_1)) (not (tail v_3 v_2)) (not (tail v_3 v_3)) (not (tail v_3 v_4))
  (not (tail v_4 v_1)) (not (tail v_4 v_2)) (not (tail v_4 v_3)) (not (tail v_4 v_4))

  (not (head v_1 v_1)) (not (head v_1 v_2)) (not (head v_1 v_3))      (head v_1 v_4)
  (not (head v_2 v_1)) (not (head v_2 v_2)) (not (head v_2 v_3)) (not (head v_2 v_4))
  (not (head v_3 v_1)) (not (head v_3 v_2)) (not (head v_3 v_3)) (not (head v_3 v_4))
  (not (head v_4 v_1)) (not (head v_4 v_2)) (not (head v_4 v_3)) (not (head v_4 v_4))

  (not (next v_1 v_1)) (not (next v_1 v_2)) (not (next v_1 v_3)) (not (next v_1 v_4))
  (not (next v_2 v_1)) (not (next v_2 v_2)) (not (next v_2 v_3)) (not (next v_2 v_4))
  (not (next v_3 v_1))      (next v_3 v_2)  (not (next v_3 v_3)) (not (next v_3 v_4))
  (not (next v_4 v_1)) (not (next v_4 v_2))      (next v_4 v_3)
(not (next v_4 v_4))
))
; assert a conjunction of the formulae defined and
(assert (and distinctNodes unaryG binaryG))
; check for satisfiability
(check-sat)
(get-model)
```

Node Abstraction If f maps more than one node in U_G to one node $u \in U_S$, then u must be a summary node in S.

We want the embedding to be a mapping between nodes in the graph and nodes in the shape. Making sure that this is the case in SMT, where f is a regular binary predicate on the node sort, is rather laborious. It is also unnecessary, since SMT already provides us with the tools to automatically enforce what is, essentially, a typing rule.

Thus, we separate the universe of the graph and the universe of the shape by using an additional sort, called `ANode` (Abstract Node). The embedding can then be represented by a function mapping the `Node` sort into the `ANode` sort.

This necessitates a change in the SMT prelude defined above, adding the new sort, as well as the embedding function.

Code Template 8 (SMT Prelude for Graph Embeddings). *Each SMT Script encoding a graph embedded into a shape begins with the prelude specified in Code Template 1, followed by*

```
    (declare-sort ANode 0)
    (declare-fun _F (Node) ANode)
```

Having defined the new sort `ANode`, we can now declare the nodes in the graph and the shape (i.e. the source and target of the embedding function) as `Node` and `ANode` constants, analogous to the graph encoding above.

Code Template 9 (Universe Constants (Graph Embedding)). *Let G be a graph and S be a shape. Let $v_1, \ldots, v_k$ be the nodes of the graph, and let $u_1, \ldots, u_m$ be the nodes of the shape. Then, the encoding of the universe of both the graph and the shape take the following form:*

```
  (declare-fun v_1 () Node)
  ⋮
  (declare-fun v_k () Node)

  (declare-fun u_1 () ANode)
  ⋮
  (declare-fun u_m () ANode)
```

Since each node in a shape is distinct from every other node in the shape, the `distinctNodes` formula can just be expanded to also include the node constants for the shape. Note that inequality terms between `Node` and `ANode` constants are unnecessary, as this is already taken care of by the typing system.

Code Template 10 (Pairwise Inequality (Graph Embedding)). *The pairwise inequality of concrete graph nodes is given by the first-order formula*

$$\bigwedge_{i=1}^{k}\bigwedge_{j=i+1}^{k}\neg\left(v_i = v_j\right) \wedge \bigwedge_{i=1}^{m}\bigwedge_{j=i+1}^{m}\neg\left(u_i = u_j\right)$$

and encoded into SMTLib in a straightforward fashion as the function `distinctNodes`*, using an appropriate function definition.*

As with Code Template 3, a proof that any model of this formula will induce k different `Node` constants and m different `ANode` constants is deemed unnecessary.

In the next step, we need to limit the range of `_F` to make sure that its interpretation will not necessitate creating individuals in the `ANode` sort that go beyond the constants we have defined. Thus, we need to make sure that for each `Node` x, the `ANode` y it is mapped to will be one of the constants we defined.

Code Template 11 (Embedding Range (Graph Embedding)). *The range of the embedding function is encoded by the first-order formula*

$$\bigwedge_{v\in U_G}\left(\bigvee_{u\in U_S} u = \_F\left(v\right)\right)$$

and encoded into SMTLib in a straightforward manner as the function `function`*, using an appropriate function definition. For each node $v \in U_G$, this SMT function thus encodes that* `_F` *must map it to one of the nodes $u \in U_S$.*

As above, proving that an assertion of this formula the range of the function `_F` to be restricted to the `ANode` constants $u_1, \ldots, u_m$ is trivial.

Now that the function property is assured, we move on to *surjectivity* and *node abstraction*. Both of these will be combined in a single formula. Surjectivity states, that at least one node from U_G must map to each node in U_S. On the other hand, node abstraction states that, if and only if more than one node from U_G maps to a node in S, that node must be a summary node.

This can be combined to the statement that for each non-summary node in U_S, there must be *exactly* one node from U_G, that maps to it, and for each summary node there must be *at least* one such node. We encode this property in a formula called `summarization`.

Code Template 12 (Summarization (Graph Embedding)). *Let $u_1, \ldots, u_{m'}$ denote the non-summary nodes of S, and $u_{m'+1}, \ldots, u_m$ the summary nodes. The summariza-*

tion property of the embedding function is encoded by the first-order formula

$$\bigwedge_{u \in U_S^c} \left[\bigvee_{v \in U_G} \left(u = \_F(v) \wedge \bigwedge_{v' \in U_G \setminus \{v\}} \neg \left(u = \_F(v') \right) \right) \right] \wedge \bigwedge_{u \in U_S^s} \left[\bigvee_{v \in U_G} u = \_F(v) \right]$$

where $U_S^c := \{u \in U_S \mid \iota_S(sm)(u) = \mathbf{0}\}$ *and* $U_S^s := \{u \in U_S \mid \iota_S(sm)(u) = ½\}$, *and encoded into SMTLib in a straightforward manner as the function* `summarization`*, using an appropriate function definition.*

This guarantees both surjectivity and the node abstraction property for the interpretation of `_F`, as the following lemma shows.

Lemma 10 (Correctness of Summarization encoding). *Any function* $\_F : U_G \rightarrow U_S$ *that satisfies the formula defined in Code Template 12 is a surjective function that satisfies the summarization property, given that* $v_1, \ldots, v_k, u_1, \ldots, u_m$ *are pairwise distinct.*

Proof. See Appendix A, page 243. □

What remains is the edge abstraction property. In order to ensure that the encoded embedding function respects this property, we must first introduce an encoding of the edges in the target shape. We can provide such an encoding by just slightly modifying the graph encoding shown in Sec. 4.3. Since every graph is a shape, we already have an encoding for a special subset of shapes (the graphs). Now, the idea is to remove certain restrictions from the graph encoding to make it suitable for shapes.

The main differences between a graph and a shape are the existence of summary nodes and ½-edges. The proper use of summary nodes in our encoding has already been taken care of in the `summarization` formula. Thus, what remains are only the ½-edges.

When a shape contains a ½-edge, the intended meaning is that any graph embedded into it can choose whether to contain the corresponding edge or not, i.e. the interpretation of the predicate for that particular node tuple is unconstrained. In the SMT encoding we can take this literally and simply remove the restrictions on a given predicate and node pair if the shape contains a ½-edge for that predicate and node pair, instead of a regular edge or no edge.

So, for example, for a node pair u_1, u_2 in the shape, and a binary predicate p, we would encode

`(not (p` u_1 u_2`))` iff $\iota_S(p)(u_1, u_2) = \mathbf{0}$ and
`(p` u_1 u_2`))` iff $\iota_S(p)(u_1, u_2) = \mathbf{1}$ and
(true) iff $\iota_S(p)(u_1, u_2) = ½$

This idea allows us to modify the formulas `unaryG` and `binaryG` (see Code Templates 5 and 6) to obtain `unaryS` and `binaryS`, their equivalents for shapes. Of course, since the nodes of the shape are represented in the encoding by constants of the `ANode` sort, rather than the `Node` sort, we can not use the same predicate set that we used for the encoding of the graph edges. We must thus duplicate the predicate set for the shape, using the prefix `a` (again, for "abstract") to distinguish predicates on the shape from predicates in the graph.

Code Template 13 (Label / Predicate Set (Graph Embedding)). *Given a graph G and a shape S over the same predicate set $\mathcal{P}$, that predicate set is encoded as a set of uninterpreted predicates. Let $s_1, \ldots, s_l$ be the unary predicates, and let $s_{l+1}, \ldots, s_n$ be the binary predicates. The encoding is then given by the following code template:*

```
; first the unary predicates
(declare-fun s₁ (Node) Bool)
(declare-fun as₁ (ANode) Bool)
⋮
(declare-fun sₗ (Node) Bool)
(declare-fun asₗ (ANode) Bool)
; then the binary predicates
(declare-fun sₗ₊₁ (Node Node) Bool)
(declare-fun asₗ₊₁ (ANode ANode) Bool)
⋮
(declare-fun sₙ (Node Node) Bool)
(declare-fun asₙ (ANode ANode) Bool)
```

Once the predicates for the shape have been established, the encoding of the edges can follow.

Code Template 14 (Unary Edges (Shape)). *For nodes $u_1, \ldots, u_m$ and unary labels $s_1, \ldots, s_l$ the function* `unaryS` *is given by:*

```
(define-fun unaryS () Bool (and
  B(s₁, v₁) ⋯ B(sₗ, v₁)
  ⋮
  B(s₁, vₖ) ⋯ B(sₗ, vₖ)
))
```

where each block $B(s, v)$ stands for either `(as v)`, `(not (as v))`, *or* `(true)` *depending on whether $\iota_S(s)(v)$ is* **1**, **0**, *or ½ in the shape. Thus, it effectively represents* *the logical formula*

$$\bigwedge_{v \in V} \left[\bigwedge_{\iota_S(s,1)(v)=\mathbf{1}} s(v) \wedge \bigwedge_{\iota_S(s,1)(v)=\mathbf{0}} \neg s(v) \right]$$

Code Template 15 (Binary Edges (Shape)). *For binary labels $s_{l+1}, \ldots, s_n$ and node pairs*

$$(v_1, v_1), \ldots, (v_1, v_k), (v_2, v_1), \ldots, (v_k, v_{k-1}), (v_k, v_k)$$

we obtain the following function:

```
(define-fun binaryS () Bool (and
; we begin with the first binary label $s_{l+1}$
; each B-block encodes a (possibly) (non-)existing edge
; labeled $s_{l+1}$ on the given node pair
B($s_{l+1}$,$v_1$,$v_1$) ··· B($s_{l+1}$,$v_1$,$v_k$) B($s_{l+1}$,$v_2$,$v_1$) ··· B($s_{l+1}$,$v_k$,$v_k$)
⋮
B($s_n$,$v_1$,$v_1$) ··· B($s_n$,$v_1$,$v_k$) B($s_n$,$v_2$,$v_1$) ··· B($s_n$,$v_k$,$v_k$)
))
```

where each `B(x, y, z)` *block is either* `(ax y z)`*,* `(not (ax y z))`*, or* `(true)`*, depending on whether the corresponding edge exists in the graph or not. Thus, the function* `binaryS` *effectively represents the logical formula*

$$\bigwedge_{v_1 \in V} \left(\bigwedge_{v_2 \in V} \left[\bigwedge_{\iota_S(s,2)(v_1,v_2)=\mathbf{1}} s(v_1, v_2) \wedge \bigwedge_{\iota_S(s,2)(v_1,v_2)=\mathbf{0}} \neg s(v_1, v_2) \right] \right)$$

Correctness proofs for these definitions are analogous to the proofs for `unaryG` and `binaryG`, from the proof of Lemma 9. Now that the (definite) edges of the shape have been encoded, all that is left is to assert that, wherever definite edges exist in the shape, they have to be mirrored by edges in the graph. Thus, for every node or node pair in the graph, and every label, the corresponding edge must have the same value as its image in the shape under `_F`.

As an example, let $v_1, v_2 \in V$ be nodes in the graph, and let p be a binary predicate. Let further $u_1, u_2 \in U_S$ be nodes in the shape and let (u_1, p, u_2) be the only ½-edge labeled p in the shape. Then, using an implication to introduce an exception for ½-edges, the property described above would take the following form as a formula:

$$\begin{aligned} &(\neg((\_F(v_1) = u_1) \wedge (\_F(v_2) = u_2))) \rightarrow (p(v_1, v_2) = p(\_F(v_1), \_F(v_2))) \\ \equiv\ &(p(v_1, v_2) = p(\_F(v_1), \_F(v_2))) \vee ((\_F(v_1) = u_1) \wedge (\_F(v_2) = u_2)) \end{aligned}$$

This basic idea enables us to complete the encoding of the edge abstraction property with two new formulas: `unaryAbstraction` and `binaryAbstraction`.

Code Template 16 (Unary Abstraction (Graph Embedding)). *Let $s_1, \ldots, s_l$ be the unary predicates in $\mathcal{P}$. Furthermore, for each s_i, let $u_1^{s_i}, \ldots, u_{k_i}^{s_i}$ be the set of nodes in the shape S for which $\iota(s_i)\left(u_j^{s_i}\right) = ½ \quad \forall 1 \leq j \leq k_i$ holds. Then, the encoding for unary equivalence between the graph and the shape is given by the following formula:*

```
(define-fun unaryAbstraction () Bool (and
  (or (= (s₁ v₁) (as₁ (_F v₁))) (= (_F v₁) u₁^s₁) ··· (= (_F v₁) u_k₁^s₁))
  ⋮
  (or (= (s₁ v_k) (as₁ (_F v_k))) (= (_F v_k) u₁^s₁) ··· (= (_F v_k) u_k₁^s₁))
      ⋮
  (or (= (s_l v₁) (as_l (_F v₁))) (= (_F v₁) u₁^s_l) ··· (= (_F v₁) u_k_l^s_l))
  ⋮
  (or (= (s_l v_k) (as_l (_F v_k))) (= (_F v_k) u₁^s_l) ··· (= (_F v_k) u_k_l^s_l))
)
```

In first-order logic, this formula can be expressed as:

$$\bigwedge_{i=1}^{l} \bigwedge_{j=1}^{k} \left[(s_i(v_j) = s_i(\_F(v_j))) \vee \bigvee_{o=1}^{k_i} (\_F(v_j) = u_o^{s_i}) \right]$$

Binary abstraction follows analogously.

Code Template 17 (Binary Abstraction (Graph Embedding)). *Let $s_{l+1}, \ldots, s_n$ be the binary predicates in $\mathcal{P}$. Furthermore, for each s_i, let $e_1^{s_i}, \ldots, e_{k_i}^{s_i}$ be the set of binary ½-edges in the shape S. Let each edge $e_j^{s_i}$ be denoted (u_j, s_i, w_j). Then, the encoding for binary equivalence between the graph and the shape is given by the following formula:*

```
  (define-fun binaryAbstraction () Bool (and
    (or
      (= (s_l+1 v₁ v₁) (as_l+1 (_F v₁) (_F v₁)))
      (and (= (_F v₁) u₁) (= (_F v₁) w₁))
      ⋮
      (and (= (_F v₁) u_k_l+1) (= (_F v₁) w_k_l+1))
    )
    (or
      (= ($s_{l+1}$ $v_1$ $v_2$) (a$s_{l+1}$ (_F $v_1$) (_F $v_2$)))
      (and (= (_F $v_1$) $u_1$) (= (_F $v_2$) $w_1$))
      ⋮
      (and (= (_F $v_1$) $u_{k_{l+1}}$) (= (_F $v_2$) $w_{k_{l+1}}$))
    )
    ⋮
    (or
      (= ($s_{l+1}$ $v_k$ $v_k$) (a$s_{l+1}$ (_F $v_k$) (_F $v_k$)))
      (and (= (_F $v_k$) $u_1$) (= (_F $v_k$) $w_1$))
      ⋮
      (and (= (_F $v_1$) $u_{k_{l+1}}$) (= (_F $v_k$) $w_{k_{l+1}}$))
    )
          ⋮
    (or
      (= ($s_n$ $v_1$ $v_1$) (a$s_n$ (_F $v_1$) (_F $v_1$)))
      (and (= (_F $v_1$) $u_1$) (= (_F $v_1$) $w_1$))
      ⋮
      (and (= (_F $v_1$) $u_{k_n}$) (= (_F $v_1$) $w_{k_n}$))
    )
    (or
      (= ($s_n$ $v_1$ $v_2$) (a$s_n$ (_F $v_1$) (_F $v_2$)))
      (and (= (_F $v_1$) $u_1$) (= (_F $v_2$) $w_1$))
      ⋮
      (and (= (_F $v_1$) $u_{k_n}$) (= (_F $v_2$) $w_{k_n}$))
    )
    ⋮
    (or
      (= ($s_n$ $v_k$ $v_k$) (a$s_n$ (_F $v_k$) (_F $v_k$)))
      (and (= (_F $v_k$) $u_1$) (= (_F $v_k$) $w_1$))
      ⋮
      (and (= (_F $v_1$) $u_{k_n}$) (= (_F $v_k$) $w_{k_n}$))
    )
  )
```

In first-order logic, this formula can be expressed as:

$$\bigwedge_{i=l+1}^{n} \bigwedge_{j_1=1}^{k} \bigwedge_{j_2=1}^{k} \left[(s_i (v_{j_1}, v_{j_2}) = s_i (\_F (v_{j_1}), \_F (v_{j_2}))) \vee \right. \\ \left. \bigvee_{o=1}^{k_i} ((\_F (v_{j_1}) = u_o) \wedge (\_F (v_{j_2}) = w_o)) \right]$$

While this code template might look significantly larger and more complicated, this is entirely due to the need to go through all combinations of nodes to form possible edges, rather than merely through the nodes themselves, as in the code template for unary abstraction. The basic principle is exactly the same.

Note that it would be possible to make this encoding much more compact by not separating the encoding of the shape and the encoding of the edge abstraction property. Both of these could be merged by simply defining a formula that encodes the presence of all definite edges implied by the shape directly on the graph. While this would be a more compact and efficient encoding, it would remove the explicit properties of the shape from the script. However, as we will see in Chap. 6, an explicit encoding of these properties is vital to the application that the formulas described here will be used for. Thus, we will stick to this more verbose encoding.

In conclusion, we have now inserted into the encoding everything we need to express the problem of finding out whether a given graph G can be embedded into an unconstrained shape S:

- We have introduced the shape itself to the encoding, with its own universe sort (`ANode`) and constants, as well as (somewhat less restricted) edge formulas.
- We have further introduced an uninterpreted function `_F` on the `Node` sort, mapping into the `ANode` sort. This function is intended to represent the potentially existing embedding between G and S.
- We have introduced formulas imposing restrictions on `_F`, enforcing its totality, correct range, surjectivity, as well as the node abstraction property.

- Lastly, we have introduced formulas enforcing the edge abstraction property on `_F`.

Thus, all that remains is to assert all these formulas together and check for satisfiability. Every satisfying model will contain an interpretation for `_F` that represents a valid embedding of G into S.

Code Template 18 (Embedding Assertion (Unconstrained)). *Let G be a graph, and let S be an unconstrained shape. Given an SMT script containing instances of the code templates 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17, the embeddability of G into S can be asserted using*

```
(assert (and
  distinctNodes unaryG binaryG
  function summarization
  unaryS binaryS
  unaryAbstraction binaryAbstraction
))
```

The correctness of this encoding is expressed in the following lemma.

Lemma 11 (Correctness of Embedding Encoding). *Let G be a graph and S be a shape. Let Sc be an SMT script containing an instance of the assertion defined in Code Template 18 for G and S. Let F be the set of possible embeddings between G and S.*

Then there is a one-to-one relationship between models of Sc and embeddings in F, i.e. each model of Sc is associated with a valid embedding of G into S, and vice-versa. Specifically, this means that Sc is UNSAT if and only if $F = \emptyset$.

Proof. See Appendix A, page 244. □

As an example, consider the shape shown in Fig. 4.3. It is clear that the graph from the previous section (see Fig. 4.2) is embedded into this shape. Listing B.2 shows the result of applying the embedding encoding described in this section to that graph and shape pair. When fed into an SMT solver, the model produced creates the following embedding: $f := \{(v_1, u_1), (v_2, u_2), (v_3, u_2), (v_4, u_2)\}$, the only valid embedding between G and S.

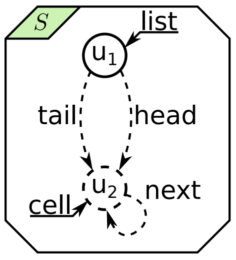


Figure 4.3: The example shape used in Listing B.2

Note that, while a model for an SMT problem as described in Code Template 18 will contain the logical structure for G, and a function `_F` representing a valid embedding, it will in general not be able to sensibly provide an interpretation for S.

This is merely due to the fact that the solver is not able to work with three-valued logics and must, for each ½-edge in the shape, decide whether to include the edge or not.

Thus far we have only dealt with unconstrained shapes in this encoding, and have obtained a way to check for embeddings that is far more laborious and involved than just using a graph algorithm to search for valid embeddings. Now we will extend our encoding to constrained shapes, which is where the true advantage of using logical encodings will become clear. As a reminder, a graph G is embedded into a constrained shape (S, Λ) (see Def. 61) if and only if:

- G is classically embedded into S (see Def. 43) via an embedding f and
- G satisfies every possible interpretation of Λ as defined by f.

In essence, the constraints further restrict our options for possible embeddings. Therefore, we can include constraints in our encoding simply by using the encoding we have created thus far, and add a formula that represents adherence to the constraints implied by `_F` and Λ.

The basic idea of how we will achieve this is to derive, from each constraint, a formula that is attached to particular nodes in the shape. Then, we will write out a formula that states that, if a given graph node tuple is mapped to the shape node tuple the constraint is attached to, then that graph node tuple has to satisfy the constraint. Of course, at this point we have to restrict the constraints we can support to those represented by quantifier-free formulas, since we intend to encode the constraint formula as-is into an SMT script using the `QF_UF` logic. In general, however, this restriction is not necessary, as we will see in Sec. 4.7, where we will construct a constraint encoding that allows quantifiers in order to check constrained shapes for feasibility.

As specified in Def. 55, every constraint consists of a formula and an assignment of the free variables of that formula into the shape the constraint is attached to. Multiple constraints may use the same formula, but different assignments. In order to increase clarity and remove redundancies, we will begin by encoding each constraint *formula* as its own function in our script.

Code Template 19 (Constraint Formulas (Graph Embedding)). *Let α be a (constraint) formula with free variables* $\mathrm{F}(\alpha) = \{x_1, \ldots, x_k\}$. *Then the encoding of α is given by the following code template:*

```
    (define-fun constraint_α ((x_1 Node) ··· (x_k Node)) |α|)
```

where $|\alpha|$ in this code represents a direct SMT encoding of the formula itself, using the predicate set for the graph, rather than that for the shape.

For each unique constraint formula in Λ, our encoding will contain a function as defined above. These functions will then be used in the formula encoding the adherence of the graph to each constraint implied by `_F`.

Code Template 20 (Constraint (Graph Embedding)). *Let (α, m) be a constraint, let $k := |\mathrm{F}(\alpha)|$, and let $\mathrm{F}(\alpha) = \{x_1, \dots, x_k\}$. Let further $\dot{V}^k$ be the set of all node tuples $(v_1, \dots, v_k)$ such that $\forall i, j \in \{1, \dots, k\} : i \neq j \to v_i \neq v_j$. Let $n := \left|\dot{V}^k\right|$ and let $\left(v_1^i, \dots, v_k^i\right)$ denote the i-th tuple in $\dot{V}^k$. Then the encoding of the adherence of a graph G to this particular constraint is given by the following code template, providing the prior existence of an instance of Code Template 19 for α:*

```
(define-fun $constraint_{\alpha,m}$ () Bool (and
(=>
  (and (= (_F $v_1^1$) $m(x_1)$) $\cdots$ (= (_F $v_k^1$) $m(x_k)$))
  ($constraint_\alpha$ $v_1^1$ $\cdots$ $v_k^1$)
)
⋮
(=>
  (and (= (_F $v_1^n$) $m(x_1)$) $\cdots$ (= (_F $v_k^n$) $m(x_k)$))
  ($constraint_\alpha$ $v_1^n$ $\cdots$ $v_k^n$)
)
))
```

In effect, this code template represents the first-order function

$$\bigwedge_{i=1}^{n} \left[\bigwedge_{j=1}^{k} \left(\left(\_F\left(v_j^i\right) = m\left(x_j\right)\right) \to \alpha_{\left[x_1 \mapsto v_1^i, \dots, x_k \mapsto v_k^i\right]} \right) \right]$$

Having encoded each individual constraint in this way, encoding adherence to all of them is then a simple matter of creating the appropriate conjunction.

Code Template 21 (Constraints (Graph Embedding)). *Let (S, Λ) be a constrained shape. Let $\alpha_1, \dots, \alpha_\lambda$ be the unique constraint* formulas *in Λ. For each α_i, let $m_1^i, \dots, m_{\mu_i}^i$ be the assignments for that formula that occur in Λ. Then, the encoding for the adherence of the graph G to the constraint set implied by `_F` is given by the following code template:*

```
(define-fun constraints () Bool (and

  $constraint_{\alpha_1,m_1^1}$ $\cdots$ $constraint_{\alpha_1,m_\lambda^{\mu_1}}$
```

```
          ⋮
  $constraint_{\alpha_\lambda, m_1^\lambda}$ ⋯ $constraint_{\alpha_\lambda, m_\lambda^{\mu_\lambda}}$
))
```

This leads to a final extension of the overall assertion at the end of the encoding.

Code Template 22 (Embedding Assertion (Constrained)). *Let G be a graph, and let (S, Λ) be a constrained shape. Given an SMT script containing instances of the code templates 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, and 17, as well as an instance of Code Template 21 and all its required precursor instances (instances of Code Template 19 and CoT. 20), the embeddability of G into (S, Λ) can be asserted using*

```
(assert (and
  distinctNodes unaryG binaryG
  function summarization
  unaryS binaryS
  unaryAbstraction binaryAbstraction
  constraints
))
```

As an example, let's assume that we added the constraint

$$(\alpha, m) := (\neg next\,(x, x)\,, [x \mapsto u_2])$$

to the shape S shown in Fig. 4.3. For this constraint, we have $k = 1$ (cf. CoT. 19). Thus, its constraint formula is given by

```
(define-fun $constraint_\alpha$ (($x_1$ Node)) (not (next $x_1$ $x_1$)))
```

In order to derive the appropriate encoding for the adherence of the graph G (see Fig. 4.2) to this constraint, we first need to determine the set $\dot{V}^1$, i.e. the set of 1-tuples such that in every tuple each node occurs at most once. Fortunately, this is very easy: $\dot{V}^1 = \{(v_1)\,, (v_2)\,, (v_3)\,, (v_4)\}$. The encoding for the adherence of G to the constraint (α, m) is therefore given by

```
(define-fun $constraint_{\alpha,m}$ () Bool (and
  (=> (and (= (_F $v_1$) $u_2$) true) ($constraint_\alpha$ $v_1$))
  (=> (and (= (_F $v_2$) $u_2$) true) ($constraint_\alpha$ $v_2$))
  (=> (and (= (_F $v_3$) $u_2$) true) ($constraint_\alpha$ $v_3$))
  (=> (and (= (_F $v_4$) $u_2$) true) ($constraint_\alpha$ $v_4$))
))
```

The whole constraint adherence formula, then, since this is the only constraint in our example, is simply

```
(define-fun constraints () Bool (and
  $constraint_{\alpha,m}$ true
))
```

Adding these formulas to Listing B.2 yields Listing B.3, which still returns SAT, since the graph G does in fact satisfy the constraint. Introducing a slight change to G, however, like adding a self-edge labeled $next$ to v_2 (included in the code via the commented-out line 55), changes the result to UNSAT, since $G \models concr\_{F}\left(\{(\alpha, m)\}\right)$ no longer holds. In analogy to Lemma 11, this extension of the embedding encoding is also correct.

Lemma 12 (Correctness of Constrained Embedding Encoding). *Let G be a graph and (S, Λ) be a constrained shape. Let Sc be an SMT script containing an instance of the assertion defined in Code Template 22 for G and (S, Λ). Let F be the set of possible embeddings between G and (S, Λ) (see Def. 61).*

Then there is a one-to-one relationship between models of Sc and embeddings in F, i.e. each model of Sc is associated with a valid embedding of G into (S, Λ), and vice-versa. Specifically, this means that Sc is UNSAT if and only if $F = \emptyset$.

Proof. This follows directly from Lemma 11 and the fact that CoT. 20, by Definition, requires that for any given $\_F$, the graph satisfy all possible interpretations of the match of the constraint through that $\_F$. Therefore, the models (more specifically, the interpretation of $\_F$ in those models) represent *exactly* those embeddings that additionally satisfy the constraint embedding condition. □

4.5 Encoding of a Trace

In Section 4.1, we briefly touched on the importance of obtaining an encoding for the existence of a concrete trace corresponding to an abstract trace. Having created encodings for graphs and embeddings in the previous section, we now turn to creating such an encoding, starting with its overall structure, and deferring the details of its components to later sections. Before we can start to build this encoding, however, we first need to formally establish the context in which it would be applied.

Ultimately, the goal for all techniques presented in this thesis is to verify a safety property represented by some error pattern E for a given, possibly infinite-state graph transformation system. Let $\mathcal{G} = (I, \mathcal{R})$ be such a system. If the GTS is infinite-state, or has an otherwise prohibitively large state space, we would abstract it into a shape transformation system. This can be done by merely creating the canonical abstraction

$\hat{I}$ of I (cf. Def. 50). Thus, $\mathcal{S} = \left(\hat{I}, \mathcal{R}\right)$ would be the corresponding STS. Now suppose that, in the course of creating the shape transition system for $\mathcal{S}$, a shape S_k (and a path of length k to that shape) were discovered which potentially matches E. Such a path is called an abstract error trace.

Definition 65 (Abstract Error Trace). *Let $\mathcal{S} = \left(\hat{I}, \mathcal{R}\right)$ be an STS, and let $trans\left(\mathcal{S}\right) = \left(N, E^1, E^2\right)$ be its shape transition system. Let further $E = (L, id_L, L)$ be an error pattern. For any $k \in \mathbb{N}$, an* abstract error trace *of $\mathcal{S}$ through $trans\left(\mathcal{S}\right)$ is a path*

$$\left(S_0, \left(P_0, m_0, S_0^m\right), S_1\right) \cdots \left(S_{k-1}, \left(P_{k-1}, m_{k-1}, S_{k-1}^m\right), S_k\right) = \pi \in E^{2^*}$$

of length k such that

$$\begin{aligned} S_0 &= \hat{I} && \textit{and} \\ [\![\varphi_E]\!]_{m_e}^{S_k} &\geq 1/2 \end{aligned}$$

for some match m_e.

As stated in Chapter 3, the existence of such an abstract error trace does not mean that a corresponding concrete error trace also exists. The question of the existence of such a trace is what we are aiming to answer with our encoding. But, before we can do so, we must define exactly what "corresponding" means in this context.

Definition 66 (Corresponding Concrete Error Trace). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, $trans\left(\mathcal{G}\right) = \left(N, E^1, E^2\right)$ be its transition system, and let $\mathcal{S} = \left(\hat{I}, \mathcal{R}\right)$ be an STS such that $I \sqsubseteq_{f_0} \hat{I}$ for some f_0. Let $E = (L, id_L, L)$ be an error pattern and let $\pi = e_0 e_1 \cdots e_k$ be an abstract error trace for E in $trans\left(\mathcal{S}\right)$, where*

$$e_i = \left(S_i, \left(P_i, m_i, S_i^m\right), S_{i+1}\right).$$

Then a corresponding error trace *π' for π is a path*

$$\left(I, \left(P_0, m_0\right), G_1\right) \cdots \left(G_{k-1}, \left(P_{k-1}, m_{k-1}\right), G_k\right) = \pi' \in E^{2^*}$$

of length k and a match m'_e of E into G_k, such that

$$\begin{aligned} G_i &\sqsubseteq_{f_i} S_i^m && \textit{and} \\ f_i \circ m_i &= \mathrm{id}_{N_{L_{P_i}}} && \textit{and} \\ [\![\varphi_E]\!]_{m'_e}^{G_k} &= \mathbf{1} && \textit{and} \\ f_k \circ m'_e &= \mathrm{id}_{N_{L_E}} \end{aligned}$$

Thus, a concrete trace, i.e. a path through the transition system of a graph transformation system, corresponds to an abstract error trace if

- it is of the same length,
- begins at the initial graph,
- ends in a graph that matches the error pattern,
- each of its intermediate graphs is embedded into the materialized shapes of the respective steps in the abstract error trace,
- and each match used is compatible with the corresponding match used in the abstract trace via that embedding.

This situation is depicted in Fig. 4.4. Note that, while the embeddings into materialized shapes will completely determine the match used for any concrete graph, this does not mean that each abstract trace corresponds to only one concrete error trace at most. This is due to the fact that, while the embeddings determine the matches, the embeddings themselves are not fully determined.

Lemma 13 (Number of corresponding concrete traces). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, let $\mathcal{S} = \left(\hat{I}, \mathcal{R}\right)$ be an STS covering $\mathcal{G}$, and let $trans\left(\mathcal{S}\right) = \left(N, E^1, E^2\right)$ be its shape transition system. Let further π be an abstract error trace of length k in $trans\left(\mathcal{S}\right)$. Let π'_i be the set of corresponding concrete error paths to π_i, the prefix of the abstract error path of length i. Let further $G\left(\pi'_i\right)$ be the set of terminal graphs of these paths, i.e. the nodes in $trans\left(\mathcal{G}\right)$ that these paths terminate at. Then every graph G in $G\left(\pi'_i\right)$ creates at most as many graphs in $G\left(\pi'_{i+1}\right)$ as it has distinct embeddings into S^m_i.*

Proof. Let $0 < i < k$, and $G \in G\left(\pi'_i\right)$. The definition of a corresponding concrete trace now demands that $G \sqsubseteq_f S^m_i$. It also demands that the match m' of the rule P_i to G is essentially the inverse of the embedding into the part of S^m_i that is $N_{L_{P_i}}$, i.e. $f_i \circ m_i = \text{id}$. Since $N_{L_{P_i}}$ must be a fully concrete subgraph of S^m_i, f_i must be injective when restricted to it, meaning that m_i is fully determined by f_i. Since P_i and m_i are now fixed, the result of applying P_i to G at m_i is also fixed, meaning that every $G \in G\left(\pi'_i\right)$ creates at most as many non-isomorphic graphs in $G\left(\pi'_{i+1}\right)$ as it has distinct embeddings f_i into S^m_i. □

Thus, a corresponding error trace can be constructed by starting from I and continuing with the matches as defined by the subsequent embeddings into the materialized shapes, until either no next embedding can be constructed, or the end of the abstract trace is reached. The idea that there can be several possible embeddings of the same

$$\begin{tikzcd}
\hat{I} \arrow[d, squiggly] & & & S_1 \arrow[dr, squiggly] & & & & S_2 \arrow[dr, squiggly] & & & & & S_k \arrow[dr, squiggly] & & \\
S_0^m \arrow[rr, squiggly, "{P_0, m_0}"] & & S_1^t \arrow[ur, squiggly] & & S_1^m \arrow[rr, squiggly, "{P_1, m_1}"] & & S_2^t \arrow[ur, squiggly] & & S_2^m \arrow[rrr, squiggly, dotted] & & & S_k^t \arrow[ur, squiggly] & & S_k^e & \\
& (L_0, R_0) \arrow[ul, "id"'] \arrow[dl, "m_0"] & & & & (L_1, R_1) \arrow[ul, "id"'] \arrow[dl, "m_1"] & & & & (L_2 \arrow[ul, "id"'] \arrow[dl, "m_2"] & \cdots & R_{k-1}) & & & E \arrow[ul, "id"'] \arrow[dl, "m_e"] \\
I \arrow[u, dashed, "f_0"] \arrow[rrrr, squiggly, "{P_0, m_0}"] & & & & G_1 \arrow[u, dashed, "f_1"] \arrow[rrrr, squiggly, "{P_1, m_1}"] & & & & G_2 \arrow[u, dashed, "f_2"] \arrow[rrrrr, squiggly, dotted, "{P_2, m_2}" near start, "P_{k-1} m_{k-1}" near end] & & & & & G_k \arrow[u, dashed, "f_k"] &
\end{tikzcd}$$

Figure 4.4: Schematic view of a concrete trace corresponding to an abstract trace

graph into a materialized shape such that different graphs result might seem counter-intuitive. After all, the materialization is there to *remove* all ambiguity with regard to the match. The uncertainty regarding the embeddings stems from the materialization of concrete nodes out of summary nodes. The concrete graph that was embedded into the un-materialized shape might have embedded more than one node into such a summary node. The materialization does not prescribe which of these will be the concrete, materialized node, and which will remain embedded into the summary node (if it persists). Figure 4.5 illustrates such a situation[‡]. In rare cases where such situations do not occur, there can only be at most one corresponding concrete trace, since there is only one initial graph, and all embeddings and thus matches are prescribed by the abstract trace.

Remark 1. *In Def. 66 and the proof of Lemma 13, as well as in the remainder of this thesis, matches as defined in Def. 11 are often concatenated with mappings that operate on node sets alone. This is a slight abuse of notation and means that the mapping that is concatenated is the function that induces the match, as per Def. 8.*

Having established just what it is we wish to encode, we can now move on to think about the actual encoding.

[‡]edge labels have been omitted, since they are not relevant to the problem.

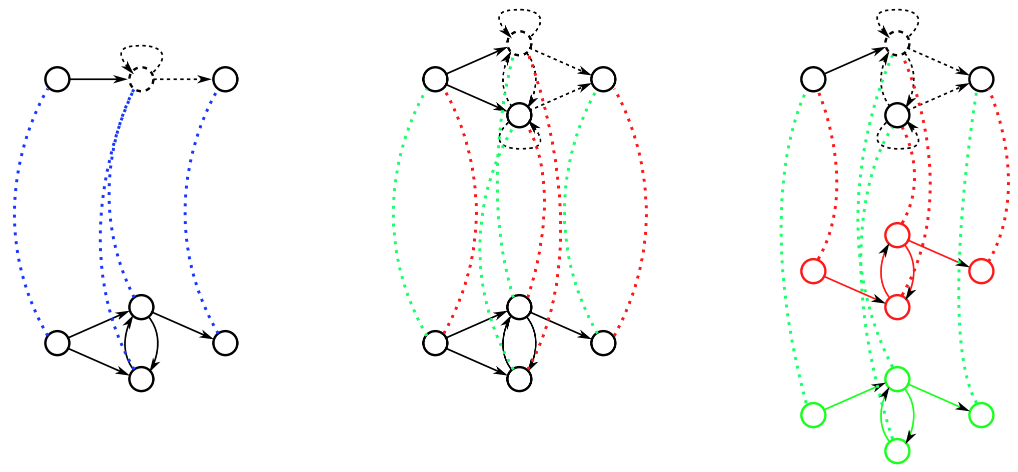


Figure 4.5: Two different embeddings of the same graph induce different results

4.5.1 Overview and Challenges

The basic idea for creating an encoding of a sequence of actions within a system is to encode the initial state, declare all subsequent states, and have the interpretation of these subsequent states be restricted by a conjunction of encoded transition relations that mirror the actions performed. In our case, the initial state would be the initial graph, whereas the actions performed would be the graph rules applied in the trace. Thus, our trace encoding will consist of an encoding of the initial graph, and a sequence of rule application encodings, all combined in one big conjunction.

Building on Def. 66, this encoding must not just encode a sequence of graph rule applications at the concrete level, but should also make sure that the rules are applied in such a way as to be consistent with the way they were applied at the abstract level. This "consistency" is provided by the embedding functions f_i between the graphs G_i and the shapes S_i^m, as shown in Fig. 4.4. Thus, for each step and each graph G_i we need to encode two things: that G_i is embedded into S_i^m, and that G_i is transformed by the i-th rule application into G_{i+1} using a compatible match as determined by f_i.

Embedding encodings have already been dealt with in Sec. 4.4. We will thus reuse that encoding and focus on the encoding of the rule applications themselves. The encoding of such a rule application, i.e. a transformation of a graph into another graph, as opposed to merely encoding static graphs, brings with it a new set of challenges.

One of these challenges is the fact that, in one encoding, we need to represent several entirely different graphs or graph sets – before and after each rule application. The question then is how to separate them. This can be achieved in one of two ways.

One possibility would be to separate the universes, i.e. to create separate sets of node constants for the different graphs. This seems the most intuitive option, since in the graph formalism the graph (sets) are also separated chiefly by their universes / node sets.

The other way would be to separate not the universes, but the predicate sets. Thus, we would have one set of edge predicates for the graph before a rule application, and one for after. Both of these predicate sets would operate on the same universe, thus realizing two different graphs.

In terms of performance, tests performed by Tobias Isenberg[96] using similar encodings suggest that neither option is superior. For this thesis, we choose the latter option. We will index our edge predicates in order to distinguish graphs before and after (i.e., in between) rule applications, in analogy to the static single assignment approach in predicate abstraction[81].

Another challenge is the fact that all our encodings so far have relied on a fixed, stable universe of nodes. This makes the encoding of rule applications that add or remove nodes difficult. For individual (and indeed any a priori fixed number of) rule applications, the solution to this is simple, and employed in our previous work[97] based on Tobias Isenberg's master thesis[96] as well.

Rather than attempting to encode the actual creation or destruction of nodes, we pre-compute the maximum number of nodes that will be needed by any graph that we encode and then create that many universe constants. The distinction between "currently existing" and "currently deleted" or "reserve" nodes can then be made by additional predicates.

These predicates will be called `past`, `present` and `future`, representing nodes that were deleted, nodes that are currently "active", and reserve nodes that may (or may not) later on become active. Any graph rule that adds or removes nodes can easily be converted into one that manipulates these three predicates instead.

Both of these challenges – how to separate the individual graphs and how to deal with node dynamics – must be dealt with before an encoding of a trace can take place.

4.5.2 Indexing and GTS / STS Modification

We begin by describing how we deal with node dynamics, as this solution must be applied before the trace is even constructed. As stated above, the basic idea of dealing with node dynamics is to "emulate" the actual creation and deletion of nodes by manipulating special predicates.

This works very well on the abstract level of shapes and shape transformations, because the abstraction allows us to create (potentially infinite) "reservoirs" for new nodes and "repositories" for deleted nodes. This is done with summary nodes.

Let $\mathcal{S} = (I, \mathcal{R})$ be a regular shape transformation system In order to modify it to emulate node creation rather than actually performing it, the initial shape I is first modified to contain two new summary nodes P and F, as well as three new predicates *past*, *present*, and *future*. These new predicates are interpreted such that *future* is $\mathbf{1}$ on F and $\mathbf{0}$ everywhere else, *past* is $\mathbf{1}$ on P and $\mathbf{0}$ everywhere else, and finally *present* is $\mathbf{0}$ on P and F, but $\mathbf{1}$ everywhere else.

The intention of this modification is clear: F serves as a reservoir for nodes to be created in the future, P serves as a repository for nodes that were deleted, and *present* marks all nodes that "actually" belong to the shape. New nodes are materialized out of F and marked *present*, whereas deleted nodes are marked *past* after being stripped of all other edges. This has the added benefit that the canonical abstraction described in Def. 50 will always restore a transformed shape to the form where there is exactly one node labeled *past*, and exactly one labeled *future*. Formally, the modification of the starting shape I is defined by the following definitions. First, an auxiliary definition meant to make the presentation of the actual definition more concise.

Definition 67 (Characteristic Function). *Let A be a set. For any given subset $B \subseteq A$, the* characteristic function χ_B^A *of that subset, is defined by*

$$\begin{aligned} \chi_B^A : A &\to \mathcal{K} \\ x &\mapsto \begin{cases} \mathbf{1} & \textit{if } x \in B \\ \mathbf{0} & \textit{else} \end{cases} \end{aligned}$$

Thus, the characteristic function identifies *its subset.*

With that, we can easily define the modification of a shape as described above.

Definition 68 (Modified Shape). *Let $S = (U, \mathcal{P}, \iota)$ be the a shape over the predicate set $\mathcal{P}$. The* modified shape $S' = (U', \mathcal{P}', \iota')$ *is defined by*

$$\begin{aligned} \mathcal{P}' &:= \mathcal{P} \cup \{\textit{past}, \textit{present}, \textit{future}\} \\ U' &:= U \cup \{p, f\} && \textit{(add two nodes)} \\ \iota' &:= \iota'' \cup \iota_{pa} \cup \iota_{pr} \cup \iota_{fu} \\ \iota'' &:= (s, k) \mapsto \begin{cases} \iota(p)(u_1, \ldots, u_k) & \textit{if } (u_1, \ldots, u_k) \in U^k \\ \mathbf{0} & \textit{otherwise} \end{cases} \\ \iota_{pa} &:= \textit{past} \mapsto \chi_{\{p\}}^{U'}, \quad \iota_{pr} := \textit{present} \mapsto \chi_U^{U'}, \quad \iota_{fu} := \textit{future} \mapsto \chi_{\{f\}}^{U'} \end{aligned}$$

applying consistent renaming where necessary.

Figure 4.6 shows an example of the modification process described in this definition. Having modified the shape, we now turn to the graph rules. They must be modified as

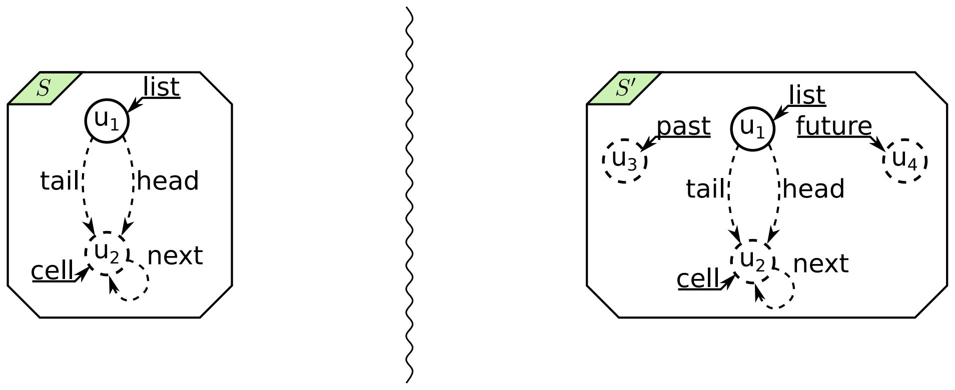


Figure 4.6: Modification of a shape representing a non-empty linear list

well in order to turn node creation into node materialization, and node deletion into node labeling.

Let $P = (L, p, R)$ be a graph rule. In order to modify it to emulate rather than perform node creation and deletion, we first take the node set $N_R \setminus N_L$, i.e., the set of nodes created by the rule, and add it to the left hand side. Then we add *future* edges to these newly added nodes, as well as *present* edges to all other nodes in the left and right hand side, and add the identity map on all these added elements to p. This has the effect that, instead of actually creating new nodes, the rule now tries to match nodes marked *future* for the nodes it wants to create, and nodes marked *present* for all other nodes.

After having thusly dealt with node creation, we turn to node deletion. Similarly to before, we add the nodes in $N_L \setminus N_R$ to R, and label them *past* in R, again adding the identity map on the newly created elements to the rule isomorphism p. This has the effect that, instead of actually deleting the nodes, the rule now labels them as *past* and removes all edges from them that it used to match them.

Note that, while this removes some of the edges from the "deleted" nodes, it does not necessarily remove *all* of them, since deleted nodes may be incident to edges that were not part of the match – this is the "dangling edge" problem familiar from Chap. 2. This necessitates a small change in the SPO rule application semantics, in order to accommodate deleting nodes using a *past* label. Formally, this modification of rules is defined as follows.

Definition 69 (Modified Rule). *Let $P = (L, p, R)$ be a graph rule over a predicate set $\mathcal{P}$. The* modified rule *$P' = (L', p', R')$ is defined using the modified graphs L' and R', which are defined as follows*

$$L' := \left(N'_L, {E'}^1_L, E^2_L\right)$$

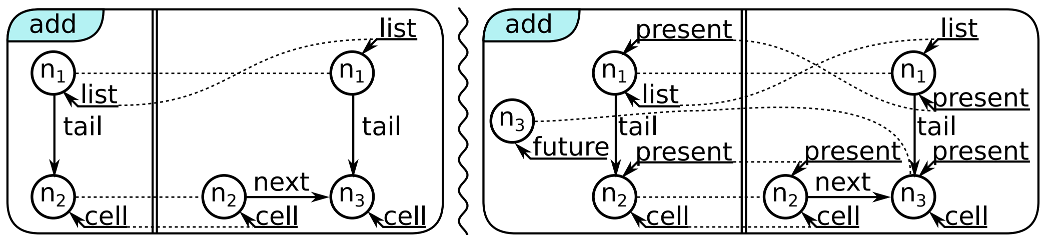


Figure 4.7: Modification of the add-rule of the linear list GTS

$$
\begin{aligned}
N'_L &:= N_L \cup (N_R \setminus N_L) \\
{E'}^1_L &:= E^1_L \cup \{(\mathit{present}, n) \mid n \in N_L\} \cup \big\{(\mathit{future}, n) \mid n \in \big(N'_L \setminus N_L\big)\big\} \\
R' &:= \Big(N'_R, {E'}^1_R, E^2_R\Big) \\
N'_R &:= N_R \cup (N_L \setminus N_R) \\
{E'}^1_R &:= E^1_R \cup \{(\mathit{present}, n) \mid n \in N_R\} \cup \big\{(\mathit{past}, n) \mid n \in \big(N'_R \setminus N_R\big)\big\}
\end{aligned}
$$

The new isomorphism between L' and R' is defined by

$$
\begin{aligned}
p' &:= p_r \cup p_p \\
p_r &:= p \cup id_{N_L \setminus N_R} \cup id_{N_R \setminus N_L} \\
p_p &:= \big\{\big((\mathit{present}, n_l), (\mathit{present}, n_r)\mid (\mathit{present}, n_l) \in E^1_L \\
&\qquad\qquad \wedge (\mathit{present}, n_r) \in E^1_R \wedge (n_l, n_r) \in p_r\big)\big\}
\end{aligned}
$$

Note that, when P neither adds nor deletes nodes, P' only differs from P by requiring (and maintaining) present edges on all its nodes.

Figure 4.7 shows an example of the modification process described in this definition. The *add*-rule is changed from creating a new node to add to the list to matching a *future*-node and turning it into a *present*-node. In the course of this thesis, we will often use these modified rules instead of regular rules. As stated above, these require a slightly modified version of the SPO rule application definition.

Definition 70 (Modified Rule Application (SPO)). *Let G be a graph, $P = (L, p, R)$ be a rule modified as per Def. 69, $m : L \to G$ be a valid match, and G' be the result of* applying P to G at m using the single pushout (SPO) method. *We define the match on the nodes of the right hand side as $f := p \circ (m \mid N_{L_c}) \cup id_{N_{R_c}}$. The graph G' is then defined as follows:*

$$
\begin{aligned}
N_{G'} &:= N_G \\
E^1_{G'} &:= \Big[E^1_G \setminus m\left(E^1_L \setminus E^1_{L_c}\right) \cup \hat{f}\left(E^1_R \setminus E^1_{R_c}\right)\Big] \setminus \big\{(u, n) \mid \big(\mathit{past}, m^{-1}(n)\big) \in E^1_R\big\}
\end{aligned}
$$

$$E^2_{G'} := \left[E^2_G \setminus m\left(E^2_L \setminus E^2_{L_c}\right) \cup \hat{f}\left(E^2_R \setminus E^2_{R_c}\right)\right] \setminus \left\{(n_1, b, n_2) \mid \left(\mathit{past}, m^{-1}(n)\right) \in E^2_R\right\}$$

In the original Definition 13, the removal of dangling edges was taken care of by restricting the edge set to the new node set. With the node set being constant for modified rules, this had to be changed. In the above definition, the job of removing dangling edges is performed by removing all edges on nodes that were matched by rule nodes labeled *past* in the right hand side. Since these are exactly the nodes that the original, unmodified rule would have deleted, it is easy to see that the modified rule application has the same effect as the original one.

Lemma 14 (Equivalence of Modified Rules). *Let S be a shape, let $P = (L, p, R)$ be a graph rule, and let P^{mod} be the result of modifying P according to Def. 69. For any shape S, let $\mathrm{mod}(S)$ be the result of modifying S according to Def. 68. Let*

$$mod(S) \xrightarrow{P^{mod}, m, mod(S)^m}_{\mathcal{C}} S'$$

be the normalized *transition resulting from applying P^{mod} to $\mathrm{mod}(S)$ at some potential match m, provided such a match exists. Let further $S \xrightarrow{P, m', S^{m'}}_{\mathcal{C}} S''$ be the normalized transition resulting from applying P to S at $m' := m{\downarrow}_{N_{L_P}}$. Then we have*

$$S' = mod\left(S''\right),$$

i.e. GTS modification and shape transition commute.

Proof Sketch. Modified rule application differs from regular rule application only in the treatment of added and deleted nodes. Every *past*-node created by the deletion of a node by a modified rule will by Def. 50 be merged with the node P created by the shape modification (see Def. 68). Thus, they are absent from the *present*-part of the shape after rule application. Similarly, every *future*-node created by materialization out of the node F created by the shape modification will have its *future*-edge removed by the rule application and take the same place that the created node would have taken in the original rule application. Therefore, S' can only differ from S'' in the existence of the P and F nodes, and the *present* edges on the shape body, exactly what the mod-operation would add.

Note that this even works with Constraint-safe Abstraction (see Def. 64), since constraints can never touch *past* or *future* nodes (by Def. 70), and are thus always mergable by the basic canonical abstraction.

□

Thus, applying a modified rule to a modified shape yields the same shape as modifying the shape resulting from applying the unmodified rule to the unmodified shape. Now that we are able to modify shapes and rules to emulate rather than perform node creation and deletion, we can modify an entire shape transformation system to do the same.

Definition 71 (Modified Shape Transformation System). *Let $\mathcal{S} = (I, \mathcal{R})$ be a shape transformation system. Let I' be the shape I, modified according to Def. 68, and let $\mathcal{R}'$ be the result of modifying every rule in $\mathcal{R}$ according to Def. 69. Then the* modified shape transformation system *$\mathcal{S}'$ is given by $\mathcal{S}' = (I', \mathcal{R}')$.*

This modified shape transformation system is equivalent to the original shape transformation system, provided the modified SPO semantics are used.

Lemma 15 (Modified STS Equivalence). *Let $\mathcal{S} = (I, \mathcal{R})$ be an STS, and let $\mathcal{S}' = (I', \mathcal{R}')$ be its modification according to Def. 71. Let $T := trans\,(\mathcal{S})$ be the transition system for $\mathcal{S}$, and let $T' := trans\,(\mathcal{S}')$ be the transition system for $\mathcal{S}'$, obtained using the modified SPO semantics specified in Def. 70. Then there exists an isomorphism between T and T'.*

Proof Sketch. This follows inductively from Lemma 14. Since the modification commutes with every individual transition that is created, we can create the isomorphism just by binding I to I' and simultaneously constructing the state spaces of both transition systems, constructing and updating the isomorphism as we go. □

From now on, whenever we consider the encoding of a trace, we will assume that the trace is part of the transition system of a *modified* transformation system, as defined above. To illustrate the effect of this whole process of modifying the shape transformation system, consider the example shown in Fig. 4.8. Here, the modified start rule is applied to the modified start shape, and upon the result, the modified end rule is applied. After every application, the resulting shape is abstracted to its canonical form (see Def. 50). This example illustrates nicely how new nodes are materialized out of the *future*-labeled node, and deleted nodes are absorbed (by the canonical abstraction) into the *past*-labeled node.

Having solved the issue of node dynamics, what remains is the issue of graph separation. As stated in Sec. 4.5.1, the graphs in the trace are separated by indexing. This indexing happens in a very straightforward manner. The trace encoding begins with an encoding of the initial graph. Each rule application will define a new graph or graph set constructed as the result of applying the rule to the respective previous graph (set).

We give the initial graph the index 0, and increment that index by 1 with each rule application. Thus, in the context of a single application of a rule $P = (L, p, R)$ to a graph $G = (U, \mathcal{P}, \iota)$, yielding a result graph $G' = (U', \mathcal{P}', \iota')$, we add some integer i

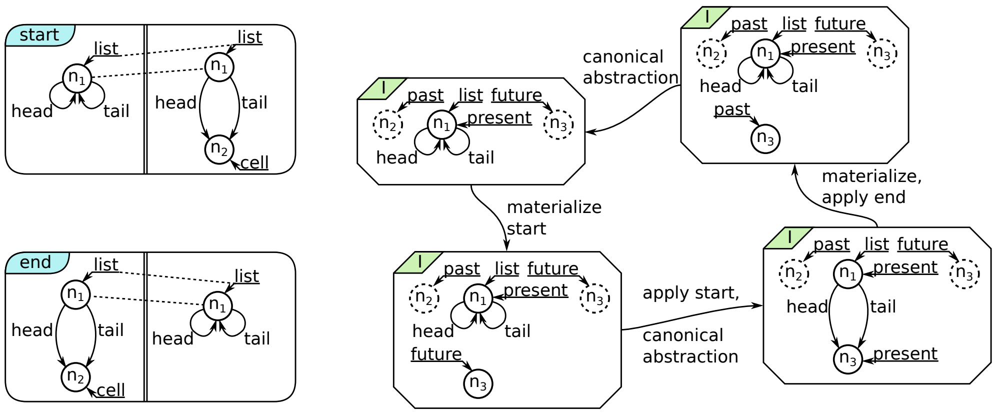


Figure 4.8: Example application of modified start and end rules

as an index to all labels in $\mathcal{P}$, and $i+1$ as an index to all predicates in $\mathcal{P}'$. This approach creates a situation where each satisfying interpretation for the predicate set with index i corresponds to one possible graph obtained from the initial graph by applying the first i rules of the trace. In the following, we will often need to refer to specific indices of predicate sets in order to specify exactly which graph we are talking about. This is done with the help of an indexed predicate set.

Definition 72 (Indexed Predicate Set). *Let $\mathcal{P}$ be a set of predicates, and let $k \in \mathbb{N}$ be a positive integer. Then we define the k-indexed predicate sets of $\mathcal{P}$ by*

$$\mathcal{P}_i := \{(s_i, k) \mid (s, k) \in \mathcal{P}\} \qquad 0 \leq i \leq k$$

Thus, given a graph transformation system $\mathcal{G} = (I, \mathcal{R})$, and an abstract trace π of length k, we can think of each graph in a corresponding concrete trace π' as using one of the k-indexed predicate sets of $\mathcal{P}$.

4.5.3 High-Level Encoding of a Trace

Now that all the trace-level parts are in place, we can construct the framework for the encoding of a concrete trace corresponding to an abstract trace. Since we have not defined all of the components of this encoding yet, we will use placeholder names, thereby giving a good overview over the high-level structure of a trace encoding, without going into too much detail.

As stated before, every trace encoding must begin with an encoding of the initial graph. This encoding can be done using the basic graph encoding detailed in Sec. 4.3. Note, however, that since we will need this graph to be embedded into the *modified* initial shape of the shape transformation system, we must modify it as well, on top of using the predicate set $\mathcal{P}_0$ instead of $\mathcal{P}$. This is done analogously to Def. 68, with the exception that the added nodes can of course not be summary nodes and must thus be regular nodes. Apart from this, the graph encoding can proceed just as normal, creating an instance of Code Templates 3, 5, and 6. These code templates will require a number of predicate declarations in the prelude of the encoding. We will denote these declarations with the shorthand `declare_graph` in the encoding of the trace.

Next, we must take care that enough additional nodes are present to account for potentially created nodes in the trace. Since at the concrete level, we can not use summary nodes, we must determine how many nodes any graph in the trace is going to need, and then create additional "overflow" nodes such that the largest graph in the trace can be expressed on the nodes of the original graph plus the overflow nodes. We also need to express that these overflow nodes are not part of the actual start graph. This is achieved by marking them *future* nodes and encoding the absence of any other edges touching these nodes. The following code template realizes this.

Code Template 23 (Overflow Nodes). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, and let $\pi = (S_0, (S_0^m, P_0, m), S_1) \cdots \left(S_{k-1}, \left(S_{k-1}^m, P_{k-1}, m\right), S_k\right)$ be an abstract trace of length k in the corresponding modified STS. For each rule $P \in \mathcal{R}$, let $|P| \in \mathbb{N}_0$ be the number of nodes in its left hand side[§]. Then $\lambda := \sum_{i=0}^{k-1} |P_i|$ is the number of nodes required by the matches in the trace π and we encode the formula*

$$\bigwedge_{x \in O} \bigwedge_{\substack{y \in O \cup N_I \\ x \neq y}} \neg (x = y) \wedge \bigwedge_{x \in O} \mathit{future}(x) \wedge \bigwedge_{\substack{(s,1) \in \mathcal{P}' \\ s \neq \mathit{future}}} \bigwedge_{x \in O} \neg s(x)$$
$$\wedge \bigwedge_{x \in O} \bigwedge_{y \in O \cup N_I} \bigwedge_{(s,2) \in \mathcal{P}'} \neg s(x, y)$$

as the function `overflow` *in SMTLib, using an appropriate function definition. Here, O is the set of overflow nodes and N_I is the node set of the modified initial graph.*

Note that technically, only the actually created nodes would have to be declared as overflow nodes for the trace. However, here, we use the entire left hand side of the rules, in order to make sure inapplicability of rules is always a matter of missing edges, rather than missing nodes. The specific reason for this will become clear in Chapter 5.

Prior to defining this function, the node constants used to represent the overflow nodes have to be declared. Once this is done, the function `overflow` states that the

[§]remember that in a modified rule, the created nodes are in the left hand side as well overflow nodes are pairwise distinct from each other and from all nodes in the initial graph, and that except for the *future* predicate, no predicate in $\mathcal{P}_0$ evaluates to $\mathbf{1}$ on them or between them and the nodes of the initial graph.

Lemma 16 (Correctness of Overflow Encoding). *Let $\mathcal{S} = (I := (U, \mathcal{P}, \iota), \mathcal{R})$ be a modified STS over a predicate set $\mathcal{P}$, π be an abstract error path for $\mathcal{S}$, k be the number of nodes created in π, and let Sc be an SMT script containing an instance of Code Template 3, 5, 6, and 23, as well as their prerequisite declarations. Then there is only one satisfying model for Sc, containing the logical structure of the following shape.*

$$S := (U', \mathcal{P}, \iota')$$
$$U' := U \mathbin{\dot\cup} \{o_1, \dots, o_k\}$$
$$\iota' := \iota \cup \iota_1 \cup \iota_2$$
$$\iota_1 := \bigcup_{(s,k) \in \mathcal{P} \setminus \{\mathit{future}\}} \left\{ (s,k) \mapsto \left((x_1, \dots, x_k) \mapsto \begin{cases} \mathbf{0} & \textit{iff } x_i \notin U \\ \iota(s,k)(x_1, \dots, x_k) & \textit{else} \end{cases} \right) \right\}$$
$$\iota_2 := \left\{ (\mathit{future}, 1) \mapsto \left(x \mapsto \begin{cases} \mathbf{1} & \textit{iff } x \notin U \\ \iota(\mathit{future})(x) & \textit{else} \end{cases} \right) \right\}$$

Proof. Obvious from Lemma 9 and the construction of `overflow`. □

Having taken care of the initial graph and the overflow nodes, we can now turn to the encoding of the actual transitions that make up the corresponding concrete trace. Each such transition determines the form of the graph that follows it (G_{i+1}) based on the form of the graph that precedes it (G_i). This has two parts, as depicted in Fig. 4.4: an embedding of G_i into S_i^m, as well as an encoded relationship between the edges of G_i and G_{i+1}.

Encodings of Graph embeddings have already been defined in Sec. 4.4. We can thus just use Code Template 18 or 22 as appropriate, as well as their prerequisite code templates, together with the required declarations (`declare_embedding`) in the prelude of the script to realize this part of the transition. In order to distinguish the individual embeddings from each other, we index the prerequisite function definitions for these assertions, as well as the declarations with the index of the shape that they encode the embedding into.

Note that, since the initial graph has already been defined, and the edges of each subsequent graph in the trace graphs is determined by the encoding of the rule applications, the portions of the embedding encoding that encode the graph to be embedded, i.e. `distinctNodes`[¶], `unaryG` and `binaryG`, can be omitted. Each such embedding encoding thus uses the Code Templates 11, 12, 14, 15, 16, 17, and, if necessary, 21.

[¶]The part about the nodes of the *graph*, not the part about the nodes of the *shape*

What remains for each step is the encoding of the rule application itself. A thorough definition of this encoding is given in Sec. 4.6. For now, we will assume that an application encoding consists of two functions: `rule_selection` and `rule_application`, each of which can also be indexed with the appropriate step number. Declarations necessary for these templates are referred to by their placeholder name `declare_rule` in the trace encoding.

Now that the initial graph, the overflow nodes, the embeddings and the rule applications are all accounted for, the trace encoding is complete. The initial graph and the overflow encoding define the starting point, the embeddings and rule applications generate the resulting restrictions on the graphs along the trace, and the final embedding (into the final shape in which the error was materialized) makes sure the last graph in the trace matches the error. This leads us to the following code template for the trace encoding.

Code Template 24 (Corresponding Concrete Trace). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, and let $\pi = (S_0, (S_0^{m_0}, P_0, m_0), S_1) \cdots (S_{k-1}, (S_{k-1}^{m_{k-1}}, P_{k-1}, m_{k-1}), S_k)$ be an abstract trace of length k in the corresponding modified STS. Let further S_k^e be a materialization of the error pattern E out of S_k. Then the corresponding concrete trace can be encoded using using appropriate instances of Code Templates 3, 5, 6, 23, 11, 12, 14, 15, 16, 17, 21, 26, and 27. For the full listing, see Code Listing B.4 in Appendix B, page 260.*

Any models of an SMT script containing an instance of the above template represents a sequence of graphs that constitute a valid path through the concrete transition system of $\mathcal{G}$, such that each graph in the path embeds into its corresponding shape in the abstract trace, and each rule application is matched in such a way that it is compatible with the corresponding rule application in the abstract trace, as defined in Def. 66.

Lemma 17 (Correctness of Trace Encoding). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, and let $\pi = (S_0, (S_0^{m_0}, P_0, m_0), S_1) \cdots (S_{k-1}, (S_{k-1}^{m_{k-1}}, P_{k-1}, m_{k-1}), S_k)$ be an abstract trace of length k in the corresponding modified STS. Let further S_k^e be a materialization of the error pattern E out of S_k. Let Sc be an SMT script containing an instance of Code Template 24 and its prerequisite templates. Then any model of Sc contains the logical structures of graphs $G_0, \ldots, G_k$ as well as the logical structures of instances of the rules $P_0, \ldots, P_{k-1}$, and interpretations of functions $f_0, \ldots, f_k$, such that*

$$\pi' := (G_0, (P_0, m_0'), G_1) \cdots (G_{k-1}, (P_{k-1}, m_{k-1}'), G_k)$$

is a corresponding concrete trace for π, and $G_k \sqsubseteq S_k^e$.

Proof. First, Lemma 16 implies that G_0 is part of any model for the script, along with a set of k_o overflow nodes where k_o is the sum of the left hand sides of the rules in the

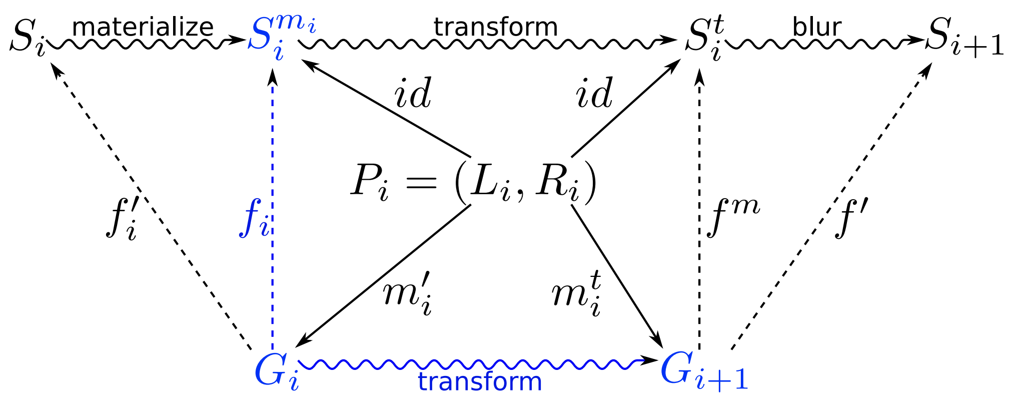


Figure 4.9: A schematic of the application of a rule to a shape and its effect on embedded graphs

trace. This ensures that there are always enough nodes of the `Node` sort to embed into the materialized shapes of the abstract trace. Furthermore, Lemma 11 implies that an embedding function $\_F\_0$ embedding G_0 into S_0^m is also part of the model.

We will now assume that the rule application encoding implies that a graph G_1 such that $G_0 \xrightarrow{P_0,\_F\_0^{-1}\circ \mathrm{id}_{L_{P_0}}} G_1$ is part of the model, i.e. that the rule application encoding constructs G_1 from G_0 using the match implied by the embedding. Now that G_1 is part of the model, we can proceed as with G_0 and obtain G_2 and so on. We also obtain $G_k \sqsubseteq_{\_F\_k} S_k^e$ (form the correctness of the k-th embedding by Lemma 11).

Thus, given the correctness of the rule application encoding, the trace encoding is correct. □

Having established the overall structure of a trace encoding, we can now move on to the final missing piece, the rule application encoding.

4.6 Encoding of a Rule Application

What remains of the trace encoding described in Sec. 4.5 is the encoding of the actual rule application. We will base this encoding on previous work[96, 97], with a few modifications to take advantage of the different context.

Each individual transformation $G_i \xrightarrow{P,m} G_{i+1}$ contains 4 essential mathematical objects – the graph G_i, the rule P, the match m, and the result graph G_{i+1}. When G_i,

P, and m are fixed, the resulting graph G_{i+1} is uniquely determined. As shown in the proof of Lemma 13, the match m is actually fully defined by the embedding into S_i^m. Thus, our encoding will be based on G_i, S_i^m and P, leaving the solver to create the matching m from f_i and a result graph G_{i+1} to complete the model, creating a valid graph transformation in the process.

Figure 4.9 shows a schematic of the context from which the encoding is constructed. The highlighted parts are the ones that will actually be encoded, with the "transform" part being described in this section.

Since each transformation encoding is part of a sequence, the graph G_i need never be explicitly encoded, since it can be assumed to have been provided by the encoding of the previous step, or the encoding of the initial graph if $i = 0$. Furthermore, S_i^m is, in the context of a trace encoding, already encoded by the embedding encoding. Therefore, the only mathematical object that actually must be added to the encoding is P.

Now, before the actual encoding can be constructed, we need to decide how the various parts will be represented. We need to somehow represent a match, i.e. a function mapping nodes onto other nodes, and we need to refer to that match when determining the effect of the rule P at that match on G_{i+1}. If this function were to be encoded explicitly, this referral would be rather cumbersome, requiring scores of "if-then" statements making the value of certain edges in G_{i+1} dependent on the (at the time of the encoding unknown) values of the match. Such effects can be seen in the embedding encoding (see Sec. 4.4), where the embedding itself is represented as such a function.

In this particular context, however, we can take advantage of the fact that, since matches are injective, we can emulate them by representing them as an equivalence relation ($=$) between the nodes of the rule and the nodes of the graph, rather than an explicit function. This has the benefit that we can refer directly to the rule nodes when specifying the effect on G_{i+1}, because by the definition of $=$ the formula must then also hold for the graph node to which the rule node was matched. Using equality instead of an explicit matching function brings with it the problem that the same rule cannot be used twice, but this is easily solved by simply creating a new (indexed) instance of the rule for every step in the trace. Having established this basic idea, we can move on to the actual encoding.

As stated above, the rule application encoding consists of two parts, called `rule_selection` and `rule_application`. In `rule_selection`, the match between P and G_i is established. In `rule_application`, the edges of G_{i+1} are encoded as they result from the edges of G_i and the encoded match. Beyond the declarations for the edges of G_i and G_{i+1}, as well as the embedding of G_i into S_i^m, the only declarations that are strictly necessary for this part of the encoding are the ones that establish the existence of the rule nodes. We will begin with these declarations.

Code Template 25 (Declarations for Rule Application Encodings). *Let* $(S_i, (S_i^m, P_i, m), S_{i+1})$ *be a single transition in an abstract error trace* π*, where* $P_i = (L, p, R)$ *with* $N_L = N_R$ *is a modified graph rule. The instances of the rule nodes* $n_0, \ldots, n_{\lambda_i}$ *for this single transition are then declared in the following way.*

```
(declare-fun P_i_n_0 () Node)
⋮
(declare-fun P_i_n_λi () Node)
```

This merely establishes the existence of these rule nodes. It does not establish that they are pairwise distinct or in any way encode the edges of the rule itself, since neither of these encodings are necessary. Any interpretation of these node constants will have to make them pairwise distinct, because they are later set equal to nodes that are required to be pairwise distinct by an instance of Code Templates 3 and 23. The edges of the rule would theoretically be necessary to establish a match. However, the embedding into a shape in which the left hand side of that rule has been materialized (S_i^m) is already equivalent to this. Thus, only the nodes themselves are required to act as a proxy between the edges of G_{i+1} and G_i.

We continue by defining the actual match condition. As noted earlier, the fact that a graph G_i is embedded into a shape S_i^m, in which a rule P is materialized using a match m, already establishes that the rule P matches the graph at $\mathrm{id}_L \circ f^{-1}$, a function which at this point is well-defined. With that in mind, all that remains to do to create the match is to require the nodes in the rule to be equal to the nodes that are embedded into the corresponding rule nodes in the abstract trace.

Code Template 26 (Match Encoding). *Let* $(S_i, (S_i^m, P_i, m), S_{i+1})$ *be a single transition in an abstract error trace* π*, where* $P_i = (L, p, R)$ *with* $N_L = N_R$ *and* $|N_L| := \lambda_i$ *is a modified graph rule. Let* $G_i \sqsubseteq_{f_i} S_i^m$ *be the graph at the* i*-th place in the corresponding concrete trace* π' *Let* $n_0, \ldots, n_{\lambda_i} \in N_L$ *be the rule node instances, and let* $v_0, \ldots, v_{\lambda_i} \in f_i^{-1}(N_L)$ *be the nodes of the graph* G_i *that are embedded into the materialized part of* S_i^m*. Then the encoding of the match is given by the formula*

$$\bigwedge_{j=0}^{\lambda_i} (n_j = v_j)$$

and encoded into SMTLib in a straightforward fashion as the function `rule_selection`*, using an appropriate function definition.*

The correctness of this encoding is trivial. An instance of this code template establishes the identity between the rule nodes and the graph nodes to which they must be matched, according to the embedding into the corresponding materialized shape.

This fairly simple encoding stands in stark contrast to the second part of the application encoding. Here, we must express that the graph G_{i+1} contains the edges P_i creates, does not contain the edges P_i removes, according to the match defined previously, and, crucially, *looks exactly like G_i in those places that are unaffected by the rule.*

More formally, for any node tuple (v_0) or (v_0, v_1) in the universe consisting of the nodes of the initial graph and the overflow nodes, and each predicate p_{i+1}, we must express the value of p_{i+1} at that tuple. Our encoding must do this without knowing a priori which nodes in G_i were matched by the rule. Here our earlier decision to encode matches as equality pays off in a significant simplification – we can simply check for any given node tuple in the universe whether it is matched by (read: equal to) a node tuple in the rule where a change occurs, and can then assert that change for G_{i+1}.

In order to avoid having to perform this check for absolutely every node tuple and predicate in the graph, we introduce a slight optimization. Most graph rules only change the values of a few predicates, leaving the others untouched. If, for example, a given rule has absolutely no effect on p-edges, it makes sense to simply encode that all p-edges stay the same, and not to check for any matches regarding p. Thus, given a rule P_i, all predicates p that are not changed by P_i can be updated for G_{i+1} using the formula

$$\bigwedge_{j=1}^{v} p_{i+1}(v_j) = p_i(v_j) \qquad \text{if } p \text{ is unary}$$

$$\bigwedge_{j_1=1}^{v} \bigwedge_{j_2=1}^{v} p_{i+1}(v_{j_1}, v_{j_2}) = p_i(v_{j_1}, v_{j_2}) \qquad \text{if } p \text{ is binary}$$

where v is the size of the universe, i.e. the number of nodes in the initial graph, plus the added past and future nodes (i.e. 2), plus the number of overflow nodes.

When the predicate p is in principle modified by the rule P_i, then we need to modify this formula to insert an exception. The value of the predicate at a given node tuple should remain the same, unless the node tuple was the target of the match of one of the p-edges in the rule that are changed by it. To ease the definition of this new formula, we first define the set of created and deleted edges for a rule.

Definition 73 (Changed Edge Sets). *Let P be a rule modified according to Def. 69. Let E^1_L, E^2_L be the edge sets of the left-hand side, and E^1_R, E^2_R be the edge sets of the right-hand side. Then the* changed edge sets E^1_+, E^1_-, E^2_+, *and E^2_- are defined as follows.*

$$E^1_+ := E^1_R \setminus E^1_L \qquad \textit{unary edges added}$$

$$E^1_- := E^1_L \setminus E^1_R \qquad \textit{unary edges removed}$$

$$E_+^2 := E_R^2 \setminus E_L^2 \qquad \textit{binary edges added}$$
$$E_-^2 := E_L^2 \setminus E_R^2 \qquad \textit{binary edges removed}$$

Equipped with these edge sets, we can now specify a formula that expresses that edges should remain the same, unless they are the match target of a changed edge, which is given by

$$\bigwedge_{j=1}^{v} \left[\neg \left(\bigvee_{(p,n) \in E_+^1 \cup E_-^1} (n = v_j) \right) \vee p_{i+1}(v_j) = p_i(v_j) \right],$$

if p is unary, and

$$\bigwedge_{j_1=1}^{v} \bigwedge_{j_2=1}^{v} \left[\bigvee_{(n_1,p,n_2) \in E_+^2 \cup E_-^2} (n_1 = v_{j_1} \wedge n_2 = v_{j_2}) \vee p_{i+1}(v_{j_1}, v_{j_2}) = p_i(v_{j_1}, v_{j_2}) \right],$$

if p is binary. Thus, the persistence of edges is only required if they are matched to none of the changed edges.

What remains is to express, as a formula, what happens to those edges that *do* match changed edges. This is done easily enough, by simply expressing the new value of the edge. Again, we reap the benefits of using equality as the match, because it enables us to simply express the edges on the rule nodes and have that automatically carry over to the graph nodes to which they are matched. The following formula demonstrates this.

$$\bigwedge_{(p,n) \in E_+^1} p(n) \wedge \bigwedge_{(p,n) \in E_-^1} \neg p(n) \wedge \bigwedge_{(n_1,p,n-2) \in E_+^2} p(n_1, n_2) \wedge \bigwedge_{(n_1,p,n-2) \in E_-^2} \neg p(n_1, n_2)$$

Building on these formulas, our encoding for the application of a rule will consist of four parts. The first part will define added edges, the second part will define deleted edges, the third part will encode definitely unchanged edges, and the fourth and last part will take care of conditionally unchanged edges.

Code Template 27 (Application Encoding). *Let $(S_i, (S_i^m, P_i, m), S_{i+1})$ be a single transition in an abstract error trace π, where $P_i = (L, p, R)$ with $N_L = N_R$ and $|N_L| := \lambda_i$ is a modified graph rule. Let $N_I \cup \{p, f\} \cup O = \{v_1, \ldots, v_v\}$ be the set of universe nodes, where O is the set of overflow nodes. Let $|U| = v$ and let $\mathcal{P}^0 = \{(s,k) \mid \forall (n_1, \ldots, n_k) \in N_L^k : \iota_L(s)(n_1, \ldots, n_k) = \iota_R(s)(n_1, \ldots, n_k)\}$ be the set of predicates that are not changed by P. Let further*

$$E_+^1 = \left\{ \left(u^{+,1}, n^{+,1}\right), \ldots, \left(u^{+,a}, n^{+,a}\right) \right\}$$

$$\begin{aligned} E_-^1 &= \left\{ \left(u^{-,1}, n^{-,1}\right), \ldots, \left(u^{-,b}, n^{-,b}\right) \right\} \\ E_+^2 &= \left\{ \left(n^{+,1}, b^{+,1}, m^{+,1}\right), \ldots, \left(n^{+,c}, b^{+,c}, m^{+,c}\right) \right\} \qquad \textit{and} \\ E_-^2 &= \left\{ \left(n^{-,1}, b^{-,1}, m^{-,1}\right), \ldots, \left(n^{-,d}, b^{-,d}, m^{-,d}\right) \right\} \end{aligned}$$

Then the encoding of the application step for P_i is given by the formula

$$\begin{aligned} & \bigwedge_{(p,n) \in E_+^1} p_{i+1}(n) \wedge \bigwedge_{(p,n) \in E_-^1} \neg p_{i+1}(n) \\ \wedge & \bigwedge_{(n_1,p,n_2) \in E_+^2} p_{i+1}(n_1, n_2) \wedge \bigwedge_{(n_1,p,n_2) \in E_-^2} \neg p_{i+1}(n_1, n_2) \\ \wedge & \bigwedge_{(p,1) \in \mathcal{P}^0} \bigwedge_{j=1}^{v} p_{i+1}(v_j) = p_i(v_j) \\ \wedge & \bigwedge_{(p,2) \in \mathcal{P}^0} \bigwedge_{j_1=1}^{v} \bigwedge_{j_2=1}^{v} p_{i+1}(v_{j_1}, v_{j_2}) = p_i(v_{j_1}, v_{j_2}) \\ \wedge & \bigwedge_{(p,1) \in \mathcal{P} \setminus \mathcal{P}^0} \bigwedge_{j=1}^{v} \left[\neg \left(\bigvee_{(p,n) \in E_+^1 \cup E_-^1} (n = v_j) \right) \vee p_{i+1}(v_j) = p_i(v_j) \right] \\ \wedge & \bigwedge_{(p,2) \in \mathcal{P} \setminus \mathcal{P}^0} \bigwedge_{j_1,j_2=1}^{v} \left[\bigvee_{\substack{(n_1,p,n_2) \\ \in E_+^2 \cup E_-^2}} (n_1 = v_{j_1} \wedge n_2 = v_{j_2}) \vee p_{i+1}(v_{j_1}, v_{j_2}) = p_i(v_{j_1}, v_{j_2}) \right] \end{aligned}$$

This is encoded in SMTLib as the function `rule_application`. *See Code Listing B.5 in Appendix B, page 261 for the full listing of the template.*

With this code template we express that, for every node tuple and every predicate that is potentially changed by the rule, the node tuple is either matched to a changed edge in the rule, in which case the change is reflected in the result graph, or it is not, in which case the value of the predicate at that node tuple must remain constant. Predicate values of predicates that are not touched by the rule are to remain constant by default.

Lemma 18 (Correctness of Application Step Encoding). *Let G_i be a (modified) graph, let S_i be a (modified) shape such that $G_i \sqsubseteq_{f_i'} S_i$, let $P_i = (L, p, R)$ be a rule, let S_i^m be a materialization of P_i out of S_i for some match m, and let $G_i \sqsubseteq_{f_i} S_i^m$. Let G_{i+1} be the result of applying P_i to G_i at $f_i^{-1}\big\downarrow_L$. Let further Sc be an SMT script containing some form of encoding of G_i, S_i^m, and f_i, as well as instances of Code Templates 26 and 27. Then, any model of Sc contains the logical structure of G_{i+1}.*

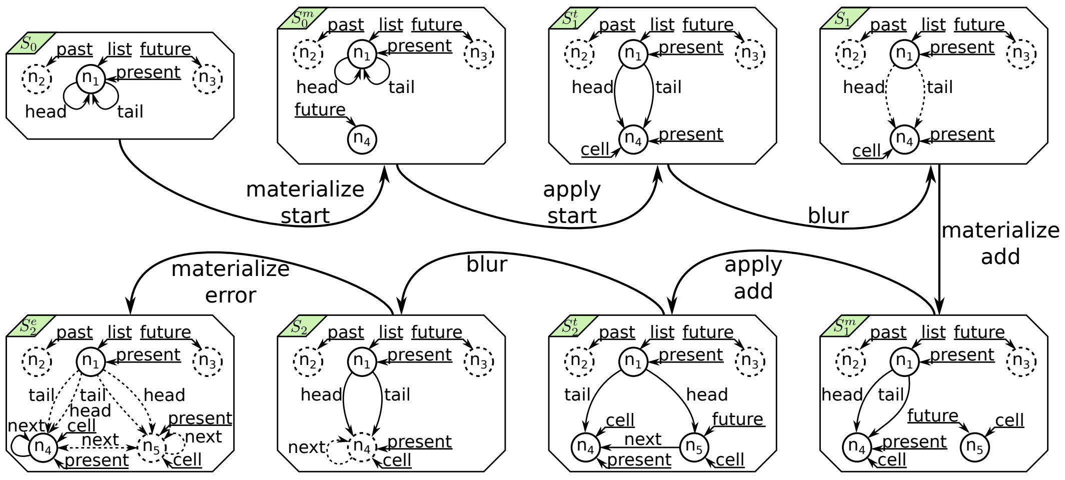


Figure 4.10: An abstract error trace (Apply start, then add, then materialize error)

Proof. See Appendix A, page 247. □

With this, we have completed our encoding of rule application, and thus of corresponding concrete traces in general.

For an example of an application of the rule application encoding in the context of a trace encoding, consider the trace shown in Fig. 4.10. In this abstract trace, a list is created using start, then a node is added using add. At this point, the two cell nodes are merged to maintain canonical abstraction. This then allows for the materialization of the error pattern, which consists of a single cell node with a *next*-self-edge.

Clearly, this trace is spurious, since at this point there is no way in the original GTS that a *next*-self-edge could exist. We can now use our corresponding concrete trace encoding to prove this.

The encoding can also give us much more information about this trace, ranging from where in the trace the abstraction strays from the original system to the kind of constraints that could be used to reign it back in. These uses of the encoding are explored in Chapter 6.

4.7 Encoding of the Emptiness of a Shape

Besides the verification and analysis of abstract error traces, there is another application for SMT encodings in the context of our approach that was alluded to in the description of constraints in Sec. 3.6. For an unconstrained shape there is always at least one graph that is embedded into it. This can easily be seen by remembering that any graph is a shape and that adding ½-edges and summary nodes only ever *adds* possible embeddings, but never subtracts them. The introduction of constraints, however, changes this, since they explicitly exclude certain embeddings from the overall set of possible embeddings into the shape they are attached to.

While for individual constraints this usually does not exclude *all* of the possible embeddings, multiple constraints together can easily collide, rendering the shape useless. This situation is called an *empty shape*.

Definition 74 (Empty Shape). *Let* (S, Λ) *be a constrained shape.* (S, Λ) *is said to be an* empty *or* infeasible shape *iff* $\mathcal{G}(S, \Lambda) = \emptyset$*. If* (S, Λ) *is not empty, it is said to be* feasible.

Determining whether or not a constrained shape is empty (or feasible) is non-trivial. Evaluating the constraints in Λ on S can yield **0**, of course, but that only considers one constraint at a time, and only detects trivially unusable shapes. The much more common, and much more difficult to detect case occurs when two or more constraints contradict each other, e.g. if $\Lambda = \{(c_1, m_1), (c_2, m_2)\}$ and $\mathcal{G}(S, \{(c_1, m_1)\}) \cap \mathcal{G}(S, \{(c_2, m_2)\}) = \emptyset$.

In Sec. 4.4, we have already developed an encoding that can decide whether a given graph G can be embedded into a given constrained shape (S, Λ). If the shape is empty, this encoding would surely yield UNSAT. Unfortunately, that would only mean that G, specifically, cannot be embedded into (S, Λ), not that no graph can possibly be embedded. In Sec. 4.5 we have seen that the encoding of the graph can be made external to the embedding encoding, and that the graph used need not be fully defined. But even if all constraints on the graph are lifted, the encoding would still always define a fixed number of nodes to make up the graph to be embedded.

Clearly, the embedding encoding, as defined so far, cannot be used to check whether *any* graph of *any* size can be embedded into a given shape. This is because in order to express that, we would need to quantify over the `Node` sort, which we cannot do in the encodings defined so far, since they are all written for the QF_UF logic. In this section, we will develop a quantified version of the embedding encoding which will enable us to detect any empty shapes that are constructed over the course of any algorithm exploring the state space of an STS.

The purpose of the new encoding will be to give a simple yes or no answer to the question "$\mathcal{G}(S, \Lambda) \neq \emptyset$?", and will thus be called the "shape feasibility encoding". Therefore, the explicit encoding of the embedding, which was previously used in the embedding encoding in order to be able to glean the actual embedding from the model created, is no longer necessary. We will thus base the shape feasibility encoding on the graph encoding described in Sec. 4.3, rather than the embedding encoding. This meshes well with the intuition that a concrete graph is the simplest form of a shape, and trivially feasible. From this we will derive successively more complex encodings that incorporate more and more features of a shape, beginning with ½-edges, continuing with summary nodes and ending with constraints.

As a reminder, the graph encoding consists of a prelude which establishes the edge labels as unary and binary predicates on a `Node` sort, as well as `Node` constants representing the node set, and continues to assert the pairwise distinctiveness of the nodes (`distinctNodes`, see CoT. 3), as well as the unary and binary edges of the graph (`unaryG` and `binaryG`, see CoT. 5 and 6). This encoding yields exactly one model, representing the logical structure of the graph itself.

In order to introduce the first relaxation, ½-edges, to this encoding, we use the same reasoning as we used for the embedding encoding: an edge that is neither required to exist nor not to exist by the edge encoding can be interpreted by the solver in both ways – mimicking the meaning of a ½-edge. We thus use a modified version of `unaryG` to encode unary ½ edges.

Code Template 28 (Unary Edges in Feasibility Encoding). *Let $S = (U, \mathcal{P}, \iota)$ be a shape, with $|U| = k$ and $\mathcal{P}$ consisting of l unary predicate symbols and $n - l$ binary predicate symbols. The unary edges in S that are not incident to any summary nodes are encoded using the following code template.*

```
(define-fun unaryG () Bool (and
  B(s_1 v_1) ··· B(s_l v_1)
  ⋮
  B(s_1 v_k) ··· B(s_l v_k)
))
```

where each `B(`s_i v_j`)` *stands for*

`(not (`s_i v_j`))`	*if* $\iota_S(s_i)(v_j) = \mathbf{0}$
`(`s_i v_j`)`	*if* $\iota_S(s_i)(v_j) = \mathbf{1}$
`(true)`	*if* $\iota(s_i)(v_j) = ½ \vee \iota(sm)(v_j) = ½$

It thus encodes the first-order formula

$$\bigwedge_{\substack{\iota(s)(v)=\mathbf{1}\\ \iota(sm)(v)=\mathbf{0}}} s(v) \wedge \bigwedge_{\substack{\iota(s)(v)=\mathbf{0}\\ \iota(sm)(v)=\mathbf{0}}} \neg s(v)$$

Analogously, a modified version of `binaryG` takes care of binary ½-edges.

Code Template 29 (Binary Edges in Feasibility Encoding). *Let $S = (U, \mathcal{P}, \iota)$ be a shape, with $U = \{v_1, \ldots, v_k\}$ and $\mathcal{P}$ consisting of l unary predicate symbols and $n - l$*

binary predicate symbols. The binary edges in S that are not incident to any summary nodes are encoded using the following code template.

```
(define-fun unaryG () Bool (and
B($s_{l+1}$ $v_1$ $v_1$) ⋯ B($s_n$ $v_1$ $v_1$)
B($s_{l+1}$ $v_1$ $v_2$) ⋯ B($s_n$ $v_1$ $v_2$)
⋮
B($s_{l+1}$ $v_{k-1}$ $v_k$) ⋯ B($s_n$ $v_{k-1}$ $v_k$)
B($s_{l+1}$ $v_k$ $v_k$) ⋯ B($s_n$ $v_k$ $v_k$)
))
```

where each `B(`s_i v_{j_1} v_{j_2}`)` *stands for*

`(not (`s_i v_{j_1} v_{j_2}`))`	$\mathit{ifi}_S(s_i)(v_{j_1}, v_{j_2}) = \mathbf{0}$
`(`s_i v_{j_1} v_{j_2}`)`	$\mathit{ifi}_S(s_i)(v_{j_1}, v_{j_2}) = \mathbf{1}$
`(true)`	$\mathit{ifi}(s_i)(v_j) = \tfrac{1}{2} \vee \iota(sm)(v_{j_1}) = \tfrac{1}{2} \vee \iota(sm)(v_{j_2}) = \tfrac{1}{2}$

It thus encodes the first-order formula

$$\bigwedge_{\substack{\iota(s)(v_1,v_2)=\mathbf{1}\\ \iota(sm)(v_1,v_2)=\mathbf{0}}} s(v_1, v_2) \wedge \bigwedge_{\substack{\iota(s)(v_1,v_2)=\mathbf{0}\\ \iota(sm)(v_1,v_2)=\mathbf{0}}} \neg s(v_1, v_2)$$

These encodings are correct by the same arguments that were used in Lemma 9 and Lemma 11. An encoding using these modified versions of `unaryG` and `binaryG` is thus capable of expressing the presence of ½-edges. This enables us to encode the (trivially given) feasibility of shapes without summary nodes and constraints – each such shape has 2^e embedded graphs, where e is the number of ½-edges in the shape.

Thus far, the modifications have been fairly straightforward. However, the addition of summary nodes to the picture requires more effort on our part. As stated above, we do not want to explicitly encode the embedding function and therefore, using the embedding function to incorporate the semantics of summary nodes is not an option.

Instead we will encode summary nodes not, like regular nodes, as `Node` constants, but rather as unary `Node` predicates. A node is embedded into a summary node n, if the associated predicate N evaluates to $\mathbf{1}$ for that node. Every such predicate will therefore act as the characteristic function (see Def. 67) of the set of nodes embedded into that particular summary node.

In order to accomplish this, we will need to change the prelude. Let $S = (U, \mathcal{P}, \iota)$ be an unconstrained shape, and let $\{v_1, \ldots, v_m\} =: S \subseteq U$ be the set of summary nodes. For each summary node v_i, a unary `Node` predicate `_Vi` is constructed. This gives us the prelude for the feasibility encoding, based on the prelude of the graph encoding.

Code Template 30 (Prelude for Feasibility Encoding). *Let $S = (U, \mathcal{P}, \iota)$ be an unconstrained shape, let $|U| = v$ and let $\mathcal{P}$ consist of l unary predicates and $n - l$ binary predicates. Let further $\{v_1, \ldots, v_m\} =: S \subseteq U$ be the set of summary nodes in S, and let $N := U \setminus S = \{u_1, \ldots, u_k\}$ be the set of non-summary nodes in S, where $|N| = k$. Then the SMT prelude for the feasibility encoding of S is given by the following code template.*

```
(set-logic UF)
(declare-sort Node 0)
⋮
; non-summary nodes and predicates as in Sec. 4.3
⋮
; declare each summary node as a unary node predicate
(declare-fun _V1 (Node) Bool)
⋮
(declare-fun _Vm (Node) Bool)
```

The addition of the node predicates for the summary nodes necessitates a change to the encoding that ensures that the nodes are distinct. Instead of simply requiring that nodes are pairwise distinct, we will now require in addition that every member of the `Node` sort is either equal to exactly one of the `Node` constants, or belongs to exactly one of the sets defined by the summary node predicates. This essentially translates to one of the conditions a mapping must meet in order to be considered an embedding function – each node in the graph must be embedded into exactly one of the nodes in the shape, and only summary nodes are allowed to have more than one node embedded into them. The following code template is the result.

Code Template 31 (Exclusivity Encoding). *Let $S = (U, \mathcal{P}, \iota)$ be an unconstrained shape, and let $|U| = v$. Let further $\{v_1, \ldots, v_m\} =: \Sigma \subseteq U$ be the set of summary nodes in S, and let $N := U \setminus S = \{u_1, \ldots, u_k\}$ be the set of non-summary nodes in S, where $|N| = k$. Then the encoding of the exclusivity of the* `Node` *sort for S is given by the following first-order formula*

$$\bigwedge_{i=1}^{k} \bigwedge_{j=i+1}^{k} \neg (v_i = v_j) \wedge \bigwedge_{i=1}^{k} \bigwedge_{j=1}^{m} \neg \_Vj\,(v_i) \wedge \forall x : \bigoplus_{i=1}^{k} (x = v_i) \oplus \bigoplus_{j=1}^{m} \_Vj\,(x)$$

which is encoded into SMTLib in s straightforward fashion as the function `exclusivity`. *For the complete Code Template, please see Code Listing B.6, page 262.*

The correctness of this encoding is expressed in the following lemma.

Lemma 19 (Correctness of exclusivity encoding). *Let S be an unconstrained shape, and let Sc be a script containing the corresponding instances of Code Templates 30, 28, 29, and 31. Let M be a model of Sc using a* `Node` *universe U, and let f be the mapping $f : U \to U_S$ given by*

$$f(x) := \begin{cases} v_i & \textit{if } \iota(=)(v_i, x) \\ v_j & \textit{if } \iota(\_Vj)(x) \end{cases}$$

Then f is a total function that obeys the node abstraction property, i.e.

$$\left|f^{-1}(x)\right| > 1 \geq \iota(sm)(x)$$

Proof. Since the distinctiveness of all node constants is encoded directly in `exclusiv-ity`, and the term

$$\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{m} \neg\_Vj(v_i) \qquad\qquad \text{implies}$$
$$\iota_M(\_Vj)(x) = \mathbf{0}$$

for all summary nodes v_j and node constants x, the injectivity of f on the set of non-summary nodes is guaranteed. Further, if there existed a universe element x such that it is neither equal to one of the node constants nor belongs to one of the sets defined by the $\_Vj$ predicates, this would mean

$$\exists x : \neg \left(\bigoplus_{i=1}^{k} (x = v_i) \oplus \bigoplus_{j=1}^{m} \_Vj(x) \right)$$

which is literally the negation of the last part of the `exclusivity` formula. Thus, since m is a model, every universe element is either equal to one of the node constants of satisfies one of the summary node predicates, which makes f total. □

This, of course only takes care of the existence of the summary nodes and their semantics with regard to the (implicitly given) embedding function. What remains to be defined is how edges that are incident to summary nodes are encoded, since our encoding for edges so far (Code Templates 28 and 29) explicitly exclude such edges. As before, unary and binary predicates are handled separately, with binary predicates being further subdivided into two separate code templates.

Unary edges that are incident to summary nodes are handled similarly to unary edges on concrete nodes. The only difference is that now, rather than naming particular nodes for which we define predicate values, we universally quantify over all nodes that belong to a given summary node.

Code Template 32 (Unary Edges on Summary Nodes). *Let $S = (U, \mathcal{P}, \iota)$ be a shape with $\mathcal{P}$ consisting of l unary predicate symbols and $n - l$ binary predicate symbols. Let further $\{v_1, \ldots, v_m\} =: \Sigma \subseteq U$ be the set of summary nodes in S. Then the encoding of the unary edges on the summary nodes of S is given by the following code template.*

```
(define-fun unaryS () Bool (forall ((x (Node))) (and
  ; first the positive then the negative assertions
  (=> (_V1 x) (and B(s1 x) ... B(sl x)))
  ⋮
  (=> (_Vm x) (and B(s1 x) ... B(sl x)))
)))
```

where, as in previous code templates, for each $1 \leq j \leq m$, each Block `B(`s_i x`)` *stands for*

`(not (`s_i x`))`	*if* $\iota(s_i)(v_j) = \mathbf{0}$
`(`s_i x`)`	*if* $\iota(s_i)(v_j) = \mathbf{1}$
`(true)`	*if* $\iota(s_i)(v_j) = ½$

This encodes the following first-order formula:

$$\forall x : \left[\bigwedge_{i=1}^{m} \left(\_Vi(x) \rightarrow \left(\bigwedge_{\substack{(s,1)\\ \iota(s)(v_i)=\mathbf{0}}} \neg s(x) \wedge \bigwedge_{\substack{(s,1)\\ \iota(s)(v_i)=\mathbf{1}}} s(x) \right) \right) \right]$$

Note that, in the case of summary nodes, definite knowledge about edges is in general much rarer than for definite nodes, so these formulas are usually rather short, since all literals concerning ½-edges are optimized out. We can now move on to the binary edges of S.

As mentioned above, this part of the shape encoding is split into two sub-parts. First, we will deal with binary relations between summary nodes and definite nodes, and vice-versa. Then, we handle binary relations between pairs of summary nodes. The following encoding defines the first part.

Code Template 33 (Binary Edges Between Concrete and Summary Nodes). *Let $S = (U, \mathcal{P}, \iota)$ be a shape with $\mathcal{P}$ consisting of l unary predicate symbols and $n - l$ binary predicate symbols. Let further $\{v_1, \ldots, v_m\} =: \Sigma \subseteq U$ be the set of summary nodes in S. Then the encoding of the binary edges between summary nodes and concrete nodes in S is given by the following code template.*

```
(define-fun binaryS1 () Bool (forall ((x (Node))) (and
  (=> (_V1 x) (and B(s_{l+1} x v_1) ··· B(s_{l+1} x v_k)
                   B(s_{l+1} v_1 x) ··· B(s_{l+1} v_k x)))
  ⋮
  (=> (_Vm x) (and B(s_{l+1} x v_1) ··· B(s_{l+1} x v_k)
                   B(s_{l+1} v_1 x) ··· B(s_{l+1} v_k x)))
)))
```

where, again, each `B(x y z)` *block is either* `(x y z)`, `(not (x y z))`, *or* `(true)`, *depending on whether the corresponding edge exists in the shape or not. The corresponding first-order formula is as follows*

$$\forall x : \left[\bigwedge_{i=1}^{m} \left(\_Vi\,(x) \rightarrow \left(\bigwedge_{\substack{(s,2),v \\ \iota(s)(v_i,v)=\mathbf{0}}} \neg s\,(x,v) \wedge \bigwedge_{\substack{(s,2),v \\ \iota(s)(v_i,v)=\mathbf{1}}} s\,(x,v) \right.\right.\right.$$
$$\left.\left.\left. \wedge \bigwedge_{\substack{(s,2),v \\ \iota(s)(v,v_i)=\mathbf{0}}} \neg s\,(v,x) \wedge \bigwedge_{\substack{(s,2),v \\ \iota(s)(v,v_i)=\mathbf{1}}} s\,(v,x) \right)\right)\right]$$

The second part, dealing exclusively with edges between summary nodes, is encoded as follows.

Code Template 34 (Binary Edges Between Summary Nodes). *Let $S = (U, \mathcal{P}, \iota)$ be a shape with $\mathcal{P}$ consisting of l unary predicate symbols and $n - l$ binary predicate symbols. Let further $\{v_1, \ldots, v_m\} =: \Sigma \subseteq U$ be the set of summary nodes in S. Then the encoding of the binary edges between summary nodes in S is given by the following code template.*

```
(define-fun binaryS2 () Bool (forall ((x (Node)) (y (Node))) (and
  (=> (and (_V1 x) (_V1 y)) (and B(s_{l+1} x y) ··· (and B(s_n x y))))
  ...
  (=> (and (_V1 x) (_Vm y)) (and B(s_{l+1} x y) ··· (and B(s_n x y))))
  (=> (and (_V2 x) (_V1 y)) (and B(s_{l+1} x y) ··· (and B(s_n x y))))
  ...
  (=> (and (_Vm x) (_Vm y)) (and B(s_{l+1} x y) ··· (and B(s_n x y))))
)))
```

with the `B` *blocks defined as before, representing the following first-order function:*

$$\forall x, y : \left[\bigwedge_{i,j=1}^{m} \left((\_Vi\,(x) \wedge \_Vj\,(y)) \rightarrow \left(\bigwedge_{\substack{(s,2) \\ \iota(s)(v_i,v_j)=\mathbf{0}}} \neg s\,(x,y) \wedge \bigwedge_{\substack{(s,2) \\ \iota(s)(v_i,v_j)=\mathbf{1}}} s\,(x,y) \right) \right) \right]$$

All three of these code templates follow the same basic idea. Every node that is embedded into a given summary node must mirror all of its definite properties. The set of nodes that are embedded into a given summary node v_i is identified by the node predicate `_Vi`. Thus, every node for which a given node predicate is true, must exhibit all of the definite properties associated with the corresponding summary node. This is expressed in the following lemma.

Lemma 20 (Correctness of Edge Encodings for Summary nodes). *Let S be an unconstrained shape, and let Sc be a script containing the corresponding instances of Code Templates 30, 28, 29, 31, 32, 33, and 34. Let M be a model of Sc using a* `Node` *universe U, and let f be the mapping $f : U \rightarrow U_S$ given by*

$$f\,(x) := \begin{cases} v_i & \textit{if}\ \iota\,(=)\,(v_i, x) \\ v_j & \textit{if}\ \iota\,(\_Vj)\,(x) \end{cases}$$

Then f is a total function satisfying the node abstraction property and the edge abstraction property, i.e.

$$\iota_M\,(p)\,(v_1, \ldots, v_k) \sqsubseteq \iota_S\,(p)\,(f\,(v_1)\,, \ldots, f\,(v_k))$$

for all predicates p and all node tuples $v_1, \ldots, v_k$.

Proof. By Lemma 19, f is a total function that satisfies the node abstraction property. In analogy to the proof of Lemma 11, we now assume that there is a binary predicate $(p, 2)$ and a node tuple (v_1, v_2) such that

$$½ \neq \iota_S\,(p)\,(f\,(v_1)\,, f\,(v_2)) \neq \iota_M\,(p)\,(v_1, v_2)$$

holds. Keep in mind that as an interpretation in boolean logic, ι_M cannot produce the truth value ½. We distinguish four cases.

$\mathbf{f\,(v_1)\,, f\,(v_2) \notin \Sigma}$ **:** In this case, by the correctness of the Formulae `unaryG` and `binaryG` (see CoT. 28 and CoT. 29), the assumption is already contradicted.

$\mathbf{f\,(v_1) \in \Sigma \wedge f\,(v_2) \notin \Sigma}$ **:** In this case, let $v_i := f\,(v_1)$. M must satisfy the following term present in the formula binaryS1 (see CoT. 33):

$$\forall x : \left[\left(\_Vi\left(x\right) \rightarrow \left(\bigwedge_{\substack{(s,2),v\\ \iota(s)(v_i,v)=\mathbf{0}}} \neg s\left(x,v\right) \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v_i,v)=\mathbf{1}}} s\left(x,v\right) \right.\right.\right.$$
$$\left.\left.\left. \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v,v_i)=\mathbf{0}}} \neg s\left(v,x\right) \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v,v_i)=\mathbf{1}}} s\left(v,x\right)\right)\right)\right]$$

Thus, every universe element, including v_2, must satisfy

$$\left(\bigwedge_{\substack{(s,2),v\\ \iota(s)(v_i,v)=\mathbf{0}}} \neg s\left(x,v\right) \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v_i,v)=\mathbf{1}}} s\left(x,v\right) \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v,v_i)=\mathbf{0}}} \neg s\left(v,x\right) \wedge \bigwedge_{\substack{(s,2),v\\ \iota(s)(v,v_i)=\mathbf{1}}} s\left(v,x\right)\right)$$

Since $v_2 \notin \Sigma$, we know that there exists a node constant v_j such that $v_2 = v_j$, and thus the above formula enforces that

$$\iota_S\left(p\right)\left(f\left(v_1\right), f\left(v_2\right)\right) = \iota_M\left(p\right)\left(v_1, v_2\right)$$

holds.

$\mathbf{f}\left(\mathbf{v_2}\right) \in \mathbf{\Sigma} \wedge \mathbf{f}\left(\mathbf{v_1}\right) \notin \mathbf{\Sigma}$: analogous to the above.

$\mathbf{f}\left(\mathbf{v_1}\right), \mathbf{f}\left(\mathbf{v_2}\right) \in \mathbf{\Sigma}$: Let $v_i = f\left(v_1\right) \wedge v_j = f\left(v_2\right)$. Then the formula binaryS2 (see CoT. 34) contains the term

$$\forall x, y : \left[\left(\left(\_Vi\left(x\right) \wedge \_Vj\left(y\right)\right) \rightarrow \left(\bigwedge_{\substack{(s,2)\\ \iota(s)(v_i,v_j)=\mathbf{0}}} \neg s\left(x,y\right) \wedge \bigwedge_{\substack{(s,2)\\ \iota(s)(v_i,v_j)=\mathbf{1}}} s\left(x,y\right)\right)\right)\right]$$

and thus, any pair of universe elements belonging to $\_Vi$ and $\_Vj$, respectively, has to satisfy

$$\left(\bigwedge_{\substack{(s,2)\\ \iota(s)(v_i,v_j)=\mathbf{0}}} \neg s\left(x,y\right) \wedge \bigwedge_{\substack{(s,2)\\ \iota(s)(v_i,v_j)=\mathbf{1}}} s\left(x,y\right)\right)$$

and thus mirror all definite edges present in the shape, implying

$$\iota_S(p)(f(v_1), f(v_2)) = \iota_M(p)(v_1, v_2)$$

for the particular pair v_1, v_2.

Therefore, in all cases, the assumption is refuted and thus the edge abstraction property is upheld by any model of Sc. □

Thus far we have made sure that the definite nodes in the graph are distinct, that every node in existence is associated either with a definite node, or with a summary node, and that every node in existence mirrors the definite properties of the node it is associated with. This is an almost complete encoding of the existence of an embedding function from an undetermined graph of undetermined size (the solver will attempt to find a universe that satisfies the model) into the given shape. The only remaining property of an embedding is surjectivity.

We need to make sure that the predicates we introduced for the summary nodes will be interpreted as $\mathbf{1}$ for at least one node each. A similar statement is not needed for the definite nodes, since their existence is already assured by the corresponding node constants. The surjectivity property is encoded using the following code template.

Code Template 35 (Surjectivity Property). *Let $S = (U, \mathcal{P}, \iota)$ be a shape and let $\{v_1, \ldots, v_m\} =: S \subseteq U$ be the set of summary nodes in S. Then the encoding for the surjectivity property of the implicit embedding in the encoding for the emptiness of S is given by the formula*

$$\exists x_1, \ldots, x_m : \left(\bigwedge_{i=1}^{m} \_Vi\,(x_i) \right)$$

which is encoded into SMTLib in a straightforward fashion as the function `surjectivity`*, using an appropriate function definition.*

The correctness argument for this formula is very straightforward, and will thus be omitted here. This assures that each summary node has at least one actual node assigned to it. The encoding, as it is defined so far, can already be used to check for the emptiness of unconstrained shapes. However, as we have seen at the beginning, that is not useful, since unconstrained shapes never can be empty.

Lemma 21 (Models for Unconstrained Shape Encoding). *Let $S = (U, \mathcal{P}, \iota)$ be a shape, and let Sc be a script containing one corresponding instance each of Code Templates 30, 28, 29, 31, 32, 33, 34, and 35. Let Sc further contain assertions for these code instances. Then Sc is satisfiable, and every model for Sc contains the logical structure of a graph*

G that can be embedded into S. There are furthermore exactly as many models for Sc as there are graphs that can be embedded into S.

Proof. This follows from the correctness of the individual formulas that make up the unconstrained shape encoding, and specifically from Lemma 19 and Lemma 20. □

In order to obtain an actually useful encoding for the emptiness of a shape, we need to add support for constraints. According to Def. 60, graphs are embedded into constrained shapes if and only if they satisfy all shape constraints that, under the embedding, are equivalent to the shape constraints on the shape. Thus, for our encoding we need a way to express this condition, i.e. we need a way to encode the formulas of the shape constraints, and a way to go through all possible interpretations of their assignments under all possible embeddings.

Fortunately, our constraints are already formulated as first-order formulas, which makes the encoding simple. As in the Embedding Encoding in Sec. 4.4, we can encode the constraint formulas using Code Template 19. Thereby, every distinct formula α_i that is used by the constraints in Λ is encoded as an SMT function `constraint`$_{\alpha_i}$, with each free variable being converted into a function parameter.

In order to to then take all possible assignments into account, we first start with the simplest possible case. Let (α_i, m_i) be a shape constraint, and let M_i be the image of m_i. In the simplest possible case, all nodes in M_i are concrete nodes. This is the simplest case, because all embedded graphs must embed exactly one node into each of the nodes in M_i, creating only one possible interpretation for m_i. Thus, in that case, we would only need to assert the truth of `constraint`$_{\alpha_i}$ with the nodes in M_i as parameters.

The next issue is how to deal with summary nodes. Let $s \in M_i$ be a summary node, and let all nodes in $M_i \setminus \{s\}$ be concrete nodes. Let U be the universe of the shape, $\bar{V}_i$ be the set of variables assigned to nodes other than s by m_i, and let $\bar{\alpha}_i$ be α_i, with every variable in $\bar{V}_i$ replaced by its image under m_i. Thus, $(\bar{\alpha}_i, m_i{\downarrow_s})$ is a shape constraint with a single free variable v, assigned to the summary node s. Now, this constraint must hold for any embedded graph, regardless of which node is embedded into s. Therefore, it must hold for all nodes for which the corresponding node predicate `_Vs` holds, i.e. we get

$$\forall x : \_Vs\,(x) \rightarrow (\alpha_i)_{[v \mapsto x]}$$

This basic idea is extended to arbitrary numbers of summary nodes in the following code template.

Code Template 36 (Constraint Encoding). *Let (α, m) be a constraint, and let* `constraint`$_\alpha$ *be the encoding of its formula according to Code Template 19. Let further*

$$\{y_1, \ldots, y_{a'}, y_{a'+1}, \ldots, y_a\}$$

be the set of free variables of α, sorted in such a way that the first a' variables are assigned to summary nodes by m and the remaining variables are assigned to concrete nodes. Let $v_i = m(y_i)$ be the assigned nodes for these variables. Then, the encoding of the adherence of the potentially embedded graph encoded by the feasibility encoding to the constraint (α, m) is given by the following code template.

```
(define-fun constraint_α_m () Bool (forall
  ((x1 (Node)) ... (xa' (Node)))
  (=>
    (and (_V1 x1) ... (_Va' x1))
    (constraintα x1 ··· xa' va'+1 ··· va)
  )
))
```

This effectively encodes the first-order formula

$$\forall x_1, \ldots, x_{a'} : \bigwedge_{i=1}^{a'} \_Vi\,(x_i) \rightarrow \left(\alpha_{\left[y_1 \mapsto x_1, \ldots, y_{a'} \mapsto x_{a'}, y_{a'+1} \mapsto m\left(y_{a'+1}\right), \ldots, y_a \mapsto m(y_a)\right]}\right)$$

Since this first-order formula for constraint satisfaction follows directly from Def. 60 (using the implied embedding), the correctness of this formula is trivially given. Having defined all constituent parts, we can now encode the feasibility of any given constrained shape as an SMT problem. The following code template creates the required assertion and pulls all the required parts together.

Code Template 37 (Shape Feasibility Assertion). *Let $(S = (U, \mathcal{P}, \iota), \Lambda)$ be a constrained shape, with Λ having the form*

$$\Lambda = \{(\alpha_1, m_1), \ldots, (\alpha_1, m_{a_1}), (\alpha_2, m_1), \ldots, (\alpha_c, m_{a_c})\}$$

Given an SMT script containing corresponding instances of the Code Templates 30, 28, 29, 32, 33, 34, 35, 19, and 36, the feasibility of (S, Λ) can be asserted by

```
  (assert (and
    exclusivity
    unaryG binaryG
    unaryS binaryS1 binaryS2
    surjectivity
    constraint_$\alpha_1$_$m_1$ ⋯ constraint_$\alpha_1$_$m_{a_1}$
    ⋮
    constraint_$\alpha_c$_$m_1$ ⋯ constraint_$\alpha_c$_$m_{a_c}$
  ))
```

The constraints encoding restricts the meaning of the actual shape, and can in the extreme case render the resulting script unsatisfiable, as the following Lemma shows.

Lemma 22 (Correctness of Constrained Shape Feasibility Encoding). *Let $(S = (U, \mathcal{P}, \iota), \Lambda)$ be a constrained shape, with Λ having the form*

$$\Lambda = \{(\alpha_1, m_1), \ldots, (\alpha_1, m_{a_1}), (\alpha_2, m_1), \ldots, (\alpha_c, m_{a_c})\}$$

Let Sc be a script containing a corresponding instance each Code Templates 37, as well as all its prerequisite Code Templates. Then there are exactly as many models for Sc as there are graphs that can be embedded into S. Specifically, Sc is unsatisfiable if and only if $\mathcal{G}(S, \Lambda) = \emptyset$.

Proof. Direct corollary of the previous lemmata and definitions. □

As an example for this encoding, consider the shape shown in Fig. 4.11. It represents (among other things) an arbitrarily large cycle of *cell* nodes, comprised of *next* edges. In order to exclude some of the interpretations that are not cycles of *cell* nodes and *next* edges, a number of constraints have been added to the shape. In total, there are 4 constraints, made from a total of 3 different formulas:

$$\Lambda = \{(\alpha, [v \mapsto n_1]), (\alpha, [v \mapsto n_2]), (\beta, [v \mapsto n_3]), (\gamma, [v \mapsto n_3, w \mapsto n_3])\} \quad \text{where}$$
$$\alpha := \exists x, y : (\mathit{next}(x, v) \wedge \mathit{next}(v, y) \wedge \neg (x = v) \wedge \neg (y = v))$$
$$\beta := \forall x : (\mathit{next}(v, v) \wedge \neg (x = v) \rightarrow \neg \mathit{next}(x, v) \wedge \neg \mathit{next}(v, x))$$
$$\gamma := (v = w) \wedge \mathit{next}(v, v)$$

Here, the formula α expresses that there is an incoming *next* edge, as well as an outgoing *next* edge incident to its free variable. This constraint is attached to n_1 and n_2. The edge $(n_1, \mathit{next}, n_2)$ satisfies half of the condition α for both nodes, leaving us with the constraint that there must be at least one incoming edge to n_1, and at least one outgoing edge to n_2.

The formula β expresses that if the node it is attached to has a self-edge labeled *next*, then there must be no other *next*-edges attached to it. This constraint is attached to the node n_3. Since n_3 is a summary node, this constraint must therefore hold for all nodes embedded into n_3, i.e. it holds for all nodes in the cycle that are not n_1 or n_2.

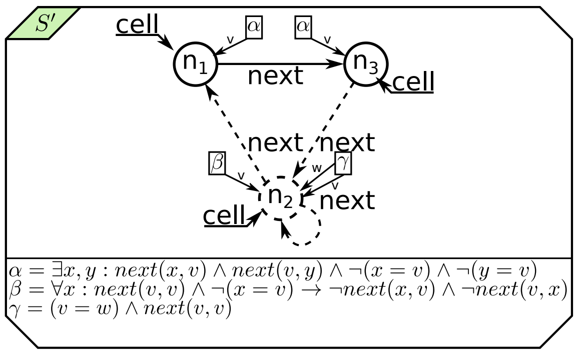

S'
cell
α
v
α
v
n1
next
n3
cell
next
next
β
v
w
γ
v
n2
next
cell
$\alpha = \exists x, y : next(x, v) \wedge next(v, y) \wedge \neg(x = v) \wedge \neg(y = v)$
$\beta = \forall x : next(v, v) \wedge \neg(x = v) \rightarrow \neg next(x, v) \wedge \neg next(v, x)$
$\gamma = (v = w) \wedge next(v, v)$

Figure 4.11: A non-trivially infeasible shape

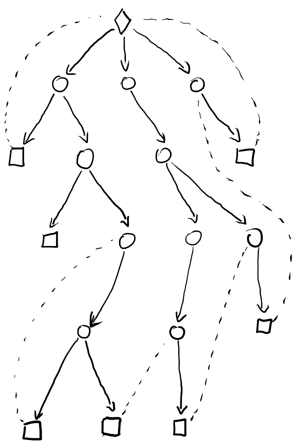
Finally, the formula γ expresses that its two free variables are identical, and that the node they are attached to has a self-edge. This constraint is also attached to n_3. Due to the definition of the embedding conditions for constraints, this has the effect that only one node can be embedded into n_3, since as soon as there are more than one, the condition that they are all identical no longer holds. Thus, this constraint expresses that n_3 is a single node with a self-edge labeled *next*.

Each of these constraints individually are easily satisfiable by the shape. Even any two of them together can still be satisfied. Taken together however, these three constraints make the shape infeasible. The constraint $(\gamma, [v \mapsto n_3, w \mapsto n_3])$ determines that n_3 is a single node with an *next*-self-edge. The constraint $(\beta, [v \mapsto n_3])$ on the same node tells us that since n_3 has a self-edge labeled *next*, it can have no other *next*-edges incident to it. And finally, this contradicts the α-constraints, which required an incoming *next*-edge into n_1, and an outgoing *next*-edge from n_2, neither of which is still possible since the only possible source (or target, respectively) of these edges was n_3. Therefore, we have $\mathcal{G}(S) = \emptyset$ for this shape S.

Just by reasoning on the shape itself it would be fairly difficult to deduce this. Using the SMT encoding defined in this section, however, we can query any SMT solver capable of processing the UF theory with our encoding and receive the answer almost instantly. Listing B.7 shows the encoding for the shape shown in Fig. 4.11. Note that the encoding is small, positively tiny compared to the trace encoding we defined before. This is due, among other factors, to the fact that only definite knowledge needs to be encoded, the embedding does not need to be encoded explicitly, and quantifiers allow us to take many shortcuts unavailable to us before.

Of course, the runtime is not always so favorable. As stated earlier, our use of first-order logic makes all tests that have to do with constraints in particular, or quantifiers in general (as in this encoding) undecidable. However, the vast majority of shapes can be checked for feasibility in a handful of seconds using the methods presented in this sections, and as we will see in Chapters 5 and 6, state space construction and refinement can be organized such that undecidable instances (or rather, timeouts) cannot lead to an unsound analysis, but rather to a slightly less efficient one. One can of course manually construct cases that lead to timeouts for the solver, but for the average use case, our approach seems to be feasible.

In conclusion, we have now constructed SMT encodings that let us check for the existence of concrete traces corresponding to abstract error traces, and for the feasibility of any given shape, constrained or not. Together with the definitions of shape transformation systems obtained in the previous chapter, this gives us all the tools we need to construct the verification algorithm that is the centerpiece of this thesis. We will properly define this algorithm in the following two chapter, and examine an example implementation of it, as well as example applications of that implementation, in Chapter 7.



5 Lazy State Space Construction

The provenance of the constraints used to refine shapes has not been specified so far. Clearly, requiring all such constraints to be created by hand at the start of the analysis is not feasible. On the other hand, the construction of the abstract state space as defined so far does not lend itself well to deriving constraints during construction. Thus, in this chapter and the next one, we will attempt to create another approach at state space construction and semi-automatic abstraction refinement to address this issue. We begin by looking to the literature for guidance on automatic abstraction refinement.

All verification schemes that use abstraction and aim to be automatic rather than interactive have to incorporate some form of this process. For this reason, plenty of research has been performed for a large variety of verification techniques and use cases, yielding several general templates for methods to refine the abstraction. One such template is called CEGAR[51] – counterexample guided abstraction refinement.

The central idea of CEGAR is that a spurious abstract error trace is a failure in the abstraction, and that that failure is most specifically expressed in the error trace itself. After all, if the algorithm that created the trace concluded that it is an error, even though it is not (a *spurious error*), then at some point along that trace it must have left the actual state space of the original problem, and considered states that are not actually part of the system. Finding this point, and figuring out what information the algorithm was lacking there, is the key to refining the abstraction just enough to rule out that specific spurious trace. This allows a verification system to automatically refine its abstraction, but do so as little as possible, in order to obtain just enough precision to verify the system, and not more.

Since its introduction[51], the CEGAR technique has found widespread use across the field of model checking and is now considered to be a state-of-the-art technique. CEGAR is used in many successful model checking tools, such as SLAM[18], which has found use in verifying Windows device drivers, the more recent BLAST[32] model checker, which has an even wider range of target applications, or a number of other tools, such as Moped[101], SatAbs[54], and F-Soft[98] While most of these tools perform some kind of predicate abstraction[81], CEGAR has been and is being applied to many other abstraction approaches as well[8, 121, 156]. Even though the target use cases and the abstraction formalisms may differ, the fundamental concept of CEGAR remains the same.

A basic abstraction system implementing CEGAR works as follows. First, it chooses an initial abstraction. This abstraction is usually chosen to be as coarse as possible, unless domain knowledge is available that delivers refinement information that is guaranteed to be useful. Then, the state space of the system is constructed with respect to that abstraction. This state space is searched for potential errors, and such errors, when found, are tested against the original, unabstracted system. If no error is found, the system is deemed safe. If a non-spurious error is found, the system is proven to be faulty. If, however, only spurious errors are found, these errors are used to refine the abstraction. Then, the state space with respect to the new abstraction is constructed and the process continues. Note that CEGAR itself does not specify how the abstraction should be refined, and especially does not force either a lazy or a non-lazy approach – SLAM recomputes the state space upon adoption of a new abstraction (non-lazy) while BLAST only refines the part of the abstraction that exhibited the spuriou serror (lazy).

In the previous chapters, we have defined algorithms to construct the state space of an STS, as well as SMT encodings that can determine whether or not a given abstract trace through such a state space is spurious or not. Thus, it seems that we have all that we need to implement the CEGAR paradigm and complete the feature set of our approach. However, applying CEGAR directly to shape transformation systems, as we have defined them so far, would be difficult.

The main reason for this is that the CEGAR approach requires that, when an error is found, a finite number of counterexamples can be identified. In the domain where the idea of CEGAR originated, the static analysis and verification of sequential programs, this is always the case. Each counterexample corresponds to an execution path through the program, represented by its control-flow graph. Any construction of an abstract state space is grounded in this control flow graph and can rely on it as a structure that does not change with the abstraction.

In our approach, on the other hand, such a structure that could ground the analysis is conspicuously absent. As in all basic graph transformation systems, we assume that graph rules are applied continuously and atomically, whenever they match some part of the graph. Translated into the language of sequential programs, this would mean that a GTS with n rules is like a system with n parallel processes, each one consisting of an infinite loop executing only one check (is the rule applicable?) and one action (apply the rule), if the check succeeds. As far as control flow goes, this is not helpful. One could of course argue that the control-flow graph of a graph transformation system is its transition system. That, however, mixes control flow and knowledge about the state of the system, meaning that in infinite-state GTS, the transition system is of course infinite, making it equally useless as a fixture for the abstraction system.

Thus, we are stuck with the following situation. Our abstract state space construction algorithm (see Chapter. 3) creates the abstract reach set of an STS related to a given GTS by the canonical abstraction (see Def. 50). It then uses an existential abstraction (see Def. 54) to create a (finite) shape transition system to correspond to the (possibly) infinite graph transition system. As detailed in Def. 53, in doing so, it uses the covering relation between shapes to discard shapes that are covered by other shapes. This is done in order to limit the size of the state space. When an error shape is found, i.e. a shape that potentially matches the error pattern, we are faced with the task of creating a counterexample for it, i.e. an *abstract error trace* from the initial shape to the error shape. However, since we have potentially discarded many shapes during the construction of the reach set, the construction of such a trace can be difficult, if not impossible.

This can go so far that the initial shape and the error shape are subsumed by the same shape in the reach set of the STS, leaving us with absolutely no trace information to go on. Furthermore, even when an actual trace from the initial shape to the potential error exists, it will, due to the nature of an existential abstraction, almost always encounter self-transitions on the way, where certain rules applied to certain shapes lead back to those shapes themselves. In such cases, there are an infinite number of possible traces that could be gained for a given potential error, and it is not clear that any particular one of them, such as, e.g., the shortest one, is the best one to use.

Clearly, if we are to employ CEGAR, a different approach to state space construction and refinement is required.This new approach will be a relatively direct implementation of the principle of *lazy abstraction*, or *lazy state space construction* (LSSC), already employed by several CEGAR-approaches, such as BLAST[32] and the approach by McMillan[113]. In LSSC, the state space is explored by unfolding it like a tree, rooted in the initial state. Every rule applicable to a shape represented by a node in that tree creates a number of child nodes, which will in turn have their own child nodes, and so on. Where in our previous approach of a transition system, states that are found to be covered by other states are used to fold the transition system back in on itself, in order to obtain a concise representation, a covered node in LSSC is simply marked as such and left unexplored.

When an error node is found in the unfolding of the tree, there is only one possible path from the initial node to it. This path is taken to be the abstract error trace, and used to obtain a refinement of the abstraction, such as a new set of shape constraints. Crucially, this abstraction refinement is then not applied to the entire tree, but only along the path to the error, leading to a representation of the state space in which different parts of it are represented with different levels of abstraction. This solves both our issues that were raised above – paths are now guaranteed to exist and be unambiguous.

Over the course of this chapter, we will first examine a representative example of the LSSC paradigm in Section 5.1. Having obtained a more detailed idea of how LSSC works, we apply this knowledge in Section 5.2 to the definition of the *shape transition tree*, the basic data structure that will support the state space we construct. Based on this definition, we then create the algorithm that constructs this data structure in Section 5.3. There we will focus on what actions are performed on the tree and how they work together in creating the outer construction loop of the LSSC approach. The description of the inner refinement loop is left to Chapter 6.

5.1 The Principle of Lazy State Space Construction

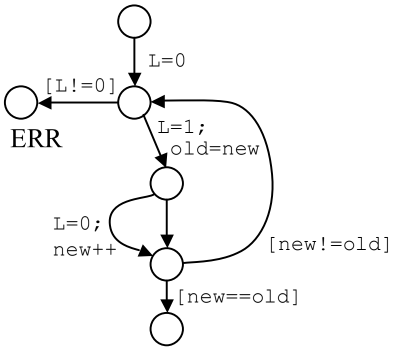
The observation that originally motivated lazy state space construction or, more specifically, lazy abstraction refinement, is that in many cases, an error is localized to a very small portion of the state space, while other, large sections of the state space are error-free. In verification approaches that rely on a successively refined, global abstraction, such cases incur huge, unnecessary costs as they dutifully explore safe sections of the state space in great detail. Thus, much could be gained by creating a verification approach that does not refine its abstraction globally, but locally – only on those parts of the state space that cannot be verified using a coarser, and therefore less costly, abstraction.

In this chapter, we will apply this basic idea to the exploration of the state space of an STS, and to the refinement of its abstraction. In doing this, we will follow the template given by Kenneth McMillan in his 2006 paper on lazy abstraction[113]. This paper contains a very clear description of lazy state space construction, which is helped by the fact that it completely omits abstract post operators, to speed up construction. While this work was done on sequential programs and is most directly comparable to predicate abstraction[81] of such programs, the basic principles employed in it can be transfered to our use case as well.

McMillan's paper focused on programs written in a C-like, sequential programming language. A fragment of such a program is shown in Fig 5.1. In McMillan's approach,

```
do {
  lock();
  old = new;
  if (*) {
    unlock();
    new++;
  }
} while (new != old)
```

(a) C-like source code



(b) Control-Flow Graph

Figure 5.1: The program fragment used by McMillan[113] to showcase his lazy abstraction approach

the program to be verified is first converted into a *control-flow graph* (CFG). In this graph, each program location corresponds to a node, and each transition between program locations corresponds to a directed edge, connecting these program locations. A transition representing the execution of a statement is labeled with the effect of that statement on the variables of the program. A conditional branch, on the other hand, is labeled with the condition that has to be met so that that particular branch can be taken. In the particular example shown in Fig. 5.1, the initial location describes the state of the system before entering the piece of code shown in the figure, and the error location is implicitly reached when `lock()` is called while $L \neq 0$ holds.

Once the control-flow graph has been constructed, a set of starting abstraction predicates is chosen. These are the predicates that form the abstract domain that will be used to describe (and over-approximate) the state of the program. In the abstraction, the state of the program will be described only by the truth values of these predicates. Usually, the analysis begins with an empty set of predicates, meaning that the abstraction is unable to express any knowledge about the program state whatsoever[*].

Once a predicate set has been chosen, an *abstract reachability tree* (ART) is constructed from the CFG using those predicates. The constructed tree is rooted in the initial state of the program, corresponding to the starting location in the CFG. This state is labeled with the knowledge about the initial state of the variables in the program, expressed using the predicates from the predicate set chosen at the beginning. Since one usually starts with an empty predicate set, this label will usually be `true`.

Now, each so far unexplored node is explored in the following way, until no unexplored nodes remain.

Each edge in the CFG that goes out from the CFG location corresponding to the

[*]excluding, of course, the program location itself

node is converted into an outgoing edge to a new node in the ART. If the edge is a statement, it is labeled with the effect of that statement. In the lazy abstraction paradigm, this effect must be over-approximated, meaning that the following state must be labeled with a formula over the abstraction predicates that is implied by the actual effect of the statement and the formula of the source node. This can be achieved by computing the abstract-post operator, yielding the strongest formula expressible in the abstraction predicates that satisfies the above condition. But, since computing the abstract post-operator usually incurs high costs, this can also be avoided by just labeling the successor state `true`. McMillan uses this latter version in his paper[113].

If the edge, on the other hand, is a conditional edge, then it is checked whether the condition conflicts with the label of the parent node. If it does, the edge is discarded. If it does not, the edge is retained and the label for the new node, as above, will be implied by the source node formula and the condition, e.g., `true`.

Once a new node has been constructed, it is checked against existing nodes to see if any of them *cover* the new node, i.e. represent the same program location and have a label that is implied by the label of the new node. If the node is covered, it represents a *special case* of a node that has already been explored, meaning that exploring it would be redundant. Such nodes are marked as covered and not explored. This process of exploring and potentially covering nodes continues until an error location is reached in the ART.

Figure 5.2 shows an ART for the program in Fig. 5.1. The initial state represents the state of the program before the loop is entered. It is labeled `true`, since we have no knowledge of the program state at that point. As the CFG shows, we know that `L` is initialized to zero at the beginning of the program. This is represented by the first edge in the ART. However, since no abstraction predicates were used, we cannot utilize this information, i.e. we have no choice but to label the new node with `true` as well. This node represents the call to `lock()` at the beginning of the loop. Now, we have no way to derive whether `L` is zero or non-zero on the call to `lock()`, forcing us to consider the possibility that it might be non-zero. This leads to the existence of a corresponding edge and a node corresponding to the error location in the CFG.

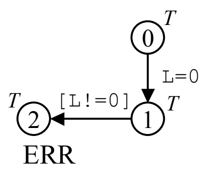
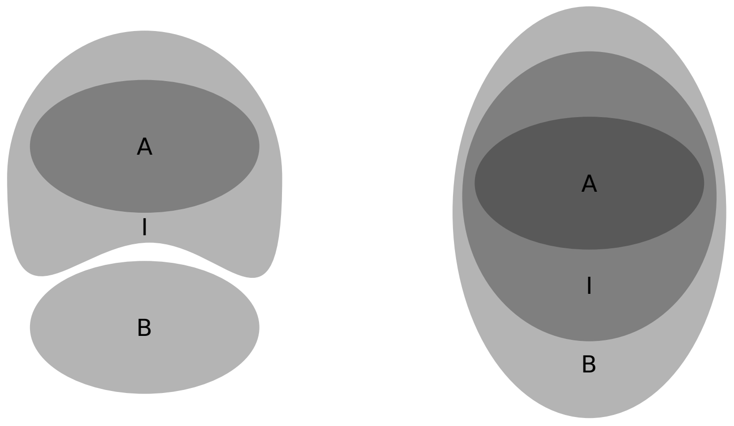


Figure 5.2: ART of Fig. 5.1 with no abstraction predicates[113]

At this point, counterexample-guided abstraction refinement is employed by examining the path `0`→`1`→`2` from the initial node to the error location and deducing what predicates might be necessary to exclude this path (given that the path is spurious). In order to understand how this works, and how our encoding from Chap. 4 figures into all this, we must first take a quick detour and explore the concept of *Craig Interpolation*.

Craig Interpolation is a mathematical concept that was first made feasible in 1957 by



(a) I implied by A, inconsistent with B **(b)** I implied by A, implying B

logician William Craig in his paper "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory"[60]. In essence, Craig proved that a formula satisfying the conditions specified by the following definition always exists:

Definition 75 (Interpolant). *Let $\lambda(\varphi)$ be the set of non-logical symbols, such as predicates, contained in a first-order logical formula φ. Let further A and B be inconsistent first-order formulas, i.e. $\sigma_S(A \land B) = \emptyset$ (see Def. 32). Then there exists a logical formula I such that*

- *$A \models I$, i.e. all models of A are models of I (A* implies *I),*
- *$I \land B \models \mathbf{0}$, i.e. I and B are inconsistent, and*
- *$\lambda(I) \subseteq \lambda(A) \cap \lambda(B)$, i.e. I uses only the common symbols of A and B.*

Such a formula is known as an interpolant *for A and B.*

Intuitively speaking this means that whenever there are two inconsistent first-order formulas A and B, there exists a generalization I of A, which is still inconsistent with B, but only uses symbols common to both A and B. Figure 5.3a shows a sketch of the relationship of the model sets of the three formulas in this scenario. Simple rearrangement of these symbols gives us also an alternative way of thinking about interpolation, namely that, given formulas A and B, such that $A \models B$, there is a formula I over their common symbols such that $A \models I \models B$. This interpretation is shown in Fig. 5.3b. The fact that the formula I in both of these interpretations lies "between" A and B motivated the term "interpolation".

Another way of thinking of an interpolant, as defined in Def. 75 and depicted in Fig. 5.3a, is that it represents the essence of *why* A and B are inconsistent, condensed into a form compatible with both formulas. As we will see, McMillan made use of this interpretation of interpolants in his paper[113] in an ingenious way.

Returning to the example shown in Fig. 5.2, the counterexample `0`$\rightarrow$`1`$\rightarrow$`2` is first converted into a so-called *path formula*, consisting of a formula I describing the initial state of the program (`true` here), and a sequence of transition formulas $T_{i,i+1}$ describing the relationship between the program variables of step i in the path and the variables of step $i+1$. In order to express this in logic, the program variables are *indexed*, so that every state along the path has its own set of program variables. This is called *static single assignment* (SSA). So, in our case, we would end up with a formula

$$I \wedge T_{0,1} \wedge T_{1,2}$$

describing an execution path in the original program, where the first call to `lock()` fails since $L \neq 0$. Because L is initialized to 0 in the program (visible in the control flow graph in Fig. 5.1), this path formula is infeasible.

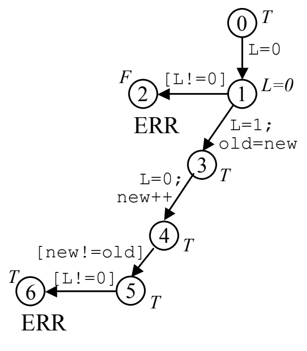
Since we now have an infeasible conjunction of subformulas, by inserting cuts into the conjunction we can construct from this two instances of the situation described in Def. 75.

$$\underbrace{I}_{A} \wedge \underbrace{T_{0,1} \wedge T_{1,2}}_{B} \tag{5.1}$$

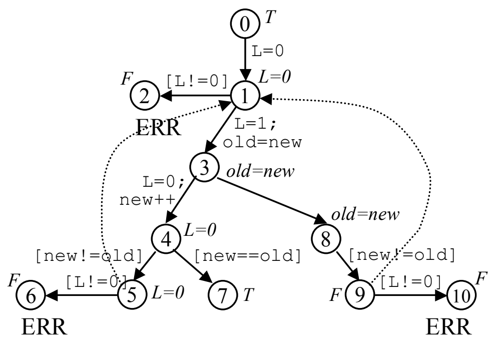
$$\underbrace{I \wedge T_{0,1}}_{A} \wedge \underbrace{T_{1,2}}_{B} \tag{5.2}$$

Thus, two interpolants result from this. I_1, the interpolant for cut (5.1) is a formula that is implied by the initial state, but inconsistent with the application of the two transitions. Thus, intuitively, it represents the answer to the question "what information does the initial state contain that makes the following two transitions infeasible?". The interpolant for cut (5.2), I_2, can be interpreted similarly. It is implied by $I \wedge T_{0,1}$, but inconsistent with $T_{1,2}$, the transition that goes from `lock()` to the error state. Intuitively, I_2 is thus the answer to the question "What information does the state at `lock()` contain that would prevent the following transition to the error state". In other words, each interpolant is essentially an answer to the question what information the abstraction was missing at that point that could have been helpful to prevent the creation of the spurious error.

In this particular case, we could for example obtain the interpolants $I_1 = \mathbf{1}$, since the initial state contains no information, and $I_2 = (L = 0)$, since that is implied by the transition $T_{0,1}$, but inconsistent with $T_{1,2}$. Through these interpolants, we have



(a) Second spurious error found



(b) ART complete and error-free

additional information that we can add to the states along the path. We add `true` to the initial state (to no effect), $L = 0$ to state `1`, and `false` to state `2`, since the path leading to it was infeasible (this is essentially the interpolant for $A := I \wedge T_{0,1} \wedge T_{1,2}$ and $B := \mathbf{1}$). Once this is done, the error state is no longer reachable in the ART as constructed so far and we can continue exploring it.

It should be noted that refining nodes in such a manner can mean that nodes that previously covered other nodes may now cease to do so, necessitating the further exploration of the now uncovered nodes. Once the new information has been added to the nodes, and any now-faulty covering information removed, exploration of the ART continues until another error is found or no nodes remain unexplored. In the particular example used by McMillan, the refinement process repeats once more when the error shown in Fig. 5.4a is found. Note that, again, all labels on the newly constructed nodes are `true`. An interpolation of the path formula corresponding to that new error path leads via further exploration to the ART shown in Fig. 5.4b. This ART contains another spurious counterexample, the refinement of which does not lead to any new unexplored nodes. After this has been concluded, the system is declared safe since no nodes are left unexplored and no error node is reachable.

In this thesis, we will define an adaptation of the above approach for shape transformation systems. Much of the basic principle can be used directly, without many modifications. In order to do this, we need the following:

- A tree-like data structure that represents the state space of an STS, as an analogue to the ART. (→Sec. 5.2)
- A notion of when any node in this data structure covers another node. (→Sec. 5.2)
- A construction algorithm, capable of creating that data structure, starting with the initial shape. (→Sec. 5.3)

- A procedure which derives from a path through this tree an abstract error trace, and its corresponding encoding (see Chapter 4). (→Chap. 6)
- A strategy of where in this encoding to place the interpolation cuts (→Chap. 6)
- An algorithm that takes resulting interpolants and uses them to refine a path through the tree. (→Chap. 6)

We will gather these remaining parts over the course of the next sections, going into more detail about McMillan's lazy abstraction approach as needed.

A Short Survey of Lazy Abstraction

Before we move on to define our own approach, we will take a short detour to give the reader a quick overview over the topic of lazy abstraction. The idea of lazy abstraction is of course older than the paper by McMillan that we used to illustrate the process. While the term "lazy abstraction" was used before 2002 to informally refer to application-specific abstraction schemes that were "lazy" in a sense specific to the respective approach, for the purposes of this overview, the seminal paper for lazy abstraction is the one written by Henzinger, Jhala, Majumdar, and Sutre[91]. In this paper the basic principles of lazy abstraction in the context of software verification were described, and demonstrated with the introduction of the BLAST tool, which was later described in its own paper[32].

The concept of lazy abstraction offered the attractive prospect of removing a lot of redundant work from state space construction and was thus quickly adopted by a number of tools, such as KRATOS[48], SAFARI[9], Wolverine[106], and IC3[47]. It was then applied, often via those tools, to industrially relevant problems, such as device drivers[90], SystemC[48], and Hardware models[151].

The combination of craig interpolation with lazy abstraction (and CEGAR)[89] marked an important step forward for lazy abstraction since it provided a widely usable, relatively simple way to provide the local abstraction refinements needed for lazy abstraction. Use of interpolation was soon widespread in lazy abstraction and expanded the reach of lazy abstraction further with the introduction lazy abstraction for richer languages, e.g. by including support for arrays[10] and full Presburger Arithmetic[93], or by using it for termination proofs[55]. Lazy abstraction has also been combined with shape analysis[33] (see Chapter 3, page 26) and partial order reduction[49] to further reduce the state space that still needs to be explored.

Clearly, lazy abstraction is a very successful abstraction technique that is still evolving[†]. This thesis represents a new entry into this field by applying this successful paradigm to the verification of abstract graph transformation systems.

[†]for example, through generalizations into even more flexible abstraction methods[38]

We now begin thepresentation of our own approach by defining shape transition trees, the data structure that will serve as our analogue to the abstract reachability tree.

5.2 Shape Transition Trees

In McMillan's lazy abstraction approach, the control flow graph of the program is *unfolded* to obtain the so-called *abstract reachability tree*. Each node in this tree represents a state of the program, made up of a location in the control flow graph and a formula restricting the state of the programs variables at that location. A single path reaches from the initial state to every such node, clearly identifying the execution path in the original program that corresponds to it.

If we are to apply the lazy state space construction paradigm to shape transformation systems, we need a tree-like data structure that would similarly represent an "unfolding" of an STS. Thus, we need to find a way to represent states of the system, i.e. nodes in the tree, in such a way that they can be refined without jeopardizing the structure of the tree.

In our scenario, we will assume that we are tasked with verifying the absence of an error pattern E from an infinite-state GTS $\mathcal{G} = (I, \mathcal{R})$, using an abstraction scheme $\mathcal{A}$. We make the following observation of how state space construction for the corresponding STS $\mathcal{S} = (\mathcal{A}(I), \mathcal{R})$ proceeds with regard to a single, non-initial shape S_i. First, a shape transformation creates a new shape S_i^t. This new shape is the direct result of a rule application to a materialized shape and thus not on the same level of abstraction as its source shape. Then, the abstraction scheme $\mathcal{A}$ is applied to the shape, yielding a new, "normalized" shape S_i. For this shape, potential matches for all the rules in $\mathcal{R}$ are computed, yielding a set of materializations $S_i^{m_0}, \ldots, S_i^{m_k}$, from which new shapes are derived via rule application.

Thus, for each actual shape produced by the STS, there is an "incoming shape", and a potentially large but finite and constant number of "outgoing shapes", or materializations. This observation leads us to our definition of the tree nodes we will use: each such node consists of a *central shape* representing the state described by the node, an single *entry point*, describing how the shape was constructed, and a number of *materialization points*, providing anchors for transitions to subsequent states. In order to keep the definitions more concise from this section forward, we first introduce a shorthand notation for constrained shapes.

Definition 76 (Shorthand for Constrained Shapes). *Let* (S, Λ) *be a constrained shape. As a shorthand, this can be written as* $\underline{S}$*. The constraint set itself can be referred to using* $\underline{\Lambda_S}$*.*

We will now go through the mathematical definitions necessary to establish the full concept of an *STT node*. As the central shape is just a shape as defined in Def. 56, we begin with the entry point.

Definition 77 (Entry Point). *Let $\underline{S}, \underline{S'}$ be shapes, such that $\mathcal{G}(\underline{S}) \subseteq \mathcal{G}(\underline{S'})$. Then $\underline{S}$ is called an* entry point *of the shape $\underline{S'}$ (and vice-versa).*

Thus, an entry point to a shape $\underline{S}$ (and later an STT node) is just another shape that, covers some (but not necessarily all) of the same graphs as $\underline{S}$. In the context of an STT, entry points are either unabstracted results of rule applications, or the initial graph of a GTS. We move on to materialization points.

Definition 78 (Materialization Point). *Let $\underline{S}$ be a shape, let P be a graph rule, and let m be a match such that $[\![\varphi_P]\!]_S^m \geq ½$. Let further $\underline{S^m}$ be the corresponding materialization, and f^m be its embedding into $\underline{S}$. The tuple (S^m, f^m) is called a* materialization point *of S. For any fixed rule set $\mathcal{R}$, the* materialization set *$\mathcal{M}_{\mathcal{R}}(S)$ is the set of all possible materialization points of S. When the rule set is clear from context, it may be omitted.*

A materialization point for a shape $\underline{S}$ represents a single materialization of some rule out of $\underline{S}$. The rule itself, the match, as well as the constraints derived for the materialization are not part of the definition, since they can be easily derived from the materialization and the embedding. For every shape, there can only ever be a finite amount of different materialization points, given that the rule set is finite.

Lemma 23 (Materialization Sets are Finite). *Let $\underline{S}$ be a shape, and let $\mathcal{R}$ be a finite set of graph rules. Then we have $|\mathcal{M}_{\mathcal{R}}(\underline{S})| \in \mathbb{N}_0$.*

Proof. It is well known that, for any two sets A, B, the number of different mappings from B to A is given by $|A|^{|B|} \in \mathbb{N}_0$ Thus, for any rule P, the number of mappings from L_P into $\underline{S}$ is finite, and therefore the number of potential matches m is also finite. As per Def. 46, the number of materializations for any given potential match is given by 2^n, where n is the number of summary nodes in the match. Therefore, every rule $P \in \mathcal{R}$ has a finite number of materializations for any given shape $\underline{S}$. Since there is one materialization point for every possible materialization of each rule in $\mathcal{R}$, we can thus conclude that $|\mathcal{M}_{\mathcal{R}}(\underline{S})| \in \mathbb{N}$. □

Having defined entry and materialization points, we can move on to defining the STT nodes themselves.

Definition 79 (STT Node). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system. Let further $\mathcal{A}$ be an abstraction scheme and $\mathcal{S} = (\mathcal{A}(I), \mathcal{R})$ be the shape transformation system for $\mathcal{G}$ w.r.t. $\mathcal{A}$. An* STT node *in this context is a triple $\left(\underline{S^{in}}, \underline{S}, M\right)$, where $\mathcal{G}\left(\underline{S^{in}}\right) \cap \mathcal{G}(\underline{S}) \neq \emptyset$ and $M = \mathcal{M}_{\mathcal{R}}(\underline{S})$.*

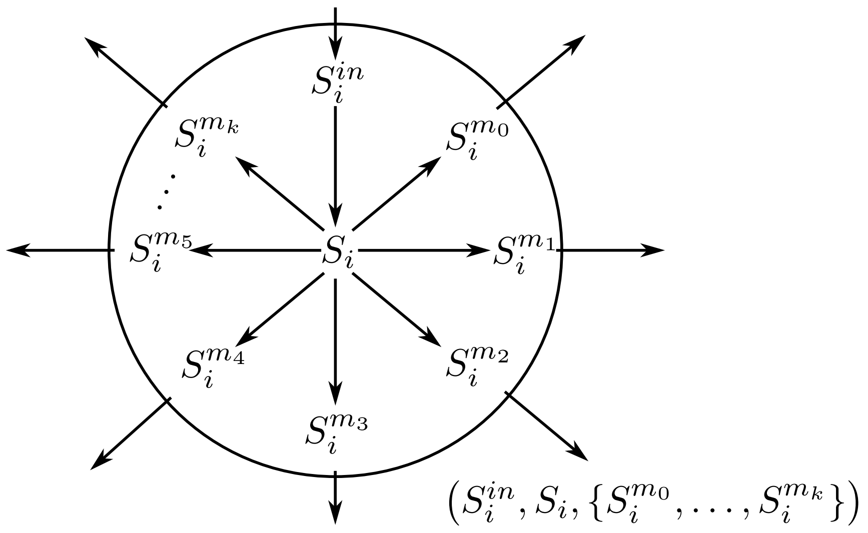


Figure 5.5: Schematic of an STT node

Figure 5.5 shows a schematic of an STT node $\left(\underline{S_i^t}, \underline{S_i}, \mathcal{M}\left(\underline{S_i}\right)\right)$. As the definition above states, each STT node has one entry point, here designated S_i^t. The intention of this entry point is to serve as the anchor point for the single incoming edge for the node. For inner nodes and leaves, this shape will be the unabstracted result shape of the transformation represented by the incoming edge. For the root, having no incoming edge, this shape will be the initial graph of the GTS.

The central shape $\underline{S}$ serves as a label for the entire node. All other shapes in the node are embedded into it, cutting out sections of the graph set covered by $\underline{S}$.

The materialization points, here labeled just with the materialized shapes $\underline{S_i^{m_0}, \ldots, S_i^{m_k}}$ themselves, serve as anchor points for outgoing edges. They can also represent a potential error in the state, if they contain a materialization of the error pattern rather than the left hand side of a rule.

In a tree made up of these nodes, edges connect STT nodes. Each such edge is associated with exactly one materialization point and one entry point. They are labeled with the rule that is applied in the corresponding transformation and the materialization point that serves as the basis for that transformation. These transition edges give the STT its tree structure.

In analogy to the abstract reachability trees used by McMillan, STTs use a second kind of edge to indicate a coverage relationship. Covering edges go from covered nodes to the nodes that cover them. The thusly defined covering relationship means that the state represented by the covered node is already completely represented by the covering node. Further exploration beyond that node is thus unnecessary. In order to properly define the STT, we first state a basic definition of a generic tree.

Definition 80 (Tree). *A graph* $G = \left(N, E^1, E^2\right)$ *over a label set* $L = (U_L, B_L)$ *is a* tree *if and only if*

$$\exists! \rho \in N : \forall \rho \neq n \in N : \exists! \pi \in {E^2}^* : \pi = (\rho, l_1, x) \cdots (y, l_k, n) \qquad , k \in \mathbb{N}$$

The unique node ρ *is then called the* root *of* G*, and for each node* $n \in N$ *the unique path* π *from* ρ *to* n *is called the* root path $\pi(n)$*. A tree defines a partial order on its nodes, called the* ancestor relation $\preceq$*. It is defined by*

$$\preceq := \left\{(n, m) \in N \times N \mid \exists \pi \in {E^2}^* : \pi = (n, l, x) \cdots \left(y, l', m\right)\right\}$$

This structure forms the basis of the STT. Similar to graphs, there are various ways to define substructures of trees that can be removed, modified, or added. For the algorithms manipulating STTs that we will define in Section 5.3 and Chapter 6, we will specifically need two concepts: *subtrees*, and *tree pruning*.

Definition 81 (Subtree, Tree Pruning). *Let* $T = \left(N, E^1, E^2\right)$ *be a tree over a label set* L*. Let* $n \in N$*. Then the tree* $T_n = \left(N_n, E^1_n, E^2_n\right)$ *over* L *is the* subtree rooted in n*, if and only if*

$$\begin{aligned} N_n &= \{m \in N \mid n \preceq m\} \\ E^1_n &= \left\{(u, m) \in E^1 \mid m \in N_n\right\} \\ E^2_n &= \left\{(m_1, b, m_2) \in E^2 \mid m_1, m_2 \in N_n\right\} \end{aligned}$$

By removing the subtree for a node n *from a tree* T*, it can be* pruned at n*, leading to the tree* $T' = T \setminus T_n$*, defined by*

$$\begin{aligned} N' &= N \setminus N_n \cup \{n\} \\ {E^1}' &= E^1 \setminus E^1_n \cup \left\{(u, n) \in E^1\right\} \\ {E^2}' &= E^2 \setminus E^2_n \end{aligned}$$

Having established these basic structures, we can now move on to extend them for the purpose of representing unfoldings of STSs. The following definition states the basic structure of an STT.

Definition 82 (Shape Transition Tree). *Let* $\mathcal{G} = (I, \mathcal{R})$ *be a GTS, let* $\mathcal{A}$ *be an abstraction scheme, and let* $\mathcal{S} = (\mathcal{A}(I), \mathcal{R})$ *be the corresponding STS. Let* $\mathcal{N}$ *be the set of all possible STT nodes using* $\mathcal{R}$*, let* $\mathcal{M}$ *be the set of all materialization points and* $\mathcal{E}$ *the set of all possible embeddings.. A* Shape Transition Tree *is a tuple* $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$*, where*

- *N is an arbitrary set,*
- *$E^1 \subseteq \mathcal{N} \times N$, $E^2 \subseteq N \times (\mathcal{R} \times \mathcal{M}) \times N$ are sets such that $\left(N, E^1, E^2\right)$ is a tree over the label set $L = (\mathcal{N}, \mathcal{R} \times \mathcal{M})$, rooted in $\rho_{\mathcal{T}} \in N$,*
- *$\forall n \in N : \left|\{x, n\} \in E^1\right| = 1$, i.e. all nodes have exactly one label given by a unary edge,*
- *$C \subseteq N \times N$ is a set such that $\forall\,(n, m) \in C : n \not\leq m$,*
- *$l_c : C \to \mathcal{E}$ is a labeling function.*

E^2 is the set of transition edges*, while C is the set of* covering edges*. For any transition edge e, r_e denotes the rule applied, while $\left(\underline{S_e^m}, f_e^m\right)$ denotes the materialization point used, i.e. $l_e\,(e) = \left(r_e, \left(\underline{S_e^m}, f_e^m\right)\right)$. For any node n, $\underline{S_n}$ denotes the central shape of its associated STT node (via the unary edge), while $\underline{S_n^{in}}$ denotes its incoming shape, i.e. $\left(\left(\underline{S_n^{in}}, \underline{S_n}, \mathcal{M}\left(\underline{S_n}\right)\right), n\right) \in E^1$. The set of* covered nodes*, i.e. the set of nodes with* outgoing *covering edges, is denoted $N_C = \{n \in N \mid \exists m \in N : (n, m) \in C\}$.*

This definition establishes the "syntactical" structure of a shape transition tree. Subtrees and tree pruning extend to this data structure in the obvious way, by restricting C and l_c to the respective part of the tree. For ease of presentation, we will occasionally conflate the concepts of a node in the STT and an STT node, e.g. by stating that transition edges connect STT nodes even though this is technically not the case.

So far, our definition allows many STTs that are not in any way valid unfoldings of an STS. This property of an STT is defined separately.

Definition 83 (STT Validity). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS. An STT $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ is called a* valid unfolding of *or a* valid STT for *$\mathcal{G}$, if and only if*

1. $\left(\left(I, \underline{S_0}, \mathcal{M}\left(\underline{S_0}\right)\right), \rho_{\mathcal{T}}\right) \in E^1$, *with $I \in \mathcal{G}\left(\underline{S_0}\right)$*
   i.e. the root is the abstraction of I,
2. $\forall e = (n, m) \in E : \left(\underline{S_e^m}, f_e^m\right) \in \mathcal{M}\left(\underline{S_n}\right) \wedge \underline{S_n} \xrightarrow{r_e, m, \underline{S_e^m}} \underline{S_m^{in}}$,
   i.e. transition edges represent shape transitions,
3. $\forall n \in N \setminus N_C : \forall\left(\underline{S^m}, f^m\right) \in \mathcal{M}\left(\underline{S_n}\right) :$
   $$\left[\mathcal{G}\left(\underline{S^m}\right) = \emptyset \vee \exists e \in E^2 : \left(\left(\underline{S^m}, f^m\right) = \left(\underline{S_e^m}, f_e^m\right)\right)\right],$$
   i.e. all materialization points of non-covered nodes are either invalid or explored,
4. $\forall n, m \in N : (\exists\,(n, m) \in C) \to \underline{S_n} \sqsubseteq_{l_c(n,m)} \underline{S_m} \wedge \neg\exists\,(n, m') \in E^2$,
   i.e. covering edges imply an embedding and no covered nodes are explored, and

5. $\forall n : \underline{S_n^{in}} \sqsubseteq \underline{S_n}$,
   i.e. the incoming shape is covered by the central shape.

Note that, since error materializations cannot be explored (because patterns cannot be applied), condition 3 implies that there are no valid materializations of the error anywhere in $\mathcal{T}$. Note also, that in a valid STT, the set of leaf nodes is formed by N_C, the set of covered nodes, and by nodes with materialization sets that contain no valid shapes.

A *valid* STT represents a complete unfolding of a (possibly infinite-state) graph transformation system. This idea of an STT "representing a GTS" motivates the following definition, which provides the answer to the question which particular graphs from a GTS are represented by any given shape in the STT. The concept of a *local graph set* is a slight extension of the concept of corresponding concrete traces from Sec. 3.5 and Sec. 4.5.

Definition 84 (Local Graph Set). *Let $\mathcal{G} = (I, \mathcal{R})$ be a GTS, let $\mathcal{A}$ be an abstraction scheme, and let $\mathcal{S} = \left(\underline{S_0} := \mathcal{A}(I), \mathcal{R}\right)$ be the corresponding STS. Let $\underline{S_0} \xrightarrow{P_1, m_1, S^{m_1}} \cdots \xrightarrow{P_k, m_k, S^{m_k}} \underline{S_k}$ be an abstract trace of length k. Then, for every $i \in \{1, \ldots, k\}$, let the* local graph set Γ_i *be defined by*

$$\Gamma_0 := \{I\}$$
$$\Gamma_i := \left\{G \mid \exists G' \in \Gamma_{i-1} : \left[G' \sqsubseteq_f \underline{S^{m_i}} \wedge \exists m' : \left(m = f \circ m' \wedge G' \xrightarrow{P_i, m'} G\right)\right]\right\}$$

The local graph set defines for every prefix of an abstract trace the set of graphs that can be produced by the original GTS by following the constraints of the abstract trace. In other words, for every shape $\underline{S_i}$ along the trace, Γ_i defines the set of graphs from the original reach set it actually represents, as opposed to the totality of all graphs that are embedded into it. Note that Γ_i can be empty, if the abstraction is so coarse that it includes behavior that can not be replicated by the GTS at all. This will usually be the case. Since every node in an STT has an associated root path, and thus an abstract trace, we denote the local graph set for an STT node n by $\Gamma(n)$.

We will now show that a graph transformation system can only have a valid STT if it is free of the error pattern.

Figure 5.6 shows an example for a valid STT for the linear list example using the canonical abstraction from Def. 64. In the interest of a concise presentation, unary edges have been omitted – *list* nodes are denoted by diamond-shaped nodes, while *cell* nodes are denoted by regular, circular nodes. Solid edges are transition edges, dashed edges are covering edges. Each STT node is denoted by its central shape only.

We can see that the STT does not contain any materializations of the error pattern – a single cell node with a *next* self-edge. Each shape in which the error could potentially

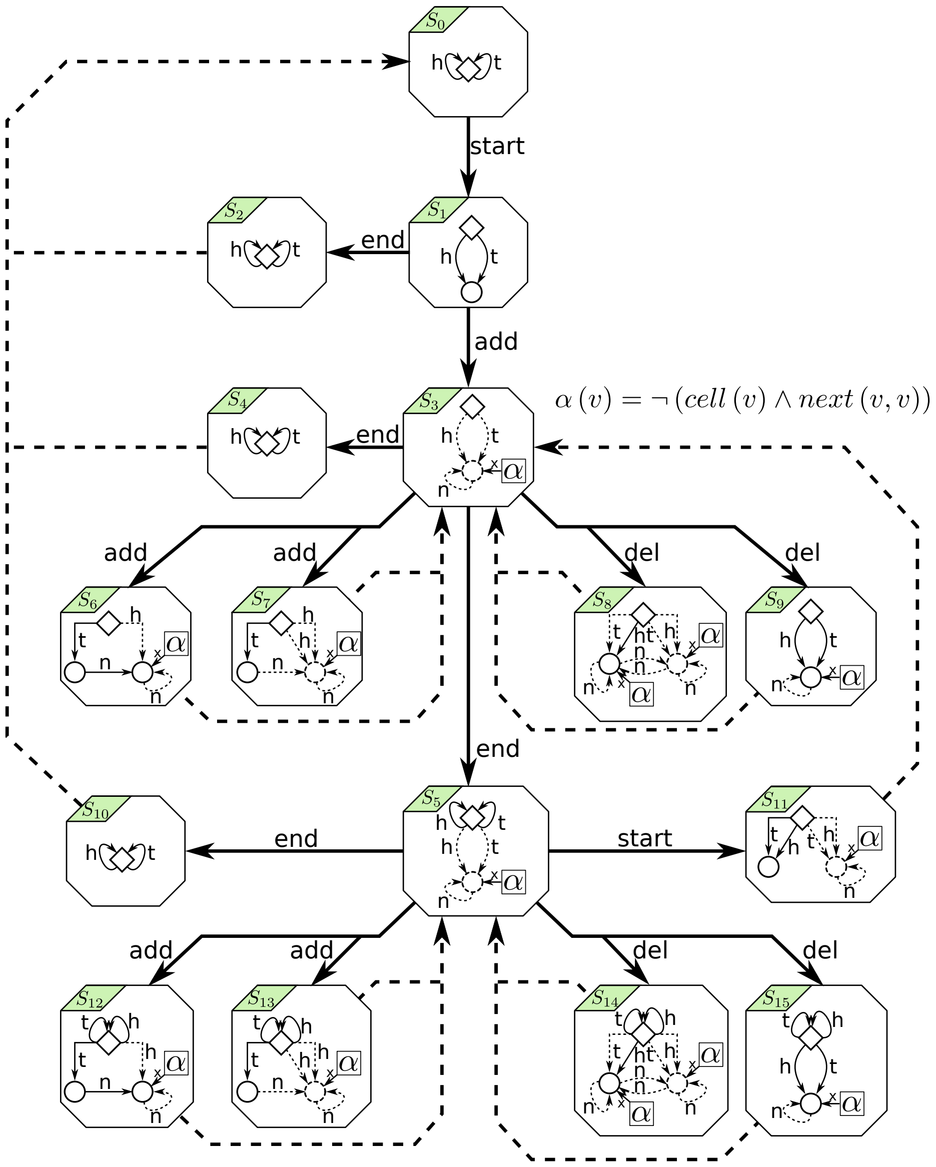


Figure 5.6: A valid shape transition tree for the linear list example.

materialize has been annotated with a constraint forbidding exactly that concretization. Thus, from this STT we conclude that the linear list example is free of that particular error pattern.

In order to prove that this is sound, we need to prove that a valid STT can only cover an error-free GTS. For this, we need to prove that

1. no covered node could potentially produce paths in its subtree that are not covered by paths in the covering node's subtree, and
2. each path in the transition system of the GTS corresponds to a path in the STT (possibly using covering edges).

From Definition 83, we already know that the root of a valid STT is an overapproximation of the initial graph of a GTS (Condition 1). We further know that each transition edge represents a valid shape transition (Condition 2). Thus, by the soundness of shape transitions (see Lemma 6), we already know that paths through the STT that do *not* involve covering edges represent valid overapproximations of paths through the transition system of the GTS. What remains to be seen is that an STT node is marked as covered only when exploring it could absolutely not yield any new behavior. Closely linked to this is the following lemma. It states that when a shape $\underline{S}$ is embedded into another shape $\underline{S'}$, then every one of its materializations (w.r.t. a fixed rule set $\mathcal{R}$) is embedded into one of the materializations of $\underline{S'}$.

Lemma 24 (Embedding implies Materialization Coverage). *Let $\mathcal{R}$ be a set of rules, and let $\underline{S}$, $\underline{S'}$ be shapes such that $\underline{S} \sqsubseteq_f \underline{S'}$. Then for every shape $\underline{S^m} \in \mathcal{M}_{\mathcal{R}}(\underline{S})$, there exists a shape $\underline{S'^{m'}} \in \mathcal{M}_{\mathcal{R}}(\underline{S'})$, such that $\underline{S^m} \sqsubseteq_{f'} \underline{S'^{m'}}$, for some embedding function f'.*

Proof. Let $\underline{S^m} \in \mathcal{M}_{\mathcal{R}}(\underline{S})$ be a materialization for a rule $P \in \mathcal{R}$, with $\underline{S^m} \sqsubseteq_{f^m} \underline{S}$ that preserves a set of summary nodes I. By the embedding property, the potential match m used to construct the embedding has a corresponding match $m' = f \circ m$ for the shape $\underline{S'}$. By the transitivity of embeddings, we have $\underline{S^m} \sqsubseteq_g \underline{S'}$, where $g := f \circ f^m$. We now choose a materialization $S'^{m'} \in \mathcal{M}_{\mathcal{R}}(\underline{S'})$ for the rule P such that the set I' of preserved summary nodes satisfies $f(I) \subseteq I'$, i.e., it contains the embedding targets of all summary nodes preserved by $\underline{S}^m$. We can now construct an embedding $\underline{S^m} \sqsubseteq_{f'} \underline{S'^{m'}}$ as follows.

$$f'(n) := \begin{cases} n & \text{if } n \in N_{L_P} \\ f(n) & \text{otherwise} \end{cases}$$

Since a materialization is constructed (see Def. 46) by removing the match, inserting the left hand side of the rule, while preserving all summary nodes in I, this is indeed a valid function mapping $\underline{S^m}$ into $\underline{S'^m}$. It is also a valid embedding, since it is in effect a union on disjoint supports between a valid embedding and the identity function on N_{L_P}. □

Every materialization point of a covered shape is thus covered by at least one materialization point of the covering shape. From the soundness of shape transformation (see Lemma 6), we know that the coverage relationship is preserved in rule applications. Now, the goal is to show that any behavior that follows from a covered node is necessarily covered by the behavior that follows the covering node. All we really need to know is whether any actual graph traces that might possibly be covered by the abstract trace through a covered node is covered by the behavior that follows the covering node. For this reason, we can ignore the abstraction for the hypothetical behavior induced by the covered node. Thus, the following lemma states that the subtree of concrete graphs under a covered node, is also covered by the subtree under the covering node.

Lemma 25 (Node Coverage implies Subtree Coverage). *Let $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ be a valid shape transition tree for a GTS $\mathcal{G} = (I, \mathcal{R})$ and an abstraction scheme $\mathcal{A}$, and let $n, n' \in N$ be two nodes in that tree such that $\left(\left(\underline{S^{in}}, \underline{S}, \mathcal{M}_{\mathcal{R}}\left(\underline{S}\right)\right), n\right) \in E^1$, $\left(\left(\underline{S'^{in}}, \underline{S'}, \mathcal{M}_{\mathcal{R}}\left(\underline{S'}\right)\right), n'\right) \in E^1$, and $(n, n') \in C$. Let further $\mathcal{T}_{n'}$ be a shape transition tree for $\mathcal{R}$, rooted in n', and using the abstraction scheme $\mathcal{A}$. Let $\Gamma(n)$ be the local graph set of n. Then we have that for every sequence of graph transitions*

$$G_0 \xrightarrow{P_1, m_1} \cdots \xrightarrow{P_k, m_k} G_k$$

of length k with $G_0 \in \Gamma(n)$, there is a mapping $\pi : \{G_0, \ldots, G_k\} \to N$, such that $\forall i \in \{0, \ldots, k-1\}$ we have

$$\exists \left(\underline{S^m}, f^m\right) \in \mathcal{M}\left(\underline{S_{\pi(G_i)}}\right) : G_i \sqsubseteq_{f_i} \underline{S^m} \wedge \underline{S_{\pi(G_i)}} \xrightarrow{P_{i+1}, \underline{S^m}, f_i \circ m_{i+1}} \underline{S_{\pi(G_{i+1})}} \tag{5.3}$$

Note that the path in the image of π might contain covering edges.

Proof. See Appendix A, page 248. □

Thus we can conclude that it is sound to terminate the exploration of an STT at covered nodes, since any behavior that could be discovered beyond the covered node is surely represented by the behavior beyond the covering node. Note that, in particular, the embedding theorem implies that any match for the error pattern that could be found beyond the local graph set of a covered node is guaranteed to turn up in the path through the tree described by π. The subtree coverage lemma, together with the soundness of shape transformations, essentially implies that all paths of a GTS are represented by any valid shape transition tree that was constructed for it on the basis of its initial graph and some abstraction scheme. The following lemma restates this in a more explicit manner.

Lemma 26 (Path Coverage Lemma). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system, and let $\mathcal{A}$ be an abstraction scheme. Let $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ be a* valid STT *corresponding to $\mathcal{G}$ with root node $\rho_{\mathcal{T}} = \left(I, \underline{S_0} := \mathcal{A}(I), \mathcal{M}\left(\underline{S_0}\right)\right)$. Let $\pi := I \xrightarrow{P_0, m_0} G_1 \xrightarrow{P_1, m_1} \cdots$ be some (possibly infinite) path through* $\operatorname{trans}(\mathcal{G})$. *Then there exists a mapping g that maps every graph in π to an edge in E^2, such that $\forall i \geq 0$*

$$G_i \sqsubseteq \underline{S_{g(G_i)}} \tag{5.4}$$

$$G_i \sqsubseteq_{f_i} \underline{S^m_{g(G_i)}} \tag{5.5}$$

$$\underline{S_{g(G_i)}} \xrightarrow{P_i, \underline{S^m_{g(G_i)}}, f_i \circ m_i} \underline{S_{g(G_{i+1})}} \tag{5.6}$$

holds.

Proof. Lemma 25 implies that such a function g exists for any finite path up to an arbitrary length k. The existential nature of the abstraction (expressed in the Embedding Theorem) ensures that whenever the concrete path can be extended, the abstract path can be extended as well. □

We can now formulate the central point of shape transition trees, i.e., that the existence of a valid STT for a given GTS implies that that GTS is free of the error pattern..

Theorem 4 (Valid STTs cover valid GTSs). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system, let E be an error pattern, and let $\mathcal{A}$ be an abstraction scheme. Let $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ be a* valid STT *corresponding to $\mathcal{G}$ with root node*

$$\rho_{\mathcal{T}} = \left(I, \underline{S_0} := \mathcal{A}(I), \mathcal{M}\left(\underline{S_0}\right)\right).$$

Then there is no $G \in$ $\operatorname{trans}(\mathcal{G})$ *such that $[\![\varphi_E]\!]^m_G = \mathbf{1}$ for any m.*

Proof. Assume that there is a node $G_e \in \operatorname{trans}(\mathcal{G})$ such that $[\![\varphi_E]\!]^m_{G_e} = \mathbf{1}$ for some error match m. Then there is a shortest path from the initial graph I to G_e through $\operatorname{trans}(\mathcal{G})$. Let $I \xrightarrow{P_0, m_0} G_1 \xrightarrow{P_1, m_1} \cdots G_k \xrightarrow{P_k, m_k} G_e$ be that path. Then, the Path Coverage Lemma (26) implies that there is a node $n \in \mathcal{T}$ such that $\underline{S_n}$ potentially matches the error. This contradicts the validity of $\mathcal{T}$. □

This shows that a GTS can be considered free of a given error pattern, if a valid STT for that GTS and error pattern can be constructed. The question that remains is how to construct an STT for a given GTS in such a way that the result is guaranteed to be valid if the GTS is error-free. Section 5.3 deals with the basic construction algorithm that delivers this, assuming a refinement algorithm that is explained in Section 6.

5.3 Basic Construction Loop of an STT

The basic idea of how to construct a valid STT for a given GTS (if possible) is as follows. We begin with a graph transformation system $\mathcal{G} = (I, \mathcal{R})$, an abstraction scheme $\mathcal{A}$, and an error pattern E. We construct the root node $\rho_{\mathcal{T}} = (I, S_0, \mathcal{M}_{\mathcal{R}}(S_0))$ of the STT by abstracting I to S_0 using $\mathcal{A}$, and computing $\mathcal{M}_{\mathcal{R}}(S_0)$. Using the shape feasibility encoding (see Sec. 4.7), we mark all materialization points that have empty graph sets. For each remaining materialization point, we then construct an outgoing transition edge by applying the associated rule to the materialized shape. We obtain an unabstracted result shape to which we can apply the abstraction scheme, creating a new STT node as the target of the edge in the process. For these newly created STT nodes, we check whether their central shapes are embedded into the central shapes of any of the already explored nodes (currently only the root node). We add covering edges where such embedding relations exist, and compute the materialization sets where they do not. With each of the remaining new nodes, we now proceed exactly as we did with the root node. This continues until we have constructed a tree where all leaf nodes are covered or contain no valid materializations.

Two issues can prevent us from completing this process successfully.

If the GTS is not safe, or if the initial abstraction is so coarse that it creates spurious errors, we will eventually produce an STT node that contains a feasible materialization point for the error pattern. In such cases we must examine the error, decide whether the error is real, and, if it is not, somehow improve the abstraction. This is covered in Chapter 6.

It is also possible that the exploration does not terminate, if the abstraction is too precise. This is an expected and unavoidable complication, since we are dealing with an undecidable problem.

Assuming that the construction of the STT terminates and that no error nodes are encountered, the resulting STT will be valid (by construction) and prove $\mathcal{G}$ safe with respect to E.

In the following, we will properly define the construction process sketched above, identify and describe the individual sub-algorithms that compose it, and prove that an error-free termination does in fact always produce a valid STT.

Examining the high-level process described above, we observe that there are a number of discrete actions that are taken. For the purpose of specifying the overall construction algorithm, we will assume that each of these actions is performed by its own sub-algorithm. Data that pertains to the overall algorithm, such as the rule set, error pattern, or abstraction scheme used, is assumed to be globally defined and shared among the individual sub-algorithms. The actions that we will integrate in this manner are the following

- Sets of materialization points are computed – this is called *node expansion*.
- Materialization points are checked for emptiness of their graph sets – this is called the *feasibility check*.
- Shape transformations are computed for feasible, non-error materialization points, yielding new STT nodes – this is called *node exploration*.
- Central shapes of newly constructed nodes are checked for embeddings into the central shapes of existing nodes during a *coverage check*.
- Finally, upon discovering a feasible error materialization $\underline{S^e}$, *error handling* is invoked to refine the abstraction. This semi-automatic sub-algorithm either terminates the analysis with a real error, or restructures the STT in such a way that the (infeasible) error can no longer occur.

Before we can define the main algorithm or examine its correctness, we must define the exact effect of each of these algorithms on the STT. For this, we need to slightly adjust our definition of an STT node in order to account for the fact that the creation of a node (done during *exploration*) is separated in the construction process from the computation of its materialization points (done during *expansion*).

Definition 85 (Tentative STT Node). *Let $\mathcal{A}$ be an abstraction scheme. A tuple $\left(\underline{S_{in}}, \underline{S}, \emptyset\right)$ is called a* tentative STT node *iff $\underline{S_{in}}$ is a valid entry point for $\underline{S}$.*

Using this new concept, we now understand that node exploration produces *tentative* STT nodes, while node expansion turns them into fully realized STT nodes. This is useful in the formal definition of the effects of these sub-algorithms, which we will now establish to facilitate the proof of correctness of the main algorithm. We begin with the `EXPAND`-algorithm.

Definition 86 (Valid `EXPAND`-algorithm). *An algorithm taking as input a tentative STT node $n = \left(\underline{S^{in}}, S, \emptyset\right)$ of an STT $\mathcal{T}$, a rule set $\mathcal{R}$ and an error pattern P is said to be a* valid **`EXPAND`**-algorithm*, if and only if after its execution the following conditions hold:*

(a) $n = \left(\underline{S^{in}}, \underline{S}, \mathcal{M}_{\mathcal{R}}\left(\underline{S}\right)\right)$*, and*

(b) no other parts of $\mathcal{T}$ are modified.

Thus, a valid `EXPAND` algorithm computes the materialization set for a tentative node, turning it into a regular STT node. Not all of those materializations will be feasible. In order to identify those that are not, a valid `CHECKFEASIBLE` algorithm can be used.

Definition 87 (Valid `CHECKFEASIBLE`-algorithm). *An algorithm is a* valid **`CHECK-FEASIBLE`**-algorithm *if and only if it takes a shape $\underline{S}$ as input, returns a value $b \geq (\mathcal{G}(\underline{S}) \neq \emptyset)$, always terminates, and has no side-effects.*

Note that this definition does not require that infeasible shapes are always detected. As in other instances in this thesis when the satisfiability of a first-order formula is determined, there is the possibility that this analysis will not terminate (or time out, depending on implementation). If an infeasible shape is not detected, this does not compromise the soundness of the overall analysis – it just means that more spurious parts of the state space will be explored. On the other hand, if a feasible shape were deemed infeasible, this could lead to error paths being overlooked. Thus, in our definition for the `CHECKFEASIBILITY` algorithm we demand that a timeout (or some other measure guaranteeing termination) be implemented and that the algorithm err on the side of feasibility.

Once individual materialization points have been deemed feasible, they can be explored, i.e. the shape transitions they represent computed and new STT nodes created. We now define the specific conditions that the exploration needs to satisfy.

Definition 88 (Valid `EXPLORE`-algorithm). *Let $n = \left(\underline{S^{in}}, \underline{S}, M\right)$ be an STT node in an STT $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$, and let x be the node it labels (in E^1). For every $\left(\underline{S_i^m}, f_i^m\right) \in M$ with $i \in \{1, \ldots, |M|\}$ such that $\mathcal{G}\left(\underline{S_i^m}\right) \neq \emptyset$, let P_i be the rule used to construct the materialization, and let $\underline{S_i^t}$ be the shape such that $\underline{S_i^m} \xrightarrow{P_i, \mathrm{id}} \underline{S_i^t}$. An algorithm taking as input a unary edge $\left(n = \left(\underline{S_{in}}, \underline{S}, M\right), x\right)$ of an STT $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ and returning a set T of tentative STT nodes (again, as unary edges), is a* valid **`EXPLORE`**-algorithm *if and only if after its execution the following conditions on the resulting STT $\mathcal{T}' = \left(N', E^{1'}, E^{2'}, C, l_c\right)$ are satisfied:*

$$
\begin{aligned}
T &= \left\{n_1, \ldots, n_{|M|}\right\} \\
N' &= N \cup T \\
E^{2'} &= E^2 \cup \left\{\left(n, \left(P_i, \left(\underline{S_i^m}, f_i^m\right)\right), n_i\right) \mid i \in \{1, \ldots, |M|\} \wedge \mathcal{G}\left(\underline{S_i^m}\right) \neq \emptyset\right\} \\
E^{1'} &= E^1 \cup \left\{\left(\left(\underline{S_i^t}, \mathcal{A}\left(\underline{S_i^t}\right), \emptyset\right), n_i\right) \mid (n, l, n_i) \in E^{2'}\right\}
\end{aligned}
$$

This relatively complex definition merely assures that transition edges are added to the STT labeled with correct transitions and only for those materialization points that are feasible. Furthermore it ensures that the tentative nodes created by the algorithm are constructed by merely applying the abstraction scheme to the transformed shape. Before these tentative nodes can be transformed into full STT nodes we must make sure that their graph set is not already covered by some other full STT node in the tree.

This is done by using the COVER-algorithm to check for embedding relations between a given pair of nodes,

Definition 89 (Valid COVER-algorithm). *An algorithm taking as input two unary edges (n,x) and (m,y) of an STT $\mathcal{T}$ is a* valid **COVER**-algorithm *if and only if it always terminates and returns a function f, such that*

$$(f=\emptyset)\vee\left(x\not\preceq y\wedge\underline{S_n}\sqsubseteq_f\underline{S_m}\right)$$

Note that, again, as in the case of the CHECKFEASIBILITY algorithm, we do not demand that the COVER-algorithm return an embedding if one exists. We only require that any function it does return is an actual embedding and that the resulting embedding edge would not connect n to a node m within its own subtree.

These algorithm descriptions cover everything the main algorithm does except for error handling. Analyzing a trace and refining the abstraction along its path is a much more complex task than the tasks performed by the other sub-algorithms, leading to an equally complex set of requirements for a valid algorithm that accomplishes said task. However, if we want to prove correctness of the main algorithm effectively, we need to specify, as abstractly as possible, the effect of the HANDLEERROR-algorithm on the STT.

Definition 90 (Valid HANDLEERROR-algorithm). *Let A be an algorithm operating on an STT $\mathcal{T}$ for a GTS $\mathcal{G}$, taking an STT node n and one of its materialization points $\left(\underline{S^m_\epsilon},f^m_\epsilon\right)$ for an error pattern E as an input. Let*

$$\pi\left(n\right)=e_1\cdots e_\epsilon \textit{ with } e_i=\left(n_{i-1},\left(P_{i-1},\left(\underline{S^m_{i-1}},f^m_{i-1}\right)\right),n_i\right)$$

be the root path of n. A is called a valid **HANDLEERROR**-algorithm*, if and only if it always terminates and for any given parameters, and its effect is exactly one of the following:*

(a) If $\Gamma\left(n_\epsilon\right)\cap\mathcal{G}\left(\underline{S^m_\epsilon}\right)\neq\emptyset$, it returns $(\pi,(n))$, where π is an error path through $\mathrm{trans}\left(\mathcal{G}\right)$ corresponding to $\pi\left(n\right)$, and does not modify $\mathcal{T}$.

(b) If $\Gamma\left(n_\epsilon\right)\cap\mathcal{G}\left(\underline{S^m_\epsilon}\right)=\emptyset$, let k be the smallest integer such that $\Gamma\left(n_k\right)\cap\mathcal{G}\left(\underline{S^m_k}\right)=\emptyset$. For any $0\leq i\leq k$, let further m_i, P_i, and I_i be the match, rule, and subset of summary nodes used to construct $\underline{S^m_i}$, and let $T_i=\left(N_i,E^1_i,E^2_i,C_i,l_{ci}\right)$ be the subtree rooted in n_i. Finally, let $0\leq j\leq k$. Then HANDLEERROR *returns $(\emptyset,(n_j,\ldots,n_k))$ and modifies $\mathcal{T}=\left(N,E^1,E^2,C,l_c\right)$ such that the resulting STT $\mathcal{T}''=\left(N'',E^{1''},E^{2'},C'',l''_c\right)$ is defined by $\left(N',E^{1'},E^{2'},C',l'_c\right)=$*

$\mathcal{T}' := \mathcal{T} \setminus T_j$ and the following conditions:

$$\begin{aligned}
N'' &= N' \cup \{n_j, n_{j+1}, \ldots, n_k\} \\
E^{1''} &= E^{1'} \setminus \{(x_j, n_j), (x_{j+1}, n_{j+1}), \ldots, (x_k, n_k)\} \\
&\quad \cup \{(y_j, n_j), (y_{j+1}, n_{j+1}), \ldots, (y_k, n_k)\} \\
y_i &= \left(\underline{S'^{in}_{k-i}}, \left(S_{k-i}, \Lambda'_{k-i} := \Lambda_k \cup \{(\alpha_i, m_i)\}, \mathcal{M}_{\mathcal{R}}\left((S_{k-i}, \Lambda'_{k-i})\right)\right)\right) \\
E^{2''} &= E^{2'} \cup \Big\{ \left(n_j, \left(P_j, \left(\left(S^m_j, \Lambda^m_j \cup \mathrm{concr}_{f^m_j}(\alpha_j, m_j)\right), f^m_j\right)\right), n_{j+1}\right), \ldots, \\
&\quad \left(n_{k-1}, \left(P_{k-1}, \left(\left(S^m_{k-1}, \Lambda^m_{k-1} \cup \mathrm{concr}_{f^m_{k-1}}(\alpha_{k-1}, m_{k-1})\right), f^m_{k-1}\right)\right), n_k\right) \Big\} \\
C'' &= C' \setminus \{(x, l, y) \mid y \in \{n_j, \ldots, n_k\}\} \\
l''_c &= l'_c{\downarrow}_{C''}
\end{aligned}$$

where each (α_i, m_i) is a shape constraint such that the materialization $\underline{S^{m_i}_i}$ from (S_i, Λ'_i) using m_i, P_i, and I_i is infeasible, i.e.

$$\begin{aligned}
\mathcal{G}\left(\underline{S^{m_i}_i}\right) &= \emptyset && \textit{, as well as} \\
\underline{S'^{in}_i} &\sqsubseteq (S_i, \Lambda'_i) && \textit{, and thus} \\
\Gamma(n_i) &\subseteq \mathcal{G}\left((S_i, \Lambda'_i)\right).
\end{aligned}$$

This fairly complex definition merits a more intuitive explanation. Essentially, the HANDLEERROR sub-algorithm decides whether the abstract trace it is given covers a real error. If so, this error is returned and the tree is not modified. If the error is spurious, HANDLEERROR determines the failure point k – the point at which the abstract trace becomes spurious. It also chooses an index $j < k$ and then refines the central shapes of the nodes n_j through n_k with one additional constraint each. The transformed shapes along the refined path segment, along with the materialization sets of the refined nodes are then recomputed. All behavior under the refined nodes except for the refined path is removed. Figure 5.7 shows a schematic of this process.

Note that the HANDLEERROR sub-algorithm depends on solving a long first-order formula. This does not create termination problems, since by construction our trace encoding uses a finite domain, guaranteeing decidability.

However, as we will see in Chapter 6, the automatic part of the analysis is not guaranteed to find a suitable refinement. This motivates the implementation of HANDLEERROR as a *semi-automatic* algorithm, with a human designer complementing the automatic refinement process where necessary. Due to this human component, the concept of termination becomes somewhat fuzzy, since it depends on the ability of the

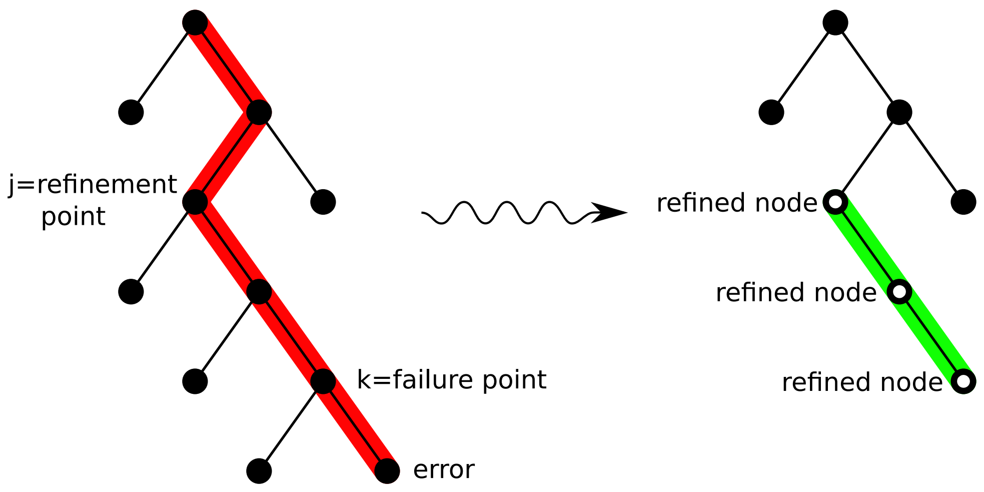


Figure 5.7: Schematic Visualization of the refinement process

user to create refinements. For the remainder, we will assume that, when queried, a human designer can, in cooperation with the algorithm (see Chapter 6), always produce a suitable refinement.

Now, using valid implementations of the sub-algorithms defined above, we can finally formulate the overall construction algorithm, and convince us of its correctness. Listing 5.1 shows the Pseudocode formulation of the algorithm.

Theorem 5 (Correctness of Construction Algorithm). *Let $\mathcal{G} = (I, \mathcal{R})$ be a graph transformation system, let P be an error pattern, and let $\mathcal{A}$ be an abstraction scheme. Let the algorithms* `EXPAND`, `EXPLORE`, `CHECKFEASIBILITY`, `COVER` *and* `HANDLEERROR` *be valid, according to Definitions 86, 87, 88, 89 and 90. Then the main construction algorithm (see Listing 5.1), applied to the inputs $\mathcal{G}$ and P, will either*

(i) not terminate,

(ii) terminate and produce a concrete error path through $\operatorname{trans}(\mathcal{G})$*, or*

(iii) terminate and produce a valid STT $\mathcal{T}$.

Proof. See Appendix A, page 249. □

The above proof and thus the validity of Theorem 5 hinges on the correctness of the individual sub-algorithms. Thus, we will now examine each of them in detail and prove that they conform to Definitions 86, 87, 88, 89, and 90, respectively.

Listing 5.1: Main construction algorithm

```
Input: Initial graph $I$, rule set $\mathcal{R}$, error pattern $P$
$\underline{S_0}$ <- $\mathcal{A}(I)$; $\rho$ <- $\left(I, \underline{S_0}, \mathcal{M}_{\mathcal{R}}\left(\underline{S_0}\right)\right)$;
$\mathcal{T}$ <- $(r, \{(\rho, r)\}, \emptyset, \emptyset, \emptyset)$;
$E$ <- $\{(\rho, r)\}$; $K$ <- $\emptyset$;
while ($E \neq \emptyset$) do
  Choose and remove $(n, x)$ from $E$;
  for each $(m, y) \in K$ do
    $F$ <- COVER($n$, $m$);
    if $\exists f \in F$ then
      $C_{\mathcal{T}}$ <- $(n, m)$; $l_c$ <- $l_c \cup \{((n, m), f)\}$;
      break;
    fi;
  od;
  if ($\exists (n, l, m) \in C$) continue;
  EXPAND($n$);
  for each $\left(\underline{S^m, f^m}\right)$ in ($\mathcal{M}\left(\underline{S_n}\right)$) do
    if not CHECKFEASIBLE($S^m$) then
      mark materialization point $\left(\underline{S^m, f^m}\right)$ invalid in $n$;
    fi;
  od;
  for each valid $(S^e, f^e) \in \mathcal{M}\left(\underline{S_n}\right)$ do
    $(p, (n_1, \ldots, n_l))$ <- HANDLEERROR($n$, $(S^e, f^e)$, $\mathcal{T}$);
    if not ($p$ is empty) then
      return $p$;
    else
      $K$ <- $K \setminus \{n_1, \ldots, n_l\}$;
      $n$ <- $n_1$;
      $E$ <- $E \cup \{x \mid n \in (n_1, \ldots, n_l) \wedge (x, n) \in C_{\mathcal{T}}\} \cup \{n_2, \ldots, n_l\}$;
    fi;
  od;
  $E$ <- $E \cup$ EXPLORE($(n, x)$); $K$ <- $K \cup \{(n, x)\}$;
od;
return $\mathcal{T}$
```

Listing 5.2: EXPAND sub-algorithm

```
Input: STTNode $(\underline{S_{in}}, \underline{S}, M)$
  $M$ <- $\emptyset$;
  for each $r$ in ($\{P\} \cup \mathcal{R}$) do
    for each potential match $m$ for $r$ do
      for each keepSet $i$ for $r, m$ do
        $\underline{S^m}$ <- materialization for $S, r, m, i$;
        $f^m$ <- Embedding of $\underline{S^m}$ into $\underline{S}$
        $M$ <- $M \cup \{(\underline{S^m}, f^m)\}$;
        attach materialization point for $S^m$ to $n$
      od;
    od;
  od;
```

The EXPAND sub-algorithm

The EXPAND sub-algorithm is a simple computation of the materialization set of a given STT node. Listing 5.2 shows a Pseudocode representation of this algorithm. No further properties than the correct computation of materializations is required here. Thus we can assume that EXPAND works as required in Def. 86.

The EXPLORE sub-algorithm

The EXPLORE sub-algorithm similarly simply performs shape transformations as defined in Chapter 3. Examining Listing 5.3, it is easy to see that, in terms of Def. 88 and the proof of Thm. 5, EXPLORE maintains the loop invariant in the sense that it reestablishes the condition that all nodes must either be covered, marked for expansion (here, returned in T), or have transition edges attached to all feasible materialization points. The loop on lines $4 - 13$ goes through all feasible materializations. For each such materialization, lines $6 - 7$ compute the result shape, line 8 constructs a new tentative STT node for the result shape using the abstraction scheme, and lines $9 - 12$ add that node and the corresponding label and transition edge to the correct sets. Thus, EXPLORE works as required by Def. 88.

The CHECKFEASIBILITY sub-algorithm

The objective of the CHECKFEASIBILITY sub-algorithm is to determine whether the graph set for the given constrained shape is empty. This is achieved using the feasibility encoding described in Sec. 4.7. The given shape is encoded into an SMTLib script,

Listing 5.3: EXPLORE sub-algorithm

```
Input: Unary Edge $\left(\left(\underline{S_{in}}, \underline{S}, M\right), n\right)$
  $\mathcal{T} = \left(N, E^1, E^2, C, l_c\right)$ <- STT of $n$;
  $T$ <- $\emptyset$;
  for each $(\underline{S^m}, f^m)$ in $M$ do
    if ($\mathcal{G}(\underline{S^m}) = \emptyset$) then continue;
    $P$ <- rule used to construct $\underline{S^m}$;
    $\underline{S^t}$ <- Apply $P$ to $\underline{S^m}$ at id;
    $m$ <- $(\underline{S^t}, \mathcal{A}(\underline{S^t}), \emptyset)$;
    $T$ <- $T \cup \{(m, x)\}$;
    $N$ <- $N \cup \{x\}$;
    $E^1$ <- $E^1 \cup \{(m, x)\}$
    $E^2$ <- $E^2 \cup \{(n, (P, (S^m, f^m)), x)\}$;
  od;
  return $T$;
```

processed by a solver, and the result returned as a simple boolean response. The resulting algorithm is exceedingly simple, restricted to simply call the encoder, and then the solver. In the case of a timeout, **1** is returned. Otherwise, the return value is

$$b := \begin{cases} \mathbf{1} & \text{, if result is SAT} \\ \mathbf{0} & \text{, if result is UNSAT} \end{cases}$$

The `COVER`-algorithm has a similar part, so we will forgo making the CHECKFEASIBILITY algorithm explicit here, and just assume that we have a valid implementation of the algorithm conforming to Def. 87, based on the correctness of the feasibility encoding established in Lemma 22.

The `COVER` sub-algorithm

The `COVER` algorithm is a bit more involved than `EXPAND`, `EXPLORE`, and `CHECK-FEASIBILITY`. The objective is, given two STT nodes n and m, to compute a valid embedding from $\underline{S_n}$ to $\underline{S_m}$, if such an embedding exists. This is the first instance in this thesis, where we have to decide if there exists an embedding relationship between to *possibly constrained* shapes. Thus far, we have not properly defined how such a check would be performed (see Def. 61).

As a reminder, $\underline{S_n}$ can, axiomatically speaking, be thought of as being embedded into $\underline{S_m}$, if and only if $\mathcal{G}\left(\underline{S_n}\right) \subseteq \mathcal{G}\left(\underline{S_m}\right)$, as specified in Def. 61. This is the case if a classical embedding f exists such that $S_n \sqsubseteq_f S_m$, *and* the constraints of $\underline{S_m}$, projected onto $\underline{S_n}$, do not further restrict the meaning of $\underline{S_n}$. If we think of a shape as formulating conditions on the graphs embedded into it, we can express this same thought in the language of logic – the restrictions imposed by $\text{concr}_f\left(\underline{\Lambda_{S_m}}\right)$ on embedded graphs are implied by the restrictions imposed by S_n and $\underline{\Lambda_{S_n}}$, respectively. In other words, if $\underline{S_n} \sqsubseteq_f \underline{S_m}$ holds, then no graph can exist that is embedded into $\underline{S_n}$ but violates some constraint in $\text{concr}_f\left(\underline{\Lambda_{S_m}}\right)$.

Lemma 27 (Constraint Implication implies Embedding). *Let* (S, Λ) *and* (S', Λ') *be shapes such that* $S \sqsubseteq_f S'$ *for some* f. *Then we have* $(S, \Lambda) \sqsubseteq_f (S', \Lambda')$ *if and only if*

$$\forall G : G \sqsubseteq_g (S, \Lambda) \to G \models concr_{f \circ g}(\Lambda')$$

Proof. First, $(S, \Lambda) \sqsubseteq_f (S', \Lambda') \Rightarrow \forall G : G \sqsubseteq_g (S, \Lambda) \to G \models concr_{f \circ g}(\Lambda')$.

If $(S, \Lambda) \sqsubseteq_f (S', \Lambda')$, then all $G \in \mathcal{G}(S, \Lambda)$ must be embedded in (S', Λ'). Since $S \sqsubseteq_f S'$, the existence of an embedding $G \sqsubseteq_g S$ implies the existence of an embedding $G \sqsubseteq_{f' := f \circ g} S'$. This embedding is valid for the *constrained* shape (S', Λ'), iff $G \models concr_{f'}(Lambda')$ by Def. 60.

Now, $(S, \Lambda) \sqsubseteq_f (S', \Lambda') \Leftarrow \forall G : G \sqsubseteq_g (S, \Lambda) \to G \models concr_{f \circ g}(\Lambda')$.

We know that $S \sqsubseteq_f S'$, and that all $G \in \mathcal{G}(S, \Lambda)$ satisfy $concr_{f'}(Lambda')$. Therefore $G \sqsubseteq_{f'} S'$ for all $G \in \mathcal{G}(S, \Lambda)$, and the claim follows directly from Def. 60. $\square$

This last formulation suggests a simple solution to determining the embedding relation between constrained shapes – we can use the shape feasibility encoding from Sec. 4.7. This encoding can be thought of as consisting of two parts, two formulas: the part encoding embeddability into the shape itself (comprised of Code Templates 30, 28, 29, 32, 33, 34, and 35), and the part encoding adherence to the constraints attached to the shape (comprised of Code Templates 19 and 36). For any shape $\underline{S}$, we will denote these parts $\|S\|$ and $\|\underline{\Lambda}_{\|}S$, for the sake of argument. Now, for the shape $\underline{S_n}$, any model of $\|S_n\| \wedge \|\underline{\Lambda_{S_n}}\|$ represents a graph embedded into it. In accordance with our argument above, we can extend that to

$$\|S_n\| \wedge \|\underline{\Lambda_{S_n}}\| \wedge \neg\| \text{concr}_f\left(\underline{\Lambda_{S_m}}\right) \|.$$

Any model of this formula would represent a graph that is embedded in $\underline{S_n}$, but does not satisfy at least one of the constraints implied by $\underline{S_m}$. In other words, it would be evidence, that the graph set of $\underline{S_n}$ is not fully covered by the graph set of $\underline{S_m}$, refuting the embedding relation. On the other hand, if the formula is unsatisfiable, then no such graph exists, and a valid embedding relation is established. That is the

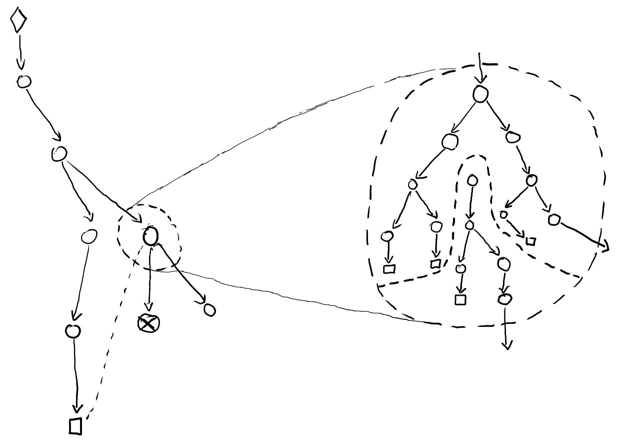
Listing 5.4: COVER sub-algorithm

```
1  Input: STTNode $n$, STTNode $m$
2
3    $F$ <- Compute all classical embeddings from $S_n$ to $S_m$
4    for each $f$ in $F$ do
5      $\mathcal{S}_n$ <- EncodeFeasibility($\underline{S_n}$); // see $Sec.$ 4.7
6      $\Lambda$ <- $\mathrm{concr}_f\,(\Lambda_m)$
7      $\mathcal{C}$ <- EncodeConstraints($\Lambda$); // see $CoT.$ 36
8      result <- SolveWithTimeout($\mathcal{S}_n \land \neg\mathcal{C}$); // any SMT solver (UF logic)
9      if (result == UNSAT) then
10       return $f$;
11     fi;
12   od;
13   return $\emptyset$;
```

basis of the COVER algorithm, shown in Listing 5.4. The correctness of the embeddings computed by it follows from the correctness of the shape feasibility encoding and Lemma 27.

As was already hinted at in the definition of a *valid* `COVER`-algorithm (Def. 89), the potential undecidability caused by quantifiers in the constraint encodings leads to a problem that we have to handle via a timeout on line 8. In case the unsatisfiability of the encoding cannot be decided within a certain time frame, SAT is assumed, i.e. we assume that the embedding we were testing is invalid. This ensures that we only return valid embeddings. We can also occasionally miss embeddings, but just as in the `CHECKFEASIBILITY` case, this does not compromise soundness, just efficiency and, potentially, termination.

With these straightforward sub-parts, the construction algorithm is almost complete. The last missing piece is the algorithm that handles errors when found. Since this process is much more complicated, it will be handled in its own chapter.



6 Interpolation-guided Refinement Loop

The sheer expressive power of first-order logic, used to create constraints in our shape abstraction approach, enables us to craft very precisely tuned abstractions. The downside of this is the size of the design space for these constraints. When a spurious error is found in the course of the construction algorithm described in Section 5.3, we must create a constraint, or possibly a series thereof, that removes the error from the analysis. Left solely to a human designer, this is a very complex task, error-prone and time-consuming. However, a fully automatic approach faces stiff challenges due to the nature of the formalism, and might not always be able to craft constraints that are as meaningful as constraints created by a human mind.

In this thesis, we adopt a hybrid approach. We will introduce two different procedures to automatically generate constraints from a spurious error. Both of these procedures create constraints applicable to a single point in the trace. In cases where only a series of constraints would soundly exclude the error, this will be detected and a human designer queried for those constraints. In designing the constraints, the designer will be assisted by results from the automatic attempt at constraint design, as well as methods to check the soundness of any constraints designed so far.

A Short Examination of Abstraction Refinement Approaches

Before we move on to describe our own approach to abstraction refinement in detail, we will give the reader a short (if almost necessarily incomplete) overview over abstraction refinement approaches. As the basic idea of abstraction is widespread in all fields of analysis, not just in software verification, so is the basic concept of abstraction refinement. We will thus only look at a few examples in the field of verification in general, and shape analysis in particular, that offer alternatives to the counterexample-based refinements we already discussed on page 132.

Any approach at software verification that uses abstraction in any variable way, that is, any approach that is able to use different abstractions for different problem instances (or even the same instance) must incorporate some mechanism to configure the abstraction. We will distinguish two ways of doing so: a priori and a posteriori approaches.

A priori abstraction refinement (or more accurately, abstraction configuration) approaches seek to adapt the abstraction to the problem at hand *before* applying it. This usually involves heuristics about the structure of the system[117] or the error that is searched for. Such approaches are orthogonal to the a posteriori approaches and can be applied in conjunction with them. There has also been work to apply an a priori approach to abstraction refinement to our abstraction process[42], using knowledge derived from the structure of the graph transformation rules and the error pattern.

A posteriori abstraction refinement approaches are those which we normally associate with the term "abstraction refinement". Starting from a first, usually very rough guess at an abstraction, the abstraction is iteratively refined, in either lazy or non-lazy fashion, until the abstraction is precise enough to prove or disprove the desired property. The quintessential example of this is of course counterexample-guided abstraction refinement. Before the introduction of CEGAR, though, and in many specialized cases where for whatever reason CEGAR is not applicable, a number of other approaches to abstraction refinement have been explored. For the purposes of this overview, an approach is only a true CEGAR approach when the entire counterexample (meaning a path from an initial state to an error state) is used for refinement. By this definition, the following kinds of approaches outside the CEGAR approach exist.

In some cases, the abstraction is parametrized by a relatively small number of factors, such as the radius in neighborhood abstraction (see Sec. 8.1), or even the path length in bounded model checking*. The standard approach to refining such abstractions is very simple – just keep increasing the numbers until the abstraction is sufficiently precise. Heuristics may be applied to the stepping of those increases, such as using the "size" of the property as a lower bound in neighborhood abstraction, but the basic principle

*For this, bounded model checking must be seen as an abstraction insofar as it represents an *under*-approximation of the state space remains the same.

When more flexible abstractions are used, more complex refinements are required. As an example, Loginov, Reps and Sagiv[110] extend their shape analysis approach[134] with an abstraction refinement approach that uses subformula-based heuristics combined with inductive learning to gain new instrumentation predicates for their abstraction. If a verification approach uses theorem proving, the information by the prover can often be used to refine the abstraction used, as done for example by Saïdi and Shankar[135]. Other approaches may source their abstraction refinements from sources completely external to the analysis itself – for example, Bauer, Toben and Westphal[23] use separate analyses to gain insights about the system, such as topology invariants, that can be used for abstraction refinement. The diversity of abstraction refinement approaches is only bounded by the diversity of abstraction approaches.

The remainder of this chapter will be structured as follows. First, we will describe how to differentiate spurious errors from real ones, and analyze the error trace itself as well as the conditions that any useful refinement that excludes a spurious error would have to meet in Section 6.1. Next, we will explore the possibilities for automatic abstraction refinement in Section 6.2. Finally, in Section 6.3, we describe the tools available to human designers when crafting constraints in cases where the automatic abstraction refinement methods fail. On the whole, this will constitute a semi-automatic algorithm conforming to Def. 90 from the previous chapter.

6.1 Error Analysis and Conditions on Refinement

Before any abstraction refinement can take place, we must first make sure that it is even necessary, that is, we must determine whether the error is really spurious. This is done by checking if there is a corresponding concrete error in the original graph transformation system. Should such an error exist, we have proven the GTS unsafe and are finished. On the other hand, if no such concrete error exists, we have detected a *spurious error*, and must somehow refine the STT in such a manner as to exclude a re-occurrence of that particular spurious error.

The starting point of our approach will be the trace encoding established in Chapter 4, Section 4.5. Each materialization point for the error pattern that we might find is attached to an STT node e, which defines a unique, abstract path from the root STT node ρ to e. From the information contained in that root path, an abstract error trace according to Def. 65 can easily be constructed. Thus, we are in a situation where we can check for the existence of a corresponding concrete error trace, i.e. an “actual” error in the original GTS (see Def. 66), by using the trace encoding described by Code Template 24.

Clearly, if this encoding is unsatisfiable, then there is no corresponding concrete error trace and therefore no behavior in the original GTS, erroneous or otherwise, that is reflected by this abstract trace, which means that it can safely be removed from the STT. On the other hand, since the encoding does use the original, initial graph for its encoding and tries to find actual sequences of applications of the original rules to mirror the behavior represented by the abstract trace, any model for this encoding will represent an actual path through the transition system for the original GTS, ending in a graph that matches the error pattern, in other words, an actual error. Thus, with the encoding and solving of the trace encoding, the first task is complete – if the error is real, we can construct a corresponding error path and return it.

To get to this point, the original formulation of the trace encoding was sufficient. In order to extract useful constraint information from the trace, additional work is needed. We will first focus on extracting additional information from the given encoding, and then enhance the encoding to make it more useful for abstraction refinement.

If the encoding is unsatisfiable, then we know that at some point along the path, the abstraction ceased to cover any actual behavior. It is this point that we need to find and change in such a way that the remainder of the path can no longer be explored.

This change will be in the form of new shape constraints, attached to the central shapes of STT nodes along the abstract trace in such a way that the constraints do not allow the abstract trace to proceed to the error pattern, but also do not exclude any of the behavior of the original graph transformation system. We codify these properties of a useful and correct refinement in the following definition.

Definition 91 (Valid Refinement). *Let* $\pi = e_1 \cdots e_k$ *with*

$$e_i = \left(n_{i-1}, \left(P_i, \left(\underline{S_{i-1}^m}, f_{i-1}^m\right)\right), n_i\right)$$

be the root path of an STT node $n_k = \left(\underline{S_k^{in}}, \underline{S_k}, M_k\right)$ *such that* $\left(\underline{S_k^e}, f_k^e\right) \in M_k$ *is a materialization for the error pattern. Let* $n_{j_1}, \ldots, n_{j_2}$*, with* $n_i = \left(\underline{S_i^{in}}, \underline{S_i}, M_i\right)$ *be a sequence of STT nodes along that path. A sequence of constraints* $(\alpha_{j_1}, m_{j_1}), \ldots, (\alpha_{j_2}, m_{j_2})$ *is a* valid refinement *for* π *if and only if*

$$\mathcal{G}\left(S_{j_2}^m, \Lambda_{j_2}^m \cup \mathrm{concr}_{f_{j_2}^m}\left(\{(\alpha_{j_2}, m_{j_2})\}\right)\right) = \emptyset$$

the materialization at j_2 *is made invalid*

$$\forall i \in \{j_1, \ldots, j_2\} : \underline{S_i^{in}} \sqsubseteq \left(S_i, \Lambda_i \cup \{(\alpha_i, m_i)\}\right)$$

the incoming shape remains embedded

$$\forall i \in \{j_1, \ldots, j_2\}, f, G : \left[G \in \Gamma(n_i) \wedge G \sqsubseteq_f \underline{S_i} : G \models \mathrm{concr}_f\left(\{(\alpha_i, m_i)\}\right)\right]$$

no actual graph is excluded

Note that the second condition implies the third. Note also that the first and last condition taken together mean that j_1 and j_2 have to be chosen such that the materialization at j_2 is spurious. The objective of the semi-automatic algorithm described in this section is to find such indices, construct a valid refinement as defined above and add it to the STT at the appropriate point along the path. We will focus first on finding this point, the exact spot where the abstract trace becomes spurious.

It stands to reason that, if there is some index i in the path π, after which no rule applied in the path has a concrete counterpart, that the parts of the encoding that encode for steps $i + 1$ and later are inconsequential for that insight. This is because the encoding of concrete traces corresponding to an abstract trace (see CoT 24) has the form

$$\varphi := \|I\| \wedge \|step_0\| \wedge \cdots \wedge \|step_k\|,$$

where $\|I\|$ is the encoding of the initial state, consisting of Code Templates 3, 5, 6, and 23, while each $\|step_i\|$ is an encoding of a rule application, consisting of instances of Code Templates 11, 12, 14, 15, 16, 17, 21, 26, and 27. Note that in this context the match of the error is also an application step, albeit with an empty effect.

Thus, each prefix of initial state encoding and i step encodings encodes a set of graphs that result from applying the corresponding rules to the initial graph. This set was called Γ_i in Sec. 5.3. The subsequent step will build on that graph set, meaning that if the graph set becomes empty, e.g. because there simply is no corresponding valid rule application in the concrete GTS, every consecutive graph set must be empty as well. We will call the index in the path at which this occurs the *failure point* (denoted k in Def. 90).

Definition 92 (Failure Point). *Let π be an abstract error trace, and let*

$$\varphi := \|I\| \wedge \|step_0\| \wedge \cdots \wedge \|step_n\|,$$

be its encoding according to Code Template 24. For any $n \geq i \in \mathbb{N}$, let

$$\varphi_i := \|I\| \wedge \|step_0\| \wedge \cdots \wedge \|step_i\|,$$

be the i-th prefix of φ. Then, the failure point of *φ is defined by*

$$\min\{i \in \mathbb{N} \mid i \leq k \wedge \varphi_i \models \bot\}$$

It is worth noting, that the actual failure can only occur at a specific point within the $\|step_i\|$ identified by the failure point. Each $\|step_i\|$ is defined, according to Code Template 24, as

$$||embed_i|| \wedge ||apply_i||,$$

where $||embed_i||$ encodes the embedding of the current graph (set) into the *materialized* shape of step i, as well as an instance of Code Template 21 and all its required precursor instances (instances of CoT. 19 and CoT. 20), and $||apply_i||$ encodes the actual application of the rule (using Code Templates 26 and 27). Since, once the embedding into the materialized shape is established, the match of the rule is given and thus the application of the rule necessarily possible, the failure *must* lie in the embedding. In other words, when trace encodings fail to be satisfiable, it is *always* due to the embedding into the materialized shape at the failure point.

Having thusly determined the failure point, we turn to the subject of refining the abstraction. Since the failure occurs when a materialized shape represents a subset of the graph set of the central shape of an STT node that does not actually represent any graphs in the original system, the goal will be to add constraints to the central shape of that node that preclude that particular materialization. If the error is already implied by the incoming shape, additional constraints added to nodes further back in the trace might also be necessary. Adding these constraints should be done in such a way that the thusly gained information is carried over to the other materialization points to refine the entire subtree defined by the STT node. We will discuss two separate automatic ways to achieve this, as well as explore tools to make this task easier for human designers.

Before we can do this, however, we will reexamine the trace encoding from Chapter 4 and make two adjustments. The first adjustment will concern the encoding of constraints, while the second will add to the conditions that the embeddings must satisfy.

When we first created our trace encoding, we assumed nothing about the constraints that might exist on the abstract shapes along the way. This lead to an encoding that is able to handle any kind of constraint. However, in our application of the trace encoding for the purposes of abstraction refinement, we know exactly where these constraints come from, and what properties they have. This allows us to introduce a significant simplification of the encoding.

Consider the following argument to motivate the simplification. Since we begin our state space construction process with no abstraction refinement whatsoever, the part of the encoding that accounts for the constraints (CoT. 21) will always reduce to `(true)`, adding no information to the encoding. On subsequent trace encodings, for every shape we encode an embedding into, we have exactly two kinds of constraints – those that were added to the central shape of the respective node directly by the refinement, and those that were added to nodes further back in the trace and carried over to this shape via shape transformations. In both cases, we can conclude that the encoding of these constraints is superfluous. In the first case, the constraint is a valid refinement according to Def. 91, and thus cannot evaluate to false on any graph embedded into the shape. In the second case, the constraint was a valid refinement for a prior shape, and the soundness of the transformation of constrained shapes guarantees us that since the constraint did not contradict any graph at the concrete level when it was introduced, it still does not do so.

This intuitive argument suggests that accounting for constraints that were obtained from the encoding itself in future encodings is redundant. The following definitions and lemmata confirm this intuition. In order to make assertions about the process of refinement, we first need to establish the starting point to which refinements are applied.

Definition 93 (Original Central Shape). *Let $\mathcal{T}$ be an STT produced by a (not necessarily complete) run of the construction algorithm described by Listing 5.1. For every node n in $\mathcal{T}$, the shape $\underline{S_n^o}$ denotes its* original central shape*, i.e. the shape the node was originally created with. By definition, every later, refined central shape $\underline{S_n}$ is embedded into $\underline{S_n^o}$.*

Being thusly able to distinguish between constraints that were present from the beginning and thus implied by the shape transformation that created a node, and constraints that were added as valid refinements, we can now state the validity lemma hinted at by the definition of valid refinements.

Lemma 28 (Valid Refinements Remain Valid). *Let $\mathcal{T}$ be an STT. Let $n = \left(\underline{S^{in}}, (S_n, \Lambda_n), M\right)$ be an STT node of $\mathcal{T}$, let (S_n^o, Λ_n^o) be its original central shape with $\underline{S_n} \sqsubseteq_{f^o} \underline{S_n^o}$, and let every $(\alpha, m) \in \Lambda_n \setminus \operatorname{concr}_{f_n^o}(\Lambda_n^o)$ be a valid refinement. Let $\Gamma(n)$ be the local graph set of n, let $\pi(n)$ be the root path for n, $\pi'(n)$ be the same path with all constraint sets replaced by $\emptyset$, and let $\Gamma'(n)$ be the local graph set for n implied by that modified root path. Then*

$$\Gamma(n) = \Gamma'(n)$$

Proof. We will prove this by induction over the root path, beginning with the first refined node (for all prior nodes, $\Gamma(n) = \Gamma'(n)$ must obviously hold). Let n_0 be that node. Then we know that $\Lambda_{n_0}^o = \emptyset$, i.e. all $(\alpha, m) \in \Lambda_{n_0}$ are valid refinements as defined by Def. 91. Because of this, Def. 91 implies that $\Gamma(n_0) = \Gamma'(n_0)$, since valid refinements, by definition, do not contradict the local graph set.

Now, let n_i be a node along the path such that $\Gamma(n_i) = \Gamma'(n_i)$. Then we know that for all constraints in Λ_{n_i}, and all graphs $G \in \Gamma'(n_i)$, $G \models \operatorname{concr}_f(\Lambda_{n_i})$ holds, where f is the function that embeds G into S_{n_i}. The soundness of constrained shape transformation (established by Theorem 3) the guarantees that the claim holds for the unabstracted result shape $\underline{S_i^t}$ as well. The main algorithm (see Listing 5.1) now guarantees that for the resulting STT node n_{i+1}, i.e. the next STT node in the root path, the original central shape $\underline{S_{i+1}^o}$ is obtained by applying the abstraction scheme to $\underline{S_i^t}$. Thus, since $\underline{S_i^t} \sqsubseteq \underline{S_i^o}$, we know that $\forall G \in \Gamma'(n_{i+1}) : G \sqsubseteq \underline{S_{i+1}^o}$. In analogy to the start of the induction, we can now use Def. 91 to conclude that the same holds for any refinement of the original central shape as well, completing the induction. □

Thus, we now know that all valid refinements we add will, as they should be, always be satisfied on the actual concrete graphs embedded into the central shapes of our STT nodes. This implies that encoding the implied constraints for the embedding into any materialized shape is redundant, as the following Lemma shows.

Lemma 29 (Classical Embedding Implies Constraint Satisfaction). *Let G be a graph, and let $\underline{S} = (S, \Lambda)$ be a shape, and let F be the set of all f such that $G \sqsubseteq_f S$. Let furthermore $\forall f \in F : G \sqsubseteq_f \underline{S}$. Let $\underline{S^m}$ be a shape such that $\underline{S^m} \sqsubseteq_{f^m} \underline{S}$ for some f^m and $\Lambda^m = \mathrm{concr}_{f^m}(\Lambda)$. Then*

$$\forall g : G \sqsubseteq_g S^m \rightarrow G \sqsubseteq_g \underline{S^m},$$

i.e. every classical embedding of G into S^m is also a constrained embedding of G into $\underline{S^m}$.

Proof. Let g be a function such that $G \sqsubseteq_g S^m$. Then $f := f^m \circ g$ is a function such that $G \sqsubseteq_f S$. Since F contains all such functions, we know that $G \sqsubseteq_f \underline{S}$ and thus $G \models \mathrm{concr}_f(\Lambda)$. We obtain

$$\begin{aligned} G \models\ & \mathrm{concr}_f(\Lambda) \\ &= \mathrm{concr}_{f^m \circ g}(\Lambda) \\ &= \mathrm{concr}_g(\mathrm{concr}_{f^m}(\Lambda)) \\ &= \mathrm{concr}_g(\Lambda^m) \end{aligned}$$

Thus we get $G \sqsubseteq_g \underline{S^m}$. □

Put together, these lemmata prove that we can omit the constraints encoding. The unconstrained embedding encoding always has all valid classical embeddings as models, making sure that the premise for Lemma 29 is and remains given. Thus, whenever a classical embedding into a materialized shape exists, the implied constraints are always satisfied.

This concludes our first adjustment. To motivate the second adjustment, we observe that in the current encoding, there are four code templates that together define the embeddings used. These make sure that the embedding function maps to the target

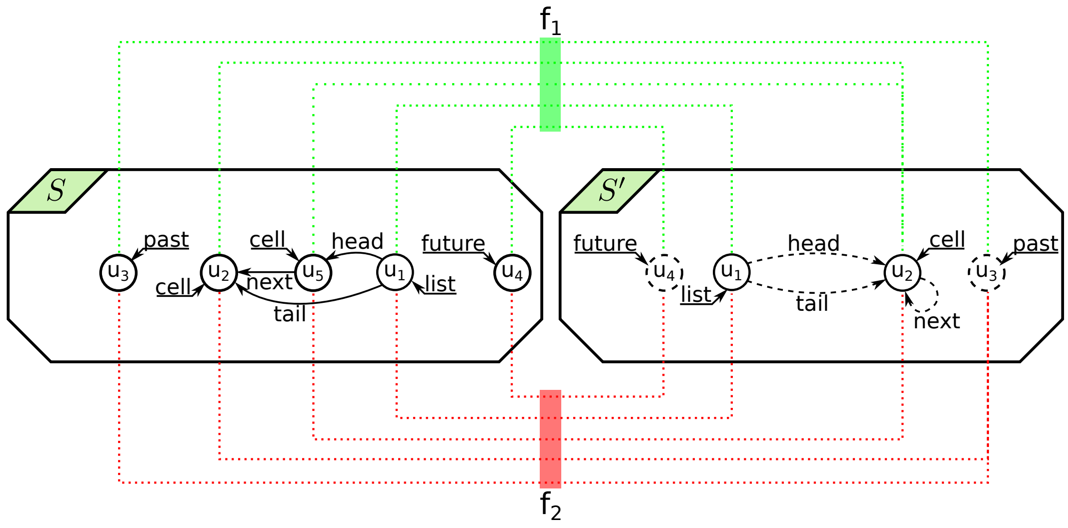


Figure 6.1: Embedding fails due to missing nodes, while the *past* node hides that fact

shape (CoT. 11), obeys the summarization property and is surjective (CoT. 12), and maps graph nodes to shape nodes in such a way that their edges are compatible (CoT. 16 and CoT. 17). Considering that we will attempt to find the reason for the non-existence of embeddings into materialized shapes using interpolation, this is not quite ideal.

To understand this, note that, given shapes S and S', there are two different groups of reasons why S might not be embeddable into S' – the embedding may fail because of missing or surplus nodes, or due to incompatible edge sets. It is important to understand that these issues are not independent of each other. As an example, consider the situation shown in Fig. 6.1.

It is clear that, intuitively, the embedding between these two shapes fails because S' assumes that there is only one *cell* node, while S contains two. This is illustrated by the green mapping which is an embedding except for the fact that the *cell* node in S' is not a summary node. When encoding a trace that includes such embeddings, we would like to detect the fact that the node set appears to be the problem.

However, since the various properties of an embedding are encoded separately, the mapping shown in red is, for the encoding, just as viable a candidate for an embedding as the green mapping. The red mapping fails for an entirely different reason, however – the fact that a *cell*-labeled node should not be embedded into an *past*-labeled node, and the fact that incoming *tail* and *next* edges are disallowed. While this is a perfectly valid result, it does not help us much, for example, in crafting a constraint to attach to a central shape to exclude a materialization. In short, we would like the encoding to ignore embeddings that fail "for the wrong reason".

In general, it is not possible to properly define "right" or "wrong" reasons for non-embeddability. However, in our particular context, where every embedding we care about is an embedding of a graph into a materialized shape, where we know that the graph is already embedded into the central shape, we can improve the situation.

Let G be a graph, let S be a shape, let $S^m \sqsubseteq_{f^m} S$ be a materialization from S, and let $F \neq \emptyset$ be the set of all embeddings f such that $G \sqsubseteq_f S$. Now, let g be an embedding such that $G \sqsubseteq_g S^m$. From the transitivity of embeddings, we know that $G \sqsubseteq_{f^m \circ g} S$, and thus $f^m \circ g \in F$. Thus, every embedding of G into S^m (if any exist) is an f^m-factor of an embedding into S.

This fairly straightforward insight allows us to add to the encoding a formula which restricts the choices for possible embeddings into the materialized shape to only such functions that, when concatenated with f^m, yield an embedding into the central shape. Not only does this potentially make searching for a model for the encoding easier, it has the more important benefit that we can now distinguish between a trace that failed due to node set concerns and a trace that failed only because of missing / superfluous edges. Before we can see this, however, we must formally define the formulas we add to the encoding.

Given an embedding f into the central shape, we can easily formulate the constraints it places on possible embeddings g into the materialization:

$$\forall x : g\left(x\right) \in f^{m-1} \circ f\left(x\right)$$

In order to encode a condition like this, however, our trace encoding would have to include information about the embedding into the central shape. Currently, it only includes this information indirectly.

As Def. 62 implies, the embedding of a graph into a materialized shape is the exact same embedding as the embedding of the transformed graph into the transformed, materialized shape. This is because the same rule is applied to the same, concrete subpart of the graph and the shape. Our encoding thus contains an embedding into the unabstracted, transformed shape S_i^t at any step i.

The obvious idea would be to utilize this information to restrict the materialization embedding. After all, we know how S_i^t embeds into S_i via f_i^t, and thus can construct by $f_i = f_i^t \circ f_{i-1}^m$ valid embeddings into the central shape. However, doing this would not quite be correct. While the formula $f_i = f_i^t \circ f_{i-1}^m$ would yield an embedding into the central shape for every embedding into the incoming shape, the fact that the central shape is (usually) more abstract means that more embeddings than that are possible. That is not to say that more graphs are allowed – the graph set is still constrained by the concrete trace set. Rather, the abstracted central shape allows for reinterpretations of the graphs that exist via new embeddings. These new embeddings translate to new projected matches, in much the same way as depicted in Fig. 4.5. Thus, using the embeddings implied by the embeddings into the incoming shape is insufficient – we need the actual embeddings into the central shape.

We are going to achieve this by simply adding an additional embedding encoding to each step of the trace. Thereby, each step then encodes

(a) an embedding into the central shape,

(b) a restriction of embeddings into the materialized shape,

(c) an embedding into the materialized shape, and

(d) a rule application based on that embedding.

The only new component in this is the encoding for the restriction of the embedding into the materialized shape. We will define it now.

Code Template 38 (Restriction of Materialization Embedding). *Let $\mathcal{S}$ be a script wherein an instance of Code Template 18 encodes embeddings into a shape S. Let S^m be a materialization for S, where $S^m \sqsubseteq_{f^m} S$. Let the embedding function for S used be named _F_i and let a further function with the same signature names _F_j be declared. Then the* encoding of the restriction of the materialization embedding *is given by the first-order formula*

$$\bigwedge_{x \in U} \bigwedge_{y \in U_{S^m}} \left[(\_F\_i\,(x) = y) \rightarrow \left(\bigvee_{z \in {f^m}^{-1}(y)} (\_F\_j\,(x) = z) \right) \right]$$

This is encoded into SMTLib in the straightforward manner, defining a function named `corr_`*i*`_`*j*.

The correctness of this code template is established in the following lemma.

Lemma 30 (Correctness of Embedding Restriction). *Let $\mathcal{S}$ be a script wherein an instance of Code Template 18 encodes embeddings into a shape S. Let S^m be a materialization for S and let $\mathcal{S}$ furthermore contain an instance* `corr_`*i*`_`*j of CoT. 38 for S, S^m. Let the embedding functions used be named _F_i and _F_j, respectively. Then any model for $\mathcal{S}$ will contain interpretations of _F_i and _F_j such that*

$$\_F\_i = f^m \circ \_F\_j$$

Proof. Assume that we have a model for $\mathcal{S}$ wherein there exists a node n such that $\_F\_i\,(n) \neq f^m \circ \_F\_j\,(n)$. Then, from CoT. 38 we can conclude that we can reorganize $\mathcal{S}$ to represent a first order formula given by

$$\varphi \equiv \alpha \wedge \bigwedge_{y \in U_{S^m}} \left[(\_F\_i\,(n) = y) \rightarrow \left(\bigvee_{z \in f^{m-1}(y)} (\_F\_j\,(n) = z) \right) \right].$$

Let now $x_1 := \_F\_i\,(n)$ and $x_2 := f^m \circ \_F\_j\,(n)$, with $x_1 \neq x_2$. We obtain

$$\begin{aligned}
\varphi &\equiv \alpha \wedge \bigwedge_{y \in U_{S^m}} \left[(\_F\_i\,(n) = y) \rightarrow \left(\bigvee_{z \in f^{m-1}(y)} (\_F\_j\,(n) = z) \right) \right] \\
&\equiv \alpha \wedge \bigwedge_{y \in U_{S^m}} \left[(x_1 = y) \rightarrow \left(\bigvee_{z \in f^{m-1}(y)} (\_F\_j\,(n) = z) \right) \right] \\
&\equiv \alpha \wedge \left(\bigvee_{z \in f^{m-1}(x_1)} (\_F\_j\,(n) = z) \right) \qquad (6.1) \\
&\equiv \alpha \wedge \mathbf{0} \equiv \mathbf{0} \qquad (6.2)
\end{aligned}$$

Where (6.1) holds because the implication can only not evaluate to true for that part of the conjunction where $x_1 = y$ holds, and (6.2) holds because $\_F\_i\,(n) \neq f^m \circ \_F\_j\,(n)$ implies that $\neg \exists z \in f^{m-1}(x_1)$ s.t. $\_F\_j\,(n) = z$, because otherwise $f^m(\_F\_j\,(n)) = f^m(z) = x_1 = \_F\_i\,(n)$ would violate the assumption. Thus the contradiction is established, and we obtain that any model of $\mathcal{S}$ satisfies $\_F\_i = f^m \circ \_F\_j$, which was the claim. □

With this encoding, we can define our new trace encoding. We only have to make a final small adjustment to the indexing used. Since, for each step, there are now two embedding encodings to consider, we will adopt the following indexing scheme. For each step i in the trace encoding,

- The application encoding and the graphs themselves will be indexed by i.
- The embedding into the central shape and the central shape itself will be indexed by $2i$.
- The embedding into the materialized shape and the materialized shape itself will be indexed by $2i + 1$.

With this in mind, we define the new trace encoding.

Code Template 39 (Unconstrained, Embedding Restricted Trace Encoding). *Let* $\mathcal{G} = (I, \mathcal{R})$ *be a GTS, and let* $\pi = (S_0, (S_0^m, P_0, m), S_1) \cdots (S_{k-1}, (S_{k-1}^m, P_{k-1}, m), S_k)$

be an abstract trace of length k in the corresponding modified STS. Then the corresponding concrete trace can be encoded using, using appropriate instances of Code Templates 3, 5, 6, 23, 11, 12, 14, 15, 16, 17, 21, 26, 27, and 38. For the complete template, see Code Template B.8, Appendix B, page 264.

This modified trace encoding can now be used as the base of the automatic abstraction refinement techniques described in the next section.

6.2 Automatic Abstraction Refinement via the Trace Encoding

To begin, we establish a simple sanity check to figure out whether an automatically generated constraint can be soundly added to a shape. Looking to Def. 91, we see three conditions. The first condition concerns the *usefulness* of the constraint, rather than its soundness, and the third condition is implied by the second. Thus, we focus on the second condition. For any central shape $\underline{S_i}$ to which we add a constraint, yielding the new shape $\underline{S_i'}$, the incoming shape must remain embedded, i.e.

$$\underline{S_i^t} \sqsubseteq \underline{S_i'}$$

must hold.

This is a simple embedding condition that can be checked using the already established methods used in the COVER-sub-algorithm. If this embedding check is positive, then we have established that no concrete graph that could conceivably be represented by an abstract trace ending in this node is omitted by the newly refined central shape. Equipped with this sanity check, we move on to the two automatic abstraction refinement techniques we will cover in this thesis: materialization-based refinement and interpolation-based refinement.

First, the – ostensibly – simple option: materialization-based refinement. The idea here is this: since the abstraction failure is expressible by a single materialization that does not cover any actual graphs in the concrete system, we can fix the abstraction by excluding the graph set defined by the materialization from the graph set defined by the central shape. This would require creating a shape constraint that evaluates to false exclusively on the target materialization (and on all shapes embedded into it).

While this is a very straightforward idea, it is complicated by the fact that the materializations created by a single match for a single rule (but different "keep sets" of summary nodes) are not independent of each other. In fact, the materializations in this scenario are linked together into a lattice structure by the subgraph relation. This is shown in Fig. 6.2.

Lemma 31 (Materialization Lattice). *Let S be a shape, let P be a rule and let m be a potential match in S for P. Let A be the set of summary nodes in the image of m,*

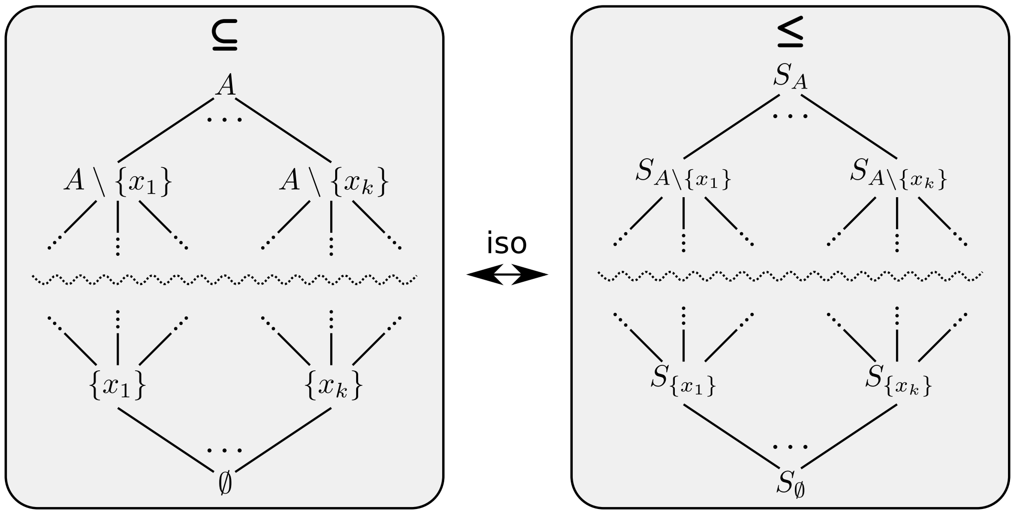


Figure 6.2: The materialization set for a given S, P, m forms a subgraph-lattice isomorphic to the subset lattice of the summary nodes touched by the match

and for any $I \subseteq A$, let S^m_I be the corresponding materialization. Then the set of all materializations for S, P, and m forms a lattice w.r.t. the subgraph relation $\leq$ that is isomorphic to the subset lattice on A.

Proof. Let 2^A be the power set of A, and let $\mathcal{S}^m$ be the set of materializations for S, P, and m. The subset relationship $\subseteq$ forms a partial order on all sets, and thus also does so on 2^A. The subgraph relationship forms a partial order on all shapes and thus also does so on $\mathcal{S}^m$.

We define f as the obvious bijective mapping between 2^A and $\mathcal{S}^m$:

$$\begin{aligned} f : 2^A &\rightarrow \mathcal{S}^m \\ X &\mapsto S^m_X \end{aligned}$$

Now, let $X, Y \in 2^A$ such that $X \subseteq Y$. Then, by Def. 46, $f(X) = S^m_X$ and $f(Y) = S^m_Y$ differ only in the summary nodes that they preserve and are otherwise identical. Specifically, S^m_Y preserves, by definition, all summary nodes that S^m_X preserves, and then preserves a set of additional nodes $Y \setminus X$. Thus we can obtain S^m_Y from S^m_X by adding the nodes in $Y \setminus X$ via the construction described in Def. 46. Therefore, S^m_X is a part of S^m_Y, i.e. $S_X \leq S_Y$.

On the other hand, let $S^m_X \leq S^m_Y$. Since $X \subseteq U_{S^m_X}$ and, by the definition of the subgraph relation (Def. 5), $U_{S^m_X} \subseteq U_{S^m_Y}$, we thus know that $X \subseteq U_{S^m_Y}$, and therefore, $X \subseteq Y$. □

The presence of this materialization lattice means that excluding singular materializations is bound to be very difficult. Consider a constraint (α, m) attached to the central shape $\underline{S}$ that is intended to exclude a materialization point $(\underline{S^m}, f^m)$. If that is to work, there must by an assignment m' for F (α) into $\underline{S^m}$ such that $[\![\alpha]\!]^{m'}_{S^m} = \mathbf{0}$ and $m = f^m \circ m'$.

That in itself is not problematic, of course, but if $\underline{S^m}$ is not the top element in the materialization lattice, then there is at least one other materialization point $(\underline{S'^m}, f'^m)$ such that $S^m \leq S'^m$. Furthermore, by construction, we have $f'^m\downarrow_{S^m} = f^m$ and thus m' has the property $m = f'^m \circ m'$. Therefore, constructing a constraint that excludes $\underline{S^m}$ but not $\underline{S'^m}$ is impossible without referencing nodes outside the assignment of the constraint, i.e. the use of quantifiers. In the face of the obvious decidability issues that we encounter when adding quantifiers to constraints, such a step should only be taken when absolutely necessary.

Instead of creating highly complex, quantified constraints, we recognize that abstraction errors that invalidate some elements of the materialization lattice but not others are created by node set constraints, i.e. the fact that at the concrete level, more or fewer nodes of a particular type are available than the particular materialization assumes. Such cases are better dealt with by manipulating the node set of the central shape, rather than adding constraints to it. We will therefore restrict ourselves to the most specific constraint we can create that remains unquantified.

Definition 94 (Negative Materialization Constraint). *Let $\underline{S}$ be a shape over a predicate set $\mathcal{P}$, let P be a rule, m be a potential match of P to $\underline{S}$, and let I be a subset of S, the set of summary nodes touched by m. Let N be the universe of the left hand side of P, and let ι^m be the interpretation defining the logical structure of the materialization $\underline{S^m}$ (see Def. 46). Then the* negative materialization constraint (α, ν) *for $\underline{S^m}$ is defined by*

$$\begin{aligned}
N' &:= \left\{n' \mid n \in I\right\} \\
\nu &:= m \cup \left\{\left(n', n\right) \mid n \in I\right\} \\
\alpha &:= \neg\left(\alpha_{set} \wedge \alpha_{un} \wedge \alpha_{bin}\right) \\
\alpha_{set} &:= \bigwedge_{\substack{n,m \in N \cup N' \\ n \neq m}} \neg\left(n = m\right) \\
\alpha_{un} &:= \bigwedge_{(s,1) \in \mathcal{P}} \left[\bigwedge_{\substack{n \in N_L \\ \iota_{S^m}(n) = \mathbf{1}}} s\left(n\right) \wedge \bigwedge_{\substack{n \in N_L \\ \iota_{S^m}(n) = \mathbf{0}}} \neg s\left(n\right)\right]
\end{aligned}$$

$$\alpha_{bin} := \bigwedge_{(s,2)\in\mathcal{P}} \left[\bigwedge_{\substack{n_1,n_2\in N_L \\ \iota_{S^m}(n_1,n_2)=\mathbf{1}}} s(n_1,n_2) \wedge \bigwedge_{\substack{n_1,n_2\in N_L \\ \iota_{S^m}(n_1,n_2)=\mathbf{0}}} \neg s(n_1,n_2) \right]$$

where f^m is the embedding of $\underline{S^m}$ into $\underline{S}$.

Intuitively speaking, α_{set} is true whenever the additional variables made for the summary nodes that the materialization preserves are assigned to nodes outside of the left hand side of the rule. The subformulas α_{un} and α_{bin} on the other hand, ensure that the part of the materialization that is mapped to the match really does instantiate the left hand side of the rule. Adding this constraint to a shape $\underline{S}$ will thus cause any materialization equal to $\underline{S^m}$ or covered by $\underline{S^m}$ to become infeasible. However, as implied by Lemma 31, it will also invalidate any materialization that is an upper bound for the target materialization in the materialization lattice. Often, this will not be a problem, as those materializations are also spurious. The sanity check introduced at the beginning of the chapter will indicate whether this is the case.

As a side note, this constraint can be used for applications other than oure abstraction refinement. If we remove the α_{set} part of the formula, we obtain a constraint that expresses the non-applicability of a graph transformation rule. An important extension of the basic graph transformation formalism is the so-called *negative application condition*, or NAC, which expresses that a rule should only be applied, if its left hand side matches a graph, *and* at the same time none of its NACS match the graph. It is thus clear that, using a simplified negative materialization constraint, support for NACs could be added to our theory. For now, we will refrain from doing so and focus on the basic formalism. We continue by asserting the correctness of the negative materialization constraint defined above.

Theorem 6 (Negative Materialization Constraints Exclude Materializations). *Let $\underline{S} = (S, \Lambda)$ be a shape, and let $\underline{S^m}$ be a materialization from that shape. Let (α, m) be the negative materialization constraint for $\underline{S^m}$, and let $\Lambda' := \Lambda \cup \{(\alpha, m)\}$. Then we have*

$$\mathcal{G}(\underline{S}) \setminus \mathcal{G}(S, \Lambda') = \mathcal{G}(\underline{S^m})$$

or, in other words, the graphs that are removed from the graph set of $\underline{S}$ by the negative materialization constraint are embedded into the materialization $\underline{S^m}$ or a materialization $\underline{S'^m}$ from the same lattice such that $\underline{S^m} \sqsubseteq \underline{S'^m}$.

Proof. See Appendix A, page 252. □

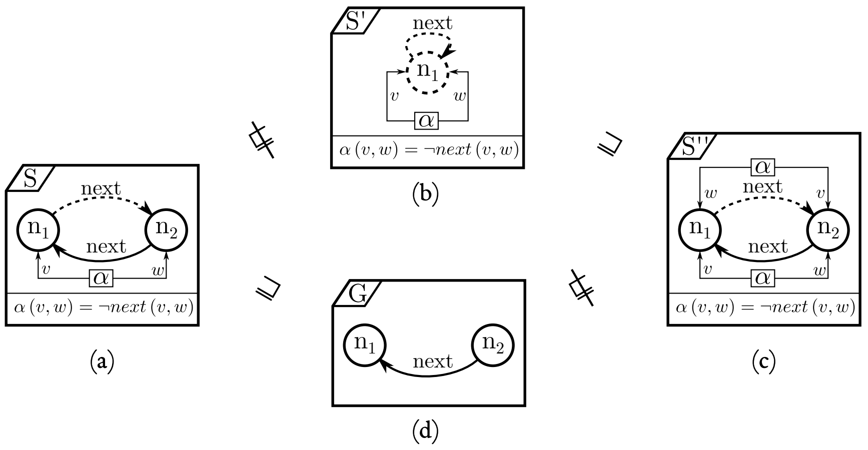


Figure 6.3: Merging of the variables of a constraint strengthens it

The advantage of this method is that it is easy to perform, and is always applicable. The disadvantage is that it is very specific to the materialization excluded, generates large constraints with many variables, which are harder to work with, and always encodes the entire structure of the rule as a negative constraint, even when only a specific, small part of it is the actual problem. That can lead to the abstraction becoming burdened with useless information. Also, since the materializations are created from the rule set, the kinds of constraints that this method can produce are rather limited.

These shortcomings motivate the second approach we will cover in this thesis. Inspired by the use of Craig Interpolation in other lazy abstraction approaches, we will use interpolants to construct new shape constraints. The idea here would be to interpolate the trace encoding after every step, and use the interpolants gained thereby to refine the central shapes of the respective STT nodes. This naïve approach, however, is made infeasible by several complications present in our domain of abstract graph transformations.

The first such complication was already hinted at in the definition of the canonical abstraction for constrained shapes (see Def. 64). A constraint attached to a shape does not easily transfer to a shape it is embedded into. To understand this, consider the effect that we intuitively associate with the transfer of a constraint from a shape $\underline{S}$ to a shape $\underline{S}'$ it is embedded into. A constraint removes a certain subset of graphs from the graph set of $\underline{S}$. Moving this constraint – soundly – to $\underline{S}'$, we would expect that the constraint continues to exclude the same graph set, thus not further restricting $\underline{S}'$. However, with the naïve method of merely projecting the constraint along the embedding, this is not what happens. For a quick example, consider the shape in Fig. 6.3.

It is clear that the shape under (a) has only one graph in its graph set: a graph with two nodes, n_1 and n_2, as well as a single *next* edge from n_2 to n_1, shown under (d). This is ensured by the constraint α, which eliminates another interpretation, in which n_1 and n_2 are connected by *next* in both directions. However, if we now increase the abstraction, as depicted under (b), and simply merge the nodes n_1 and n_2, this creates an additional constraint as an implication of the constraint on the abstracted shape. This is shown under (c). Since we have lost the knowledge in which direction the *next* edge was disallowed, it is now disallowed in both directions, which removes the original shape, as well as all graphs in its graph set, from the shape set of the abstracted shape. Thus, counterintuitively, merging nodes and thus using a more abstract shape in the presence of shape constraints will oftentimes *reduce* the size of the represented graph set, instead of increasing it.

This problem is relevant because our trace encoding encodes graphs at the concrete level, as well as materialized shapes at the abstract level. Thus, any interpolants obtained from the trace encoding are going to use terms local to a graph at the concrete level, a materialized shape, or both. In any of these cases, constraints derived from such interpolants will thus be attached, at best, to the materialized shape, and would then have to be transferred to the central shape.

The fact that the central shapes are part of the encoding does not help here. This is because, conceptually, as mentioned above, the point of failure in the correspondence between a concrete trace and an abstract trace will always be the embedding into the materialization. Thus, embeddings into central shapes or even unabstracted result shapes will never be relevant to the unsatisfiability of the trace encoding and would therefore be ignored by the interpolator.

Hence, we are forced to transfer the constraint from the materialized shape to the central shape, which might not always be possible.

The second complication comes into play when choosing the interpolation cuts to use. Consider the following partition of a trace encoding for an abstract trace containing two transition edges. The encoding of the initial graph, including overflow nodes is denoted by $||I||$, embedding encodings by $||embed_i||$, and rule application encodings by $||apply_i||$. Embedding encodings with odd indices are assumed to include an instance of the embedding correspondence encoding (CoT. 38). We will assume that the failure point is 2, i.e. only the embedding into the materialized error ($||embed_5||$) makes the trace infeasible.

$$\begin{array}{ccccccccc} ||I||\wedge & & ||embed_0||\wedge & & ||embed_1||\wedge & & ||apply_0||\wedge & & ||embed_2||\ldots \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & \end{array}$$

$$\begin{array}{ccccccccc} & (1) & & (2) & & (3) & & (4) & \\ \ldots\wedge & & \|embed_3\|\wedge & & \|apply_1\|\wedge & & \|embed_4\|\wedge & & \|embed_5\| \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\ & (5) & & (6) & & (7) & & (8) & \end{array}$$

If our goal is to obtain an interpolant which we can immediately use as a formula for a shape constraint, we need to make sure that the interpolant only uses symbols from the part of the trace formula that describes the materialized shape. However, at any of the eight interpolation points shown above, symbols of the concrete graph level are part of the symbol set intersection. This is because the transformation and persistence of these symbols must be continuous over the entire trace. Thus, at (1), (2), and (3), graph labels with the index 0, at (4), (5), and (6), graph labels with the index 1, and at (7) and (8), graph labels with the index 2 are part of the intersection. Clearly, if we are to eliminate graph labels from the interpolants, we need to use interpolation cuts at different places. We deal with both of these complications in the following way.

The problem of transferring a constraint from the materialized shape to the central shape of an STT node is ameliorated by taking measures to produce a constraint that is as simple as possible (see below), and using the constraint transfer mechanism of the canonical abstraction (see Def. 64) to move it to the central shape. Should this fail, we fall back on using the naïve constraint detailed in Def. 94. While this is not an ideal solution, it is a workable placeholder for a future abstraction mechanism that will replace it.

The problem of obtaining usable formulas from interpolating the trace is solved by a new interpolation point. The only feasible place for this, avoiding significant rearrangement of the terms of the encoding, is the failure point. For the purposes of interpolant generation, terms beyond the failure point may be omitted, since they cannot be part of the refutation proof that creates the interpolant. Thus, we can treat the embedding encoding at the failure point as the final term in the encoding. Within this embedding encoding, the encoding of the properties of the materialized shape is the only part that relies exclusively on terms from the materialized shape. These are instances of Code Templates 14 and 15. Since conjunctions are commutative, we can move these terms to the very end of the embedding encoding, and insert a new interpolation cut just before these terms. This changes $\|embed_5\|$ in our example to

$$\underbrace{\underbrace{\|corr\_4\_5\|}_{\text{CoT 38}} \wedge \underbrace{\|func_5\|}_{\text{CoT 11}} \wedge \underbrace{\|summ_5\|}_{\text{CoT 12}} \wedge \underbrace{\|unAbs_5\|}_{\text{CoT 16}} \wedge \underbrace{\|binAbs_5\|}_{\text{CoT 17}}}_{:=\|comp_5\|} \wedge \underbrace{\underbrace{\|unS_5\|}_{\text{CoT 14}} \wedge \underbrace{\|binS_5\|}_{\text{CoT 15}}}_{:=\|prop_5\|},$$

thus changing the encoding of the whole trace to

$$\begin{aligned}&\|I\| \wedge \|embed_0\| \wedge \|embed_1\| \wedge \|apply_0\| \wedge \|embed_2\| \ldots\\ &\quad \ldots \wedge \|embed_3\| \wedge \|apply_1\| \wedge \|embed_4\| \wedge \|comp_5\| \underset{\substack{\uparrow\\(9)}}{\wedge} \|prop_5\|\end{aligned}$$

which introduces the interpolation point (9). At this point, any solver computing an interpolant is forced by the restricted vocabulary of the interpolant to construct it using only the nodes and edges of the materialized shape.

From such an interpolant, the creation of a new shape constraint is trivial. Each occurrence of a reference to a node of the shape in the interpolant is replaced with a variable, the reference preserved by a corresponding mapping in the assignment of the constraint.

Note, however, that the actual abstraction error might be the under- or over- estimation of the sizes of certain node sets represented by (summary-) nodes. The interpolation cut described above, in contrast, attempts to express the abstraction error in terms of the edges found in the materialization at the failure point. As an example, consider again the example shown in Fig. 6.1. Here, the interpolant might attempt to express the fact that there are two nodes that have to be embedded into one non-summary node by expressing that the unary edges on all the other nodes prevent it from embedding the redundant node into those nodes, which is clearly not an ideal outcome.

Observant readers may have noticed that we have already taken action against this. The embedding correspondence encoding described in CoT. 38, together with the inclusion of the embeddings into the central shapes into the trace encoding, cause the trace encoding to actually become infeasible before reaching the interpolation point if the node set itself prevents the trace from continuing. At the point indicated here,

$$\|corr\_i-1\_i\| \wedge \|func_i\| \wedge \|summ_i\| \underset{\substack{\uparrow\\(*)}}{\wedge} \|unAbs_i\| \wedge \|binAbs_i\| \wedge \|unS_i\| \wedge \|binS_i\|$$

the functions that could possibly be embeddings into the shape $\underline{S_i^m}$ have already been reduced to surjective functions satisfying the summarization property that are f_i^m-factors of an embedding into $\underline{S_i}$. If $\underline{S_i^m}$ discarded a summary node into which more than one node was embedded by all embeddings into $\underline{S_i}$, the encoding will become unsatisfiable at this point. The following lemma states this formally.

Lemma 32 (Node Set Error Detection). *Let φ be an encoding of an abstract error trace* *according to CoT. 39. Let f be the failure point of φ, let $\underline{S_{2f}}$ and $\overline{S^m_{2f+1}}$ be the central and materialized shape at the failure point, and let φ_f be the $2f+1$-prefix of φ. Let $\|embed_{2f+1}\|$ be the encoding of the terminal embedding of φ_f, and let it be partitioned as follows:*

$$\underbrace{\underbrace{\|corr\_j-1\_j\|}_{CoT\ 38} \wedge \underbrace{\|func_j\|}_{CoT\ 11} \wedge \underbrace{\|summ_j\|}_{CoT\ 12}}_{:=\|node_j\|} \wedge \underbrace{\underbrace{\|unAbs_j\|}_{CoT\ 16} \wedge \underbrace{\|binAbs_j\|}_{CoT\ 17} \wedge \underbrace{\|unS_j\|}_{CoT\ 14} \wedge \underbrace{\|binS_j\|}_{CoT\ 15}}_{:=\|nonnode_j\|},$$

where $j := 2f+1$. Let F be the set of valid embeddings from the graphs in Γ_f into $\underline{S_{2f}}$. If

$$\neg\left[\exists f \in F, g : U_{\Gamma_f} \to U_{S^m_j} : \left(f = f^m_j \circ g\right)\right],$$

then

$$\varphi_{j-1} \wedge \|node_j\|$$

is unsatisfiable.

Proof. From CoT. 11, CoT. 12, and Lemma 10 we know that every model for $\|func_j\| \wedge \|summ_j\|$ is a surjective function $\_F\_j : U_{\Gamma_j} \to U_{S^m_j}$ satisfying the summarization property. Due to the guaranteed presence of the *future-* and *past-*summary nodes in the target shape and the fact that U_{Γ_j} is guaranteed to contain enough nodes to achieve surjectivity (see CoT. 23), such functions always exist. From Lemma 30, we know that exactly those functions $\_F\_j$ such that $\exists f \in F : f = f^m_j \circ \_F\_j$ satisfy $\|corr\_j - 1\_j\|$. Since these are exactly the functions that the assumption states do not exist, we get the claim. □

We (can) thus only proceed with the construction of an interpolation-based constraint, if the trace encoding is satisfiable up to the interpolation point. If it is not, we deduce that a node set error occurred and fall back to the materialization based constraint. If it is, we construct the interpolation-based constraint using the following definition.

Definition 95 (Interpolant-Based Constraint). *Let φ be an encoding of an abstract error trace according to CoT. 39. Let f be the failure point of φ, let $j := 2f+1$, and φ_j be the j-prefix of φ. Let $\|embed_j\|$ be the encoding of the terminal embedding of φ_j, and let $\|comp_j\|$ and $\|prop_j\|$ be its decomposition into CoT. 38, 11,12,16,17, and CoT. 14,15, respectively. Let $I \not\equiv \mathbf{0}$ be the interpolant obtained from interpolating between $\varphi_{j-2} \wedge \|embed_{j-1}\| \wedge \|comp_j\|$ and $\|prop_j\|$. Then the* Interpolant-Based

Constraint *obtained from φ is a constraint (α, m) such that*

$$\alpha := I\,[n_1/x_1]\,[n_2/x_2]\ldots[n_k/x_k]$$
$$m := [x_1 \mapsto n_1, x_2 \mapsto n_2, \ldots, x_k \mapsto n_k]$$

where for a formula ψ, $\psi\,[x/y]$ denotes the formula ψ with all occurrences of x replaced by y, and $n_1, \ldots, n_k$ are the nodes of the materialized shape $\underline{S_j^m}$ at the failure point.

Since this kind of constraint is created from an interpolant that by definition must be incompatible with the right hand side of the formula that created it, it must, by construction, evaluate to **0** on the materialized shape.

Lemma 33 (Interpolant-Based Constraint Excludes Materialization). *Let π be a root path through an STT $\mathcal{T}$ ending in an STT node with an error materialization point. Let the failure point be i, and let $(\underline{S}^m, f^m)$ be the materialized shape at i. Let (α, m) be the interpolant-based constraint obtained from this path and materialization point, as defined in Def. 95. Then $\underline{S^m} \not\models (\alpha, m)$.*

Proof. We know from the nature of interpolation that α is implied by the prefix of the path, yet incompatible with the embedding encoding for $\underline{S^m}$. Since α expressed in terms of the nodes and edges of $\underline{S^m}$, that can only mean that α evaluates to **0** on $\underline{S^m}$ with m. ☐

We will now focus on the soundness of adding the constraint to the central shape. In order to make predictions about this soundness, we must first specify the exact process, inspired by the canonical abstraction, by which the interpolation-based constraint, if possible, is transported from the materialized shape to the central shape of the STT node. This process is comprised of two steps.

1. In the first step, the constraint is evaluated with respect to the central shape via the materialization embedding. This will cause any literals in the constraint that would evaluate to a definite value on the central shape to be replaced with a constant of that value.
2. In the second step, we check whether any of the assignments of this modified constraint are actually merged by projecting the constraint onto the central shape. If so, it is unknown whether the constraint is sound and we have to perform the sanity check described at the beginning of the section. If not, the constraint is added to the central shape.

If the sanity check becomes necessary and deems the constraint unsound, we fall back to the negative materialization constraint as specified by Def. 94. We begin the formalization of this idea by defining basic constraint projection, i.e. the moving of a shape constraint from a shape to another shape it is embedded into.

Definition 96 (Constraint Projection). *Let $\underline{S} = (S, \Lambda)$ be a constrained shape, i.e. $\Lambda \neq \emptyset$, and let $(\alpha, m) \in \Lambda$. Let further $\underline{S'}$ be a shape such that $\underline{S} \sqsubseteq_f \underline{S'}$. Then the* projected constraint $\left(\alpha, m^f\right)$ *is defined by $m^f = f \circ m$.*

Having established the concept of constraint projection, we move on to the evaluation of constraints on logical structures other than the one they are attached to. This is called *projected evaluation*.

Definition 97 (Projected Evaluation of Constraints). *Let $\underline{S} = (S, \Lambda)$ be a constrained shape, i.e. $\Lambda \neq \emptyset$, and let $(\alpha, m) \in \Lambda$. Let further $\underline{S'}$ be a shape such that $\underline{S} \sqsubseteq_f \underline{S'}$. Then the* projected evaluation *of the constraint $(alpha, m)$ is defined by $\left(\alpha', m{\downarrow_{\mathrm{F}(\alpha')}}\right)$, where α' is obtained from α by replacing all predicate literals $p(x_1, \ldots, x_k)$ as follows*

$$p(x_1, \ldots, x_k) \mapsto \begin{cases} c := \iota_{S'}(p)(f \circ m(x_1), \ldots, f \circ m(x_k)) & \textit{, if } c \neq ½ \\ p(x_1, \ldots, x_k) & \textit{else} \end{cases}$$

Thus, in order to obtain a shape constraint for the central shape from the interpolant, the interpolant is transformed into a shape constraint (α, n), by attaching it to the materialization at the failure point via the obvious mapping. Then, its projected evaluation is computed with respect to the central shape, yielding a new constraint (α', m'). If $f^{-1} \circ f \circ m' = m'$, i.e. applying materialization embedding, followed by its inversion onto the assignment of the new constraint yields the same assignment, (α', m') is projected onto the central shape.

Projecting constraints along locally bijective embeddings like this is sound, as the following lemma shows.

Lemma 34 (Bijectively Projected Constraints Are Sound Abstractions). *Let G be a graph, let $\underline{S'}$ be a shape, and let f' be a surjective function $f' : U_G \to U_{S'}$. Let (α, m) be a constraint such that $G \models concr_{f'}(\{(\alpha, m)\})$. Let further $\underline{S}$ be a shape such that $\underline{S'} \sqsubseteq_f \underline{S}$ such that $f^{-1} \circ f \circ m = m$ and $f \circ f' = g$ with $G \sqsubseteq_g \underline{S}$. Then*

$$G \models concr_{f \circ f'}(\{(\alpha, f \circ m)\})$$

Proof. Since we know that $f^{-1} \circ f \circ m$ holds, we have

$$\begin{aligned} \mathrm{concr}_g(\{(\alpha, m)\}) &= \mathrm{concr}_{f \circ f'}(\{(\alpha, m)\}) \\ &= \mathrm{concr}_{f^{-1} \circ f \circ f'}\left(\left\{\left(\alpha, f^{-1} \circ f \circ m\right)\right\}\right) \end{aligned}$$

$$= \mathrm{concr}_{f'}\left(\{(\alpha, m)\}\right)$$

and thus the claim. □

We now know that, under certain circumstances, we can soundly project the interpolant-derived constraint from the materialized shape to the central shape. However, the question remains whether the constraint thusly added to the central shape could potentially exclude graphs that exist in the concrete system, making the refinement unsound. Intuitively, it seems obvious that this cannot be - the constraint was created from an interpolant which was implied by the trace encoding that encoded (among other things) the concrete graphs existent at that particular point in the tree. Thus, it has to evaluate to true on all concrete graphs present. The following lemma states this.

Lemma 35 (Interpolant-Based Constraint is Sound). *Let π be a root path through an STT $\mathcal{T}$ ending in an STT node with an error materialization point $(\underline{S}^m, f^m)$. Let k be the failure point and let, for each prefix π_i of π, $\Gamma(\pi_i)$ be the set of graphs produced by all concrete traces corresponding to π_i. This is equivalent to the i-local part of all models of π_i. Let (α, m) be the interpolant-based constraint obtained from this path and failure point, as defined in Def. 95, and let all $(\beta, n) \in \underline{\Lambda_{S_k}}$ be valid refinements in the sense of Def. 91 and Lemma 28. Then we have*

$$\forall G \in \Gamma(\pi_k) : G \sqsubseteq \left(S_k, \underline{\Lambda_{S_k}} \cup \{(\alpha, m)\}\right),$$

i.e. adding the interpolant-based constraint to the central shape of the STT node at the failure point does not exclude any graphs that actually exist at the concrete level.

Proof. Let $G \in \Gamma(\pi_k)$. Let now A be the formula to the left of the interpolation cut, and B the formula to the right of it. Then we have

$$\begin{aligned} A &= \|I\| \wedge \ldots \wedge \|step_{k-1}\| \wedge \|embed_2 k\| \wedge \|comp_{2k+1}\| \\ B &= \|unS_{2k+1}\| \wedge \|binS_{2k+1}\| \end{aligned}$$

The interpolant I is implied by A, yet $\Gamma(\pi_k)$ is already set by $\|start\| \wedge \|step_1\| \wedge \ldots \wedge \|step_{k-1}\|$. We will now prove that the remainder of A does not further restrict $\Gamma(\pi_k)$. As a shorthand, we will set $j := 2k + 1$. By CoT. 39, G is embedded into the central shape $\underline{S_{j-1}}$ by the embedding function f_{j-1} provided by the model for $\|embed_{j-1}\|$. The formulas $\|func_j\|$ and $\|summ_j\|$ encode a surjective function f_j that obeys the summarization properties of an embedding. $\Gamma(\pi_k)$ places no restrictions on f_j, and a model for f_j always exists, since CoT 23 (`overflow`) ensures that the concrete graph domain always has enough nodes to support a surjective function into any materialization along the trace. Furthermore, `corr_`j − `1_`j ensures (see

Lemma 30) that any such f_j satisfies $f_{j-1} = f_j^m \circ f_j$, where f_j^m is the materialization embedding for step k. The formulas $\|unAbs_j\|$ and $\|binAbs_j\|$ enforce that the graphs in $\Gamma(\pi_k)$ must mirror the properties of the materialized shape. However, they do not specify these properties, and therefore, they alone also do not further restrict $\Gamma(\pi_k)$. Thus, any model of A contains the full $\Gamma(\pi_k)$, as well as a surjective function f_j from the concrete node set into the materialized shape that preserves the summarization property and is an f_j^m-factor of the previously encoded embedding f_{j-1}.. Therefore, for any such f_j, we have

$$\forall G \in \Gamma(\pi_k) : G \models \mathrm{concr}_{f_j}\left(\{(I, \mathrm{id}_{U_{S^m}})\}\right),$$

where I is the interpolant, i.e. $A \models I$ and $I \wedge B \models \mathbf{0}$. Now, let (α, g') be the result of applying projected evaluation to (I, id), and let $f_j^{m-1} \circ f_j^m \circ g' = g'$. Let $m := f_j^m \circ g'$. Then we have

$$\begin{aligned} G &\models \mathrm{concr}_{f_j}\left(\{((\alpha, g'))\}\right) \\ &= \mathrm{concr}_{f_j}\left(\mathrm{concr}_{f_k^m}\left(\{(\alpha, m)\}\right)\right) \\ &= \mathrm{concr}_{f_j^m \circ f_j}\left(\{(\alpha, m)\}\right) \\ &= \mathrm{concr}_{f_{j-1}}\left(\{(\alpha, m)\}\right) \end{aligned}$$

Therefore, we can conclude that

$$\forall G \in \Gamma(\pi_k) : G \sqsubseteq \left(S_k, \underline{\Lambda_{S_k}} \cup \{(\alpha, m)\}\right),$$

holds. □

Finally, we return to the original intention of the added constraint - its objective is to exclude the materialization that was identified as the point of failure. Since the interpolant will by construction evaluate to false on the materialization, and the projection (if it occurs) does not detract from that, it seems clear that this is the case. Nevertheless, the following lemma formalizes this intuition.

Lemma 36 (Interpolation-Based Constraint Excludes Materialization at Failure Point). *Let π be a root path through an STT $\mathcal{T}$ ending in an STT node with an error materialization point $(\underline{S}^m, f^m)$. Let k be the failure point, $\underline{S_k}$ the central shape, and $\underline{S_k^m}$ be the materialized shape at the failure point. Let (α, m) be the interpolant-based constraint obtained from this path and failure point, as defined in Def. 95. Then we have*

$$\mathcal{G}\left(S_k^m, \underline{\Lambda_{S_k}} \cup \mathrm{concr}_f\left(\{(\alpha, m)\}\right)\right) = \emptyset$$

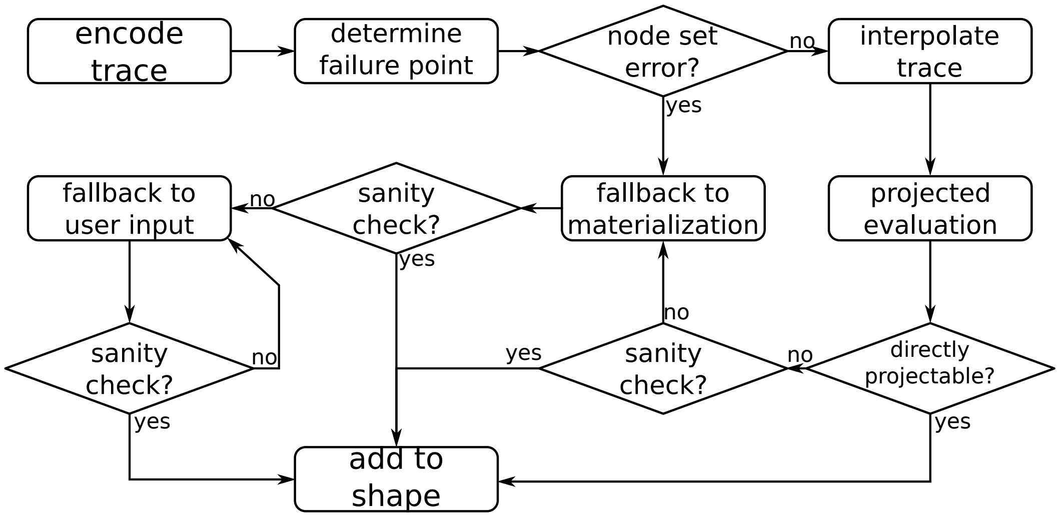


Figure 6.4: Schematic of the automatic refinement process

Proof. This follows directly from Lemma 33 and the fact that any interpolant-based constraint attached to the central shape will have been projected using a locally bijective embedding, making $\mathrm{concr}_f\left(\{(\alpha, m)\}\right)$ a singleton set. $\square$

Thus, it is both useful and sound to add the interpolant-based constraint to the central shape, if possible. This implies that the interpolant-based constraint is a valid refinement.

Theorem 7 (Interpolant-Based Constraints are Valid). *The interpolant-based constraint described in Def. 95 is a valid refinement in the sense of Def. 91.*

Proof. This follows directly from Lemmas 35 and 36. $\square$

Taken together, these definitions allow us to establish the process for abstraction refinement sketched in Fig. 6.4. We begin by encoding the trace that represents the potential error†. We then determine the failure point as specified by Def. 92. Checking for satisfiability at the point given in Lemma 32 then allows us to determine whether node set issues play a role. If they do, we skip interpolation and continue with a materialization-based refinement. Otherwise, we interpolate the trace and extract the interpolant-based constraint as specified by Def. 95. This constraint is then attached to the materialized shape and evaluated with respect to the central shape, as defined in Def. 97. For the

†we assume the trace is unsatisfiable, since otherwise the refinement process would not have been started

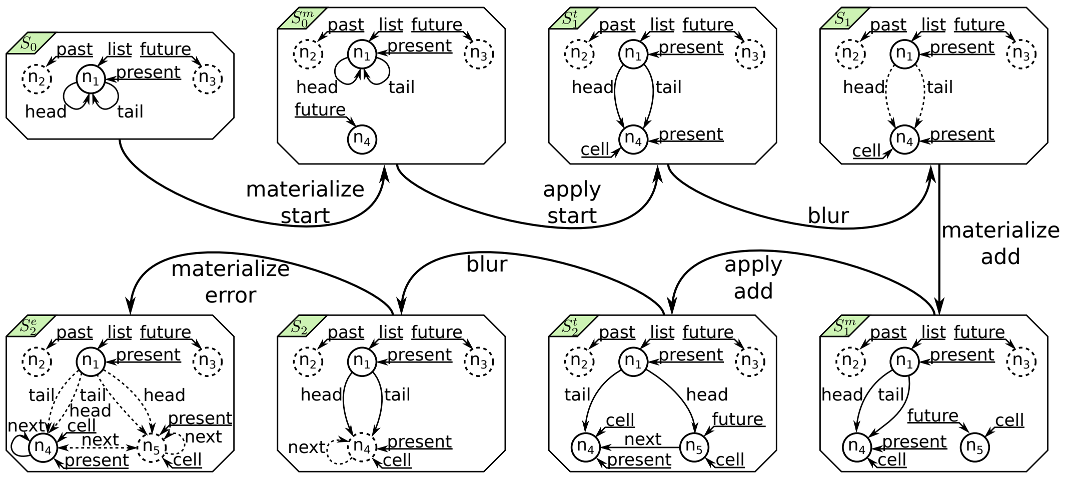


Figure 6.5: Applying $start$, then add creates an error in the initial abstraction

result, we check whether the conditions for Lemma 34 are given. If so, we can immediately use the resulting constraint for the central shape. If not, we check whether the adding of this constraint would violate the refinement conditions given in Def. 91. Should this check reveal that the constraint is valid, we add it to the shape and are done. Otherwise, we discard the constraint and construct a materialization-based constraint, which we then similarly check. In case the resulting constraint is valid, we add it to the shape. If that is not the case, we have to fall back to the user to provide a valid set of constraints. In this, we support the user by checking the constraints they provide with the same sanity check we used for the automatic refinement, and loop this until valid constraints are provided. Those constraints are then added to their respective shapes and the process ends.

As an example of the fairly complex process of adding an interpolation-based constraint to a central shape along an error path, consider the error path shown in Fig. 6.5. In our linear list example, unfolding the STT with the basic exploration loop detailed in Sec. 5.3 and the initial abstraction (no constraints), this is the first error path one encounters. The STT recognizes that applying the rule *start* once yields a linear list with one content cell. However, when the rule *add* is then applied, two content cells exist, connected via a *next* edge, which the canonical abstraction will blur into a single, summary content cell with a summary *next* self-edge. Thus, the information that there were only two nodes without self-edges is lost, and the error pattern potentially matches.

Knowing the concrete graph transformation system, however, we can immediately see that this error is spurious - it does not exist in the GTS itself. Thus, the SMT encoding of this trace returns UNSAT. Interpolating as defined in Def. 95 gives us the interpolant[‡]

$$\begin{aligned}\alpha &\equiv (\mathit{next}\,(x,x) \to (\mathit{past}\,(x) \vee (\mathit{cell}\,(x) \to (\mathit{next}\,(x,x) \to (\mathit{past}\,(x) \vee \neg\mathit{cell}\,(x))))))\\ &\quad \vee (\mathit{next}\,(x,x) \to \neg\mathit{cell}\,(x))\\ &\equiv (\neg\mathit{next}\,(x,x) \vee (\mathit{past}\,(x) \vee (\neg\mathit{cell}\,(x) \vee (\neg\mathit{next}\,(x,x) \vee (\mathit{past}\,(x) \vee \neg\mathit{cell}\,(x))))))\\ &\quad \vee (\neg\mathit{next}\,(x,x) \vee \neg\mathit{cell}\,(x))\\ &\equiv \neg\mathit{next}\,(x,x) \vee \mathit{past}\,(x) \vee \neg\mathit{cell}\,(x) \vee \neg\mathit{next}\,(x,x) \vee \mathit{past}\,(x) \vee \neg\mathit{cell}\,(x)\\ &\quad \vee \neg\mathit{next}\,(x,x) \vee \neg\mathit{cell}\,(x)\\ &\equiv \neg\mathit{next}\,(x,x) \vee \mathit{past}\,(x) \vee \neg\mathit{cell}\,(x)\end{aligned}$$

which expresses that the node represented by x, which in this case is the *cell* node n_4 in the materialization S_2^e, cannot have a self *next*-edge and not be marked either *past* or not *cell* at the same time. Attaching this to the materialized shape, we get the constraint

$$(\neg\mathit{next}\,(x,x) \vee \mathit{past}\,(x) \vee \neg\mathit{cell}\,(x)\,, [x \mapsto n_4])$$

Now, following along with the process detailed above, we evaluate this constraint in the central shape at the failure point, and find that the values of *cell* and *past* are already fully determined there. This leads to a modified constraint.

$$\begin{aligned} &(\neg\mathit{next}\,(x,x) \vee \mathit{past}\,(x) \vee \neg\mathit{cell}\,(x)\,, [x \mapsto n_4])\\ \text{becomes } &(\neg\mathit{next}\,(x,x) \vee \mathbf{0} \vee \mathbf{0}, [x \mapsto n_4])\\ = &(\neg\mathit{next}\,(x,x)\,, [x \mapsto n_4])\end{aligned}$$

This is still invalid in the materialized shape, but expresses exactly the property about the concrete graphs at this point that we would like to add to the shape. However, since the assignment m goes to node n_4, which is merged with node n_5 in the materialization embedding, the condition for Lemma 34, i.e. $f^{m-1} \circ f^m \circ m$ is not given. Following the process, we project the constraint onto S_2 anyway, and perform a sanity check. We obtain

$$\underline{S_2'} := (S_2, \{(\neg\mathit{next}\,(x,x)\,, [x \mapsto n_4])\})\,.$$

For the sanity check, we now must determine whether S_2^t remains embedded, i.e. if

[‡]This is an actual interpolant obtained by experiment with the implementation presented in Chapter 7

$S_2^t \sqsubseteq \underline{S_2'}$ holds. We encode

$$\|S_2^t\| \wedge \neg \| \operatorname{concr}\left(\{(S_2, \{(\neg \mathit{next}(x,x), [x \mapsto n_4])\})\}\right) \|,$$

using Code Templates 37 and 36, searching for graphs embedded into S_2^t that do not satisfy the new implied constraints and obtain UNSAT, thus establishing $S_2^t \sqsubseteq \underline{S_2'}$. Therefore, adding the constraint directly to the central shape is sound and the error pattern can now no longer be materialized from this shape.

Having refined the shape at the failure point, all that is left is to modify the remainder of the STT to account for the changes. At this point, any other materialization points that may already exist become invalid, since they do not account for this new constraint. This is easily fixed by projecting the new constraint downward, using the function concr_f, into all other existing materialization points. While this fixes the materialization points, the transitions that might be attached to those materialization points have to be modified, too. In the case of fully automatic abstraction refinement, as described so far, we will achieve this by simply pruning the STT at the failure point, i.e. removing all transition edges leading out from it, and recomputing those transitions. If user-generated constraints are used, multiple consecutive shapes are possible refined with constraints. In that case, we proceed just as in the automatic case, with the one difference that the transition edges that link the refined shapes are retained, instead of removed.

Before we move on to formally define the error handling algorithm, we introduce a new partitioning of the trace encoding that will simplify failure point and node set error detection.

Definition 98 (Error Trace Partitioning). *Let φ be an instance of CoT. 39. Then it can be partitioned in the following way:*

$$\|I\| \wedge \varphi_0 \wedge \ldots \wedge \varphi_{2k+1}$$

where $\|I\|$ encodes the start graph and the overflow encoding (if present) and thus consists of instances of CoT. 3, 5, 6, and 23, while φ_i is defined by

$$\begin{array}{ll}
\underbrace{\|embed_i\|}_{CoT.\ 18} \wedge \underbrace{\|corr\_i\_i+1\|}_{CoT.\ 38} \wedge \underbrace{\|func_{i+1}\|}_{CoT.\ 11} \wedge \underbrace{\|summ_{i+1}\|}_{CoT.\ 12} & \mathit{if}\, i\%2 = 0 \\
\underbrace{\|unAbs_i\|}_{CoT.\ 16} \wedge \underbrace{\|binAbs_i\|}_{CoT.\ 17} \wedge \underbrace{\|unS_i\|}_{CoT.\ 14} \wedge \underbrace{\|binS_i\|}_{CoT.\ 15} \wedge \underbrace{\|apply_{i\backslash 2}\|}_{CoT.\ 26,27} & \mathit{if}\, i\%2 = 1 \wedge i < 2k+1 \\
\underbrace{\|unAbs_i\|}_{CoT.\ 16} \wedge \underbrace{\|binAbs_i\|}_{CoT.\ 17} \wedge \underbrace{\|unS_i\|}_{CoT.\ 14} \wedge \underbrace{\|binS_i\|}_{CoT.\ 15} & \mathit{if}\, i\%2 = 1 \wedge i = 2k+1
\end{array}$$

This partitions the trace encoding by splitting each step consisting of two embeddings and one rule application along the divider used in Lemma 32, thus facilitating both failure point and node set error detection.

As Fig. 6.4 shows, at some point, we may have to rely on the user to provide shape constraints along a path to achieve refinement. The particulars of how this can be done are relegated to Sec. 6.3, but for the purposes of integration into the larger error handling algorithm, we will define the behavior of a valid interactive user refinement process here already, as we have already done for EXPAND, EXPLORE, COVER, CHECK-FESSIBILITY, and HANDLEERROR. The procedure that queries the user for shape constraints for refinement is called QUERYUSER.

Definition 99 (Valid QUERYUSER-algorithm). *Let A be an algorithm operating on an STT $\mathcal{T}$ for a GTS $\mathcal{G}$, taking a root path $\pi_k = e_1 \cdots e_k$ with*

$$e_i = \left(n_{i-1}, \left(P_{i-1}, \left(\underline{S^m_{i-1}}, f^m_{i-1}\right)\right), n_i\right)$$

through $\mathcal{T}$ and a materialization points $\left(\underline{S^m_k}, f^m_k\right)$ of the terminal node of the root path as input. A is called a valid **QUERYUSER**-algorithm*, if, given that $\Gamma_k \cap \mathcal{G}\left(\underline{S^m_k}\right) = \emptyset$ and $\Gamma_{k-1} \cap \mathcal{G}\left(\underline{S^m_{k-1}}\right) \neq \emptyset$, it returns a sequence*

$$((\alpha_j, m_j), \ldots, (\alpha_k, m_k))$$

of shape constraints for some $0 \leq j \leq k$ such that

$$\begin{aligned} \underline{S^t_i} &\sqsubseteq \underline{S'_i} && \forall j \leq i \leq k, \text{ where} \\ \underline{S'_{i-1}} &\xrightarrow{P_{i-1}, \left(\underline{S'^m_{i-1}}, f^m_{i-1}\right)} \underline{S^t_i} && \forall j < i \leq k, \text{ and} \\ \mathcal{G}\left(\underline{S'^m_k}\right) &= \emptyset \end{aligned}$$

with $\underline{S'_i} := (S_i, \Lambda_i \cup \{(\alpha_i, m_i)\})$ and $\underline{S'^m_i} := \left(S^m_i, \Lambda^m_i \cup \operatorname{concr}_{f^m_i}(\{(\alpha_i, m_i)\})\right)$.

Intuitively, this definition requires that a valid algorithm that queries the user for a refinement of constraints produces a series of constraints such that the soundness of shape transformations remains given along the refined path, and the failure point materialization is excluded. Note, specifically, that $j = k$ is allowed, i.e. the user must not necessarily generate a whole sequence of constraints – if a single constraint will work, that is fine as well. Of course, the algorithm itself in this scenario has no direct control over the properties of the constraints produced. Thus, this definition requires the QUERYUSER algorithm to incorporate some form of checking and feedback loop to make sure that the constraints that are returned do conform to the requirements.

Listing 6.1: CheckEmbedded sub-algorithm

```
Input: Shape $\underline{S_1}$, Shape $\underline{S_2}$
  $\varphi$ <- EncodeFeasibility($\underline{S_1}$); // see $CoT.$ 37
  $\psi$ <- EncodeConstraints($\underline{S_2}$); // see $CoT.$ 36
  result <- Solve($\varphi \wedge \neg\psi$); // any SMT-compatible solver
  if (result == UNSAT) then
    return true;
  else
    return false;
  fi;
```

To simplify the description of the error handling algorithm, we will encapsulate (most) actions performed directly on the tree in the sub-algorithm `ModifyTree`, shown in Listing 6.2, as well as use a sub-algorithm called `CheckEmbedded`, shown in Listing 6.1, to encapsulate the embedding check between two shapes. The following lemma establishes the effect of the `ModifyTree` algorithm.

Lemma 37 (Correctness of ModifyTree). *Given a sequence of unary edges*

$$\sigma = \left(\left(n := \left(\underline{S_1^{in}, \underline{S_1}, M_1}, x_1\right), \ldots, \underline{S_k^{in}, \underline{S_k}, M_k}, x_k\right)\right),$$

of an STT $\mathcal{T}$ and a sequence of constraints

$$\kappa = \left((\alpha_1, m_1), \ldots, (\alpha_k, m_k)\right),$$

such that

$$\begin{aligned} \underline{S_i^t} &\sqsubseteq \underline{S_i'} && \forall j \leq i \leq k, \textit{ where} \\ \underline{S_{i-1}'} &\xrightarrow{P_{i-1}, \left(\underline{S'^{m}_{i-1}}, f^{m}_{i-1}\right)} \underline{S_i^t} && \forall j < i \leq k, \textit{ and} \\ \mathcal{G}\left(\underline{S'^{m}_{k}}\right) &= \emptyset \end{aligned}$$

holds, the `ModifyTree` *algorithm modifies $\mathcal{T}$ in such a way that Condition b of Def. 90 is established.*

Proof. See Appendix A. □

This is easy to see, since the algorithm literally just adds the constraints in the right places, and removes all parts of the tree that have to be recomputed.

Listing 6.2: ModifyTree sub-algorithm

```
Input: Unary Edges $\sigma = ((n_1, x_1), \ldots (n_k, x_k))$, Constraints $\kappa = ((\alpha_1, m_1), \ldots, (\alpha_k, m_k))$
  for each $\left(n_i = \left(\underline{S_i^{in}}, \underline{S_i}, M_i\right), x_i\right) \in \sigma$ do
    $\Lambda_i$ <- $\Lambda_i \cup \{(\alpha_i, m_i)\}$; // add constraint to central shape
    for each $((S, \Lambda), f) \in M_i$ do
      $\Lambda$ <- $\Lambda \cup \{\mathrm{concr}_f(\{(\alpha_i, m_i)\})\}$; // project constraint onto materialization
      if $(\exists (a, l, b) : l = (P, ((S, \Lambda), f)))$ then
        $\mathcal{T}$ <- $\mathcal{T} \setminus \mathcal{T}_b$; // prune subtree behind materialization
      fi;
    od;
    // remove outgoing edges from node except for the edge within the sequence
    if $(i < k)$ then
      $E^2$ <- $E^2 \setminus \{(a, l, b) \mid a = n_i \land b \neq n_{i+1}\}$;
    else
      $E^2$ <- $E^2 \setminus \{(a, l, b) \mid a = n_i\}$;
    fi;
    // remove superfluous covering edges
    $\mathcal{C}_\mathcal{T}$ <- $\mathcal{C}_\mathcal{T} \setminus \{(x, l, y) \mid (x, l, y) \in \mathcal{C}_\mathcal{T} \land y = n_i\}$;
  od;
  // remove superfluous covering edge labels
  $l_{c\mathcal{T}}$ <- $l_{c\mathcal{T}}\downarrow_{\mathcal{C}_\mathcal{T}}$;
```

We now have all the automatic pieces of the error-handling algorithm in place. All that remains is to formally define it. Listing 6.3 shows a Pseudocode implementation of the algorithm. We will now go through the algorithm line by line and work out how it achieves its objective.

The algorithm works as follows. In lines 2 and 3, it determines the root path of the error node and encodes it using CoT. 39 using the partitioning described in Def. 98. It then launches a loop (line 4) that consecutively adds the parts of the encoding to an SMT script and queries a solver to determine satisfiability (line 5).

If the result is SAT, then the if-clause from line 6 through line 33, which makes up the entire loop body, is skipped. As we will see, once this branch is entered it is always left via a `return` statement, meaning that lines $35 - 37$ are only executed when the entire trace was SAT. Consequently, these lines extract the model from the solver, which is now guaranteed to exist, and use it to construct a concrete error trace, which is then

Listing 6.3: HANDLEERROR sub-algorithm

```
Input: STTNode $n := (\underline{S_{in}}, \underline{S}, M)$, Materialization Point $(\underline{S^e}, f^e)$
  $\pi$ <- Root path for $n$;
  $\varphi_0 \wedge \ldots \wedge \varphi_{2l+1}$ <- EncodeTrace($\pi$, $(\underline{S^e}, f^e)$); // see $CoT.\ 39$, $Lem.\ 32$
  for (i in [0..2l+1]) do
    result <- Solve($\varphi_0 \wedge \ldots \wedge \varphi_{2l+1}$); // any SMTLib-compatible solver
    if (result == UNSAT) then
      $k$ <- $i \backslash 2$; // failure point
      $m_k$ <- Node At Failure Point $k$;
      $\underline{S^m_k}$ <- Materialized shape at failure point $k$;
      if (i % 2 == 1) then
        $I$ <- Interpolate($\|T\|$, $k$); // interpolate as defined in $Def.\ 95$
        $(\alpha, m)$ <- EvaluateProjected($(I, m')$); // $m'$ obtained from encoding
        $\Lambda_k$ <- $\Lambda_k \cup \{(\alpha, f^m_k \circ m)\}$;
        if (($f^{m-1}_k \circ f^m_k \circ m \neq m$) || (CheckEmbedded($\underline{S^{in}_k}$, $\underline{S_k}$))) then
          ModifyTree($(m_k)$, $((\alpha, m))$);
          return ($\epsilon$, $(m_k)$);
        else
          $\Lambda_k$ <- $\Lambda_k \setminus \{(\alpha, f^m_k \circ m)\}$;
        fi;
      fi;
      $(\alpha, m)$ <- NegMaterializationConstraint($\underline{S^m_k}$); // see $Def.\ 94$
      $\Lambda_k$ <- $\Lambda_k \cup \{(\alpha, f^m_k \circ m)\}$;
      if CheckEmbedded($\underline{S^{in}_k}$, $\underline{S_k}$) then
        ModifyTree($(m_k)$, $((\alpha, m))$);
        return ($\epsilon$, $(m_k)$);
      else
        $\Lambda_k$ <- $\Lambda_k \setminus \{(\alpha, f^m_k \circ m)\}$;
      fi;
      $\{(\alpha_{i_1}, m_{i_1}), \ldots, (\alpha_k, m_k)\}$ <- QueryUser($\pi_k$, $\underline{S^m_k}$); // see $Sec.\ 6.3$
      $((m_{i_1}, x_{i_1}), \ldots, (m_k, x_k))$ <- Unary edges of nodes from $i_1..k$ out of $\pi$.
      ModifyTree($((m_{i_1}, x_{i_1}), \ldots, (m_k, x_k))$, $\{(\alpha_{i_1}, m_{i_1}), \ldots, (\alpha_k, m_k)\}$);
      return ($\epsilon$, $(m_{i_1}, \ldots, m_{i_2})$);
    fi;
  od;
  $\mathcal{M}$ <- GetModel($\|T\|$);
  $\pi^e$ <- Construct Path from $\mathcal{M}$;
  return ($\pi^e$, $n$);
```

returned.

Once the trace prefix has become unsatisfiable, the `if`-branch on line 6 is entered. Lines $7-9$ determine the failure point, and extract the relevant STT node and materialized shape.

Now, if the current index in the trace is even, then the trace became infeasible at one of the points indicated by Lemma 32, indicating that there are no functions that are factors of the embedding into the central shape with respect to the materialization embedding at the failure point. Consequently, as described above and shown in Fig. 6.4, the constraint must be constructed from the materialization rather than the interpolation. This is realized with the `if`-clause on line 10.

If the current index in the trace is odd, we enter the `if`-branch and attempt to construct an interpolation-based constraint. Line 11 queries the solver for an interpolant. This interpolant, combined with the (identity-) mapping obtained from the solver, is then simplified via projected evaluation on line 12. On line 13, the result of that process is attached to the central shape. The `if`-clause on line 14 checks for the condition used in Lemma 34, or, if that fails, whether the incoming shape is still embedded into the central shape. If that check fails, the `if`-clause is left, the new constraint is removed again on line 18 and execution then continues as if the trace index had been even, i.e. on line 21. If the check succeeds, the tree is modified, i.e. the materialization points of the node at the failure step are updated, and all outgoing transition edges removed. An empty path and the modified node are then returned on line 16, terminating the algorithm.

This pattern of adding the constraint, checking for sanity, modifying the tree and returning the node or removing the constraint again in case of failure, is repeated on lines 21-28. The only significant difference lies in the genesis of the constraint on line 21, which uses Def. 94 instead of Def. 95.

If execution reaches line 29, then a spurious error was detected, and all attempts to automatically exclude this error via new constraints have failed. We then use `QueryUser` to obtain a series of user-generated constraints. The `ModifyTree` algorithm then modifies the nodes for which constraints were generated. The modified nodes are then returned, together with an empty path.

The soundness and validity of the changes made to the STT has already been proven via the various lemmata in this section. What remains is the proof that HANDLEERRROR, as a sub-algorithm, adheres to Def. 90 and thus preserves the loop invariant used by the correctness proof for the main algorithm in Theorem 5. Intuitively, this is clear. The addition of the new constraint at the failure point invalidates at least one of the materializations along the error path, removing the error materialization (and possible a whole subtree containing it) from the STT. Outside of user intervention, transitions are only removed, and never changed, while materialization points that are affected are updated with the new constraints. The following lemma formalizes and proves this intuition correct.

Lemma 38 (Correctness of HANDLEERROR). *The algorithm specified by Listing 6.3, here referred to by R, is a valid* `HANDLEERROR` *algorithm by the terms of Def. 90.*

Proof. Let the input of the algorithm be $n = \left(\underline{S^i n}, \underline{S}, M\right)$ and $(\underline{S^e}, f^e) \in M$. Let $\pi(n) = e_1 \cdots e_k$ be the root path for n, with $e_i = \left(n_{i-1}, \left(P_i, \left(\underline{S^m_i}, f^m_i\right)\right), n_i\right)$. We need to prove that R always terminates and that its return value always satisfies conditions a or b of Def. 90.

We begin with termination. R does not use any open-ended loops, thus the only way for non-termination to occur is a procedure call that does not terminate. The only candidate for this is line 5, where an SMT solver is repeatedly called to solve a trace encoding. As mentioned earlier, this solver call handles quantifier-free formulas over a finite universe which are therefore in the decidable fraction of first-order logic. Termination of this solver call can thus be assumed.

We will now show that one of the two conditions is always fulfilled after execution of R, depending on the satisfiability of $\varphi = \varphi_0 \wedge \ldots \wedge \varphi_{2l+1}$. If φ is `SAT`, then condition a will hold. If φ is `UNSAT`, then condition b will hold. We will prove each of these in turn.

First, we assume φ to be `SAT`. Then, from the correctness of the trace encoding (see Sec. 4.5), we know that the model returned by the solver contains a concrete error trace. We also know that the `if`-clause of lines $6-33$ is never entered, leading to the eventual termination of the `for`-loop on lines $4-34$. Lines $35-37$ then extract the concrete trace from the model, and return it, satisfying condition a.

We now turn to the case where φ is `UNSAT`. If that is the case, then there is a smallest integer i such that $\varphi_0 \wedge \ldots \wedge \varphi_i$ is `UNSAT`. The `if`-branch on lines $7-32$ is entered upon the `for` loop reaching that i. Within the `if`-branch, it is obvious that there are exactly three places within the `if`-branch that have any effect on the return value and $\mathcal{T}$: lines $13-16$, $22-25$, and $31-32$. We examine each case in turn.

On line 13, the constraint created as defined in Def. 95 is added to the central shape at the failure point. If the necessary conditions for this constraint hold (line 14), the tree is modified accordingly and the modified node returned. The established correctness of the interpolant-based constraint, together with the correctness of the `ModifyTree` algorithm establish condition b upon termination of R on line 16.

On line, the same process occurs, with Def. 95 being replaced by Def. 94. The argument for correctness remains the same.

On line 29, the user is queried for a series of refinements. The result of adding these constraints to their respective STT nodes and modifying the tree accordingly follows directly from Def. 99 and the correctness of `ModifyTree`. Thus, condition b is always established when the algorithm is left from within the `if`-clause on lines $6-33$. $\square$

The valid HANDLEERROR algorithm completes the automatic portion of our tool set. We can now, using a well-defined algorithm, construct a shape transition tree from a graph transformation system, error pattern and abstraction scheme. The termination of that algorithm with a valid STT then guarantees the safety of the GTS. In the following Section, we give a framework for implementing a QUERYUSER algorithm that supports human designers as much as possible in crafting useful constraints. After that, in the next chapter, we will take a look at an example *implementation* of this algorithm.

6.3 Manual Abstraction Refinement Supported by Soundness Checks

We will now discuss the QUERYUSER sub-algorithm in more detail. Since implementation details of an interactive algorithm will revolve chiefly around presentation, interaction design, input parsing, etc., we will not give a full formal definition of this algorithm here. We will rather take a closer look at the objectives of this sub-algorithm and explore ways by which the checks and processes already developed in previous sections and chapters can be reused to aid the user in constructing sound and effective constraints.

Looking at Def. 99, we see that the `QueryUser` algorithm must essentially do two things. It must

- choose an index j lower than or equal to the failure point k, and
- provide shape constraints for the central shape of each node from index j through k such that transformation soundness is preserved and the failure materialization excluded.

Note that that does not mean that *every* node from j to k must be refined – it is perfectly acceptable to use $(\mathbf{1}, \emptyset)$ as a constraint for nodes that need not be refined. However, the utility of constraints further back in the trace that then do not influence, even indirectly, the constraint that excludes the materialization is questionable and dependent on very domain-specific circumstances.

Now, what does it mean that "transformation soundness is preserved and the failure materialization excluded"? We will begin with the simpler of the two properties, the notion of a "useful" constraint, i.e. the exclusion of the failure materialization. This was the center of our work on automatic abstraction refinement: the constraint added to the shape at the failure point must, projected onto the failure materialization, evaluate to false, i.e.

$$S_k^m \not\models \mathrm{concr}_{f_k^m}(\{(\alpha_k, m_k)\})$$

Since any sequence of constraints is useless without this property, it is sensible to start the refinement process by choosing a constraint for $\underline{S_k}$ that has that property.

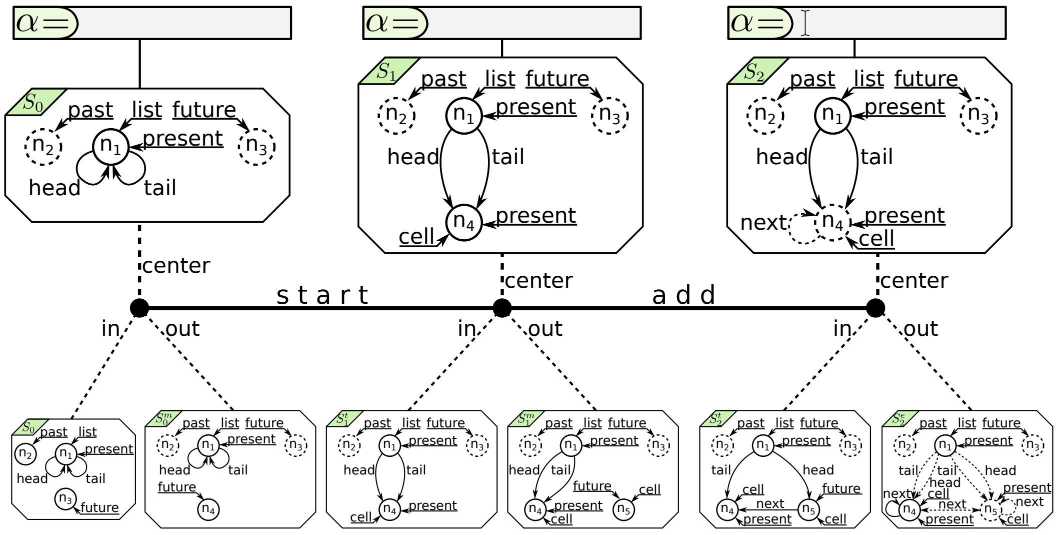


Figure 6.6: Display of one abstract error trace for manual refinement (sketch)

Turning to the other property, the "soundness" condition, we realize that such a constraint might be in conflict with the incoming shape, i.e. shrink its graph set, even though no actual graph from the concrete system is excluded by it. In the previous section, this was handled with a *sanity check* that simply checked whether the incoming shape remains embedded into the modified central shape, i.e. for an index i:

$$\underline{S_i^t} \sqsubseteq (S_i, \Lambda_i \cup \{(\alpha_i, m_i)\})\,,$$

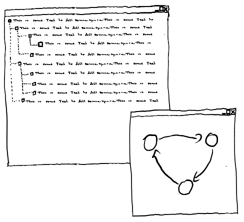
meaning the additional constraint does not shrink the graph set of the incoming shape. In the automatic refinement process, we had no way to deal with a negative result of that check. In the manual refinement process, however, we can simply bring this problem to the attention of the user, and have them fix it by also adding a constraint to the previous shape in the trace.

Note that for both properties that need to be satisfied, there are straightforward processes to check those properties defined in the previous section. The soundness property is easily checked using the `CheckEmbedded` algorithm used in the `HANDLEERROR` algorithm as a sanity check, while the "usefulness" of the constraint sequence can be established by projecting the constraint for the failure point onto the failure materialization and checking for feasibility using the algorithm `CHECKFEASIBILITY` from Chapter 5. Thus, in the manual refinement process, a user might be presented with a display like the one sketched in Fig. 6.6. If the prior attempt at automatic abstraction refinement produced one, the user can be provided with the interpolant for the failure point as a starting point. After the user has provided a constraint for the terminal shape of that path, we can check that constraint for soundness and usefulness using `CheckEmbedded` and `CHECKFEASIBILITY`.

If `CHECKFEASIBILITY` fails, we can obtain from the (satisfiable) encoding a model that represents a graph that remains embedded into the materialization, in spite of the constraint. This can be displayed to the user, who can use this information to fix the constraint, which is the only option at this point. If the constraint provided is useful, but `CheckEmbedded` fails, we again gain from the model of the encoding a graph, which in this case represents the set of graphs that are embedded into the incoming shape, but not into the refined central shape. There are now two options for the user to use this information. They can decide that the given graph represents spurious behavior and extend the refinement by refining the shape just before the shape they just refined, thus decreasing the index j for the beginning of the refinement by 1. Alternatively, they can use the graph provided as a hint to further refine the shape constraint they provided.

Every time a constraint is added to a shape $\underline{S_i}$ by the user, the materialization $\underline{S_i^m}$ and, if present, unabstracted result shape $\underline{S_i^t}$ are recomputed to account for that change. This may also cause a rechecking of the soundness of the shape constraint at $i + 1$ (if any), since the refinement of $\underline{S_i^t}$ and accompanying smaller graph set might cause it to be embedded into the central shape at $i + 1$ after all.

In this manner a sequence of shape constraints is provided by the user that is, once all checks are positive, guaranteed to conform to the conditions given in Def. 99. An example application of such a process can be found in Section 7.3, on page 218.



7 Implementation: Shape Graph Analyzer

The Shape Graph Analyzer (SGA) is the prototypical implementation created as part of this thesis. It fully implements the LSSC construction algorithm detailed in Chapter 5, including the automatic abstraction refinement process describes in Section 6.2. SGA does as yet not include an implementation for manual abstraction refinement, terminating in cases where such a refinement would become necessary. It has been implemented by binding a number of existing software packages, such as SMT solvers and a three-valued logic framework, together with an implementation of graphs, shapes, and the construction algorithm itself. In this chapter, we will gain a high-level overview over the structure, usage, and capabilities of SGA and discuss the results of applying SGA to a more complex verification for our linear list example, as well as graph transformation systems from the literature.

7.1 Description and Usage

SGA is a standalone program written in the Java programming language. It builds on the basic logical framework provided by TVLA[108] and uses SMTInterpol[46] to interface with solvers, specifically, SMTInterpol itself and Microsoft's Z3[114].

Each run of SGA will attempt to construct a valid STT for a graph transformation system supplied to SGA via four input files. These input files describe the graph labels available, the initial graph, the graph rules, and the forbidden pattern and so, together, specify a graph transformation system $\mathcal{G}$ and a forbidden pattern P. The input files are specified using a fragment of the input language for TVLA.

Figure 7.1 shows the structure of the input files by way of the linear list example used throughout the thesis. We will now examine each file individually and describe its structure and meaning. In all files, lines beginning with double-slashes (`//`) are considered comments and are ignored by SGA.

The Predicates File – *.tvp The predicates file establishes the set of predicates available for building graphs by effectively setting the label set. Each line beginning with `%p` introduces a new predicate. The format is

```
%p predicate_name(parameter$_1$, ..., parameter$_k$)
```

with `predicate_name` fixing the name of the predicate / label and k determining the arity of the predicate. The names of the parameters are placeholders and can be chosen arbitrarily.

The Initial Graph File – *.tvs The initial graph file defines its namesake. It is partitioned into two parts, defining the node and edge sets of the initial graph, respectively. The nodes part has the form

```
 %n = {v$_1$, ..., v$_k$}
```

where, again, the particular node names can be chosen arbitrarily. The part defining the predicates is comprised of a list, defining, for each label specified in the predicates file, between which nodes edges with the respective label exist. This is written

```
 %p = {u$_1$ = {v$_{i_1}$,...,v$_{j_1}$} ··· u$_n$ = {v$_{i_n}$,...,v$_{j_n}$}
       p$_1$ = {v$_{i_1}$->w$_{i_1}$,...,v$_{j_1}$->w$_{j_1}$} ··· p$_m$ = {v$_{i_1}$->$w_{i_1}$,...,v$_{j_1}$->w$_{j_1}$}}
```

The Pattern File – *.vio The pattern file specifies the forbidden pattern that defines the verification task performed by SGA. It has the same syntax as the initial graph file. The graph defined by it is taken as the left hand side of the forbidden pattern.

The Rules File – *.prod The rules file specifies the rule set of the graph transformation system and is the most complex of the four files. The layout of the file has the form

Listing 7.1: Predicates file `predicates.tvp`

```
/////////////////////////
// Unary core predicates

// list node
%p l(v)
// cell node
%p c(v)

//////////////////////////
// Binary core predicates

// head of the list
%p h(v_1, v_2)
// tail of the list
%p t(v_1, v_2)
// list link
%p n(v_1, v_2)
```

Listing 7.2: Graph file `initial.tvs`

```
%n = {v}
%p = {
        sm = {}
        l = {v}
        h = {v->v}
        t = {v->v}
        n = {}
}
```

Listing 7.3: Pattern file `forbidden.vio`

```
%n = {x}
%p = {
        sm = {}
        c = {x}
        n = {x->x}
}
```

Listing 7.4: Rules file `productions.prod`

```
%prod start {
        %left = {
                %n = {n}
                %p = {sm = {}
                        h = {n->n}
                        t = {n->n}
                        l = {n}
        }}
        %right = {
                %n = {n, x}
                %p = {sm = {}
                        h = {n->x}
                        t = {n->x}
                        l = {n}
                        c = {x}
        }}
        %g = {}
}
%prod end {
        %left = {
                %n = {n, x}
                %p = {sm = {}
                        h = {n->x}
                        t = {n->x}
                        l = {n}
                        c = {x}
        }}
        %right = {
                %n = {n}
                %p = {sm = {}
                        h = {n->n}
                        t = {n->n}
                        l = {n}
        }}
        %g = {}
}
%prod add {
        %left = {
                %n = {n, x}
                %p = {sm = {}
                        h = {n->x}
                        l = {n}
                        c = {x}
        }}
        %right = {
                %n = {n, x, y}
                %p = {sm = {}
                        h = {n->y}
                        l = {n}
                        c = {x, y}
                        n = {y->x}
        }}
        %g = {}
}
%prod del {
        %left = {
                %n = {n, x, y}
                %p = {sm = {}
                        t = {n->y}
                        l = {n}
                        c = {x, y}
                        n = {x->y}
        }}
        %right = {
                %n = {n, x}
                %p = {sm = {}
                        t = {n->x}
                        l = {n}
                        c = {x}
        }}
        %g = {}
}
```

Figure 7.1: Input File Set describing the Linear List Example

```
%prod rule$_1$ {
  %left = { G }
  %right = { G }
  %g = {}
}
⋮
%prod rule$_k$ {
  %left = { G }
  %right = { G }
  %g = {}
}
```

where every occurrence of `G` indicates a graph specification using the same syntax as the graph and pattern files. The graph given under `%left` specifies the left hand side of the rule, whereas the graph given under `%right`. The subgraph isomorphism between the left hand side L and the right hand side R of the rule is established via the implied mapping on the node sets that is specified by the node names, i.e. nodes with the same name in the specification of both sides refer to the same node. This is then extended to an isomorphism between subgraphs of L and R by setting $L_c = L \setminus (L \setminus R)$ and $R_c = R \setminus (R \setminus L)$.

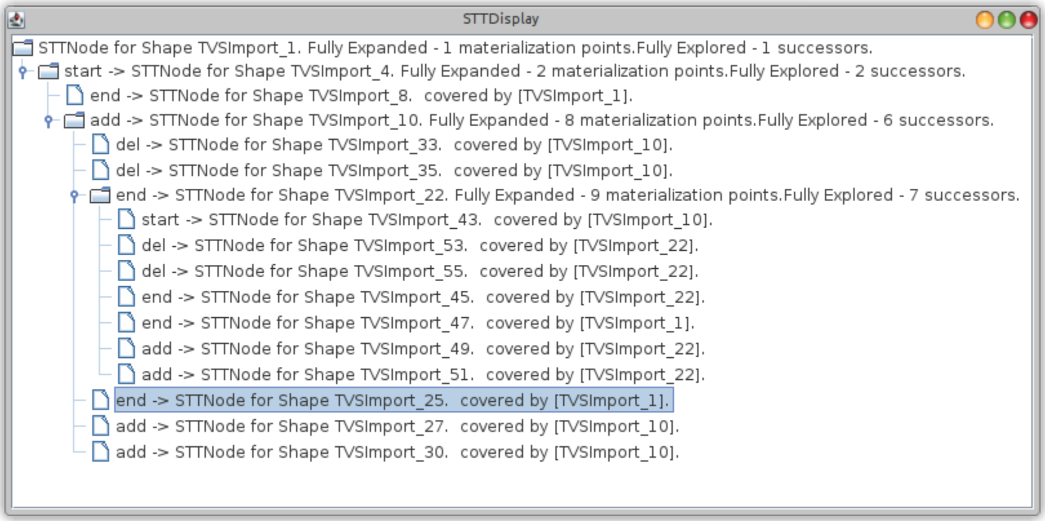
Two further configuration files, `quantifiedlogicsolver.prop` and `interpolatingsolver.prop`, determine various additional settings that pertain to the solvers used. The implementation provides default versions of these files, and it is recommended to only change them to reflect installation details, such as the executable path of the solver to be used. The syntax of these files is self-explanatory.

Once the input and configuration files have been created, the analysis can be started, substituting the proper filenames for the use case, with

```
<sga_program_call> predicates.tvp initial_graph.tvs rules.prod pattern.vio
```

During execution, the program will report on its current status using two graphical displays – the STT display and the WorkItem display. Figures 7.2 and 7.3 show these two displays as they are showing the state of the program after an execution of SGA for the linear list example.

The STTDisplay shows the tree structure of the STT. Each STT node is shown as a node in a standard Java tree display. The label of each node contains basic information about the STT node it represents. It begins with the name of the rule that produced the node (the obvious exception to this being the root node), followed by a unique name for the node, the number of materializations, outgoing connections, and whether the node is currently being expanded or explored. Covered nodes are marked by the node label ending in `Covered by` , followed by the name of the covering node. As



```
STTDisplay
STTNode for Shape TVSImport_1. Fully Expanded - 1 materialization points.Fully Explored - 1 successors.
start -> STTNode for Shape TVSImport_4. Fully Expanded - 2 materialization points.Fully Explored - 2 successors.
  end -> STTNode for Shape TVSImport_8.  covered by [TVSImport_1].
  add -> STTNode for Shape TVSImport_10. Fully Expanded - 8 materialization points.Fully Explored - 6 successors.
    del -> STTNode for Shape TVSImport_33.  covered by [TVSImport_10].
    del -> STTNode for Shape TVSImport_35.  covered by [TVSImport_10].
    end -> STTNode for Shape TVSImport_22. Fully Expanded - 9 materialization points.Fully Explored - 7 successors.
      start -> STTNode for Shape TVSImport_43.  covered by [TVSImport_10].
      del -> STTNode for Shape TVSImport_53.  covered by [TVSImport_22].
      del -> STTNode for Shape TVSImport_55.  covered by [TVSImport_22].
      end -> STTNode for Shape TVSImport_45.  covered by [TVSImport_22].
      end -> STTNode for Shape TVSImport_47.  covered by [TVSImport_1].
      add -> STTNode for Shape TVSImport_49.  covered by [TVSImport_22].
      add -> STTNode for Shape TVSImport_51.  covered by [TVSImport_22].
    end -> STTNode for Shape TVSImport_25.  covered by [TVSImport_1].
    add -> STTNode for Shape TVSImport_27.  covered by [TVSImport_10].
    add -> STTNode for Shape TVSImport_30.  covered by [TVSImport_10].
```

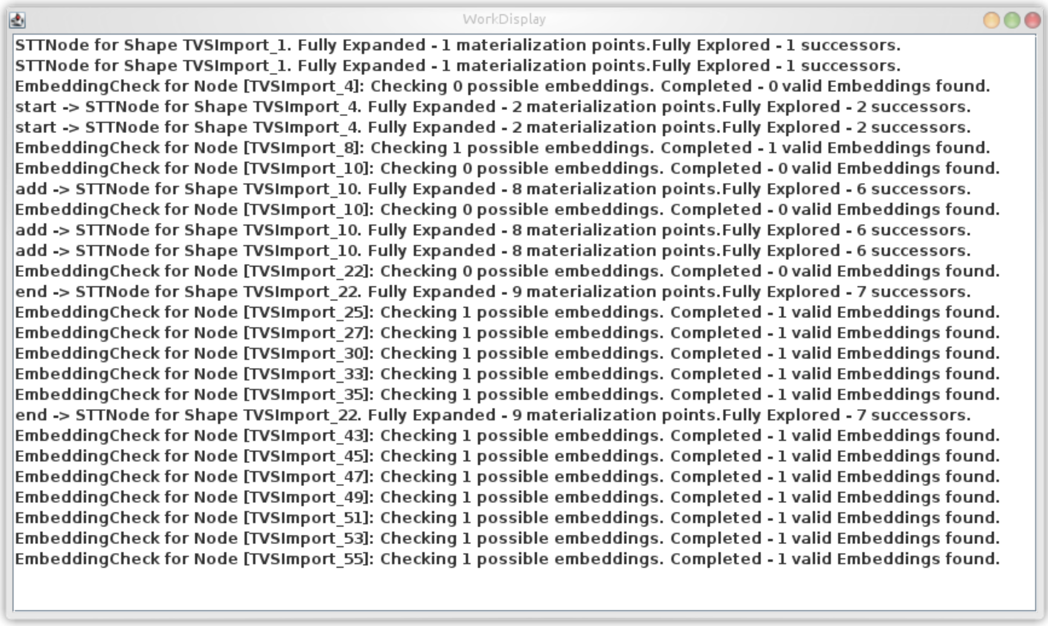
Figure 7.2: The STT display after an example execution for the linear list example

we can see here, the tree structure produced by SGA mirrors exactly the predicted tree structure shown in Fig. 5.6.

The WorkItem display shows the execution tasks that have been created so far for the construction of the STT, as well as their current status. SGA is designed for concurrent execution, so at any given time there may be many work items that are being worked on at once. The kind of work item that are created include Expansions, Explorations, and EmbeddingChecks. Expansions and Explorations are self-explanatory, while EmbeddingChecks represent calls to `COVER` in the main algorithm. Feasibility checks are not recognized as individual work items as they are always performed as part of another work item, such as a node expansion, where they are used to screen materialization points, or embedding checks, where they are used to establish graph set inclusion.

SGA also maintains log files that include more detailed information about the actions taken by the various parts of the program. These logs are saved in files called `lsscEngine.log`, `lsscExpander.log`, `lsscFeasibility.log`, and `lsscEmbedding.log`. It is here, where an error found in the system is reported. If the work item list contains only completed work items, and no error has been reported in any of the logs, then the constructed STT is valid and the system thus proven safe.

Once the analysis has finished, either because a real error was found, the tree was declared valid, or a user input was requested, a summary of the result is written into an appropriate file. In the case of a user input request or real error, the corresponding path through the STT is written into a filed called `errorpath.dot`, which then contains



```
STTNode for Shape TVSImport_1. Fully Expanded - 1 materialization points.Fully Explored - 1 successors.
STTNode for Shape TVSImport_1. Fully Expanded - 1 materialization points.Fully Explored - 1 successors.
EmbeddingCheck for Node [TVSImport_4]: Checking 0 possible embeddings. Completed - 0 valid Embeddings found.
start -> STTNode for Shape TVSImport_4. Fully Expanded - 2 materialization points.Fully Explored - 2 successors.
start -> STTNode for Shape TVSImport_4. Fully Expanded - 2 materialization points.Fully Explored - 2 successors.
EmbeddingCheck for Node [TVSImport_8]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_10]: Checking 0 possible embeddings. Completed - 0 valid Embeddings found.
add -> STTNode for Shape TVSImport_10. Fully Expanded - 8 materialization points.Fully Explored - 6 successors.
EmbeddingCheck for Node [TVSImport_10]: Checking 0 possible embeddings. Completed - 0 valid Embeddings found.
add -> STTNode for Shape TVSImport_10. Fully Expanded - 8 materialization points.Fully Explored - 6 successors.
add -> STTNode for Shape TVSImport_10. Fully Expanded - 8 materialization points.Fully Explored - 6 successors.
EmbeddingCheck for Node [TVSImport_22]: Checking 0 possible embeddings. Completed - 0 valid Embeddings found.
end -> STTNode for Shape TVSImport_22. Fully Expanded - 9 materialization points.Fully Explored - 7 successors.
EmbeddingCheck for Node [TVSImport_25]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_27]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_30]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_33]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_35]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
end -> STTNode for Shape TVSImport_22. Fully Expanded - 9 materialization points.Fully Explored - 7 successors.
EmbeddingCheck for Node [TVSImport_43]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_45]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_47]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_49]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_51]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_53]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
EmbeddingCheck for Node [TVSImport_55]: Checking 1 possible embeddings. Completed - 1 valid Embeddings found.
```

Figure 7.3: The WorkItem display after an example execution for the linear list example

the entire path, including materialization, application and blur actions, in the Graphviz description language[69]. If the tree was valid, a similar file called `finishedSTT.dot` is produced, containing the tree structure with central shapes, transformation edges, and covering edges.

7.2 Architecture

We will now obtain a birds-eye view of the architecture of the tool. This includes all the connections to external tools. The tool is roughly separated into two parts: SGA itself, and an SMTLib-library called SMTool. Both of these tools were created to form a prototype for the algorithms presented in this thesis.

SGA is essentially a tool for the state space exploration of shapes with constraints as defined in Chapter 3. It is based on the three-valued logic engine TVLA[108], in that it uses TVLA's capabilities to represent three-valued logical structures and evaluate formulas in those structures. Every aspect of the state space construction algorithm that relies on the construction and solving of SMT formulas is relegated to SMTool.

SMTool is designed to be a front end for SMTLib-compatible solvers in general that allows for the easy creation and solving of the encodings that are established in Chapter 4 and refined in Chapter 6. It is based on SMTInterpol[46], and through it, can

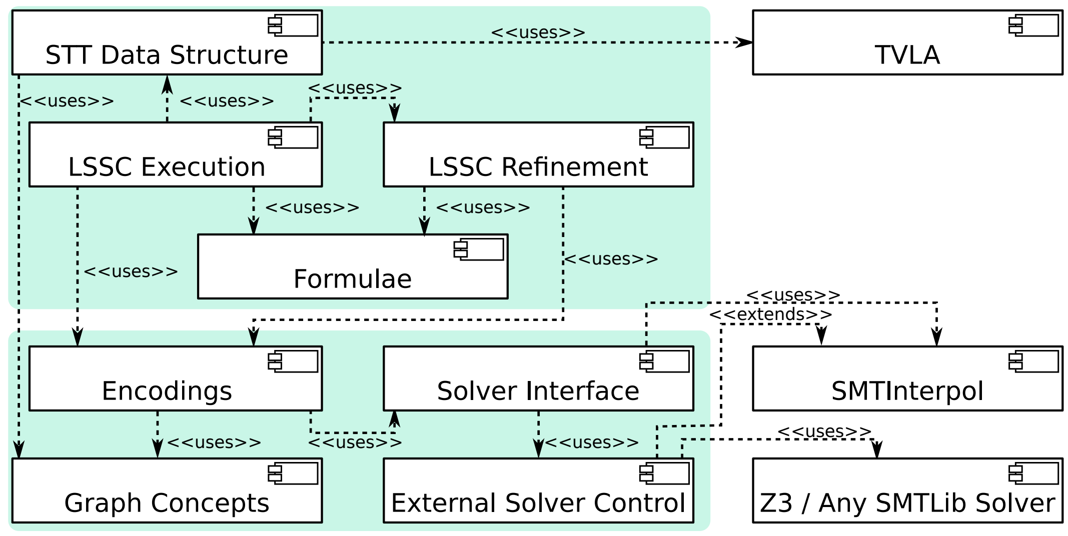


Figure 7.4: High-Level Architecture of SGA / SMTool

interface with any solver that is SMTLib-compatible. In the particular use case with SGA, we use Microsoft's Z3 for all solving of formulas with quantifiers, but SMTool is in no way restricted to Z3 and can, with very little effort, be configured to use any other SMTLib-compatible solver.

Figure 7.4 shows a simplified schematic of the high-level architecture*. The groups indicated by the shaded boxes represent software implemented as part of this thesis, in part in collaboration with Daniel Wonisch and Manuel Töws, while the remaining components represent software that is the result of other work. The upper group shows the main components of SGA.

The `STT Data Structure` component encapsulates the data structure implementing the shape transition tree, as well as all operations that can be applied to it (such as tree pruning). Here, the capabilities of `TVLA` are used to represent constraint formulas as three-valued formulas and shapes as three-valued logical structures, in order to take advantage of prior work by Daniel Wonisch which implements the transformation of shape constraints. Also used is the implementation of graph domain concepts of SMTool in order to be able to effectively communicate the state of the STT to the encodings.

This component is then used by the highly concurrent implementation of the construction algorithm, encapsulated in the `LSSC Execution` component. Internally, that is, when not communicating with either SMTool or TVLA, constraints formulas

*with transitive «`uses`»-relationships omitted where they exist, in order to keep the presentation more concise are being represented with classes from the `Formulae` component in order to avoid implicit bindings to specific solver or engine contexts that come with the respective implementations of TVLA and SMTInterpol. When an error is encountered by the `LSSC Execution` component, the creation of appropriate constraints is deferred to the `LSSC Refinement` component.

Both `LSSC`-components of SGA use the `Encodings` component provided by SMTool. For state space construction, it provides feasibility encodings, while for refinement, it provides trace encodings. The `Encodings` package uses the `Graph Concepts` package to represent the mathematical objects given to it on a conceptual level, while using the `Solver Interface` component to actually call the configured solver to solve the encodings. `SMTInterpol` as a solver is used directly by this interface. By creating a separate process and wrapping the textual logging classes of SMTInterpol, the `External Solver Control` component allows us to communicate via Pipes with any SMTLib-compatible solver, which, in our case, is `Z3`.

7.3 Application Examples

In the course of presenting the usage of SGA, we have already demonstrated the result of applying it to the linear list example. The results shown in Fig. 7.2 took around 3 seconds on an 8-core Intel Core-i7 computer, clocked at 3.4GHz, with 16GB memory. We will now introduce further examples and show the results for them achieved with SGA, reporting runtimes for those examples on that same machine.

Note that SGA is not optimized for performance, but rather for conceptual clarity, ease of maintenance, and extendability. Thus, the runtime of this example implementation should only be considered as a demonstration of the asymptotic runtime behavior of the algorithm, rather than a demonstration of feasibility in realistic settings.

7.3.1 Linear List Example with Cleanup Error

The first example we will look at uses the same GTS as our running example, but with a different forbidden pattern, shown in Fig. 7.7. This pattern essentially represents a "cleanup error", i.e. a *cell* node left over after the list has been emptied. The different error pattern necessitates different refinement steps, allowing us to observe in more detail the refinement process employed by SGA, without obtaining a state space that is too large to allow for such in-depth analysis. Overall, the algorithm performs three refinement steps – an interpolation-based refinement, followed by a materialization-based refinement, followed by a final interpolation-based refinement. Note that due to the high parallelism present in the SGA implementation, the precise state that the STT is in when a spurious error is encountered is neither entirely predictable, nor necessarily compliant with the loop invariant familiar from Chapter 5.

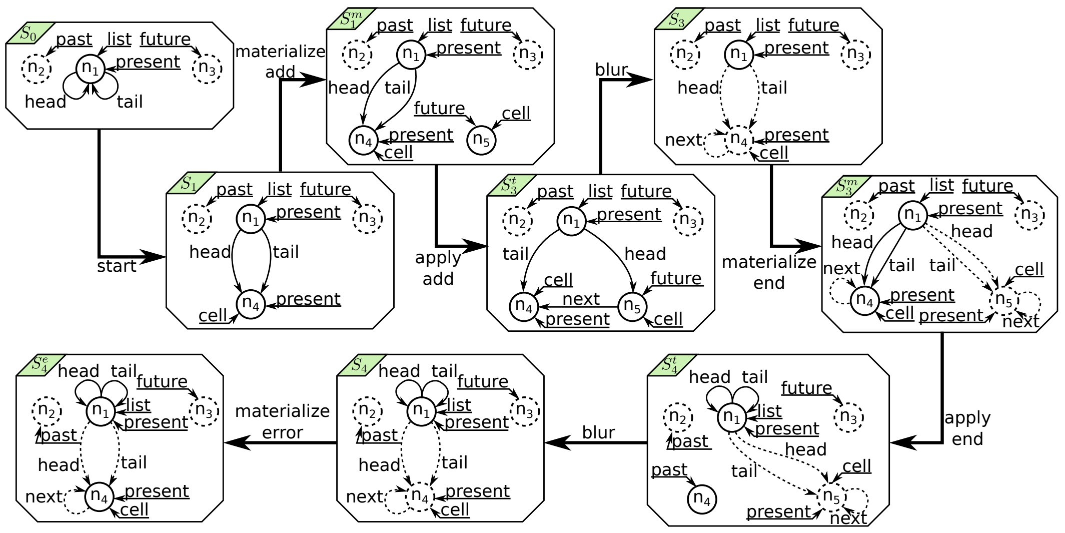


Figure 7.5: The first abstract error trace encountered by SGA on the Linear List Example with Cleanup Error

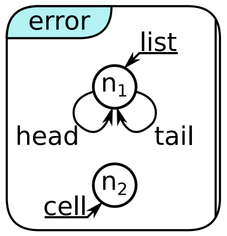


Figure 7.7: Forbidden Pattern representing a left-over *cell* node with an empty list

Now, beginning with the initial graph of the linear list GTS (see Fig. 2.5), SGA begins constructing the STT, when it encounters the first error. The STT at that point is shown in Fig. 7.6(a). The nodes that are part of the abstract error path are shown in red. As we can see, the first error occurs when the result of (abstractly) applying the start rule, followed by the add rule, i.e. a shape representing (among other things) an arbitrarily long linear list, matches the end rule. This is because "arbitrarily long" also includes lists of length 1, to which the end rule is applicable. The complete error path can be seen in Fig. 7.5.

Having established the abstract error path, SGA queries the SMTool library to perform the trace encoding and interpolation as defined in Chapters 4 and 6. The result indicates that the actual failure point is not the shape that potentially matches the error, i.e. S_4, but rather the application of the end rule to the first shape representing an arbitrarily long linear list, i.e. S_3. This is spurious, since in the actual GTS, the linear list would have two *cell* nodes at this point, one created by the start rule and one created by the add rule. The interpolation-based shape constraint created by interpolating at the failure point is

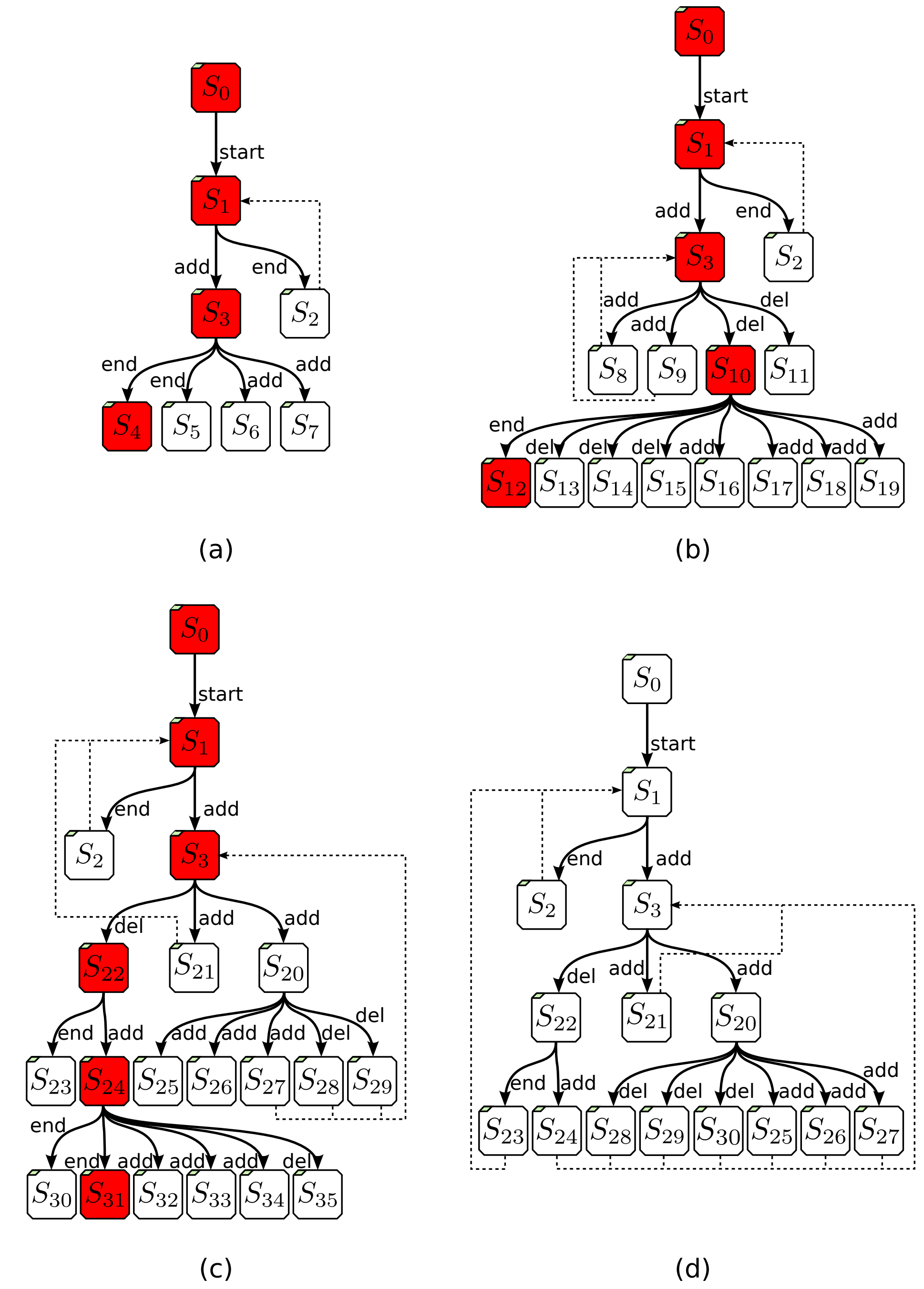


Figure 7.6: The state of the STT for the linear list example with the cleanup error pattern before each refinement step ((a) through (c)) and after construction has concluded (d).

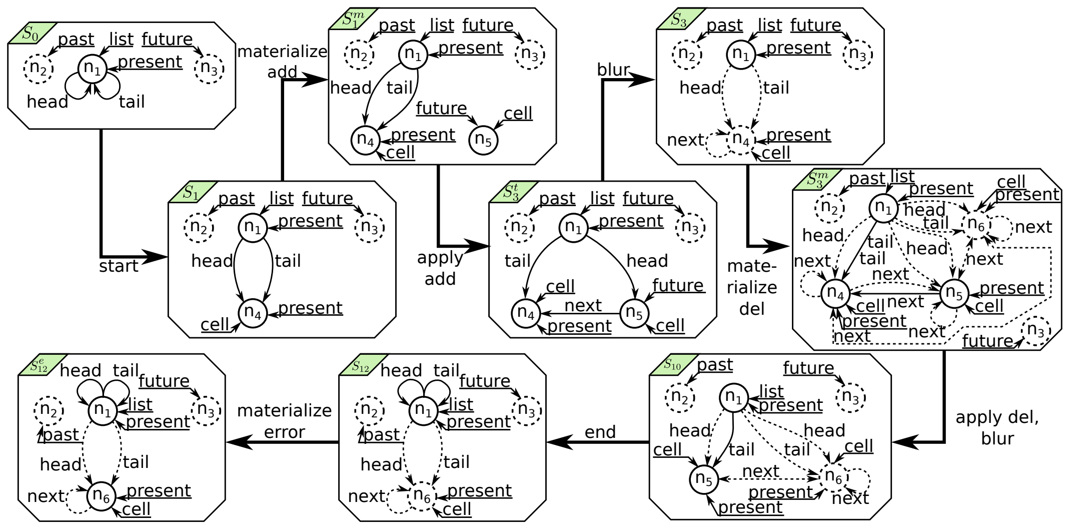


Figure 7.8: The second abstract error trace encountered by SGA on the Linear List Example with Cleanup Error

$$\begin{aligned}(\mathit{present}\,(x_4) \vee \neg\mathit{future}\,(x_1) \vee \neg\mathit{cell}\,(x_2) \vee \neg\mathit{cell}\,(x_5) \vee \neg\mathit{head}\,(x_3, x_5) \vee \neg\mathit{tail}\,(x_3, x_5)\,,\\ [x_1 \mapsto n_3, x_2 \mapsto n_4, x_3 \mapsto n_1, x_4 \mapsto n_2, x_5 \mapsto n_5])\end{aligned}$$

Originally attached to the shape of the materialization point of the end-transition edge, this constraint is then subjected to projected evaluation and subsequent projection to yield the constraint

$$(\neg\mathit{head}\,(x_3, x_5) \vee \neg\mathit{tail}\,(x_3, x_5)\,, [x_3 \mapsto n_1, x_5 \mapsto n_4])$$

which expresses, essentially, that there is no node represented by the summary *cell*-node that is the target of both a *head* and a *tail* edge, which is a prerequisite to the end rule. Since this does not conflict with S_3^t, the incoming shape for S_3, the constraint is declared valid. Materialization points are recomputed and no materialization of the end rule is feasible anymore.

The analysis then continues until the second spurious error is encountered. After having explored the STT node for $\underline{S_3}$ further, a new potential match for the error is constructed by applying del, followed by end. A detailed view of this error path can be seen in Fig. 7.8, while the state of the STT can be viewed in Fig. 7.6(b). In the interest of readability, already present constraints have been omitted, and edges of the same label that go in both directions between two nodes have been merged using two-tipped arrows. Note that the blur action that produces $\underline{S_{10}}$ does not merge nodes n_5 and n_6. This is because the interplay of the materialization of del, the application of del, and the transformation of the constraints attached to those nodes leads to there only being a single constraint on $\underline{S^t_{10}}$:

$$\left(\neg \mathit{head}\left(x_3, x_5\right) \vee \neg \mathit{tail}\left(x_3, x_5\right), \left[x_3 \mapsto n_1, x_5 \mapsto n_6\right]\right)$$

Thus, the constraint is removed from n_5, which precludes the merging of n_5 and n_6, and ultimately allows for the materialization of the error pattern after a now-valid application of end. Again, this error is spurious since, clearly, if we apply start, then add, then del, and then end, we have removed exactly as many *cell*-nodes as we added, leaving no room for any leftover nodes that the error pattern could potentially match. Thus, this is obviously a node set issue, rather than an edge set issue.

As expected, the failure point returned by SMTool for the corresponding trace indicates this. The specific point where the abstract trace leaves the concrete transition system is not, as one might suspect, the application of end, but rather the particular application of del used in the trace. This is because del was applied to a materialization $\underline{S^m_3}$ that preserved the *cell* summary node of shape $\underline{S_3}$, meaning that the materialization assumed that there were (at least) three *cell* nodes present, when in fact, there were only two. This prompts SGA to abandon the interpolation-based approach and to instead construct a negative materialization constraint for the particular materialization of the del rule at $\underline{S_3}$, leading, after evaluation, to the following constraint.

$$\begin{aligned}\left(\neg\left(\neg\left(x_1 = x_2\right) \wedge \neg\left(x_1 = a\right)\right.\right. & \left.\wedge \neg\left(x_2 = a\right) \wedge \mathit{tail}\left(x_0, x_2\right) \wedge \mathit{next}\left(x_1, x_2\right)\right), \\ & \left.\left[x_0 \mapsto n_1, x_1 \mapsto n_4, x_2 \mapsto n_4, a \mapsto n_4\right]\right)\end{aligned}$$

This constraint now precludes any materialization of del from $\underline{S_3}$ that preserves the *cell*-summary node. Equipped with this new constraint, the state space construction continues, after the subtree below $\underline{S_3}$ has been discarded.

The final spurious error is found soon after. It is constructed by applying del (without preserving the *cell*-summary node) to $\underline{S_3}$, followed by add, followed by end. The reason this is possible is because the removal of the *cell*-summary node also removes most of the constraints established earlier in the algorithm, so that another application of add reestablishes the shape that was given by $\underline{S_3}$ before the first refinement. From this shape, the same error can be reconstructed that was found when $\underline{S_3}$ was first constructed.

The complete error path is shown in Fig. 7.9, while the state of the STT is depicted

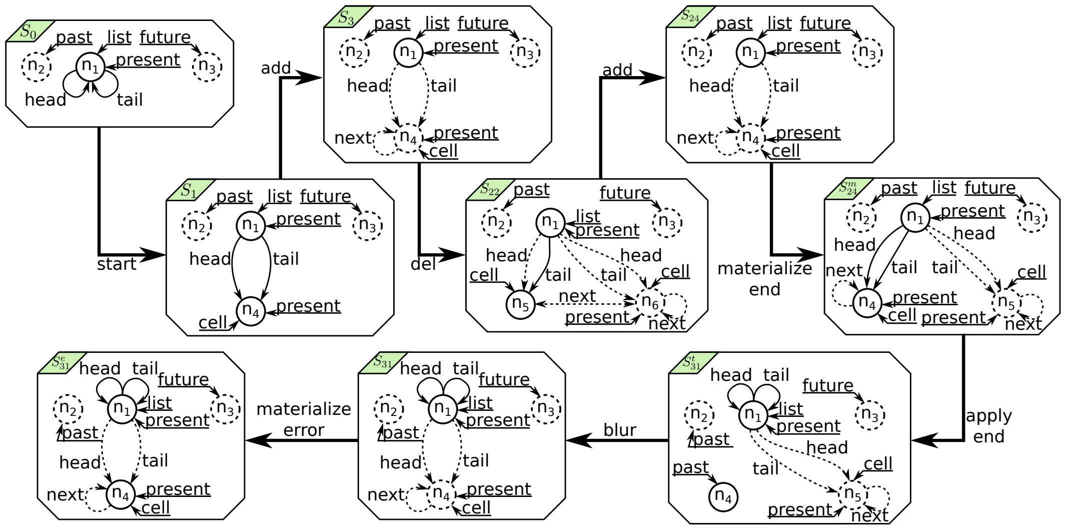


Figure 7.9: The final abstract error trace encountered by SGA on the Linear List Example with Cleanup Error

in Fig. 7.6(c). Since the error is very similar, the created constraint ends up being (semantically, not syntactically) identical to the constraint found in the first refinement. Thus, after projected evaluation and subsequent projection, we again end up with

$$(\neg \mathit{head}(x_3, x_5) \vee \neg \mathit{tail}(x_3, x_5), [x_3 \mapsto n_1, x_5 \mapsto n_4]) .$$

This, added to $\underline{S_{24}}$, again precludes any materialization of the end rule from shape $\underline{S_{24}}$, and has the added benefit that the refined shape $\underline{S_{24}}$ is now embeddable into the shape $\underline{S_3}$.

The construction algorithm continues, after having deleted the subtree under $\underline{S_{24}}$, and immediately recognizes the newly created embedding relationship. The algorithm continues to explore the remaining shapes, but finds no further errors. All discovered shapes are now either fully explored or covered by some other shape. The validity of the STT is thus established and we obtain the complete STT shown in Fig. 7.6(c). On our reference machine, this process takes about 7,5 seconds on average, varying between 6 and 9 seconds. This variance stems from the parallelism and the related nondeterminism inherent in the SGA implementation – the sequence in which shapes are expanded, checked for embedding relations, etc. are not predictable in advance and can lead to differences in how many shapes have to be considered on the way to a valid STT.

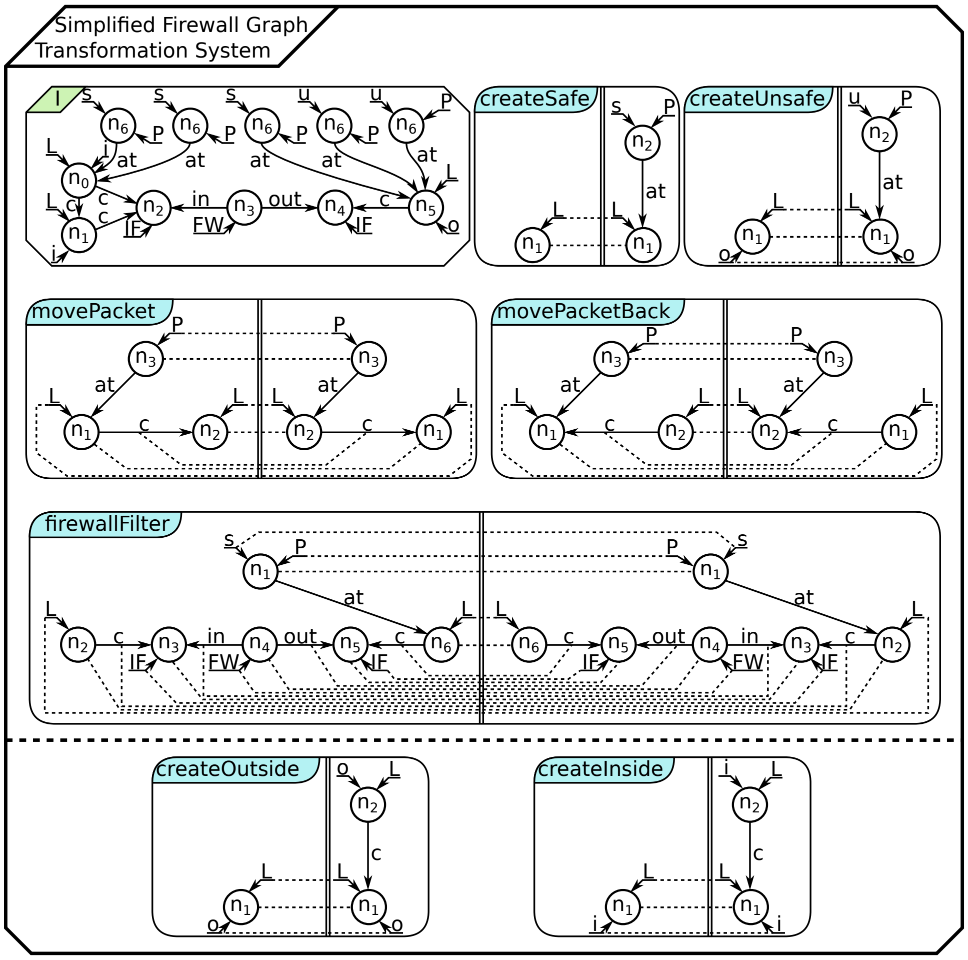
Next, we consider an example from the literature and encounter refinements that have to be referred to the user.

7.3.2 (Simplified) Firewall Example

Barbara König and Vitali Kozioura published a paper in 2006[102], in which they described their tool Augur (see Sec. 8.1), and applied it to a graph transformation system that described a computer network made up of connected locations, which in turn formed two disconnected subnetworks linked by a single firewall. The purpose of the firewall was to only let "safe" packets through. Thus, the property of the GTS that was to be verified was that "unsafe" packets, which can only be created in the part of the network that lies outside of the subnetwork shielded by the firewall, never reach the network behind the firewall.

This example was then adopted and slightly modified by Eduardo Zambon and Arend Rensink in 2012[162] to demonstrate the ability of their approach to take advantage of abstract state subsumption to curb the size of abstract state spaces in their approach (also see Sec. 8.1). Here, we will use SGA to verify this simplified version and show an example of necessary user intervention from the original, unabridged version. Figure 7.10 shows this graph transformation system, with the addition of two rules, createOutside and createInside, which add dynamic creation of connected locations, at both sides of the firewall. The result of adding these rules is the creation of arbitrary networks, both the safe part, and the unsafe part, making the infinite state space of this GTS truly huge and immensely complex. However, due to the fact that any new location has to be connected to exactly one existing location, this still means that the subnetworks remain disconnected, and the property remains satisfied.

Applying SGA to this example, at first without createOutside and createInside, we quickly obtain a very simple STT. It consists of the root node and 36 transitions going out from it, each leading to an STTNode that is then covered by the root node. This might seem strange, but it is indeed correct. The initial abstraction of the initial graph causes the safe processes, unsafe processes, and inside locations each to be grouped into summary nodes. Thus, the root node of the STT represents a state where there are an arbitrary, positive number of safe processes on an arbitrary network behind the firewall or on a single location in front of the firewall, and an arbitrary number of unsafe processes on a single location in front of the firewall. None of the rules, which govern process movement, process creation, and the filtering property of the firewall, can change anything about that state. Thus, every shape produced will be immediately embeddable into the initial shape. In total, the firewallFilter rule creates 24 of the resulting shapes (8 materializations each for 3 matches), the createSafe rule creates 3 shapes (2 materializations for 1 match, and 1 for another), the movePacket and movePacketBack rules create 4 shapes each (4 materializations for a single match), and the final resulting shape is created by the single match and single materialization for createUnsafe. On our



FW = FireWall, IF = InterFace, L = Location, P = Process, s = safe, u = unsafe, at = at, i,in = inside, o,out = outside, c = connection

Figure 7.10: The simplified Firewall GTS taken from [162] with the addition of dynamic location creation

reference machine, this computation takes 3 seconds.

Adding the createOutside and createInside rules to the GTS changes relatively little. The createInside rule behaves like the other rules in the previous STT, creating STT nodes that are covered by the root. The createOutside rule introduces a new shape (since the resulting summary location outside the firewall cannot be embedded into the single outside location that existed earlier, but the resulting subtree is just a larger version of the STT we had previously – all 82 nodes under the node created by createOutside are covered by that node. On our reference machine, this computation takes 5 seconds.

The relative ease with which we compute the state spaces for this example, and especially the fact that no refinement steps seem necessary stems from a key modification introduced by Rensink and Zambon[162]. This modification consists of adding the *i* and *o* labels which use unary labels to indicate the membership of locations in the inside and outside subnetworks. The original firewall example by König and Kozioura did not have these labels. The effect of this modification is that using an abstraction scheme that (among other things) uses unary labels to decide which nodes to merge into summary nodes, the *i* and *o* labels enforce an invariant separation of the two subnetworks.

This way, the separation of the subnetworks is never lost to the abstraction, and the property is fairly obvious. Therefore, removing the *i* and *o* labels again will have a huge impact on how difficult the example is to verify. This is achieved by simply removing all edges with those labels from the initial graph and all rules. The subnetwork membership of the locations is now a transitive property defined by the *c*, *in*, and *out* edges, and thus not readily available to the abstraction.

The result of this is immediately noticeable when applying SGA to the new version of the problem, which is now much closer to the original model as specified by König and Kozioura. The analysis begins constructing a large shape transition tree and goes through 3 refinement cycles before arriving at the situation depicted in Fig. 7.11. The error path under consideration begins with the application of createLocation to the initial shape. There is only one possible match for this rule, but since there are two summary nodes involved (n_3 and n_6), a total of four materializations are created. The particular materialization chosen here is one where the *L*-summary node is assumed to be exhausted (and is thus discarded), and the *future*-summary node is preserved. Since in the initial graph, there are two locations that were summarized by n_3, the corresponding trace encoding fails at the first step due to node set concerns. Since our formulation of negative materialization constraints cannot express that a given summary node contains *at least two* nodes, we cannot add a constraint here that removes this materialization, but leaves the ones that preserve n_3 intact.

Thus, this would be a point where we would have to fall back to the user to create an appropriate constraint. However, as specified by the refinement algorithm (see List-

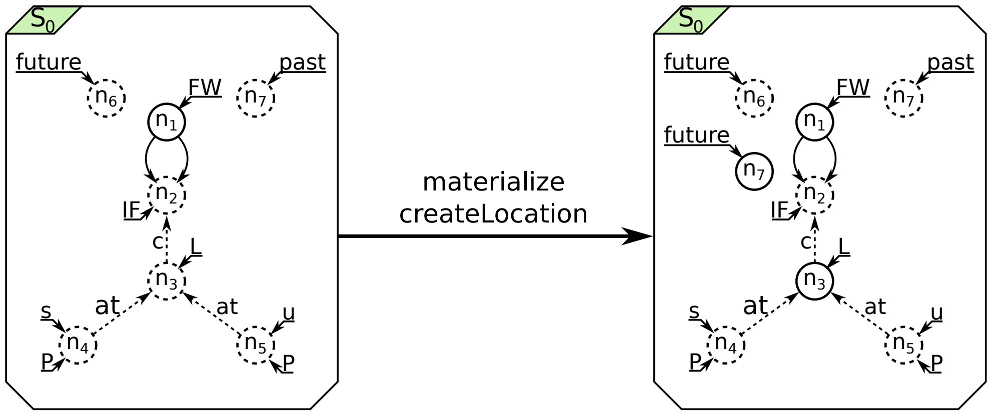


Figure 7.11: The final abstract error trace encountered by SGA on the Linear List Example with Cleanup Error

ing 6.3), SGA attempts to use the constraint anyway and use a sanity check to see if that would be unsound. SGA thus projects the negative materialization constraint for S_0^m onto S_0. There it immediately simplifies to

$$(x = y, [x \mapsto n_6, y \mapsto n_6])\,,$$

thus disallowing the materialization of any additional nodes out of the *future* reservoir.

While this is obviously not the intended outcome, it is actually sound, since the initial graph also contains only one *future* node, thereby making no statement about the minimum available space for new objects. The end effect of this is that SGA, through this interplay of materialization constraints and sanity checks, refines the initial state such that the problem is now much simpler (since no new locations or processes can be created) and solves that problem instead of the one that was originally given. This is easy enough, with the final STT being the corresponding subtree of the STT for the firewall example in the form considered by Rensink and Zambon in [162].

Still, this should not be considered more than an artifact of the implementation. The refinement step would, in a future implementation, be performed by the user, or a more sophisticated negative materialization condition. In the next and final example we will demonstrate that user-provided constraints can indeed successfully be incorporated into the process.

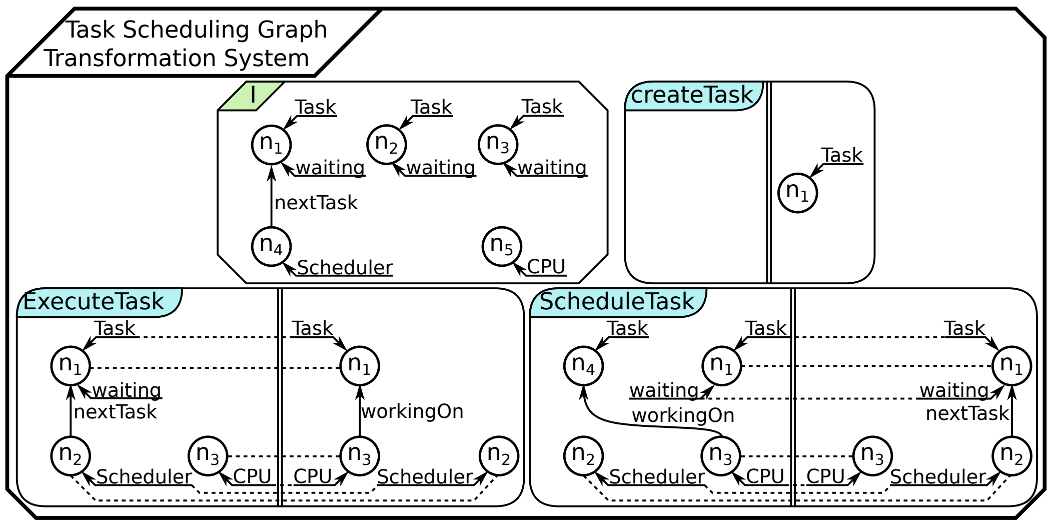


Figure 7.13: The Task Scheduling GTS

7.3.3 User Intervention Example – Process Scheduling

This example is intended to demonstrate the user intervention process, i.e. the use of user-supplied constraints in cases where both automatic processes fail. The example GTS was taken from Daniel Wonisch's master thesis[159]. It models a task scheduling system.

This system consists of a single CPU, a single scheduling agent, and a number of tasks to be scheduled. The various node types of this GTS are modeled using the unary edges *CPU*, *Scheduler*, and *Task*. If a task is scheduled to be executed, this is indicated by a *nextTask*-edge from the scheduler to the task. There can always be only one such edge. A task that is currently being processed by the CPU is indicated by a *workingOn*-edge from the CPU to the task. This edge, too, must be unique. Furthermore, a task is *waiting* until it is being worked on by a CPU.

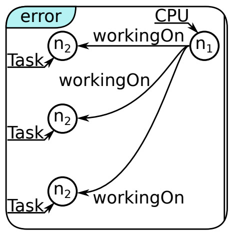


Figure 7.12: Forbidden Pattern representing a left-over *cell* node with an empty list

The rules in this GTS can create new tasks, schedule an existing task, or execute a scheduled task. Figure 7.13 shows the GTS in the form in which we will give it to SGA. The initial graph contains three tasks to begin with, one of which is already scheduled to be executed, and thus the corresponding initial shape will use a *Task*-summary node to represent these nodes.

The error that must be excluded is the scheduling of too many task for a CPU. In

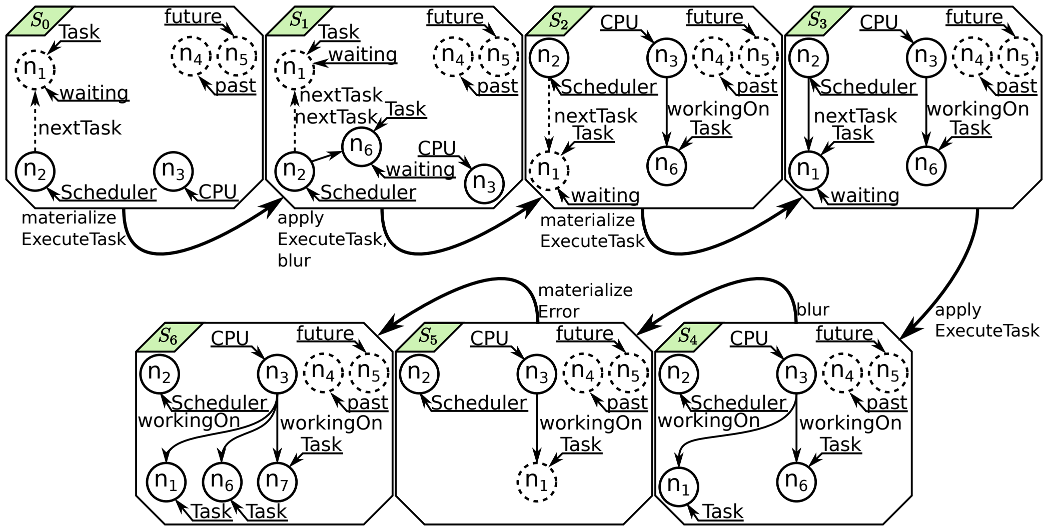


Figure 7.14: The unresolvable error path found in the first iteration of SGA for the scheduling example

our example, we will assume that the CPU node represents a dual-core CPU, meaning that at most two tasks may be executed by it. The error pattern for our GTS is thus a CPU executing three tasks simultaneously, as depicted in Fig. 7.12. Examination of the initial graph and the rules reveals that this error pattern cannot occur in the GTS, regardless of how many tasks are created – each scheduling of a task requires the completion of an executing task. Since only one task is scheduled at the beginning, there can never be more than one scheduled task and thus never more than one executing task. Note, however, that the information that there is only one *Task*-node that has an incoming *nextTask*-edge is lost in the initial abstraction.

This information loss in the initial abstraction poses a problem for the STS, which SGA quickly uncovers during the construction of the STT. SGA finds the error path shown in Fig. 7.14, which consists of two consecutive applications of the ExecuteTask rule. Clearly, ExecuteTask cannot be executed twice in succession, due to the fact that there is only one *nextTask*-edge in the initial graph and ExecuteTask does not produce additional *nextTask*-edges.

While there are many details of this trace that exhibit behavior outside the range of the concrete system[†], during the analysis of the trace, SGA discovers that the failure point is clearly the second application of ExecuteTask. Specifically, the absence of an actual (as opposed to ½) *nextTask*-edge from the scheduler to a waiting task is identified as the cause of the abstraction failure. The problem with this can be easily seen in

[†]such as the materialization of three *Task*-nodes out of a summary node that is known to only contain two

the shape S_2 in Fig. 7.14, which represents both the unabstracted result shape of the first application of ExecuteTask, and thus the incoming shape for the STT node containing the shape at the failure point, as well as the central shape of that STT node. Both shapes allowed for such edges to exist. The sanity check thus discards both the interpolant-based constraint, and the negative materialization constraint (which also disallows such an edge because the left-hand side of the rule requires it).

At this point, in a full implementation of the algorithm described in Chapters 5 and 6, the user would be given the opportunity to provide constraints for the shapes from the initial shape to the failure point. In order to simulate this exchange with SGA we modified it to use a hard-coded user-generated constraint for this specific test case. The problem with the automatically-generated constraint would be relatively obvious to the user – they would be presented with the interpolant-based constraint, the raw interpolant and the counterexample for the sanity check. This last item is the key component. The sanity check, i.e. the embedding check for the incoming shape into the central shape augmented with the proposed constraint, returns SAT when it fails. This means that, as described in Sec. 6.3, the model of this satisfiable encoding contains a graph that is embedded into the incoming shape, but not the central shape. This graph (see Fig. 7.15) is presented to the user and shows an existing *nextTask*-edge between the scheduler node and a task node – which the user knows can't exist. Therefore, the actual abstraction error happened earlier in the trace and must be dealt with at that point.

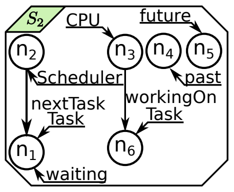


Figure 7.15: A Graph embedded into the incoming shape, but not the refined central shape at the failure point

This point, as indicated before, is the initial shape. Being the source of the initial ExecuteTask application in our error path, it does not contain the information that there is only one task with an incoming *nextTask*-edge. This is ultimately the reason why the abstraction was able to execute two (and by virtue of another overapproximation, three) *Task* nodes. The user determines that this information should thus be added to the initial shape. They provide the constraint

$$\left(\neg\left(\neg\left(x = y\right) \wedge \mathit{nextTask}\left(z, x\right) \wedge \mathit{nextTask}\left(z, y\right)\right), \left[x \mapsto n_1, y \mapsto n_1, z \mapsto n_2\right]\right),$$

which expresses exactly that there is at most one *Task*-node with an incoming *nextTask*-edge, and add it to the shape S_0.

The refinement of the central shape of the root node causes SGA to discard the STT thus far constructed and restart at S_0, now with the new user-supplied constraint. This leads to the construction of the STT shown in Fig. 7.16. The constraint provided by the user is clearly sufficient to prove the validity of the STT.

The application of ExecuteTask (yielding S_8) is now no longer followed by another such application, since this is prohibited by the constraint. Executing a task and then scheduling a new task leads back to the initial shape, as expected. Creating a task only adds a node to an existing summary node (shapes S_8 and S_0), or causes a merger with an existing non-summary node (shape S_9). The STT can even substantiate the intuition that, given this GTS, ExecuteTask and ScheduleTask actions always occur alternatingly (modulo applications of CreateTask). Note also that the actual property being verified has been generalized – the property holds for all GTSs that have an initial graph that can be embedded into the intial shape, not just the initial graph that was actually provided.

Having provided illustrative examples for every refinement method supported by our implementation, as well as user intervention, we now move on to point out those areas in which SGA has the most room for improvement in the future.

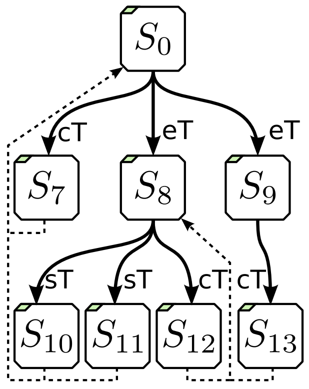


Figure 7.16: STT after user refinement (rule names abbreviated)

7.4 Possibilities for Extension

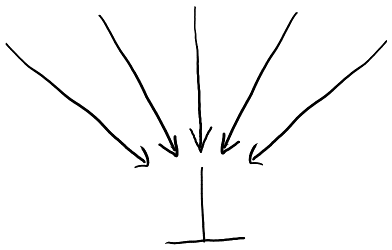
In order to avoid overlap with Sec. 8.2 (Future Work), we will focus exclusively on such improvements that could be realized without significant changes to the underlying theory.

Interactive Refinement One obvious possibility would be to add an implementation of the manual refinement process described in Chapter 6. Since implementations of all the necessary checks are already in place for the various other parts of the algorithm, the only significant investment of work necessary to enable interactive refinement would be the design and implementation of the user interface itself, as the current implementation is simply not designed to handle this kind of user interaction.

Negative Application Conditions When using shape constraints for abstraction refinement, one useful corollary one obtains from that is the ability to incorporate negative application conditions[83] into the analysis. As previously mentioned on page 178, negative application conditions can be realized using shape constraints. This was also incorporated into an earlier version of SGA. Since then, the abstraction refinement process, as well as the entire concept of state space construction has changed significantly. While implementing this, support for NACs was deemed inessential and was thus not transported into the new code. Relatively little effort would be required to update the encoding implementation to be able to handle negative application conditions again. This would enable SGA to analyze a wider variety of graph transformation system and would be an important step towards usefulness in realistic settings.

Compatibility Constraints Similar to the issue with NACs, compatibility constraints are a feature that was supported by the first iterations of SGA, and has since been rendered incompatible. Compatibility constraints are first-order formulas that state general truths about all graphs in a system. They were the starting point for Daniel Wonisch's development of Deductive Constraints, which in turn were the direct predecessors to the Shape Constraints used in this thesis. Compatibility Constraints allow for a very effective way of incorporating domain knowledge into the analysis, and have the additional advantage that they can be used to concretize shapes when they express information that is more general than would be necessary given the constraints. With relatively little effort, SGA (and, to a much lesser extent, the underlying theory) could be adapted to allow for the specification of compatibility constraints again, improving the user's influence over the abstraction further.

Most other improvements require more significant changes to the theory, and are thus relegated to the next chapter.



8 Conclusion

The verification of structurally dynamic models is a difficult problem. The new paradigm of model driven software development also makes it a very important problem, since behavioral modeling, e.g. of reconfigurable systems, is constantly growing the number of real-world examples where the safety of such models is of paramount importance. The lazy state space construction algorithm with interpolation-guided refinement presented in this thesis constitutes a new approach to checking the safety of such models. It does so by solving the coverage problem for infinite-state graph transformation systems*. The approach has been shown to be effective on some examples, but also shows definite room for improvement in more complex use-cases.

In this chapter, we will first conduct a survey of related techniques in the field. Then, we will discuss the current shortcomings of our approach and point to future work aimed at alleviating said shortcomings. Finally, we will summarize what was achieved in this thesis with respect to the goals laid out in Chap. 1, review design decisions made along the way and offer concluding thoughts.

*within the bounds of decidability

8.1 Related work

As already mentioned in the introduction (see Chap. 1), graph transformations and graph transformation systems are applied in a multitude of settings, for a variety of purposes (see, for example[68, 76, 123, 147, 150]). In many of these cases, it is important that the graph transformation systems satisfy some kind of safety and/or correctness property. And often, these systems exhibit intractably large or infinite state spaces.

It is therefore not surprising that a number of approaches have been proposed to tackle the problem of verifying infinite-state graph transformation systems. In this section, we will provide an overview of the work that has been done in this field, categorizing the various different approaches by the way they reduce the size of the state space they have to consider. Works that are related to this work by the techniques used rather than the problem tackled are referenced at the appropriate place where the techniques are discussed, such as on pages 26, 70, 132, and 140.

In the domain of model transformation, rather than basic graph transformation, there exist also a number of approaches that aim to verify certain properties of transformation languages that are in many cases based on graph transformation. The more application-focused nature of model transformation research leads to a situation where the main focus for the verification of such transformations lies on issues that cannot occur in basic graph transformation systems, such as syntax issues or semantic equivalence. There are approaches that seek to solve similar issues arising from the use of graph transformation systems[44, 78, 145, 152, 157], but to the best of my knowledge these approaches always either artificially reduce the state space by considering bounded versions of the actual problems (and are thus related to the approaches discussed in Sec. 8.1.5), rely mostly on the power of human designers, e.g. by involving interactive theorem provers in the process, or greatly reduce the expressiveness of the language concerned. For this reason, we will focus in this section on more theoretical work on basic graph transformation systems, since this achieves the goal of giving the reader a good overview over the kinds of approaches that exist, without going into any application-specific detail.

The categories that we will use to organize the various approaches are explicit state representation, abstraction by patterns, abstraction by grammars, and bounded analysis techniques. *Explicit State Representation* techniques are techniques that use abstract states that provide an abstract counterpart to every component of the concrete states they represent. This thesis is an example of an explicit state representation technique. A variation on this are techniques performing *Abstraction by Grammars*. They employ specialized grammars to construct the abstract states and vary their precision. In contrast, techniques that use *abstraction by substructures* use abstract states that describe only a part of the concrete states they represent, generally by using some form of pattern matching. Finally, *Bounded Analysis* techniques obviate the need to look at abstract states entirely by abandoning the notion that the entire state space has to be searched. Such techniques usually explore some finite prefix of the state space, and thus offer the assurance that if there is an error in the system that was not found, it must be at least some fixed number of transitions long, or at least of some fixed size, depending on the cutoff used.

Since they represent the base case of all these techniques, and because some of the related work represents extensions of them, we will begin by surveying basic state space construction and verification approaches for graph transformation systems that are not explicitly able to deal with arbitrarily large state spaces.

8.1.1 Finite-State GTS Verification Techniques

For more than a decade now the importance of graphs and graph transformations as a formal modeling tool has been steadily growing, and the advent of model-driven development processes has greatly accelerated that growth. With increased use of graph transformations, the verification of dynamic processes using them came more and more into focus.

Starting with a paper by Heckel, Ehrig, Wolter and Corradini[84], a basic formalism was developed in which the reach set of a given graph transformation system could act as a state set which would then be linked by the rule applications on those states to a transition system. Together with some notion of graph properties, such as, e.g. graph patterns, this formalism could be used as a basis for model checking graph transformation systems. From there, two main approaches to model checking finite-state graph transformation systems emerged – translating graph transformation systems to inputs for regular model checkers, and writing model checkers specifically for the use with graph transformation systems. We will survey a selection of such approaches that eventually led to the development of usable tools for the verification of graph transformation systems.

Translating graph transformation systems into model checker inputs is an approach used by several tools and their accompanying formalisms, including CheckVML[140] (Check Visual Modeling Languages) and OBGG[63] (Object Based Graph Grammars).

In CheckVML, a given graph transformation system with single pushout semantics is first translated into an abstract – read: tool-independent – transition system from which it can then be translated into any number of back-end model checkers. The first model checker used for this purpose was SPIN[94]. Note that in CheckVML, the graph transformation system is given in a special modeling language, specified by a meta-model. As such, CheckVML provides (and requires) much more context for the models it considers than similar approaches.

The OBGG approach was conceived of as a way to specify and verify distributed software written in object-oriented programming languages. As such, its formalism puts some restrictions on the models and transformations that can be used, mainly to preserve and enforce the semantics of message passing on which the systems built in OBGG are based. Furthermore, object deletion is prohibited in OBGG. These restrictions lead to many tasks that a verification or, for that matter, a simulation must perform to be much simpler than in more general approaches like CheckVML or GROOVE. The OBGG approach, like CheckVML, provides verification capabilities by translating its own modeling language into Promela[28], the input language for the SPIN[94] model checker.

As an example of an approach that does not rely on existing classical model checkers, we look at GROOVE[75], which was already introduced in Chapter 2. It represents the most direct implementation of the concept of model checking a graph transformation system, and is also the one of the most application-agnostic tools available, requiring no meta-model and implementing no additional restrictions for the graphs and rules it uses. GROOVE verifies graph transformation systems (using properties specified in CTL over graph patterns) by constructing the resulting graph transition system and treating it as a Kripke structure, upon which it performs its own implementation of the explicit CTL model checking algorithm. It is also the basis for some of the explicit state representation techniques discussed in the next section.

A further example of an approach that provides simple model checking of graph transformation systems was implemented in the highly versatile tool platform Fujaba[163]. Here, the basic graph transformation formalism was extended by merging it with the concept of an activity diagram to obtain a certain level of execution control over the graph transformation rules. This implementation was demonstrated at the transformation tool contest[111] in 2010. Further development in Fujaba that concerns the verification of graph transformation systems has focused on the support of additional modeling elements, such as timing constraints, and the subsequent verification of the result via a translation into a general-purpose model checker[66, 86], in this case UPPAAL[27].

There are of course many other tools and specification languages that model graphs and graph transformation systems to some degree, such as AGG[146], GrGen[74], and PROGRES[141]. The focus of these approaches is not on verifying graph transformation systems after they have been defined, though. They either provide correctness-by-construction properties by automatically modifying rules such that certain consistency conditions are not violated (AGG), or focus on entirely different aspects than verification altogether, such as efficiency (GrGen).

8.1.2 Explicit State Representation Techniques

For the category of explicit state representation techniques, we will discuss three approaches: Neighborhood Abstraction, Pattern Abstraction, and Petri Graph Abstraction. These each represent a completely different approach to the problem of graph abstraction and show the diversity of techniques that exist.

Neighborhood Abstraction

In its outward appearance, neighborhood abstraction looks quite similar to the approach presented in this thesis. It summarizes individual nodes using summary nodes, based on a notion of similarity, resulting in abstract graphs called shapes. It also provides algorithms for materialization and application of rules to shapes and thus creates an abstract shape transformation system. The differences between neighborhood abstraction and our approach lie in the underlying formalisms.

In a series of papers[24, 36, 125, 126]†, the authors Rensink, Distefano, Bauer, Wilhelm, Boneva, Kurbán and Kreiker developed a novel formalism for abstracting graphs. Nodes and edges in shapes are annotated with multiplicities that are precise up to a predefined bound ω. Graphs are then represented by these shapes if a relationship similar to our embedding relationship exists between the graph and the shape which additionally satisfies these multiplicity constraints.

The decision of which nodes to merge is made by a process that gives the approach its name. Nodes whose *neighborhood*, i.e. the local context in their respective graphs, made up of their adjacent nodes, up to a predefined bound k, is equivalent, are considered to be similar, and thus should be merged into summary nodes. This way, a canonical abstraction can be defined, which guarantees that for any fixed ω and k, there are only finitely many possible different shapes for any given label set. This is very advantageous, since it means that for a given ω and k a process creating the abstract state space of a GTS is guaranteed to terminate.

In order to formulate properties of graphs that can be checked with this abstraction approach, a modal logic[22, 124] for graphs and shapes was created, that allows for the specification of local transitive properties, and has the ability to precisely count up to ω. This particular modal logic is preserved and reflected by the abstraction if sufficiently large k and ω are chosen, i.e. verifying a property for the abstraction is equivalent to verifying it for any of its concretizations. This is an impressive result that has allowed the authors to verify some interesting properties of challenging case studies, such as the firewall case study also investigated in Chap. 7.

A verification of an infinite-state graph transformation system using neighborhood abstraction works by choosing very low initial values for ω and k, just enough for the properties, and constructing the corresponding state space. If this is not sufficient to prove the property, either ω or k are increased, and the process continues. Neighborhood Abstraction has been prototypically implemented in the GROOVE tool[127].

†not a complete list

The weaknesses of this approach include the following. Neighborhood abstraction has similar difficulties including a true transitive closure into its properties as we do. Introduction of such unbounded transitivity would likely render the most impressive reflection properties of this approach invalid. Furthermore, neighborhood abstraction is unable to react to abstraction failures, i.e. the abstraction being too coarse to verify a property, in a manner that is lazy and/or takes the spurious counterexample into account. The only option for refinement is to increase one of the one-dimensional precision parameters and recompute the state space. This is a weakness that is being tackled by *Pattern Abstraction*, described in the following section.

Pattern Abstraction

Pattern Abstraction, first described in a paper by Rensink and Zambon[128] and later extended in Eduardo Zambon's PhD Thesis[161], can in some ways be seen as a more sophisticated reinvention of neighborhood abstraction. It aims to alleviate the most pressing problem with neighborhood abstraction, which is its reliance on a single integer – the radius k[‡] – to define the precision of the abstraction. Not only is this abstraction refinement approach one-dimensional and unable to incorporate any additional information, experimental results[161] also show that even unit increases on small k result in massive increases in the size of the state space.

Pattern Abstraction tackles both of these problems by introducing *pattern graphs*. The basic idea is this: rather than stating that any subgraph of size k constitutes a pattern that should be preserved by the abstraction, we explicitly define a well-structured hierarchy of graph patterns that should be preserved. This gives the user the ability to identify important patterns in the graph transformation system and instruct the abstraction to preserve those. Similarly to neighborhood abstraction, multiplicities of pattern occurrences are counted precisely up to some bound ω.

The ingenious formalism developed by Zambon in his thesis makes it possible to construct well-defined shape transformation systems based on any set of sub-patterns, all while sticking mostly to constructive definitions, facilitating implementation. Furthermore, the fact that abstraction refinement is done using graphical patterns should make a tool constructed for this theory usable even for non-experts.

However, Zambon does not give any instructions about how to arrive at the insight which patterns would be useful to the abstraction, presumably leaving that to the ingenuity or domain familiarity of the user. Furthermore, as of the time of this writing, no implementation of this approach has been released and no experimental results have been published, which makes it difficult to gauge the practical applicability of the approach with any authority.

[‡]and, to a lesser extent, the multiplicity bound ω

Petri Graph Abstraction

Petri Graph Abstraction is an approach to abstract graph transformation developed and analyzed by König, Corradini, Baldan and Kozioura in a series of papers[17] reaching back to 2001[15]. The fundamental idea of Petri graph abstraction is to reduce the problem of creating the state space of an infinite-state graph transformation system to a formalism, in this case Petri nets, where the analysis methods available are more mature. This is done by creating a Petri graph, using the edges of the initial graph as the places of a Petri net, and the rules of the graph transformation system as transitions. The initial marking of this Petri net indicates the occurrences of the various edges in the initial graph, and the firing of transitions then transforms this marking to indicate the added or deleted edges. Newly created edges (and possibly the newly created nodes that support them) create new places for the Petri net and thus grow the state space. To curb this growth, the Petri graph is folded back in on itself when appropriate.

Note that while edge deletion can be modeled by removing markers from edges, node deletion can not be expressed in this formalism. Note also that unlike in many other formalisms, the verification task is not intertwined with the construction of the state space, but rather happens after the fact, as an analysis of the final Petri graph. Petri Graph Abstraction is based on hypergraphs and double pushout transformation semantics, rather than simple graphs and SPO semantics like our approach.

Like all approaches based on overapproximation, spurious counterexamples are a possibility in this approach. While their initial offering did not include a way to refine the abstraction beyond a one-dimensional parameter similar to the one used by neighborhood abstraction, this restriction has been lifted since then[102]. The approach now includes a way to analyze spurious counterexamples and identify nodes along the way whose merging caused the abstraction error. Prevention of the merging of these nodes then removes the spurious counterexamples.

König et. al were able to prove many interesting properties for a variety of systems with this approach, such as cycle-freedom properties, reachability properties, and the absence of certain patterns. Their reach thus extends to safety as well as liveness properties. The extensive body of work on this approach also contains a fairly mature implementation called AUGUR[103].

8.1.3 Abstraction by Grammars

The observation that graph abstractions essentially relate graphs to other graphs, just like graph transformations do, motivates the next set of approaches. These approaches seek to approach abstract graph transformation essentially as two “perpendicular” transformation systems – one system that models the system behavior, and one whose function it is to make the states produced by the first system more abstract, or reconcretize them when required. This is more related to the shape analysis approach[134] that our basic abstraction is based on than our approach itself, but we will include it here since it provides a distinctly different view of graph abstraction. There is quite a bit of work that falls under this category[14, 62, 107], but here we will restrict ourselves to one representative approach, since it provides a good jumping-off point to research other approaches and because it culminates in the presentation of a usable tool.

The approach we will look at is the one developed by Heinen, Noll and Rieger[85, 130]. It, and the tool that implements it, are called Juggrnaut (Just use graph grammars for nicely abstracting unbounded structures). The basic idea is to use hyperedge replacement graph grammars (HRGs) to specify valid pointer structures in the heap. Thereby, a pointer structure would be modeled by a hypergraph, and an HRG would be used to define whether the graph was *valid*, i.e. parsable using the grammar, or not.

Thus, this approach uses a single abstract state – represented by the grammar – and intermediate modified states, represented by intermediate graphs for which the parsing fails. The actions of pointer-manipulating programs, in this case using a simple programming language capable of pointer assignment, heap allocation, as well as conditional and unconditional jumps, are then interpreted as graph transformations on the abstract state. This state can be concretized using the reverse application of the parsing rules, and reabstracted using forward application after the rule has been applied. This approach was implemented in a tool[85].

The authors were able to show many relevant properties for their use case of a binary tree-manipulating algorithm, such as termination, coverage, and, of course, preservation of the tree shape. While the use of grammars to abstract states is an interesting approach, it should be pointed out that a great deal of user interaction goes into creating those analyses. The shape of the data structure to be modified, represented by the HRG, must be supplied by hand. Furthermore, the modeling of the pointer operations as graph transformation operations must also be done by hand. The results are thus not immediately transportable to other pointer manipulating languages, or, more importantly, to other data structures being manipulated.

This is an inherent downside to the idea of using grammars for abstraction. Unless the grammars are derived automatically on-the-fly to abstract common substructures, it will always be necessary to handcraft grammars for every new data structure to be verified. It is interesting to note that the pattern-based abstraction described in Sec. 8.1.2 comes relatively close to this intuition. The hierarchical pattern-graphs used there are similar to limited graph grammars specifying the structure of graph states. However, even there a process of automatic or even semi-automatic abstraction refinement (or even abstraction generation) is still missing as of this writing.

8.1.4 Substructure-based Abstraction Techniques

Unlike the previously discussed approaches that seek to present abstract states that in some way *cover* a (possibly infinite) set of concrete states, usually by incorporating special "abstract" components into graphs, the approaches described in this section take the opposite view. Rather than trying to express a high level state directly, they use certain substructures of graphs to represent all graphs that contain (or do *not* contain) those substructures.

Pattern-based Abstraction with Backwards Application

The first approach that we will discuss here was developed by Schilling, Giese, Becker, Beyer, and Klein[25] and further elaborated upon in Daniela Schilling's PhD thesis[139]. This approach was for the purpose of modeling and verifying structural adaptations in the context of distributed mechatronic systems. It uses graph patterns to represent the substructures used to group sets of graphs into abstract states. Since graph patterns have been used as "predicates" on graphs (see Sec. 8.1.1), this is strongly reminiscent of predicate abstraction, where logical predicates over program variables are used to represent the set of all states in which they evaluate to true.

More specifically, the central idea of this approach, is the representation of graphs via graph patterns, which themselves are graphs, possibly extended by negative application conditions. A graph is represented by a graph pattern, if an injective match for it can be found that can not be extended to include any of its negative application conditions. Thus, a single pattern represents an infinite set of states, and a set of patterns can be used to cover an entire state space of a graph transformation system.

However, this approach does not seek to achieve a comprehensive representation of the entire state space by forward constructing all possible states from some form of abstraction of the initial state, especially since in the intended application domain a single, well-defined initial state is often hard to come by. Rather, the opposite approach is taken. Analysis starts at a set of forbidden patterns that formulate an *invariant safety condition* for a GTS. Then, all possible combinations of how any rule in the rule set could generate, or "complete" one of these patterns are computed. Each of these rule and pattern combinations is executed backwards, to obtain a pattern that represents an abstract state leading to the error state. If, by this method, a pattern can be constructed that contains none of the forbidden patterns, i.e. a pattern representing a safe state, then the claim that the absence of the set of forbidden patterns constituted an *invariant* property is refuted and the system is declared unsafe.

By this method, infinite-state graph transformation systems of high complexity can be proven to satisfy a given inductive invariant with comparatively little effort. The analysis always terminates either with the message that the system is safe, or with a set of counterexamples.

Considering that termination could never been guaranteed by any of the previous examples, this is an impressive result. The price for termination is paid here by the restriction on the properties that can be verified, and the graph transformation semantics that can be used. This backwards application means that the approach is limited to graph transformation semantics that are reversible, such as DPO, while less restrictive ones, like SPO, are not supported. More importantly, the restriction to inductive invariants can be quite severe. For a system to be safe, it is not sufficient merely to never construct a forbidden pattern, but rather the rules must be set up so that the forbidden pattern can never be constructed regardless of circumstances. Nevertheless, many interesting properties, especially in the domain of distributed mechatronic systems fall into this category and the approach has been successfully applied to case studies from that domain[139].

The restriction to inductive invariants and the disregard for initial states are not fixed features of the basic approach, though. This is evidenced by the modification of the approach introduced by Saksena, Wibling and Jonsson[136], who, rather than doing the inductive invariant check, used backwards application of rules to forbidden patterns to perform a classical fixed-point computation. This means that, beginning with the rule-and-forbidden-pattern combinations, rules are applied backwards over and over again, creating ever new patterns that are allowed to subsume each other. Eventually, a stable state is reached, where the computed patterns cover the entire state space backward reachable from forbidden patterns. If the initial state is not covered by this pattern set, then the system is declared safe. While this adds the requirement of some kind of initial state description, and loses the termination guarantee that the approach by Schilling et.al. possessed, it also opens up the possibility to verify a much larger set of potential properties. In their paper, Saksena et.al. use this method to verify an ad-hoc routing protocol called DYMO[136].

Overapproximation of Graph Transformation Systems using Well-Quasi Orders

The second approach we will discuss was developed by König and Joshi in 2008[100]. Rather than using patterns, it relies on recent mathematical discoveries about graph substructures called *minors*. A graph G is a minor of a graph H if G can be obtained from H by deleting or contracting edges, or removing isolated nodes (or, of course, some combination thereof). Over the last couple of years, research by Robertson and Seymour[131, 132] has revealed that the graph minor relation defines a well-quasi order on graphs.

This is useful because it can be used to express certain classes of graph transition systems as well-structured transition systems (WSTS)[100]. This implies that a finite number of graph minors can be used to represent each such WSTS, and that finding this set of minors is decidable. König and Joshi present a backwards reachability algorithm for error graphs using these results and use it to verify a simple leader election protocol in their initial paper[100].

It should be noted that the restrictions placed on the graph transformations systems in this approach are really quite severe. According to König and Joshi, GTSs must either be composed exclusively of rules with disconnected nodes or edges in their left-hand side, or be transformed into a compatible GTS by adding rules that perform edge contractions for every label in the graph domain. Clearly, in many application domains, these are prohibitive restrictions.

This result was later generalized by König and Stückrath in 2014[104]. Here, the authors widen the scope of their approach to include well-quasi orders induced by other kinds of relations between graphs, specifically the subgraph relation and the induced subgraph relation. While these new well-quasi orders and the theoretical framework within which they are placed now allows for backwards reachability analyses for the coverage problem for graph transformation systems with arbitrary rule sets, this relaxation comes at the price of additional restrictions put on the kinds of graphs that can be represented. The most notable of these restrictions is given by the boundedness of the length of all undirected paths by a fixed integer k, limiting the size of contiguous components that the graph transformation system can be able to produce.

Regardless of these restrictions, the fact that this approach offers decidability for the coverage problem with caveats that may not be relevant in many application scenarios makes this an important result. A prototypical implementation exists and was used to evaluate the practical feasibility of this approach[104]. The results indicate feasibility, since all GTSs tested were verified in well under a minute, usually a few seconds.

8.1.5 Bounded Model Checking Techniques

In the field of model checking, the idea of managing intractably (or arbitrarily) large state spaces by examining only a small, well-defined part of it, ideally one that is likely to contain errors, is well known. For graph transformation systems, this idea has also been gaining traction in the past decade, even if most approaches introduce the bound on the state space often in an implicit way or otherwise do not strictly follow the example of classical bounded model checking[34]. Such approaches include, e.g., the approach by Tichy and Klöpper[148], which transforms GTSs into planning problems in the PDDL language, the approach by Baresi and Spoletini[19], which uses the logical specification language Alloy to verify properties of GTSs in a bounded way, or the approach by Lafuente and Vandin[109], which translates GTSs into rewriting logic and checks properties specified in quantified μ-calculus on bounded prefixes of the resulting state space. The graph transformation tool GROOVE[75] also has a mode that allows for bounded state space exploration and verification. Here, we will represent these bounded approaches by the approach given by Isenberg, Steenken and Wehrheim in their 2013 paper[97]. This approach includes the SMT encoding that was the inspiration for the trace encoding presented in Chapter 4.

Bounded model checking of graph transformation systems via SMT solving uses SMT encodings of GTS execution paths of incrementing lengths. All aspects of graph transformation, including isomorphism checking for the insertion of backwards transition edges into execution paths are faithfully represented by the encoding. This allows for the classical formulation of bounded LTL model checking to be directly applicable to the encoding. The LTL formula to be checked is encoded into the same SMT formula, thus creating an encoding of a path of length k that violates the k-bounded semantics of the given LTL formula. Clearly, any model of such an encoding is an error, i.e. a path through the GTS that violates the LTL property to be checked.

In contrast to our approach, thus, this approach is able to handle full LTL model checking over graph patterns as predicates, with the caveat that only errors of a length up to a certain k can be excluded. This represents a different trade-off than the one accepted in this thesis – where we trade decidability for the ability to potentially rule the entire state space safe, this approach, along with all other bounded approaches, sacrifices the ability to exclude errors in favor of decidability. However, as McMillan observed[112], it is possible, under certain circumstances, to use Craig interpolation to lift this technique from a bounded state space exploration to an unbounded state space exploration. This is an interesting avenue for future research in SMT encodings of graph transformation systems.

8.2 Future Work

While the theoretical approach taken in this thesis provides a semi-automatic verification technique for infinite-state graph transformation systems that is able to fully automatically verify some examples, the results obtained by its implementation make it clear that there is still room for improvement. The prospect of using SMT solving and interpolation to refine the abstraction is promising, and similar work in the field of program verification using such techniques suggests a great potential of this approach to find suitable abstraction refinements. However, the theory, as it has been developed so far, has as yet failed to fully deliver on this promise. In this section, we will explore a few avenues of work that have the potential to greatly increase the usefulness of our approach.

8.2.1 Shape-Based Refinement for Node Set Errors

In our lazy abstraction refinement approach, we thus far attempt an interpolation based refinement first, followed by a materialization based refinement if that fails, and finally refer to the user if the negative materialization constraint turns out to be unsound. In the vast majority of cases where the interpolation based refinement fails, it does so because the problem is that a materialization assumes that more or fewer of a particular kind of node exist than actually do exist. In the majority of cases where the subsequent materialization based refinement fails, this is due to the fact that while the materialization based refinement as given by Def. 94 can express that *too many* of a given type of node have been materialized, it is unable to express that *too few* such nodes have been materialized. This is because, fundamentally, the guaranteed existence of nodes in our formalism is something that is most naturally expressed by the inclusion of said nodes in the shape, rather than by a constraint added after the fact.

A possible solution for this would be to introduce a new refinement mechanism that splits summary nodes from which too few nodes were materialized into a number of concrete nodes plus the original summary node, in essence, performing a kind of materialization on the central shape. This would then preclude the offending materialization from being performed again. Information about which nodes to split could be obtained from the trace encoding where, with a slightly modified encoding, the information which parts of the materialization embedding could not be realized due to node set concerns could be obtained by using fine grained formula partitioning and unsat cores.

The introduction of this method into the LSSC algorithm would necessitate a change in the STT definition, since individual shape transformations can now result in different shapes and thus different nodes, based on which of the summary nodes of the resulting shapes were split during a refinement. The claim that valid instances of such modified STTs constitute proofs for the validity of the underlying GTSs must then be reexamined.

8.2.2 Transitive Closure in Constraints

One of the most difficult to verify properties in graph transformation systems are those that arise from the transitive closure of local graph properties. A good example for such a property is the network separation property in the firewall example (see Sec. 7.3.2). The membership of each location in either the network outside or inside the firewall is determined by its transitive connection to the respective interface of the firewall. The fact that the initial abstraction lacks this information lead to the summarization of both subnetworks into the same network and thereby to spurious errors as processes navigate connections into the protected network that do not exist in the concrete system. Without a transitive closure operator, it is very difficult to express such properties.

However, since the three-valued shape analysis approach by Sagiv et. al.[134] does support transitive closures, and support for transitive closures is under development for at least one of the SMT solvers used in this thesis (Z3[35]), it is conceivable that this restriction could be lifted in the future. Even if the automatic refinement procedure remains unable to produce constraints utilizing transitive closure, the ability for the user to specify such constraints would still greatly improve the reach of our approach. The most significant change to our formalism would have to be made to the shape constraint transformation definition, which would have to be amended to handle the transitive closure operator.

8.2.3 Separation of Concrete Graph Predicates

The main reason our approach currently needs the ability to fall back to a human designer to provide certain types of constraints is that automatic refinement is always localized to a single shape, the one at the failure point. This is not ideal, since in many cases, the shape at the failure point cannot effectively be refined without its predecessor in the error path also being refined. As we saw in Sec. 5.1, the usual process for abstraction refinement with interpolation does add constraints along the entire path, not just in one place for this very reason.

On page 180 we go into some detail about the reason for this shortcoming. Essentially, the way the encoding currently works, there is no way to obtain a clean, usable constraint from an interpolation at any point in the trace other than the failure point. This is because the predicates that make up the concrete graph must be carried through the entire trace and are thus part of every interpolation cut, except for the one at the failure point because the part of the trace behind the failure point can be omitted.

Solving this problem is difficult. One possible approach might be adopting a multi-layered encoding. On the first level, rule application is relegated to the shape level, while the graphs are only represented via embeddings into the various shapes. While this would introduce usable interpolation cuts at every step in the trace, it also is not an entirely concrete encoding, since even when the embeddings remain constant, the properties of the graphs that the shapes make no direct claims over are subject to change over the course of the trace. Thus, each step in such an encoding re-introduces a bit of abstraction. A second encoding level using only concrete graph applications guided by the shapes at the abstract level could complement the first level in providing both an accurate failure point and assessment of the feasibility of the error, as well as workable interpolation points.

8.2.4 Relaxation of Initial State and Error Conditions

In our approach, we have so far assumed that there is a single, well-defined start graph for the GTS, as well as a single error pattern. This decision was made to simplify the setup of the overall algorithm. However, there is no compelling reason for why these restrictions can not be overcome. It would, for example, be possible to replace the initial graph with an initial state, possibly already annotated with domain knowledge-derived shape constraints by way of user input[42]. Only the trace encoding would have to be slightly adapted, e.g. by including the initial constraints in the encoding since they are not valid refinements, for that to become a possibility.

Similarly, but requiring a bit more work, the error pattern could be replaced by an error *shape*, and the checking for a match for the error pattern could be replaced with the search for an embedding of the error shape into any subshape of a given central shape. While that would make the error checking itself more complicated, the scope of verifiable properties would grow significantly.

8.3 Summary and Concluding Thoughts

In this thesis, we set out to create a novel mechanism for formally verifying safety properties of structurally dynamic systems, such as they occur in the modeling of, e.g. reconfigurable systems, in model driven software development. In our formalization of such models, this came down to solving the coverage problem for infinite-state graph transformation systems. By way of SMT encodings of its various concepts, and the prior work on three-valued abstractions that existed at the time, we sought to leverage tried-and-true techniques from other branches of verification, such as lazy abstraction and interpolation. With this approach, we hoped to create a technique that would improve upon existing work mainly in the area of abstraction flexibility and expressive power. The final goal was to implement a tool that would showcase the new technique and prove that it could be realized.

For the most part, we have accomplished those goals. First, we focused and generalized the three-valued logic abstraction for memory stores created by Sagiv et.al.[134] by transferring it to the graph transformation domain, thus creating a system that was capable of representing infinite state sets with finite shapes, and of transforming those shapes in a sound abstraction of regular SPO semantics. Among the various available abstraction formalisms to base our approach on, we chose this one because of its direct link to formal logic. The ability of this formalism to directly evaluate first-order formulas on the structures used and thereby refine the abstraction was seen as a key component of creating a highly flexible and expressive abstraction refinement framework. We added such a refinement formalism, inspired by instrumentation predicates, to our shape system, and created an automatic update mechanism for that formalism that allows for very high precision, being not only sound, but even complete under certain circumstances[§]. Building on this formalism, we showed how an abstract shape transition system could be created that was able to finitely and soundly represent the infinite state spaces that are so characteristic for many graph transformation systems.

Subsequently, we tackled the unavoidable problem of having to identify spurious counterexamples and vacuous, i.e. internally contradictory, shapes. Since we assumed a single initial state there was a possibility to do this by, e.g., explicitly creating the concrete graph transformations that go along with the abstract counterexample and analyze the feasibility of spurious counterexamples that way. Motivated by the prospective exponential growth of these concrete traces and our intention to glean abstraction information in the form of formulas from our counterexample analysis, we instead decided to solve this problem by creating suitable SMT encodings. Using prior work on bounded model checking for graph transformation systems[97], we created a trace encoding using which an SMT solver is able to tell whether concrete traces exist that correspond to a given abstract graph transformation trace. Since we eschewed unbounded sorts and quantifiers in this encoding, we were able to keep this encoding in the decidable fragment of first-order logic. We were further able to create a feasibility encoding for shapes, able to detect conflicts between constraints attached to the same shape. Due to the unbounded nature of graphs in general, and the fact that we place no restriction on the kinds of shape constraints that can be used in our technique, we were not able to retain the same decidability result as for the trace encoding.

Put together, these parts of the technique constituted an approach to construct abstract graph transition systems with a built-in detection for spurious counterexamples and infeasible shapes. What remained missing was a way to actually use the refinement potential of our approach in a systematic way that did not place the entire burden of abstraction refinement on the user.

There are generally two ways to structure a abstract state space construction process with abstraction refinement – either the refinement loop contains the state space construction loop, or vice-versa. In the first case, one can generally arrange the abstraction in such a way that the state space construction is guaranteed to terminate. Nontermination then comes from the analysis requiring infinitely many refinement steps. In the second case, refinements are made locally and as part of the state space exploration. As a result, less redundant work needs to be done, but state space exploration itself is no longer guaranteed to terminate.

Recognizing the difficulties of creating a finite canonical abstraction with our abstraction refinement formalism, we opted for the second approach. For this, we adopted the well-known techniques of lazy abstraction and interpolation-guided refinement in

[§]these special circumstances are not given for the application scenarios we have thus far considered order to automate much, if not all of the refinement process and create a state space construction algorithm that would be able to avoid constructing most of the behavior that was unnecessary to rule out the error. The shape transition tree data structure that resulted from this proved very effective and provided a solid foundation upon which to build a lazy, i.e. local, refinement process, allowing us to only closely scrutinize that part of the state space that actually contains the error. Transferring the application of Craig interpolation to the creation of new shape constraints proved far more challenging. We were ultimately able to modify our trace encoding in such a way that a certain class of abstraction error could be automatically excluded by an interpolant extracted from the encoding. Complemented with a fallback refinement strategy that uses materializations as the source for constraints we created an automated refinement algorithm using these strategies that would ultimately fall back to a human designer in cases where the required constraints were too complex. Still, we were able to improve on the base case of the designer providing the constraints unassisted by creating auxiliary checks for the generated constraints and offering interpolants and embedding information as starting points for the manual construction of the refinement.

This approach was then implemented in a prototypical implementation in Java, using existing software like TVLA[108], SMTInterpol[46], and Z3[114] to take advantage of mature software offerings where possible. Only the automatic portion of the approach was ultimately implemented, causing the resulting program to fail with an error message whenever human input would have been required. This implementation was then tested on a few common examples of infinite-state graph transformation systems. This showed that there are non-trivial examples for which the automatic refinement process, or even the pure state space construction method were sufficient. However, it also showed that in many cases, the approach does not terminate or falls back to user input.

Verifying infinite-state graph transformation systems, even just using a single error pattern as the safety condition, is a very hard problem. Current work on this problem goes in many different directions, each of which applies different trade-offs to achieve results, usually by limiting the kind of rules or graphs that are supported. However, as the discussion of related work in Sec. 8.1 showed, there are currently a number of more mature techniques that, using different formalisms, achieve more impressive results than our new approach. Still, no single approach has as yet emerged as a clear front-runner, due to the great diversity of trade-offs and special cases that make up the field today.

As Sec. 8.2 shows there are still many improvements that could be made to make our approach more powerful, versatile, and practical. Most of these improvements are achievable with relatively little effort, suggesting that many of the shortcomings of our approach could be ameliorated in the near future.

If implemented correctly, this approach has the potential to be applicable in many different scenarios. In model-driven development of software, it could serve as a verification tool for algorithm models for manipulating unbounded data structures such as lists or trees, visual models of dynamic communication protocols such as ad hoc networking protocols, or behavioral models of structurally reconfigurable systems such as self-optimizing mechatronic systems[4].

A
Proofs

This appendix contains all the proofs that were left out of the chapters of this thesis for the sake of readability.

Proof of Lemma 2 (p38). Since both f and f' are surjective, $|U_S| = |U'_S|$, as well as the bijectivity of f and f' follow. Let $g := f' \circ f$. We distinguish two cases:

(i) If $g = \text{id}$, then $f' = f^{-1}$. Thus, for every $(p, k) \in \mathcal{P}$, and $(u_1, \ldots, u_k)$, we have

$$\begin{aligned} \iota_S(p)(u_1, \ldots, u_k) &\sqsubseteq \iota_S(p)(f(u_1), \ldots, f(u_k)) && \text{and} \\ \iota_S(p)(f(u_1), \ldots, f(u_k)) &\sqsubseteq \iota_S(p)(u_1, \ldots, u_k) && \text{and thus} \\ \iota_S(p)(u_1, \ldots, u_k) &= \iota_S(p)(f(u_1), \ldots, f(u_k)) \end{aligned}$$

i.e. $S \equiv_f S'$.

(ii) If $g \neq \text{id}$, we know that g is a bijective embedding of S into itself and that there is at least one $u \in U_S$ such that $g(u) \neq u$. Now, because g is bijective there is an n such that $g^n = g$, i.e. $g^n(u) = u$. This means that edge values are monotonically increasing along the applications of g, i.e. for any $(p, k) \in \mathcal{P}$ and $0 \leq 1 \leq k$:

$$\iota_S(p)(u_1, \ldots, u_{i-1}, u, u_i + 1, \ldots, u_k) \sqsubseteq \iota_S(p)(u_1, \ldots, u_{i-1}, u, u_i + 1, \ldots, u_k)$$

Clearly, this is only possible if edge values remain constant along all applications of g. Thus, g is a non-identity isomorphism on S. We can thus construct a new embedding of S' into S by setting $f'' = g^{-1} \circ f' = f^{-1} \circ f'^{-1} \circ f' = f^{-1}$. This brings us with $g' := f'' \circ f = \mathrm{id}$ back to the first case.

$\square$

Proof of Lemma 5 (p45). By construction, S^m contains L as a concrete subgraph. Since G is concrete, the match m' is injective. As seen in the proof of Lemma 4, $f|_{U^m \setminus N_L}$ is injective. Thus, $f' := g|_{U_G \setminus m'(N_L)} \cup m'^{-1}$ is a surjective function $f' : N_G \to U^m$. By construction, we get

$$\begin{aligned}
f \circ f' &= f \circ \left(g|_{U_G \setminus m'(N_L)} \cup m'^{-1} \right) \\
&= f \circ g|_{U_G \setminus m'(N_L)} \cup f \circ m'^{-1} && \text{non-intersecting domains} \\
&= f|_{g(U_G \setminus m'(N_L))} \circ g|_{U_G \setminus m'(N_L)} \cup f|_{N_L} \circ m'^{-1} \\
&= f|_{(U_S \setminus m(N_L))} \circ g|_{U_G \setminus m'(N_L)} \cup f|_{N_L} \circ m'^{-1} && m = g|_{m'(N_L)} \circ m' \\
&= id_{U_S} \circ g|_{U_G \setminus m'(N_L)} \cup f|_{N_L} \circ m'^{-1} \\
&= id_{U_S} \circ g|_{U_G \setminus m'(N_L)} \cup f|_{N_L} \circ \left(m^{-1} \circ g|_{m'(N_L)} \right) && m = g|_{m'(N_L)} \circ m' \\
&= id_{U_S} \circ g|_{U_G \setminus m'(N_L)} \cup \left(f \circ m^{-1} \right) \circ g|_{m'(N_L)} && m = g|_{m'(N_L)} \circ m' \\
&= id_{U_S} \circ g|_{U_G \setminus m'(N_L)} \cup g|_{m'(N_L)} \\
&= g
\end{aligned}$$

We will now show that f' is an embedding.

Let $(u, 1) \in \mathcal{P}$ and $n \in U_G \setminus m(N_L)$. Then we get $\iota_G(u)(n) \sqsubseteq \iota_S(u)(g(n)) = \iota_{S^m}(u)(f'(n))$, since g is an embedding and $m = g|_{m'(N_L)} \circ m'$. Now, let $n \in m(N_L)$ and $(u, n) \in E_L^1$. Then we have $\mathbf{1} = \iota_G(u)(n) = \iota_{S^m}(u)\left(m^{-1}(n)\right) = \iota_{S^m}(u)(f'(n))$, since m' is an injective match. On the other hand, if $n \in m(N_L)$ and $(u, n) \notin E_L^1$, then we have

$$\begin{aligned}
\iota_{S^m}(u)\left(f'(n)\right) &= \iota_S(u)\left(f \circ f'(n)\right) && \text{by Def. 46} \\
&= \iota_S(u)(g(n)) && g = f \circ f' \\
&\sqsupseteq \iota_G(u)(n)
\end{aligned}$$

Thus, we get $\iota_G\left(u\right)\left(n\right) \sqsubseteq \iota_{S^m}\left(u\right)\left(f'\left(n\right)\right)$ in all cases. For $\left(b, 2\right) \in \mathcal{P}$ and $n_1, n_2 \in U_G$, this follows analogously.

Finally, since g is an embedding, $g|_{U_G \setminus m'(N_L)}$ has the third embedding property. Since m' is an injective match, m'^{-1} has it too. Thus, f', as a union of two functions exhibiting the third embedding property, has it as well.

Therefore, f' is an embedding. □

Proof of Lemma 9 (p79). Any model of Sc has to satisfy `distinctNodes`, `unaryG`, and `binaryG`. Satisfaction of `distinctNodes` implies that the universe U of the model contains k distinct node constants, and that the nullary predicates $v_1, \ldots, v_k$ are interpreted such that there is a one-to-one mapping between them and those node constants. Since the $v_1, \ldots, v_k$ each represent a node in the graph, this establishes a one-to-one mapping between the graph nodes and the node constants in the model. From CoT. 4, we know that the model uses the same predicate set $\mathcal{P}$ that serves as the label set L of G. Now, we assume that the logical structure $(U, \mathcal{P}, \iota)$ induced by the model interprets one of the unary predicates s_i on one of the node constants v_j such that

$$\iota\left(s_i\right)\left(v_j\right) \neq \left(\left(s_i, 1\right), v_j\right) \in E^1,$$

i.e. it contradicts the graph. From the definition of `unaryG` we know that it consists of a conjunction containing, among other elements the term `B(`s_i`, `v_j`)`, where

$$\texttt{B(}s_i\texttt{, }v_j\texttt{)} = \begin{cases} s_i\left(v_j\right) & \text{if } \left(\left(s_i, 1\right), v_j\right) \in E^1 \\ \neg s_i\left(v_j\right) & \text{otherwise} \end{cases}$$

This directly implies

$$\iota\left(s_i\right)\left(v_j\right) = \left(\left(s_i, 1\right), v_j\right) \in E^1,$$

in contradiction to the assumption. The same argument holds for binary predicates. Thus, $(U, \mathcal{P}, \iota)$ is isomorphic to $ls\left(G\right)$, since there is a one-to-one correspondence between U and N, as well as between L and $\mathcal{P}$, and in addition no interpretation of a predicate in $(U, \mathcal{P}, \iota)$ can differ from the valuation of that predicate in $ls\left(G\right)$. □

Proof of Lemma 10 (p84). We will show that in a model satisfying `summarization`, the interpreted function `_F` can violate neither the surjectivity property, nor the node abstraction property.

We begin with surjectivity. Let $u_j \in U_S$ such that $\neg\exists v_i \in U_G : \_F(v_i) = u_j$. Then either $u_j \in U_S^c$ or $u_j \in U_S^s$. If $u_j \in U_S^c$, then `summarization` contains the conjunction

$$\bigvee_{v \in U_G} \left(u_j = \_F(v) \wedge \bigwedge_{v' \in U_G \setminus \{v\}} \neg \left(u_j = \_F(v') \right) \right)$$
$$\equiv \bigvee_{v \in U_G} \left(\mathbf{0} \wedge \bigwedge_{v' \in U_G \setminus \{v\}} \neg \mathbf{0} \right) \equiv \mathbf{0} \qquad (*)$$

where $(*)$ holds because $\neg\exists v_i \in U_G : \_F(v_i) = u_j$. Similarly, if $u_j \in U_S^s$, we have

$$\bigvee_{v \in U_G} u_j = \_F(v) \equiv \bigvee_{v \in U_G} \mathbf{0} \equiv \mathbf{0}$$

by the same argument.

We now turn to the node abstraction property. Let $u_j \in U_S^c$, and $X := \_F^{-1}(u_j) = \{v_1, \ldots, v_l\}$, i.e. $|X| > 1$ (due to pairwise inequality of node constants). Then, by definition, `summarization` contains the term

$$\bigvee_{v \in U_G} \left(u_j = \_F(v) \wedge \bigwedge_{v' \in U_G \setminus \{v\}} \neg \left(u_j = \_F(v') \right) \right)$$
$$\equiv \bigvee_{v \in X} \left(\mathbf{1} \wedge \bigwedge_{v' \in X \setminus \{v\}} \neg \mathbf{1} \wedge \bigwedge_{v' \in U_G \setminus X} \neg \mathbf{0} \right) \wedge \bigvee_{v \in U_G \setminus X} \left(\mathbf{0} \wedge \bigwedge_{v' \in U_G \setminus \{v\}} \neg \mathbf{0} \right) \equiv \mathbf{0}$$

Thus, any violation of either the summarization property or the node abstraction property is prohibited by `summarization`. □

Proof of Lemma 11 (p90). From Lemma 9 and Lemma 10, we already know that the model for Sc will contain an instance of $ls(G)$, and an instance of $ls(S')$, where S' is created from S by removing all of its ½-edges. Furthermore we know that the model of `_F` will be a function $\_F : U_G \rightarrow U_S$ that is surjective and satisfies the summarization condition. What remains to show is that $\_F$ satisfies the edge abstraction condition, given by

$$\forall (s,k) \in \mathcal{P} \forall v_1, \ldots, v_k \in U_G : \iota_G(s)(v_1, \ldots, v_k) \sqsubseteq \iota_S(s)(\_F(v_1), \ldots, \_F(v_k)).$$

We now assume that $\_F$ is a function that violates that condition. For that purpose, let $(s, 2) \in \mathcal{P}$ and $v_1, v_2 \in U_G$ such that $\iota_G(s)(v_1, v_2) \neq \iota_S(s)(\_F(v_1), \_F(v_1))$ and neither value is ½. Since $\iota_S(s)(\_F(v_1), \_F(v_1)) \neq$ ½, we know that $(v_1, s, v_2) \in E_S^2$ is not a ½-edge. Without loss of generality, we assume that $\iota_G(s)(v_1, v_2) = \mathbf{1}$ and $\iota_S(s)(\_F(v_1), \_F(v_1)) = \mathbf{0}$. By CoT. 17 we know that the conjunction that constitutes `binaryAbstraction` contains the term

$$(s(v_1, v_2) = s(\_F(v_1), \_F(v_2))) \vee \bigvee_{o=1}^{k_i} ((\_F(v_1) = u_o) \wedge (\_F(v_2) = w_o))$$

Since we know that $(v_1, s, v_2) \in E_S^2$ is not a ½-edge, the disjunction on the right reduces to false, leaving us with

$$(s(v_1, v_2) = s(\_F(v_1), \_F(v_2))) \vee \mathbf{0}$$

which implies $\iota_G(s)(v_1, v_2) \neq \iota_S(s)(\_F(v_1), \_F(v_1))$, in contradiction to the assumption. The same argument follows for unary predicates $(s, 1)$ and the `unaryAbstraction` formula. Thus, every non-summary edge in the shape is mirrored exactly by all its pre-images under $\_F$, and the edge abstraction condition is thus satisfied.

Now, let f be a valid embedding of G into S, i.e. $G \sqsubseteq_f S$. Let further $\_F$ be interpreted such as to mimic f, i.e.

$$\forall v \in U_G : (\iota(\_F)(v) = f(v)).$$

Then $\_F$ has the right range to satisfy CoT 11, and by Lemma 10 it also satisfies CoT. 12. Furthermore, since f satisfies the edge abstraction property, $\_F$ does so as well:

$$\begin{aligned}
&\bigwedge_{i=1}^{l} \bigwedge_{j=1}^{k} \left[(s_i(v_j) = s_i(\_F(v_j))) \vee \bigvee_{o=1}^{k_i} (\_F(v_j) = u_o^{s_i}) \right] \\
\equiv & \bigwedge_{\substack{(s_i,v) \in E_G^1 \\ \iota_S(s_i)(\_F(v)) = 1/2}} \left[(s_i(v) = s_i(\_F(v))) \vee \bigvee_{o=1}^{k_i} (\_F(v) = u_o^{s_i}) \right] \wedge \\
& \bigwedge_{\substack{(s_i,v) \in E_G^1 \\ \iota_S(s_i)(\_F(v)) \neq 1/2}} \left[(s_i(v) = s_i(\_F(v))) \vee \bigvee_{o=1}^{k_i} (\_F(v) = u_o^{s_i}) \right] \\
\equiv & \bigwedge_{\substack{(s_i,v) \in E_G^1 \\ \iota_S(s_i)(\_F(v)) = 1/2}} \left[(s_i(v) = s_i(\_F(v))) \vee \mathbf{1} \right] \wedge
\end{aligned}$$

$$\bigwedge_{\substack{(s_i,v)\in E_G^1\\ \iota_S(s_i)(\_F(v))\neq 1/2}} [(s_i(v) = s_i(\_F(v))) \vee \mathbf{0}]$$

$$\equiv \mathbf{1} \wedge \bigwedge_{\substack{(s_i,v)\in E_G^1\\ \iota_S(s_i)(\_F(v))\neq 1/2}} [(s_i(v) = s_i(\_F(v)))]$$

Therefore, this encoding has as its models exactly the valid embeddings of G into S.

□

Proof of Lemma 18 (p114). Let M be a model for Sc, and let G_{i+1}^M be the interpretation of the $i + 1$-indexed versions of $\mathcal{P}$ that M contains. We want to show that $G_{i+1} \equiv G_{i+1}^M$. Since the universes are identical and unchanging over the course of the trace, this comes down to the placement of the edges.

Let now $p \in \mathcal{P}^0$, i.e. p is a predicate that is untouched by P_i. Then by CoT. 27, there is only one term in `rule_application` that contains references to it:

$$\bigwedge_{j_1=1}^{v} \bigwedge_{j_2=1}^{v} p_{i+1}(v_{j_1}, v_{j_2}) = p_i(v_{j_1}, v_{j_2}) \qquad \text{if } p \text{ is binary,}$$

$$\bigwedge_{j=1}^{v} p_{i+1}(v_j) = p_i(v_j) \qquad \text{if } p \text{ is unary.}$$

Since M is a model for Sc, it must satisfy this term and thus

$$\iota_{G_i}(p)(v,w) = \iota_{G_{i+1}^M}(p)(v,w) \qquad \text{if } p \text{ is binary, and}$$

$$\iota_{G_i}(p)(v) = \iota_{G_{i+1}^M}(p)(v) \qquad \text{if } p \text{ is unary}$$

must hold for all nodes v, w. Thus, G_{i+1}^M and G_{i+1} are identical on all predicates in $\mathcal{P}^0$.

Let now $(p, 2) \in \mathcal{P} \setminus \mathcal{P}^0$, i.e. p is a binary predicate for which at least one value changes in an application of P_i. Then by CoT. 27, Sc contains the following conjunction (with $j := i + 1$) that references p:

$$\bigwedge_{(n_1,p,n_2)\in E_+^2} p_j(n_1, n_2) \wedge \bigwedge_{(n_1,p,n_2)\in E_-^2} \neg p_j(n_1, n_2) \wedge$$

$$\bigwedge_{j_1=1}^{v}\bigwedge_{j_2=1}^{v}\left[\bigvee_{(n_1,p,n_2)\in E_+^2\cup E_-^2}(n_1=v_{j_1}\wedge n_2=v_{j_2})\vee p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]$$

$$\equiv \bigwedge_{(n_1,p,n_2)\in E_+^2} p_j\left(n_1,n_2\right)\wedge \bigwedge_{(n_1,p,n_2)\in E_-^2} \neg p_j\left(n_1,n_2\right)\wedge$$

$$\bigwedge_{\substack{j_1,j_2\in U\\(j_1,p,j_2)\in E_+^2\cup E_-^2}}\left[\bigvee_{(n_1,p,n_2)\in E_+^2\cup E_-^2}(n_1=v_{j_1}\wedge n_2=v_{j_2})\vee p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]\wedge$$

$$\bigwedge_{\substack{j_1,j_2\in U\\(j_1,p,j_2)\notin E_+^2\cup E_-^2}}\left[\bigvee_{(n_1,p,n_2)\in E_+^2\cup E_-^2}(n_1=v_{j_1}\wedge n_2=v_{j_2})\vee p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]$$

$$\equiv \bigwedge_{(n_1,p,n_2)\in E_+^2} p_j\left(n_1,n_2\right)\wedge \bigwedge_{(n_1,p,n_2)\in E_-^2} \neg p_j\left(n_1,n_2\right)\wedge$$

$$\bigwedge_{\substack{j_1,j_2\in U\\(j_1,p,j_2)\in E_+^2\cup E_-^2}}\left[\mathbf{1}\vee p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]\wedge$$

$$\bigwedge_{\substack{j_1,j_2\in U\\(j_1,p,j_2)\notin E_+^2\cup E_-^2}}\left[\mathbf{0}\vee p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]$$

$$\equiv \bigwedge_{(n_1,p,n_2)\in E_+^2} p_j\left(n_1,n_2\right)\wedge \bigwedge_{(n_1,p,n_2)\in E_-^2} \neg p_j\left(n_1,n_2\right)\wedge \bigwedge_{\substack{j_1,j_2\in U\\(j_1,p,j_2)\notin E_+^2\cup E_-^2}}\left[p_j\left(v_{j_1},v_{j_2}\right)=p_i\left(v_{j_1},v_{j_2}\right)\right]$$

Thus, since M is a model for Sc, for any node pair v, w, G_{i+1} must contain a p-edge if P_i adds an edge there ($(n_1, p, n_2) \in E_+^2$), must not contain a p-edge if P_i removes an edge there ($(n_1, p, n_2) \in E_-^2$), and must have the same edge value as G_i if the rule did not make any changes[*]. The same holds analogously for unary predicates. Thus, G_{i+1}^M and G_{i+1} are identical on all predicates in $\mathcal{P} \setminus \mathcal{P}^0$ and therefore identical on all predicates, and thus isomorphic. $\square$

Proof of Lemma 9 (p79). Any model of Sc has to satisfy `distinctNodes`, `unaryG`, and `binaryG`. Satisfaction of `distinctNodes` implies that the universe U of the

[*]recall that matches are expressed via equalities, so the use of rule nodes in graph contexts and the use of graph nodes in rule contexts are valid here

model contains k distinct node constants, and that the nullary predicates $v_1, \ldots, v_k$ are interpreted such that there is a one-to-one mapping between them and those node constants. Since the $v_1, \ldots, v_k$ each represent a node in the graph, this establishes a one-to-one mapping between the graph nodes and the node constants in the model. From CoT. 4, we know that the model uses the same predicate set $\mathcal{P}$ that serves as the label set L of G. Now, we assume that the logical structure $(U, \mathcal{P}, \iota)$ induced by the model interprets one of the unary predicates s_i on one of the node constants v_j such that

$$\iota\left(s_i\right)\left(v_j\right) \neq \left(\left(s_i, 1\right), v_j\right) \in E^1,$$

i.e. it contradicts the graph. From the definition of `unaryG` we know that it consists of a conjunction containing, among other elements the term `B(`s_i`, `v_j`)`, where

$$\texttt{B(}s_i\texttt{, } v_j\texttt{)} = \begin{cases} s_i\left(v_j\right) & \text{if } \left(\left(s_i, 1\right), v_j\right) \in E^1 \\ \neg s_i\left(v_j\right) & \text{otherwise} \end{cases}$$

This directly implies

$$\iota\left(s_i\right)\left(v_j\right) = \left(\left(s_i, 1\right), v_j\right) \in E^1,$$

in contradiction to the assumption. The same argument holds for binary predicates. Thus, $(U, \mathcal{P}, \iota)$ is isomorphic to $ls\,(G)$, since there is a one-to-one correspondence between U and N, as well as between L and $\mathcal{P}$, and in addition no interpretation of a predicate in $(U, \mathcal{P}, \iota)$ can differ from the valuation of that predicate in $ls\,(G)$. □

Proof of Lemma 25 (p149). We will prove this by induction over the possible concrete paths starting in $\Gamma\,(n)$. Firstly, we recognize that, since $(n, n') \in C$ and $\mathcal{T}$ is valid, we know that $\underline{S_n} \sqsubseteq \underline{S_{n'}}$ and thus $\Gamma\,(n) \subseteq \mathcal{G}\left(\underline{S_n}\right) \subseteq \mathcal{G}\left(\underline{S_{n'}}\right)$. Therefore, the central shape of n' covers all graphs of the local graph set of n. Now, let $k \in \mathbb{N}$, $G_0 \xrightarrow{P_1, m_1} \cdots \xrightarrow{P_k, m_k} G_k$ be a path of length k with $G_0 \in \Gamma\,(n)$, and $k - 1 > i \in \mathbb{N}$. We assume that for all $0 \leq j < i$ the lemma holds, i.e. we have a function π such that (5.3) holds. Therefore, there is a materialization $\underline{S^m}$ of $\pi\,(G_i)$ such that $G_i \sqsubseteq_{f_i} \underline{S^m}$ and a shape $\underline{S^t}$ such that $G_{i+1} \sqsubseteq_{f'_{i+1}} \underline{S^t}$. Let $(n_{i+1}, x) \in E^1$, and n_{i+1} be the STT node for which $\underline{S^t}$ is the incoming shape. There are now two cases.

In the first case, $\neg\exists y : (x, y) \in C$, i.e. is not a covered node. We then set $\pi\,(G_{i+1}) := x$. Since $G_{i+1} \sqsubseteq \underline{S^t}$, from condition 5 we also know that $G_{i+1} \sqsubseteq \underline{S_{n_{i+1}}}$. Since P_{i+2} matches G_{i+1} at m_{i+2}, and $G_{i+1} \sqsubseteq_f \underline{S_{n_{i+1}}}$, we know by the embedding theorem (Theorem 1) that P_{i+2} *potentially* matches $\underline{S_{n_{i+1}}}$ at a match $m'_{i+2} = f \circ m_{i+2}$. n_{i+1}

therefore has a materialization point $\left(\underline{S^{m'_{i+2}}_{n_{i+1}}}, f^{m'_{i+2}}_{n_{i+1}}\right)$ such that $G_{i+1} \sqsubseteq \underline{S^{m'_{i+2}}_{n_{i+1}}}$ and an outgoing transition edge labeled with $\left(P_{i+2}, \left(\underline{S^{m'_{i+2}}_{n_{i+1}}}, f^{m'_{i+2}}_{n_{i+1}}\right)\right)$.

In the other case, x is a covered node and is thus embedded into another node x'. Then we just set $\pi\left(G_{i+1}\right) := x'$. This is valid by the same argument as above, since the existence of the covering edge (x, x') implies the existence of a corresponding embedding. With that, the induction is complete. $\square$

Proof of Theorem 5 (p156). Given that any algorithm will either terminate or not terminate, we will focus on claims ii and iii. We will prove the following: After every complete execution of the main loop (lines 5-31), $\mathcal{T}$ is an STT satisfying the validity conditions 1, 2, 4, and 5 of Def. 83, and a modified version of condition 3, specifically[†]

$$\begin{aligned} \forall n \in (N \setminus N_C) \setminus E :& \forall \left(\underline{S^m}, f^m\right) \in \mathcal{M}\left(\underline{S_n}\right) : \\ & \left[\mathcal{G}\left(\underline{S^m}\right) = \emptyset \vee \exists e \in E^2 : \left(\left(\underline{S^m}, f^m\right) = \left(\underline{S^m_e}, f^m_e\right)\right)\right] \end{aligned} \tag{A.1}$$

In other words, the loop maintains the invariant that $\mathcal{T}$ is a valid STT, with the exception of condition 3 for the nodes in E. It is fairly obvious that, if the above holds, then if the algorithm terminates, it is either because E became empty, i.e. the tree is now valid, or the loop was left through line 24, in which case we have produced a concrete counterexample.

We will begin by showing that the invariant holds when the loop is first entered. Lines 2-3 construct the starting STT $\mathcal{T}$, while line 4 sets E (the set of nodes to be processed) to be the unary edge attaching the STT node for I to the root of the tree, as well as K (the set of "completed" nodes) to the empty set. By this construction, $\mathcal{T}$ already satisfies condition 1, and since $\mathcal{T}$ contains no transition or covering edges whatsoever, conditions 2 and 4 are satisfied as well. The use of the abstraction scheme $\mathcal{A}$ to construct the central shape of the root node ensures that condition 5 also holds (in fact, the stronger condition $\mathcal{G}(I) \subseteq \mathcal{G}\left(\underline{S_0}\right)$ holds). Due to the root node being part of E, the modified condition (A.1) holds, too.

Now, we assume that the invariant holds at the beginning of the loop body, i.e. $\mathcal{T}$ satisfies conditions 1, 2, 4, and 5, as well as (A.1). Each loop iteration performs operations on a single STT node n taken from a unary edge removed from E at the beginning of the loop body. We now show for each condition in the loop invariant that it is preserved with each loop iteration, provided that the loop body is executed completely and not left via the return statement on line 24.

[†]with slight abuse of notation, since E contains unary edges, not nodes

Condition 1: The algorithms EXPAND, EXPLORE, CHECKFEASIBLE, and COVER, as well as the main algorithm itself do not touch the root node n beyond its initial expansion which cannot invalidate the condition due to Def. 86. Since $\Gamma(n) = \{I\}$ for the root node, and from Def. 90 we know that in any case where the tree is modified the root is either not modified (Case a), or refined in such a way that $\mathcal{G}\left(\underline{S_n^{in}}\right) \subseteq \mathcal{G}\left(\underline{S_n}\right)$ holds (Case b with $k \geq 0$). Thus, condition 1 is always preserved by the main loop body.

Condition 2: This condition states that every transition edge must represent a valid shape transition. Only EXPLORE and HANDLEERROR affect transition edges without removing them entirely (in which case condition 2 would be trivially satisfied). By definition (see Def. 88), EXPLORE only produces transition edges for which the condition holds. Furthermore, since HANDLEERROR never refines the unabstracted result shape, only materialization points and central shapes, and replaces invalid transitions by recomputing them, this condition is left untouched by it as well.

Condition 4: This condition concerns itself with covering edges, stating that each covering edge must be labeled with a valid embedding between the central shapes of its source and target nodes. Only the main algorithm and HANDLEERROR have an effect on this condition. In lines $7 - 13$, covering edges are added to C for all pairs (n, m) of nodes where COVER returns an embedding. Since COVER only returns an embedding when one exists between the central shapes of the parameter nodes (see Def. 89), this preserves the condition. Existing covering edges might be invalidated by a refinement in the central shape of their target node. This only happens if case b occurs in the execution of HANDLEERROR. By Def. 90, we know that all covering edges that terminate in a refined node $n_i,\ i < k$ are removed from C, and the thereby uncovered nodes added to E. Thus, condition 4 is preserved.

Condition 5: This condition describes the relationship between the entry point $\underline{S^{in}}$ and the central shape $\underline{S}$ of each STT node n in the tree, i.e. $\underline{S^{in}} \sqsubseteq \underline{S}$. Only EXPLORE and HANDLEERROR can have an effect on this condition. The definition of EXPLORE states that for any new (tentative) node n created by it, $\underline{S_n} = \mathcal{A}\left(\underline{S_n^{in}}\right)$ and thus

$$\underline{S_n^{in}} \sqsubseteq \underline{S_n}$$

holds. The HANDLEERROR sub-algorithm refines (in case b) the central shapes of nodes $n_i,\ i < k$, which could potentially invalidate condition 5. However, Def. 90(b) states that this is not the case. Thus, condition 5 is preserved.

Condition (A.1): This modified version of Condition 3 states that all feasible materialization points either have transition edges attached to them, or their STT node is covered, or their STT node is part of E. Again, we assume that this condition is satisfied at the start of the loop body. In line 6, one node n (or rather one unary edge assigning an STT node to a tree node) is removed from E, breaking Condition (A.1). We will now go through the algorithm and see that this is always rectified at some point in the loop body.

In lines 7 through 13, a loop goes through all pairs (n, m) where, syntactically, a covering edge could be added, and checks via `COVER` if a corresponding embedding exists. If one is found, the corresponding covering edge is added, thereby restoring condition (A.1). The loop is left and the main loop continued by line 14. If no embedding is found, the loop terminates, line 14 does nothing, and the node is expanded on line 15.

Lines $16 - 20$ now determine which of the materializations are feasible, i.e. which ones must be explored for Condition A.1 to hold again.

The loop on lines $21 - 30$ now handles feasible error materializations. It is only left once the current node n no longer has any. In that case, `EXPLORE` is called and attaches transition edges to all feasible materialization points of n. n is added to K, and all tentative nodes created by `EXPLORE` added to E on line 31, reestablishing Condition (A.1).

Now let's assume that on line 21, n, which is the only non-explored node outside of E, has at least one feasible error materialization $\left(\underline{S^e_n}, f^e\right)$. On line 22, `HANDLEERROR` is called. By Definition 90, `HANDLEERROR` returns either a path and a node, in which case the analysis should be terminated, or an empty path and a refined node.

Let's assume that $p \neq \epsilon$. Then the loop and the entire algorithm will be left via the return statement on line 24, satisfying option ii for the outcome of the main algorithm.

Thus, we now assume that $p = \epsilon$ and $(n_1, \ldots, n_l)$ now form the singular branch of $\mathcal{T}$ that was refined by `HANDLEERROR` in accordance with Def. 90. Then $\{n_1, \ldots, n_l\}$ will be removed from K since they are no longer explored on line 26. Then, all nodes that were uncovered by `HANDLEERROR` because they were covered either by a node in the removed subtrees or by $n_1, \ldots, n_l$, are added to E on line 28. Nodes $n_2, \ldots, n_l$ (if they exist) are also added to E, thus reestablishing Condition (A.1) for all those nodes. The particular error materialization with which the loop started has now been removed from n, either by refining n, or by removing it from the tree entirely and replacing it with a node that does not have that error materialization. Thus n is now again the only unexplored node in $\mathcal{T}$ outside of E. If it has another error materialization, the loop will continue. If not, the loop will be left, the node n explored and added to K, as stated above. The loop invariant is thus reestablished. It is worth noting that the loop from line 21 to line 30 will always terminate, since it is guaranteed to either remove an error materialization from the given node (and there is always a finite number of them) or to move back along the root path of n, which is also finite.

Thus, since the loop invariant is preserved by the loop body, and is equal to the validity condition for $\mathcal{T}$ in case $E = \emptyset$, the STT $\mathcal{T}$ is guaranteed to be valid if the main loop is left, since that guarantees $E = \emptyset$. Therefore, if the algorithm terminates, it always either returns a path or a valid STT, as claimed in the Theorem. $\square$

Proof of Theorem 6 (p178). We will first prove $S^m \not\models \text{concr}_f\left(\{(\alpha, m)\}\right)$, i.e. $\mathcal{G}\left(\underline{S^m}\right) \subseteq \mathcal{G}\left(\underline{S}\right) \setminus \mathcal{G}\left(S, \Lambda'\right)$. For this, we need to show that there is an m' such that $m = f^m \circ m' \wedge \underline{S^m} \not\models (\alpha, m')$. We construct m' by assuming an arbitrary total order $\leq$ on the nodes of N, and setting

$$m' := \text{id}_N \cup \left\{\left(n', x\right) \mid x \in f^{m-1}\left(\nu\left(x\right)\right) \setminus N\right\},$$

i.e. m' assigns each node from the left hand side of the rule to itself, and the additional variables for the summary nodes in S to the additional that were created for the preserved summary nodes (note that the second part of that definition is well-defined since for each preserved summary node in the match, only one summary node is created in the materialization, i.e. we have $\left|\left\{\left(n', x\right) \mid x \in f^{m-1}\left(\nu\left(x\right)\right) \setminus N\right\}\right| = 1$ for all n'). We now obtain, by Def. 94:

$$\begin{aligned}
f^m \circ m' &= f^m \circ \text{id}_N \cup f^m \circ \left\{\left(n', x\right) \mid x \in f^{m-1}\left(\nu\left(n'\right)\right) \setminus N\right\} \\
&= m \circ \text{id}_N \cup \left\{\left(n', f^m\left(x\right)\right) \mid x \in f^{m-1}\left(\nu\left(n'\right)\right) \setminus N\right\} \\
&= m \cup \left\{\left(n', \nu\left(n'\right)\right)\right\} \\
&= \nu
\end{aligned}$$

Thus, $(\alpha, m') \in \text{concr}_{f^m}\left(\{(\alpha, \nu)\}\right)$. Now, the constraint (α, m') can only be violated on $\underline{S^m}$ if α_{set}, α_{un} and α_{bin} are satisfied. α_{set} is satisfied by construction of m'.

Similarly, since α_{un} and α_{bin} mirror exactly the unary and binary edges found in the materialized shape, we have that on $\underline{S^m}$ α_{un} and α_{bin} are satisfied.

$$\begin{aligned}
[\![\alpha_{m'}]\!]^{\nu}_{S^m} &= \bigwedge_{(s,1)\in\mathcal{P}} \left[\bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{1}}} \iota_{S^m}(s)\left(m'(n)\right) \wedge \bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{0}}} \neg\iota_{S^m}(s)\left(m'(n)\right) \right] \\
&= \bigwedge_{(s,1)\in\mathcal{P}} \left[\bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{1}}} \iota_{S^m}(s)\left(\mathrm{id}(n)\right) \wedge \bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{0}}} \neg\iota_{S^m}(s)\left(\mathrm{id}(n)\right) \right] \\
&= \bigwedge_{(s,1)\in\mathcal{P}} \left[\bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{1}}} \iota_{S^m}(s)(n) \wedge \bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{0}}} \neg\iota_{S^m}(s)(n) \right] \\
&= \bigwedge_{(s,1)\in\mathcal{P}} \left[\bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{1}}} \mathbf{1} \wedge \bigwedge_{\substack{n\in N_L\\ \iota_{S^m}(n)=\mathbf{0}}} \neg\mathbf{0} \right] \\
&= \mathbf{1} \wedge \mathbf{1} = \mathbf{1}
\end{aligned}$$

α_{bin} follows analogously. Thus, we get

$$\begin{aligned}
&[\![\alpha]\!]^{\nu}_{S^m} = \mathbf{0} \\
\Rightarrow & S^m \not\models (\alpha, \nu) \\
\Rightarrow & S^m \not\models \mathrm{concr}_{f^m}\left(\Lambda \cup \{(\alpha, m)\}\right) \\
\Rightarrow & \mathcal{G}\left(\underline{S^m}\right) \subseteq \mathcal{G}\left(\underline{S}\right) \setminus \mathcal{G}\left(S, \Lambda'\right)
\end{aligned}$$

Now, in order for α_{set} to be satisfiable on a materialization S'^m other that S^m, but in the same materialization lattice, S'^m must preserve all the nodes that S^m preserves, otherwise the additional variable cannot be assigned to a node that is outside of N_L. Thus, every materialization other than S^m that is excluded by (α, m) is an upper bound for S^m in its materialization lattice.

Finally, if a graph G is embedded into none of the materializations, then it does not match P at a match m' compatible with m. Thus, it cannot satisfy α_{un} and α_{bin} at an assignment compatible with m, enforcing $G \models \mathrm{concr}_g\left(\{(\alpha, m)\}\right)$ for any such graph G and embedding g into S.

Therefore, the graphs excluded by (α, m) are exactly the graphs that are embeddable into S^m and its upper bounds in its materialization lattice. □

B
Code Listings

This appendix contains all code listings that, for the sake of a clear and concise presentation, could not be included in the thesis at the point where they are discussed. Each listing presented here references its section of origin.

Listing B.1: SMTLib Operators for First-Order Logic

```
not                    ; logical negation
and                    ; logical and
or                     ; logical or
xor                    ; logical exclusive or
=>                     ; logical implication
=                      ; equals
ite                    ; if-then-else
forall                 ; universal quantifier
exists                 ; existential quantifier
```

Listing B.2: Embedded Graph Encoding (see Sec. 4.4)

```
(set-option :print-success false)
;(set-option :produce-unsat-cores true)
(set-logic QF_UF)
; Sort for graph nodes
(declare-sort Node 0)
(declare-sort ANode 0)
(declare-fun _F (Node) ANode)
; four node individuals in the graph (node set / universe)
(declare-fun v_1 () Node)
(declare-fun v_2 () Node)
(declare-fun v_3 () Node)
(declare-fun v_4 () Node)
; two node individuals in the shape
(declare-fun u_1 () ANode)
(declare-fun u_2 () ANode)
; assert distinctiveness of the individuals
(define-fun distinctNodes () Bool
  (and (not (= v_1 v_2)) (not (= v_1 v_3)) (not (= v_1 v_4))
                         (not (= v_2 v_3)) (not (= v_2 v_4))
                                           (not (= v_3 v_4))
                                           (not (= u_1 u_2)))
)
; labels used in the graph become predicates
(declare-fun list  (Node) Bool)
(declare-fun cell  (Node) Bool)
(declare-fun tail  (Node Node) Bool)
(declare-fun head  (Node Node) Bool)
(declare-fun next  (Node Node) Bool)
(declare-fun alist (ANode) Bool)
(declare-fun acell (ANode) Bool)
(declare-fun atail (ANode ANode) Bool)
(declare-fun ahead (ANode ANode) Bool)
(declare-fun anext (ANode ANode) Bool)
; defines the unary edges of the graph
(define-fun unaryG () Bool (and
       (list v_1)  (not (cell v_1))
  (not (list v_2))      (cell v_2)
  (not (list v_3))      (cell v_3)
  (not (list v_4))      (cell v_4)
))
; defines the binary edges of the graph
(define-fun binaryG () Bool (and
  (not (tail v_1 v_1))      (tail v_1 v_2)  (not (tail v_1 v_3)) (not (tail v_1 v_4))
  (not (tail v_2 v_1)) (not (tail v_2 v_2)) (not (tail v_2 v_3)) (not (tail v_2 v_4))
  (not (tail v_3 v_1)) (not (tail v_3 v_2)) (not (tail v_3 v_3)) (not (tail v_3 v_4))
  (not (tail v_4 v_1)) (not (tail v_4 v_2)) (not (tail v_4 v_3)) (not (tail v_4 v_4))

  (not (head v_1 v_1)) (not (head v_1 v_2)) (not (head v_1 v_3))      (head v_1 v_4)
  (not (head v_2 v_1)) (not (head v_2 v_2)) (not (head v_2 v_3)) (not (head v_2 v_4))
  (not (head v_3 v_1)) (not (head v_3 v_2)) (not (head v_3 v_3)) (not (head v_3 v_4))
  (not (head v_4 v_1)) (not (head v_4 v_2)) (not (head v_4 v_3)) (not (head v_4 v_4))

  (not (next v_1 v_1)) (not (next v_1 v_2)) (not (next v_1 v_3)) (not (next v_1 v_4))
  (not (next v_2 v_1)) (not (next v_2 v_2)) (not (next v_2 v_3)) (not (next v_2 v_4))
  (not (next v_3 v_1))      (next v_3 v_2)  (not (next v_3 v_3)) (not (next v_3 v_4))
  (not (next v_4 v_1)) (not (next v_4 v_2))      (next v_4 v_3)  (not (next v_4 v_4))
))
; defines the range of the embedding function
(define-fun function () Bool (and
  (or (= (_F v_1) u_1) (= (_F v_1) u_2))
  (or (= (_F v_2) u_1) (= (_F v_2) u_2))
  (or (= (_F v_3) u_1) (= (_F v_3) u_2))
  (or (= (_F v_4) u_1) (= (_F v_4) u_2))
))
; defines the summarization property
(define-fun summarization () Bool (and
  (or (and (= (_F v_1) u_1)       (not (= (_F v_2) u_1)) (not (= (_F v_3) u_1)) (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1))      (= (_F v_2) u_1)  (not (= (_F v_3) u_1)) (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1)) (not (= (_F v_2) u_1))      (= (_F v_3) u_1)  (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1)) (not (= (_F v_2) u_1)) (not (= (_F v_3) u_1))      (= (_F v_4) u_1))
  )
  (or (= (_F v_1) u_2) (= (_F v_2) u_2) (= (_F v_3) u_2) (= (_F v_4) u_2))
))
; defines the unary edges of the shape
(define-fun unaryS () Bool (and
       (alist u_1)  (not (acell u_1))
  (not (alist u_2))      (acell u_2)
))
; defines the binary edges of the shape
(define-fun binaryS () Bool (and
  (not (atail u_1 u_1))
  (not (atail u_2 u_1)) (not (atail u_2 u_2))

  (not (ahead u_1 u_1))
  (not (ahead u_2 u_1)) (not (ahead u_2 u_2))

  (not (anext u_1 u_1)) (not (anext u_1 u_2))
  (not (anext u_2 u_1))
))
; assert edge abstraction for unary edges
(define-fun unaryAbstraction () Bool (and
  (or (= (list v_1) (alist (_F v_1))) false)
  (or (= (cell v_1) (acell (_F v_1))) false)
  (or (= (list v_2) (alist (_F v_2))) false)
  (or (= (cell v_2) (acell (_F v_2))) false)
  (or (= (list v_3) (alist (_F v_3))) false)
  (or (= (cell v_3) (acell (_F v_3))) false)
  (or (= (list v_4) (alist (_F v_4))) false)
  (or (= (cell v_4) (acell (_F v_4))) false)
))
; assert edge abstraction for binary edges
(define-fun binaryAbstraction () Bool (and
  (or (= (tail v_1 v_1) (atail (_F v_1) (_F v_1))) (and (= (_F v_1) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_1 v_2) (atail (_F v_1) (_F v_2))) (and (= (_F v_1) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_1 v_3) (atail (_F v_1) (_F v_3))) (and (= (_F v_1) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_1 v_4) (atail (_F v_1) (_F v_4))) (and (= (_F v_1) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_2 v_1) (atail (_F v_2) (_F v_1))) (and (= (_F v_2) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_2 v_2) (atail (_F v_2) (_F v_2))) (and (= (_F v_2) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_2 v_3) (atail (_F v_2) (_F v_3))) (and (= (_F v_2) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_2 v_4) (atail (_F v_2) (_F v_4))) (and (= (_F v_2) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_3 v_1) (atail (_F v_3) (_F v_1))) (and (= (_F v_3) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_3 v_2) (atail (_F v_3) (_F v_2))) (and (= (_F v_3) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_3 v_3) (atail (_F v_3) (_F v_3))) (and (= (_F v_3) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_3 v_4) (atail (_F v_3) (_F v_4))) (and (= (_F v_3) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_4 v_1) (atail (_F v_4) (_F v_1))) (and (= (_F v_4) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_4 v_2) (atail (_F v_4) (_F v_2))) (and (= (_F v_4) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_4 v_3) (atail (_F v_4) (_F v_3))) (and (= (_F v_4) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_4 v_4) (atail (_F v_4) (_F v_4))) (and (= (_F v_4) u_1) (= (_F v_4) u_2)))

  (or (= (head v_1 v_1) (ahead (_F v_1) (_F v_1))) (and (= (_F v_1) u_1) (= (_F v_1) u_2)))
  (or (= (head v_1 v_2) (ahead (_F v_1) (_F v_2))) (and (= (_F v_1) u_1) (= (_F v_2) u_2)))
  (or (= (head v_1 v_3) (ahead (_F v_1) (_F v_3))) (and (= (_F v_1) u_1) (= (_F v_3) u_2)))
  (or (= (head v_1 v_4) (ahead (_F v_1) (_F v_4))) (and (= (_F v_1) u_1) (= (_F v_4) u_2)))

  (or (= (head v_2 v_1) (ahead (_F v_2) (_F v_1))) (and (= (_F v_2) u_1) (= (_F v_1) u_2)))
  (or (= (head v_2 v_2) (ahead (_F v_2) (_F v_2))) (and (= (_F v_2) u_1) (= (_F v_2) u_2)))
  (or (= (head v_2 v_3) (ahead (_F v_2) (_F v_3))) (and (= (_F v_2) u_1) (= (_F v_3) u_2)))
  (or (= (head v_2 v_4) (ahead (_F v_2) (_F v_4))) (and (= (_F v_2) u_1) (= (_F v_4) u_2)))

  (or (= (head v_3 v_1) (ahead (_F v_3) (_F v_1))) (and (= (_F v_3) u_1) (= (_F v_1) u_2)))
  (or (= (head v_3 v_2) (ahead (_F v_3) (_F v_2))) (and (= (_F v_3) u_1) (= (_F v_2) u_2)))
  (or (= (head v_3 v_3) (ahead (_F v_3) (_F v_3))) (and (= (_F v_3) u_1) (= (_F v_3) u_2)))
  (or (= (head v_3 v_4) (ahead (_F v_3) (_F v_4))) (and (= (_F v_3) u_1) (= (_F v_4) u_2)))

  (or (= (head v_4 v_1) (ahead (_F v_4) (_F v_1))) (and (= (_F v_4) u_1) (= (_F v_1) u_2)))
  (or (= (head v_4 v_2) (ahead (_F v_4) (_F v_2))) (and (= (_F v_4) u_1) (= (_F v_2) u_2)))
  (or (= (head v_4 v_3) (ahead (_F v_4) (_F v_3))) (and (= (_F v_4) u_1) (= (_F v_3) u_2)))
  (or (= (head v_4 v_4) (ahead (_F v_4) (_F v_4))) (and (= (_F v_4) u_1) (= (_F v_4) u_2)))

  (or (= (next v_1 v_1) (anext (_F v_1) (_F v_1))) (and (= (_F v_1) u_2) (= (_F v_1) u_2)))
  (or (= (next v_1 v_2) (anext (_F v_1) (_F v_2))) (and (= (_F v_1) u_2) (= (_F v_2) u_2)))
  (or (= (next v_1 v_3) (anext (_F v_1) (_F v_3))) (and (= (_F v_1) u_2) (= (_F v_3) u_2)))
  (or (= (next v_1 v_4) (anext (_F v_1) (_F v_4))) (and (= (_F v_1) u_2) (= (_F v_4) u_2)))

  (or (= (next v_2 v_1) (anext (_F v_2) (_F v_1))) (and (= (_F v_2) u_2) (= (_F v_1) u_2)))
  (or (= (next v_2 v_2) (anext (_F v_2) (_F v_2))) (and (= (_F v_2) u_2) (= (_F v_2) u_2)))
  (or (= (next v_2 v_3) (anext (_F v_2) (_F v_3))) (and (= (_F v_2) u_2) (= (_F v_3) u_2)))
  (or (= (next v_2 v_4) (anext (_F v_2) (_F v_4))) (and (= (_F v_2) u_2) (= (_F v_4) u_2)))

  (or (= (next v_3 v_1) (anext (_F v_3) (_F v_1))) (and (= (_F v_3) u_2) (= (_F v_1) u_2)))
  (or (= (next v_3 v_2) (anext (_F v_3) (_F v_2))) (and (= (_F v_3) u_2) (= (_F v_2) u_2)))
  (or (= (next v_3 v_3) (anext (_F v_3) (_F v_3))) (and (= (_F v_3) u_2) (= (_F v_3) u_2)))
  (or (= (next v_3 v_4) (anext (_F v_3) (_F v_4))) (and (= (_F v_3) u_2) (= (_F v_4) u_2)))

  (or (= (next v_4 v_1) (anext (_F v_4) (_F v_1))) (and (= (_F v_4) u_2) (= (_F v_1) u_2)))
  (or (= (next v_4 v_2) (anext (_F v_4) (_F v_2))) (and (= (_F v_4) u_2) (= (_F v_2) u_2)))
  (or (= (next v_4 v_3) (anext (_F v_4) (_F v_3))) (and (= (_F v_4) u_2) (= (_F v_3) u_2)))
  (or (= (next v_4 v_4) (anext (_F v_4) (_F v_4))) (and (= (_F v_4) u_2) (= (_F v_4) u_2)))
))
; assert a conjunction of the formulae defined and
(assert (and distinctNodes unaryG binaryG function summarization unaryS binaryS unaryAbstraction binaryAbstraction))
; check for satisfiability
(check-sat)
(get-model)
(get-unsat-core)
```

Listing B.3: Embedded Graph Encoding with constraints (see Sec. 4.4)

```
(set-option :print-success false)
;(set-option :produce-unsat-cores true)
(set-logic QF_UF)
; Sort for graph nodes
(declare-sort Node 0)
(declare-sort ANode 0)
(declare-fun _F (Node) ANode)
; four node individuals in the graph (node set / universe)
(declare-fun v_1 () Node)
(declare-fun v_2 () Node)
(declare-fun v_3 () Node)
(declare-fun v_4 () Node)
; two node individuals in the shape
(declare-fun u_1 () ANode)
(declare-fun u_2 () ANode)
; assert distinctiveness of the individuals
(define-fun distinctNodes () Bool
  (and (not (= v_1 v_2)) (not (= v_1 v_3)) (not (= v_1 v_4))
                         (not (= v_2 v_3)) (not (= v_2 v_4))
                                           (not (= v_3 v_4))
                                           (not (= u_1 u_2)))
)
; labels used in the graph become predicates
(declare-fun list  (Node) Bool)
(declare-fun cell  (Node) Bool)
(declare-fun tail  (Node Node) Bool)
(declare-fun head  (Node Node) Bool)
(declare-fun next  (Node Node) Bool)
(declare-fun alist (ANode) Bool)
(declare-fun acell (ANode) Bool)
(declare-fun atail (ANode ANode) Bool)
(declare-fun ahead (ANode ANode) Bool)
(declare-fun anext (ANode ANode) Bool)
; defines the unary edges of the graph
(define-fun unaryG () Bool (and
       (list v_1)  (not (cell v_1))
  (not (list v_2))      (cell v_2)
  (not (list v_3))      (cell v_3)
  (not (list v_4))      (cell v_4)
))
; defines the binary edges of the graph
(define-fun binaryG () Bool (and
  (not (tail v_1 v_1))      (tail v_1 v_2)  (not (tail v_1 v_3)) (not (tail v_1 v_4))
  (not (tail v_2 v_1)) (not (tail v_2 v_2)) (not (tail v_2 v_3)) (not (tail v_2 v_4))
  (not (tail v_3 v_1)) (not (tail v_3 v_2)) (not (tail v_3 v_3)) (not (tail v_3 v_4))
  (not (tail v_4 v_1)) (not (tail v_4 v_2)) (not (tail v_4 v_3)) (not (tail v_4 v_4))

  (not (head v_1 v_1)) (not (head v_1 v_2)) (not (head v_1 v_3))      (head v_1 v_4)
  (not (head v_2 v_1)) (not (head v_2 v_2)) (not (head v_2 v_3)) (not (head v_2 v_4))
  (not (head v_3 v_1)) (not (head v_3 v_2)) (not (head v_3 v_3)) (not (head v_3 v_4))
  (not (head v_4 v_1)) (not (head v_4 v_2)) (not (head v_4 v_3)) (not (head v_4 v_4))

  (not (next v_1 v_1)) (not (next v_1 v_2)) (not (next v_1 v_3)) (not (next v_1 v_4))
  (not (next v_2 v_1)) (not (next v_2 v_2)) (not (next v_2 v_3)) (not (next v_2 v_4))
; (not (next v_2 v_1))      (next v_2 v_2)  (not (next v_2 v_3)) (not (next v_2 v_4))
  (not (next v_3 v_1))      (next v_3 v_2)  (not (next v_3 v_3)) (not (next v_3 v_4))
  (not (next v_4 v_1)) (not (next v_4 v_2))      (next v_4 v_3)  (not (next v_4 v_4))
))
; defines the range of the embedding function
(define-fun function () Bool (and
  (or (= (_F v_1) u_1) (= (_F v_1) u_2))
  (or (= (_F v_2) u_1) (= (_F v_2) u_2))
  (or (= (_F v_3) u_1) (= (_F v_3) u_2))
  (or (= (_F v_4) u_1) (= (_F v_4) u_2))
))
; defines the summarization property
(define-fun summarization () Bool (and
  (or (and (= (_F v_1) u_1)       (not (= (_F v_2) u_1)) (not (= (_F v_3) u_1)) (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1))      (= (_F v_2) u_1)  (not (= (_F v_3) u_1)) (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1)) (not (= (_F v_2) u_1))      (= (_F v_3) u_1)  (not (= (_F v_4) u_1)))
      (and (not (= (_F v_1) u_1)) (not (= (_F v_2) u_1)) (not (= (_F v_3) u_1))      (= (_F v_4) u_1))
  )
  (or (= (_F v_1) u_2) (= (_F v_2) u_2) (= (_F v_3) u_2) (= (_F v_4) u_2))
))
; defines the unary edges of the shape
(define-fun unaryS () Bool (and
       (alist u_1)  (not (acell u_1))
  (not (alist u_2))      (acell u_2)
```

```
))
; defines the binary edges of the shape
(define-fun binaryS () Bool (and
  (not (atail u_1 u_1))
  (not (atail u_2 u_1)) (not (atail u_2 u_2))

  (not (ahead u_1 u_1))
  (not (ahead u_2 u_1)) (not (ahead u_2 u_2))

  (not (anext u_1 u_1)) (not (anext u_1 u_2))
  (not (anext u_2 u_1))
))
; assert edge abstraction for unary edges
(define-fun unaryAbstraction () Bool (and
  (or (= (list v_1) (alist (_F v_1))) false)
  (or (= (cell v_1) (acell (_F v_1))) false)
  (or (= (list v_2) (alist (_F v_2))) false)
  (or (= (cell v_2) (acell (_F v_2))) false)
  (or (= (list v_3) (alist (_F v_3))) false)
  (or (= (cell v_3) (acell (_F v_3))) false)
  (or (= (list v_4) (alist (_F v_4))) false)
  (or (= (cell v_4) (acell (_F v_4))) false)
))
; assert edge abstraction for binary edges
(define-fun binaryAbstraction () Bool (and
  (or (= (tail v_1 v_1) (atail (_F v_1) (_F v_1))) (and (= (_F v_1) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_1 v_2) (atail (_F v_1) (_F v_2))) (and (= (_F v_1) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_1 v_3) (atail (_F v_1) (_F v_3))) (and (= (_F v_1) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_1 v_4) (atail (_F v_1) (_F v_4))) (and (= (_F v_1) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_2 v_1) (atail (_F v_2) (_F v_1))) (and (= (_F v_2) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_2 v_2) (atail (_F v_2) (_F v_2))) (and (= (_F v_2) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_2 v_3) (atail (_F v_2) (_F v_3))) (and (= (_F v_2) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_2 v_4) (atail (_F v_2) (_F v_4))) (and (= (_F v_2) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_3 v_1) (atail (_F v_3) (_F v_1))) (and (= (_F v_3) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_3 v_2) (atail (_F v_3) (_F v_2))) (and (= (_F v_3) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_3 v_3) (atail (_F v_3) (_F v_3))) (and (= (_F v_3) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_3 v_4) (atail (_F v_3) (_F v_4))) (and (= (_F v_3) u_1) (= (_F v_4) u_2)))

  (or (= (tail v_4 v_1) (atail (_F v_4) (_F v_1))) (and (= (_F v_4) u_1) (= (_F v_1) u_2)))
  (or (= (tail v_4 v_2) (atail (_F v_4) (_F v_2))) (and (= (_F v_4) u_1) (= (_F v_2) u_2)))
  (or (= (tail v_4 v_3) (atail (_F v_4) (_F v_3))) (and (= (_F v_4) u_1) (= (_F v_3) u_2)))
  (or (= (tail v_4 v_4) (atail (_F v_4) (_F v_4))) (and (= (_F v_4) u_1) (= (_F v_4) u_2)))

  (or (= (head v_1 v_1) (ahead (_F v_1) (_F v_1))) (and (= (_F v_1) u_1) (= (_F v_1) u_2)))
  (or (= (head v_1 v_2) (ahead (_F v_1) (_F v_2))) (and (= (_F v_1) u_1) (= (_F v_2) u_2)))
  (or (= (head v_1 v_3) (ahead (_F v_1) (_F v_3))) (and (= (_F v_1) u_1) (= (_F v_3) u_2)))
  (or (= (head v_1 v_4) (ahead (_F v_1) (_F v_4))) (and (= (_F v_1) u_1) (= (_F v_4) u_2)))

  (or (= (head v_2 v_1) (ahead (_F v_2) (_F v_1))) (and (= (_F v_2) u_1) (= (_F v_1) u_2)))
  (or (= (head v_2 v_2) (ahead (_F v_2) (_F v_2))) (and (= (_F v_2) u_1) (= (_F v_2) u_2)))
  (or (= (head v_2 v_3) (ahead (_F v_2) (_F v_3))) (and (= (_F v_2) u_1) (= (_F v_3) u_2)))
  (or (= (head v_2 v_4) (ahead (_F v_2) (_F v_4))) (and (= (_F v_2) u_1) (= (_F v_4) u_2)))

  (or (= (head v_3 v_1) (ahead (_F v_3) (_F v_1))) (and (= (_F v_3) u_1) (= (_F v_1) u_2)))
  (or (= (head v_3 v_2) (ahead (_F v_3) (_F v_2))) (and (= (_F v_3) u_1) (= (_F v_2) u_2)))
  (or (= (head v_3 v_3) (ahead (_F v_3) (_F v_3))) (and (= (_F v_3) u_1) (= (_F v_3) u_2)))
  (or (= (head v_3 v_4) (ahead (_F v_3) (_F v_4))) (and (= (_F v_3) u_1) (= (_F v_4) u_2)))

  (or (= (head v_4 v_1) (ahead (_F v_4) (_F v_1))) (and (= (_F v_4) u_1) (= (_F v_1) u_2)))
  (or (= (head v_4 v_2) (ahead (_F v_4) (_F v_2))) (and (= (_F v_4) u_1) (= (_F v_2) u_2)))
  (or (= (head v_4 v_3) (ahead (_F v_4) (_F v_3))) (and (= (_F v_4) u_1) (= (_F v_3) u_2)))
  (or (= (head v_4 v_4) (ahead (_F v_4) (_F v_4))) (and (= (_F v_4) u_1) (= (_F v_4) u_2)))

  (or (= (next v_1 v_1) (anext (_F v_1) (_F v_1))) (and (= (_F v_1) u_2) (= (_F v_1) u_2)))
  (or (= (next v_1 v_2) (anext (_F v_1) (_F v_2))) (and (= (_F v_1) u_2) (= (_F v_2) u_2)))
  (or (= (next v_1 v_3) (anext (_F v_1) (_F v_3))) (and (= (_F v_1) u_2) (= (_F v_3) u_2)))
  (or (= (next v_1 v_4) (anext (_F v_1) (_F v_4))) (and (= (_F v_1) u_2) (= (_F v_4) u_2)))

  (or (= (next v_2 v_1) (anext (_F v_2) (_F v_1))) (and (= (_F v_2) u_2) (= (_F v_1) u_2)))
  (or (= (next v_2 v_2) (anext (_F v_2) (_F v_2))) (and (= (_F v_2) u_2) (= (_F v_2) u_2)))
  (or (= (next v_2 v_3) (anext (_F v_2) (_F v_3))) (and (= (_F v_2) u_2) (= (_F v_3) u_2)))
  (or (= (next v_2 v_4) (anext (_F v_2) (_F v_4))) (and (= (_F v_2) u_2) (= (_F v_4) u_2)))

  (or (= (next v_3 v_1) (anext (_F v_3) (_F v_1))) (and (= (_F v_3) u_2) (= (_F v_1) u_2)))
  (or (= (next v_3 v_2) (anext (_F v_3) (_F v_2))) (and (= (_F v_3) u_2) (= (_F v_2) u_2)))
  (or (= (next v_3 v_3) (anext (_F v_3) (_F v_3))) (and (= (_F v_3) u_2) (= (_F v_3) u_2)))
  (or (= (next v_3 v_4) (anext (_F v_3) (_F v_4))) (and (= (_F v_3) u_2) (= (_F v_4) u_2)))

  (or (= (next v_4 v_1) (anext (_F v_4) (_F v_1))) (and (= (_F v_4) u_2) (= (_F v_1) u_2)))
  (or (= (next v_4 v_2) (anext (_F v_4) (_F v_2))) (and (= (_F v_4) u_2) (= (_F v_2) u_2)))
  (or (= (next v_4 v_3) (anext (_F v_4) (_F v_3))) (and (= (_F v_4) u_2) (= (_F v_3) u_2)))
  (or (= (next v_4 v_4) (anext (_F v_4) (_F v_4))) (and (= (_F v_4) u_2) (= (_F v_4) u_2)))
))
; define constraint function for alpha
(define-fun constraint_alpha ((x_1 Node)) Bool (and
  (not (next x_1 x_1))
))
; define constraint adherence for (alpha,m)
(define-fun constraint_alpha_m () Bool (and
  (=> (and (= (_F v_1) u_2) true) (constraint_alpha v_1))
  (=> (and (= (_F v_2) u_2) true) (constraint_alpha v_2))
  (=> (and (= (_F v_3) u_2) true) (constraint_alpha v_3))
  (=> (and (= (_F v_4) u_2) true) (constraint_alpha v_4))
))
; define overall constraint adherence
(define-fun constraints () Bool (and
  constraint_alpha_m
  true
))
; assert a conjunction of the formulae defined and
(assert (and distinctNodes unaryG binaryG function summarization unaryS binaryS
             unaryAbstraction binaryAbstraction constraints))
; check for satisfiability
(check-sat)
(get-model)
```

Listing B.4: Listing for Code Template 24

```
(set-option :print-success false)
(set-logic QF_UF)
(declare-sort Node 0)
(declare-sort ANode 0)
; declarations for initial graph encoding
declare_graph
; declarations for overflow encoding
(declare-fun $o_1$ () Node)
    ⋮
(declare-fun $o_\lambda$ () Node)
; declarations for embeddings (see prelude for embedding encoding)
; these contain the declarations of the predicates for $G_1 \cdots G_k$
declare_embedding_0
    ⋮
declare_embedding_k
; declarations for applications
declare_rule_0
  ⋮
declare_rule_k-1

; graph encoding
(define-fun distinctNodes () Bool ... )
(define-fun unaryG () Bool ... )
```

```
(define-fun binaryG () Bool ... )
;overflow nodes
(define-fun overflow () Bool ... )
; embedding into initial shape $S_0$
(define-fun function_0 () Bool ... )
(define-fun summarization_0 () Bool ...)
(define-fun unaryS_0 () Bool ...)
(define-fun binaryS_0 () Bool ...)
(define-fun unaryAbstraction_0 () Bool ...)
(define-fun binaryAbstraction_0 () Bool ...)
(define-fun constraints_0 () Bool ...)
; rule application transforming $I$ into $G_1$
(define-fun () rule_selection_0 Bool ...)
(define-fun () rule_application_0 Bool ...)

$\vdots$

; rule application transforming $G_{k-1}$ into $G_k$
(define-fun () rule_selection_k-1 Bool ...)
(define-fun () rule_application_k-1 Bool ...)
; embedding into materialized error shape $S_k^e$
(define-fun function_k () Bool ... )
(define-fun summarization_k () Bool ...)
(define-fun unaryS_k () Bool ...)
(define-fun binaryS_k () Bool ...)
(define-fun unaryAbstraction_k () Bool ...)
(define-fun binaryAbstraction_k () Bool ...)
(define-fun constraints_k () Bool ...)

; assertion
(assert (and distinctNodes ··· constraints_k))
```

Listing B.5: Listing for Code Template 27

```
(define-fun rule_application () Bool (and
; first, positive effects of rule (unary, binary)
($u_{i+1}^{+,1}$ $n^{+,1}$) ··· ($u_{i+1}^{+,a}$ $n^{+,a}$)
($b_{i+1}^{+,1}$ $n^{+,1}$ $m^{+,1}$) ··· ($b_{i+1}^{+,c}$ $n^{+,c}$ $m^{+,c}$)
; then, negative effects of the rule (unary, binary)
($u_{i+1}^{-,1}$ $n^{-,1}$) ··· ($u_{i+1}^{-,b}$ $n^{-,b}$)
($b^{-,1}_{i+1}$ $n^{-,1}$ $m^{-,1}$) $\cdots$ ($b^{-,d}_{i+1}$ $n^{-,d}$ $m^{-,d}$)
; then the unchanged predicates $u^1, \dots, u^{e'}, b^{e'+1}, \dots, b^e \in \mathcal{P}^0$
; (unary, binary)
(= ($u^1_{i+1}$ $v_1$) ($u^1_i$ $v_1$)) $\cdots$ (= ($u^1_{i+1}$ $v_v$) ($u^1_i$ $v_v$))
$\vdots$
(= ($u^{e'}_{i+1}$ $v_1$) ($u^{e'}_i$ $v_1$)) $\cdots$ (= ($u^{e'}_{i+1}$ $v_v$) ($u^{e'}_i$ $v_v$))
(= ($b^{e'+1}_{i+1}$ $v_1$) ($b^{e'+1}_i$ $v_1$)) $\cdots$ (= ($b^{e'+1}_{i+1}$ $v_v$) ($b^{e'+1}_i$ $v_v$))
$\vdots$
(= ($b^e_{i+1}$ $v_1$) ($b^e_i$ $v_1$)) $\cdots$ (= ($b^e_{i+1}$ $v_v$) ($b^e_i$ $v_v$))
; and finally the potentially changed predicates
; $\bar{u}^1, \dots, \bar{u}^{f'}_c, \bar{b}^{f'+1}_c, \dots, \bar{b}^e_c \in \mathcal{P} \setminus \mathcal{P}^0$ (unary, binary)
(or (= $v_1$ $n^{+,1}$) $\cdots$ (= $v_1$ $n^{+,a}$)
    (= $v_1$ $n^{-,1}$) $\cdots$ (= $v_1$ $n^{-,a}$)
    (= ($\bar{u}^1_{i+1}$ $v_1$) ($\bar{u}^1_i$ $v_1$)))
    $\vdots$
(or (= $v_v$ $n^{+,1}$) $\cdots$ (= $v_v$ $n^{+,a}$)
    (= $v_v$ $n^{-,1}$) $\cdots$ (= $v_v$ $n^{-,a}$)
    (= ($\bar{u}^{f'}_{i+1}$ $v_1$) ($\bar{u}^{f'}_i$ $v_1$)))
(or (and (= $v_1$ $n^{+,1}$) (= $v_1$ $m^{+,1}$)) $\cdots$ (and (= $v_1$ $n^{+,c}$) (= $v_1$ $m^{+,c}$) )
    (and (= $v_1$ $n^{-,1}$) (= $v_1$ $m^{-,1}$)) $\cdots$ (and (= $v_1$ $n^{-,d}$) (= $v_1$ $m^{-,d}$) )
    (= ($\bar{b}^{f'+1}_{i+1}$ $v_1$) ($\bar{b}^{f'+1}_i$ $v_1$)))
    $\vdots$
(or (and (= $v_v$ $n^{+,1}$) (= $v_v$ $m^{+,1}$)) $\cdots$ (and (= $v_v$ $n^{+,c}$) (= $v_v$ $m^{+,c}$) )
    (and (= $v_v$ $n^{-,1}$) (= $v_v$ $m^{-,1}$)) $\cdots$ (and (= $v_v$ $n^{-,d}$) (= $v_v$ $m^{-,d}$) )
    (= ($\bar{b}^f_{i+1}$ $v_v$ $v_v$) ($\bar{b}^f_i$ $v_v$ $v_v$)))
))
```

Listing B.6: Listing for Code Template 31

```
      (define-fun exclusivity () Bool (and
        ; concrete nodes are distinct from each other
        (not (= $v_1$ $v_2$)) (not (= $v_1$ $v_3$)) $\cdots$ (not (= $v_1$ $v_k$))
                              (not (= $v_2$ $v_3$)) $\cdots$ (not (= $v_2$ $v_k$))
                                                                $\vdots$
                                                            (not (= $v_{k-1}$ $v_k$))
```

```
    ; concrete nodes are distinct from summary nodes
    (not (_V1 $v_1$)) ⋯ (not (_V1 $v_k$))
        ⋮                 ⋮
    (not (_Vm $v_1$)) ⋯ (not (_Vm $v_k$))
    ; all nodes of the node sort are either concrete nodes
    ; or embedded into one of the summary nodes
    (forall ((x (Node))) (xor
      (= x $v_1$) ⋯ (= x $v_k$)
      (_V1 x) ⋯ (_Vm x)
    ))
  ))
```

Listing B.7: Encoding of the feasibility of the shape shown in Fig. 4.11)

```
(set-option :produce-proofs true)
(set-logic UF)
(declare-sort Node 0)
(push 1)
(declare-fun x1 () Node)
(declare-fun x2 () Node)
(declare-fun _X3 (Node) Bool)
(declare-fun cell (Node) Bool)
(declare-fun next (Node Node) Bool)

(define-fun exclusivity () Bool (forall ((x Node)) (or
  (and (not (= x x1)) (not (= x x2)) (_X3 x))
  (and (not (= x x1)) (= x x2) (not (_X3 x)))
  (and (not (= x x2)) (not (_X3 x)) (= x x1))
)))

(define-fun unaryG () Bool (and
  (cell x1)
  (cell x2)
))


(define-fun binaryG () Bool (and
  (not (next x2 x1))
  (not (next x1 x1))
  (next x1 x2)
  (not (next x2 x2))
))

(define-fun unaryS () Bool (forall ((x Node))
  (=> (_X3 x) (cell x))
))

(define-fun binaryS1 () Bool (forall ((x Node))
  (=> (_X3 x) (and (not (next x1 x)) (not (next x x1))))
))

(define-fun binaryS2 () Bool (forall ((x Node))
  (=> (_X3 x) true)
))


(define-fun surjectivity () Bool (exists ((x_0 Node))
  (_X3 x_0)
))

(define-fun constraint_alpha1 () Bool (exists
  ((x Node) (y Node))
  (and (next x1 y) (next x x1) (not (= y x1)) (not (= x x1)))
))

(define-fun constraint_alpha2 () Bool (exists
  ((x Node) (y Node))
  (and (next x2 y) (next x x2) (not (= y x2)) (not (= x x2)))
))

(define-fun constraint_beta () Bool (forall ((x_0 Node))
  (=>
    (_X3 x_0)
    (forall ((x Node))
      (=>
        (and (next x_0 x_0) (not (= x x_0)))
        (and (not (next x x_0)) (not (next x_0 x)))
      )
    ))
))

(define-fun constraint_gamma () Bool (forall
  ((x_1 Node) (x_0 Node))
  (=>
    (and (_X3 x_0) (_X3 x_1))
    (and (= x_0 x_1) (next x_0 x_0))
  )
))

(assert (! exclusivity :named assert_exclusivity))
(assert (! unaryG :named assert_unaryG))
(assert (! binaryG :named assert_binaryG))
(assert (! unaryS :named assert_unaryS))
(assert (! binaryS1 :named assert_binaryS1))
(assert (! binaryS2 :named assert_binaryS2))
(assert (! surjectivity :named assert_surjectivity))
(assert (! constraint_alpha1 :named assert_constraint_alpha1))
(assert (! constraint_alpha2 :named assert_constraint_alpha2))
(assert (! constraint_beta :named assert_constraint_beta))
(assert (! constraint_gamma :named assert_constraint_gamma))

(check-sat)
(pop 1)
```

Listing B.8: Unconstrained Embedding Restricted Trace Encoding

```
(set-option :print-success false)
(set-logic QF_UF)
(declare-sort Node 0)
(declare-sort ANode 0)
; declarations for initial graph encoding
declare_graph
; declarations for overflow encoding
(declare-fun o_1 () Node)
    ⋮
(declare-fun o_λ () Node)
; declarations for embeddings (see prelude for embedding encoding)
; these contain the declarations of predicates for G_1 ··· G_k
declare_embedding_0
    ⋮
declare_embedding_2k+1
; declarations for applications
declare_rule_0
  ⋮
declare_rule_k-1

```

```
; graph encoding
(define-fun distinctNodes () Bool ... )
(define-fun unaryG () Bool ... )
(define-fun binaryG () Bool ... )
;overflow nodes
(define-fun overflow () Bool ... )
; embedding into initial shape $S_0$
(define-fun function_0 () Bool ... )
(define-fun summarization_0 () Bool ...)
(define-fun unaryS_0 () Bool ...)
(define-fun binaryS_0 () Bool ...)
(define-fun unaryAbstraction_0 () Bool ...)
(define-fun binaryAbstraction_0 () Bool ...)
; embedding correspondence 0$\longrightarrow$1
(define-fun corr_0_1 () Bool ... )
; embedding into materialized shape $S_0^m$
(define-fun function_1 () Bool ... )
(define-fun summarization_1 () Bool ...)
(define-fun unaryS_1 () Bool ...)
(define-fun binaryS_1 () Bool ...)
(define-fun unaryAbstraction_1 () Bool ...)
(define-fun binaryAbstraction_1 () Bool ...)
; rule application transforming $I$ into $G_1$
(define-fun () rule_selection_0 Bool ...)
(define-fun () rule_application_0 Bool ...)

⋮

; rule application transforming $G_{k-1}$ into $G_k$
(define-fun () rule_selection_k-1 Bool ...)
(define-fun () rule_application_k-1 Bool ...)
; embedding into central shape of step $k$
(define-fun function_k () Bool ... )
(define-fun summarization_k () Bool ...)
(define-fun unaryS_k () Bool ...)
(define-fun binaryS_k () Bool ...)
(define-fun unaryAbstraction_k () Bool ...)
(define-fun binaryAbstraction_k () Bool ...)
; embedding correspondence $2k \rightarrow 2k+1$
(define-fun corr_0_1 () Bool ... )
; embedding into materialized error shape $S_k$
```

```
(define-fun function_2k+1 () Bool ... )
(define-fun summarization_2k+1 () Bool ...)
(define-fun unaryS_2k+1 () Bool ...)
(define-fun binaryS_2k+1 () Bool ...)
(define-fun unaryAbstraction_2k+1 () Bool ...)
(define-fun binaryAbstraction_2k+1 () Bool ...)
; assertion
(assert (and distinctNodes ··· binaryAbstraction_2k+1))
```

References

[1] (2007). *OMG Unified Modeling Language (OMG UML).* Technical report, Object Management Group.

[2] (2010). *OMG Systems Modeling Language (OMG SysML ™).* Technical report, Object Management Group.

[3] (2011). *OMG Unified Modeling Language*. Technical Report August, Object Management Group.

[4] (2014). Design Methodology for Intelligent Technical Systems. *Lecture Notes in Mechanical Engineering. Springer, Heidelberg.*

[5] (2014). *MDA Guide revision 2.0*. Technical Report June, Object Management Group.

[6] (2014). *OMG Object Constraint Language*. Technical Report February, Object Management Group.

[7] Abdulla, P. A., Cerans, K., Jonsson, B., & Yih-Kuen, T. (1996). General decidability theorems for infinite-state systems. *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, (pp. 313–321).

[8] Abdulla, P. A., Chen, Y.-F., Delzanno, G., Haziza, F., Hong, C.-D., & Rezine, A. (2010). Constrained Monotonic Abstraction: A Cegar for Parameterized Verification. In P. Gastin & F. Laroussinie (Eds.), *21st International Conference on Concurrency Theory (CONCUR 2010)* (pp. 86–101). Paris, France: Springer Berlin Heidelberg.

[9] Alberti, F., Bruttomesso, R., & Ghilardi, S. (2012a). SAFARI: SMT-based Abstraction for Arrays with Interpolants. In P. Madhusudan & S. A. Seshia (Eds.), *Computer Aided Verification, 24th International Conference (CAV 2012* (pp. 679–685). Berkeley, California, USA: Springer Berlin Heidelberg.

[10] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., & Sharygina, N. (2012b). Lazy abstraction with interpolants for arrays. *Logic Programming and Automated Reasoning*, (pp. 46–61).

[11] Alur, R. & Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2), 183–235.

[12] Angermann, A. (2007). *Matlab-Simullink-Stateflow: Grundlagen, Toolboxen, Beispiele*. Oldenbourg Verlag.

[13] Apt, K. R., De Boer, F. S., & Olderog, E.-R. (2009). *Verification of sequential and concurrent programs*. Springer.

[14] Bakewell, A., Plump, D., & Runciman, C. (2003). Checking the shape safety of pointer manipulations. In *Relational and Kleene-Algebraic Methods in Computer Science: 7th International Seminar on Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra, Bad Malente, Germany, Revised Selected Papers* (pp. 48–61).

[15] Baldan, P., Corradini, A., & König, B. (2001). A static analysis technique for graph transformation systems. In *Proceedings odCONCUR 2001—Concurrency Theory, 12th International Conference* (pp. 381–395). Aalborg, Denmark.

[16] Baldan, P., Corradini, A., & König, B. (2004). Verifying Finite-State Graph Grammars: an Unfolding-based Approach. In P. Gardner & N. Yoshida (Eds.), *15th International Conference on Concurrency Theory (CONCUR 2004)* (pp. 83–98). London, UK: Springer.

[17] Baldan, P., Corradini, A., & König, B. (2008). A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7), 869–907.

[18] Ball, T., Levin, V., & Rajamani, S. K. (2011). A decade of software model checking with SLAM. *Communications of the ACM*, 54(7), 68—-76.

[19] Baresi, L. & Spoletini, P. (2006). On the use of Alloy to analyze graph transformation systems. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, & G. Rozenberg (Eds.), *Third International Conference on Graph Transformations (ICGT 2006)* (pp. 306–320). Natal, Rio Grande do Norte, Brazil: Springer.

[20] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., & Tinelli, C. (2011). CVC4. In *Computer Aided Verification - 23rd International Conference (CAV 2011)* (pp. 171–177). Snowbird, UT, USA: Springer.

[21] Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A., & Moura, L. (2012). 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3), 1–35.

[22] Bauer, J., Boneva, I., Kurbán, M. E., & Rensink, A. (2008). A modal-logic based graph abstraction. In H. Ehrig, R. Heckel, G. Rozenberg, & G. Taentzer (Eds.), *4th International Conference on Graph Transformations (ICGT 2008)* (pp. 321–335). Leicester, United Kingdom: Springer.

[23] Bauer, J., Toben, T., & Westphal, B. (2007). Mind the shapes: abstraction refinement via topology invariants. In K. S. Namjoshi, T. Yoneda, T. Higashino, & Y. Okamura (Eds.), *Automated Technology for Verification and and Analysis, 5th International Symposium (ATVA 2007)* (pp. 35–50). Tokyo, Japan: Springer.

[24] Bauer, J. & Wilhelm, R. (2007). Static analysis of dynamic communication systems by partner abstraction. In H. R. Nielson & G. Filé (Eds.), *Static Analysis, 14th International Symposium (SAS 2007)* (pp. 249–264). Kongens Lynby, Denmark: Springer.

[25] Becker, B., Beyer, D., Giese, H., Klein, F., & Schilling, D. (2006). Symbolic invariant verification for systems with dynamic structural adaptation. In L. J. Osterweil, H. D. Rombach, & M. L. Soffa (Eds.), *Proceedings of the 28th international conference on Software engineering (ICSE 2006)* (pp. 72–81). New York, New York, USA: ACM.

[26] Becker, S., Dziwok, S., Gerking, C., Schäfer, W., Heinzemann, C., Thiele, S., Meyer, M., Priesterjahn, C., Pohlmann, U., & Tichy, M. (2014). *The MechatronicUML Design Method - Process and Language for Platform-Independent Modeling*. Technical report, Heinz Nixdorf Institute, University of Paderborn, Paderborn.

[27] Behrmann, G., David, A., Larsen, K. G., Hå kansson, J., Petterson, P., Wang, Y., & Hendriks, M. (2006). Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006)* (pp. 125–126). Riverside, California, USA: IEEE.

[28] Bell Labs, R. (2014). Promela Language Reference.

[29] Bengtsson, J. & Yi, W. (2004). Timed automata: Semantics, Algorithms and Tools. In J. Desel, W. Reisig, & G. Rozenberg (Eds.), *Lectures on Concurrency and Petri Nets, Advances in Petri Nets* (pp. 87–124). Eichstätt, Germany: Springer.

[30] Bertrand, N., Delzanno, G., König, B., Sangnier, A., & Stückrath, J. (2012). On the Decidability Status of Reachability and Coverability in Graph Transformation Systems. In A. Tiwari (Ed.), *23rd International Conference on Rewriting*

Techniques and Applications (RTA 2012) (pp. 1–24). Nagoya, japan: Schloss Dagstuhl - Leibniz Zentrum für Informatik.

[31] Beydeda, S., Book, M., & Gruhn, V., Eds. (2005). *Model-Driven Software Development*. Springer Berlin Heidelberg.

[32] Beyer, D., Henzinger, T. A., Jhala, R., & Majumdar, R. (2007). The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6), 505–525.

[33] Beyer, D., Henzinger, T. A., & Théoduloz, G. (2006). Lazy shape analysis. In T. Ball & B. J. Jones (Eds.), *Computer Aided Verification, 18th International Conference (CAV 2006)* (pp. 532–546). Seattle, Washington, USA: Springer.

[34] Biere, A., Cimatti, A., Clarke, E. M., & Zhu, Y. (1999). Symbolic model checking without BDDs. In R. Cleaveland (Ed.), *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference (TACAS 1999)*, number 97 (pp. 193–207). Amsterdam, The Netherlands: Springer.

[35] Bjø rner, N. (2011). Engineering theories with Z3. In H. Yang (Ed.), *Programming Languages and Systems - 9th Asian Symposium (APLAS 2011)* (pp. 4–16). Kenting, Taiwan: Springer.

[36] Boneva, I., Kreiker, J., Kurbán, M., Rensink, A., & Zambon, E. (2012). *Graph Abstraction and Abstract Graph Transformations (Amended version)*. Technical report, Centre for Telematics and Information Technology, University of Twente.

[37] Brucker, A. D. & Wolff, B. (2012). Featherweight OCL. In M. Balaban, J. Cabot, M. Gogolla, & C. Wilke (Eds.), *Proceedings of the 12th Workshop on OCL and Textual Modelling (OCL 2022)* (pp. 19–24). Innsbruck, Austria: ACM.

[38] Brückner, I., Dräger, K., Finkbeiner, B., & Wehrheim, H. (2007). Slicing abstractions. In *International Symposium on Fundamentals of Software Engineering (FSEN 2007)* (pp. 17–32). Tehran, Iran: Springer.

[39] Bruttomesso, R. (2011). Satisfiability Modulo Theories Lezione 1.

[40] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., & Sebastiani, R. (2008). The mathsat 4 smt solver. In A. Gupta & S. Malik (Eds.), *Computer Aided Verification, 20th International Conference (CAV 2008)* (pp. 299–303). Princeton, New Jersey, USA: Springer.

[41] Bruttomesso, R., Pek, E., Sharygina, N., & Tsitovich, A. (2010). The opensmt solver. In J. Esparza & R. Majumdar (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010)* (pp. 150–153). Paphos, Cyprus: Springer.

[42] Buckler, C. (2010). *Generierung von Shape-Analysen aus Graph-Spezifikationen*. Master thesis, Universität Paderborn.

[43] Burmester, S., Giese, H., & Tichy, M. (2005). Model-driven development of reconfigurable mechatronic systems with MechatronicUML. In U. Aß mann, M. Aksit, & A. Rensink (Eds.), *Model Driven Architecture, European MDA Workshops: Foundations and Applications (MDAFA 2003)* (pp. 47–61). Twente, The Netherlands: Springer.

[44] Cabot, J., Clarisó, R., Guerra, E., & de Lara, J. (2010). Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. *Journal of Systems and Software*, 83(2), 283–302.

[45] Chen, P. P. (1976). The Entity-Relationship model — Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1), 9–36.

[46] Christ, J., Hoenicke, J., & Nutz, A. (2012). SMTInterpol: An Interpolating SMT Solver. In A. F. Donaldson & D. Parker (Eds.), *Model Checking Software - 19th International Workshop (SPIN 2012)*, volume 3 (pp. 248–254). Oxford, UK: Springer.

[47] Cimatti, A. & Griggio, A. (2012). Software model checking via IC3. In P. Madhusudan & S. A. Seshia (Eds.), *Computer Aided Verification, 24th International Conference (CAV 2012* (pp. 277–293). Berkeley, California, USA: Springer.

[48] Cimatti, A., Griggio, A., & Micheli, A. (2011a). KRATOS – A Software Model Checker for SystemC. In G. Gopalakrishnan & S. Qadeer (Eds.), *Computer Aided Verification - 23rd International Conference (CAV 2011)* (pp. 310–316). Snowbird, Utah, USA: Springer Berlin Heidelberg.

[49] Cimatti, A., Narasamdya, I., & Roveri, M. (2011b). Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In P. A. Abdulla, K. Rustan, & M. Leino (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 17th International Conference (TACAS 2011)* (pp. 341–356). Saarbrücken, Germany: Springer.

[50] Clark, T. & Warmer, J. (2002). Object Modeling with the OCL - The Rationale behind the Object Constraint Language. *Lecture Notes in Computer Science*, 2263.

[51] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2000). Counterexample-guided abstraction refinement. In E. A. Emerson & A. P. Sistla (Eds.), *Computer Aided Verification, 12th International Conference (CAV 2000)* (pp. 154 —- 169). Chicago, Illinois, USA: Springer.

[52] Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. Cambridge, Massachusetts: MIT Press, 1 edition.

[53] Clarke, E. M., Grumberg, O., & Peled, D. A. (2001). *Model Checking*, volume 962. MIT Press.

[54] Clarke, E. M., Kroening, D., Sharygina, N., & Yorav, K. (2005). SATABS: SAT-based predicate abstraction for ANSI-C. In N. Halbwachs & L. D. Zuck (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS 2005)* (pp. 570–574). Edinburgh, Scotland, UK: Springer.

[55] Codish, M., Fuhs, C., Giesl, J., & Schneider-Kamp, P. (2010). Lazy abstraction for size-change termination. In C. G. Fermüller & A. Voronkov (Eds.), *Logic for Programming, Artificial Intelligence, and Reasoning, 17th International Conference (LPAR 2010)* (pp. 217–232). Yogyakarta, Indonesia: Springer.

[56] Courcelle, B. (1994). Monadic Second-Order Definable Graph Transductions: A Survey. *Theoretical Computer Science*, 126(1), 53—-75.

[57] Cousot, P. (1981). Semantic Foundations of Program Analysis. *Program Flow Analysis: Theory and Applications*, 10, 303–342.

[58] Cousot, P. (2012). Formal verification by abstract interpretation. In A. Goodloe & S. Person (Eds.), *NASA Formal Methods, 4th International Symposium (NFM 2012)* (pp. 3–7). Norfolk, Virginia: Springer.

[59] Cousot, P. & Cousot, R. (1992). Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13(2&3), 103–179.

[60] Craig, W. (1957). Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3), 269–285.

[61] Distefano, D., O'hearn, P. W., & Yang, H. (2006). A local shape analysis based on separation logic. In H. Hermanns & J. Palsberg (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference (TACAS 2006)* (pp. 287–302). Vienna, Austria: Springer.

[62] Dodds, M. & Plump, D. (2006). Extending C for Checking Shape Safety. *Electronic Notes in Theoretical Computer Science*, 154(2), 95–112.

[63] Dotti, F. L., Foss, L., Ribeiro, L., & dos Santos, O. M. (2003). Verification of Distributed Object-based Systems. In E. Najm, U. Nestmann, & P. Stevens (Eds.), *Formal Methods for Open Object-based Distributed Systems (FMOODS 2003)* (pp. 261–275). Paris, France: Springer.

[64] Drewes, F., Kreowski, H.-J., & Habel, A. (1997). Hyperedge Replacement Graph Grammars. In G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 (pp. 95–162). World Scientific.

[65] Dutertre, B. (2006). Yices 2.2. In A. Biere & R. Bloem (Eds.), *Proceedings on the 2nd SMT competition (SMTCOMP 2006)* (pp. 737 —- 744). Vienna, Austria: Springer International Publishing.

[66] Eckardt, T., Heinzemann, C., Henkler, S., Hirsch, M., Priesterjahn, C., & Schäfer, W. (2011). Modeling and Verifying Dynamic Communication Structures Based on Graph Transformations. *Computer Science - Research and Development*, 28(1), 3–22.

[67] Ehrig, H., Prange, U., & Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation - Monographs in Theoretical Computer Science*. Springer.

[68] Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., & Varró-Gyapay, S. (2005). Model Transformation by Graph Transformation: A Comparative Study. In *Proceedings of the International Workshop on Model Transformations in Practice (MTiP 2005)* Genova, Italy.

[69] Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., & Woodhull, G. (2002). Graphviz — open source graph drawing tools. In *Graph Drawing* (pp. 483–484).

[70] Engelfriet, J. & Rozenberg, G. (1991). Graph Grammars Based on Node Rewriting: An Introduction to NLC Graph Grammars. In H. Ehrig, H.-J. Kreowski, & G. Rozenberg (Eds.), *Graph Grammars and Their Application to Computer Science, 4th International Workshop* (pp. 12 —- 23). Bremen, Germany: Springer.

[71] Fischer, T., Niere, J., Torunski, L., & Zündorf, A. (1998). Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Theory and Application of Graph Transformations, 6th International Workshop (TAGT 1998)* (pp. 296–309). Paderborn, Germany: Springer.

[72] Franssen, M. (1999). Cocktail: A Tool for Deriving Correct Programs. In *Workshop on Automated Reasoning* (pp. 39 —- 40). Edinburgh, Scotland, UK: The Society for the Study of Artificial Intelligence and Simulation Behaviour.

[73] Frei, R. & Serugendo, G. D. M. (2014). *Self-Healing Software*, chapter 5, (pp. 71–82). World Scientific.

[74] Geiß, R., Batz, G. V., Grund, D., Hack, S., & Szalkowski, A. (2006). GrGen: A fast SPO-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, & G. Rozenberg (Eds.), *Graph Transformations, Third International Conference (ICGT 2006)* (pp. 383–397). Natal, Rio Grande do Norte, Brazil: Springer.

[75] Ghamarian, A. H., Mol, M., Rensink, A., Zambon, E., & Zimakova, M. (2011). Modelling and Analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1), 15–40.

[76] Ghezzi, C., Mocci, A., & Monga, M. (2009). Synthesizing intensional behavior models by graph transformation. In *Proceedings of the 31st International nference on Software Engineering (ICSE 2009)* (pp. 430–440). Vancouver, Canada: IEEE.

[77] Ghiya, R. & Hendren, L. (1996). Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In H.-J. Boehm & G. L. Steele Jr. (Eds.), *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 1—-15). St. Petersburg Beach, Florida, USA: ACM Press.

[78] Giese, H., Glesner, S., & Leitner, J. (2006). Towards verified model transformations. In D. Hearnden, J. G. Süß, B. Baudry, & N. Rapin (Eds.), *3rd Workshop on Model Development , Validation and Verification (MoDeVVa 2006)* Genova, Italy: Le Commissariat à l'Energie Atomique - CEA.

[79] Giese, H. & Henkler, S. (2006). A Survey of Approaches for the Visual Model-Driven Development of Next Generation Software-Intensive Systems. *Journal of Visual Languages & Computing*, 17(6), 528–550.

[80] Giese, H., Hildebrandt, S., & Lambers, L. (2010). Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. In L. Lucio, E. Vieira, & S. Weiß leder (Eds.), *Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa 2010)* (pp. 19–24). Oslo, Norway: IEEE.

[81] Graf, S. & Saïdi, H. (1997). Construction of abstract state graphs with PVS. In O. Grumberg (Ed.), *Computer Aided Verification, 9th International Conference (CAV 1997* (pp. 72 – 83). Haifa, Israel: Springer.

[82] Gupta, A. (1992). Formal Hardware Verification Methods: A Survey. In R. Kurshan (Ed.), *Formal Methods in System Design*, volume 1 (pp. 5–92). Springer US.

[83] Habel, A., Heckel, R., & Taentzer, G. (1996). Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3/4), 287–313.

[84] Heckel, R., Ehrig, H., Wolter, U., & Corradini, A. (1997). Integrating the Specification Techniques of Graph Transformation and Temporal Logic. In I. Prívara & P. Ruzicka (Eds.), *Mathematical Foundations of Computer Science, 2nd International Symposium (MFCS 1997)* (pp. 219 —- 228). Bratislava, Slovakia: Springer.

[85] Heinen, J., Noll, T., & Rieger, S. (2010). Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. *Electronic Notes in Theoretical Computer Science*, 266(August), 93 —- 107.

[86] Heinzemann, C., Suck, J., & Eckardt, T. (2010). Reachability Analysis on Timed Graph Transformation Systems. *Electronic Communications of the European Association of Software Science and Technology*, 32.

[87] Heisserman, J. (2004). *Exploring the Future of Design: Formal engineering design synthesis*, volume 36. Cambridge University Press.

[88] Henzinger, T. A. (1996). *The theory of hybrid automata*. New Brunswick, New Jersey, USA: IEEE Computer Society.

[89] Henzinger, T. A., Jhala, R., Majumdar, R., & McMillan, K. L. (2004). Abstractions from Proofs. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2004)*, 39(1), 232–244.

[90] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., & Weimer, W. (2002a). Temporal-safety proofs for systems code. In E. Brinksma & K. G. Larsen (Eds.), *Computer Aided Verification, 14th International Conference (CAV 200* (pp. 526–538). Copenhagen, Denmark: Springer.

[91] Henzinger, T. A., Jhala, R., Majumdar, R., & Sutre, G. (2002b). Lazy Abstraction. In J. Launchbury & J. C. Mitchell (Eds.), *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, volume 37 (pp. 58–70). Portland, Oregon, USA: ACM.

[92] Hoare, C. (1985). *Communicating sequential processes*, volume 21. Prentice-Hall.

[93] Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., & Rümmer, P. (2012). A verification toolkit for numerical transition systems. In D. Giannakopoulou & D. Méry (Eds.), *18th International Symposium on Formal Methods (FM 2012)* (pp. 247–251). Paris, France: Springer.

[94] Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 279–295.

[95] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kleburtz, D., Nikhil, R., Partain, W., & Peterson, J. (1997). Report on the Programming Language Haskell: a non-strict, Purely Functional Language Version 1.2. *ACM SIGPLAN Notices - Haskell special issue*, 27(May), 1 – 164.

[96] Isenberg, T. (2012). *Bounded Model Checking für Graphtransformationssysteme als SMT-Problem*. Master thesis, University of Paderborn.

[97] Isenberg, T., Steenken, D., & Wehrheim, H. (2013). Bounded Model Checking of Graph Transformation Systems via SMT Solving. In D. Beyer & M. Boreale (Eds.), *Formal Techniques for Distributed Systems - Joint International Conference (FMOODS/FORTE 2013)* (pp. 178–192). Florence, Italy: Springer.

[98] Ivančić, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., & Ashar, P. (2005). F-Soft: Software verification platform. In K. Etessami & S. K. Rajamani (Eds.), *Computer Aided Verification, 17th International Conference (CAV 2005)* (pp. 301–306). Edinburgh, Scotland, UK: Springer.

[99] Jackson, D., Shlyakhter, I., & Sridharan, M. (2001). A micromodularity mechanism. In *International Symposium on Foundations of Software Engineering*, volume 26 (pp.62̄). Vienna· Austria: ACM.

[100] Joshi, S. & König, B. (2008). Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems. In A. Gupta & S. Malik (Eds.), *Computer Aided Verification, 20th International Conference (CAV 2008)* (pp. 214–226). Princeton, New Jersey, USA: Springer.

[101] Kiefer, S., Schwoon, S., & Suwimonteerabuth, D. (2009). Moped - A Model-Checker for Pushdown Systems. *Date of Access: January*, 20.

[102] König, B. & Kozioura, V. (2006). Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In H. Hermanns & J.

Palsberg (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference (TACAS 2006)*, volume 627 (pp. 197–211). Vienna, Austria: Springer.

[103] König, B. & Kozioura, V. (2008). Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 211, 201–210.

[104] König, B. & Stückrath, J. (2014). A general framework for Well-Structured Graph Transformation Systems. In P. Baldan & D. Gorla (Eds.), *Concurrency Theory, 25th International Conference (CONCUR 2014)* (pp. 467–481). Rome, Italy: Springer.

[105] Kreowski, H.-J., Kuske, S., & Wille, R. (2010). Graph Transformation Units Guided by a SAT Solver. In H. Ehrig, A. Rensink, G. Rozenber, & A. Schürr (Eds.), *Graph Transformations, 5th International Conference (ICGT 2010)*, volume 637 (pp. 27–42). Enschede, The Netherlands: Springer.

[106] Kroening, D. & Weissenbacher, G. (2011). Interpolation-based software verification with WOLVERINE. In G. Gopalakrishnan & S. Qadeer (Eds.), *Computer Aided Verification, 23rd International Conference (CAV 2011)* (pp. 573–578). Snowbird, Utah, USA: Springer.

[107] Lee, O., Yang, H., & Yi, K. (2005). Automatic Verification of Pointer Programs Using Grammar-based Shape Analysis. In S. Sagiv (Ed.), *Programming Languages and Systems, 14th European Symposium on Programming (ESOP 2005)* (pp. 124–140). Edinburgh, Scotland, UK: Springer.

[108] Lev-Ami, T. & Sagiv, M. (2000). TVLA: A System for Implementing Static Analyses. In J. Palsberg (Ed.), *Static Analysis, 7th International Symposium (SAS 2000)* (pp. 280–302). Santa Barbara, California, USA: Springer.

[109] Lluch-Lafuente, A. & Vandin, A. (2011). Towards a Maude Tool for Model Checking Temporal Graph Properties. *Electronic Communications of the European Association of Software Science and Technology*, 41.

[110] Loginov, A., Reps, T., & Sagiv, M. (2005). Abstraction refinement via inductive learning. In K. Etessami & S. K. Rajamani (Eds.), *Computer Aided Verification, 17th International Conference (CAV 2005)* (pp. 519–533). Edingurgh, Scotland: Springer.

[111] Mazanek, S., Rensink, A., & Van Gorp, P. (2010). *Transformation Tool Contest 2010*. Technical Report July, Malaga, Spain.

[112] McMillan, K. L. (2003). Interpolation and SAT-based Model Checking. In W. A. Hunt Jr. & F. Somenzi (Eds.), *Computer Aided Verification, 15th International Conference (CAV 2003)* (pp. 1–13). Boulder, Colorado, USA: Springer.

[113] McMillan, K. L. (2006). Lazy Abstraction with Interpolants. In T. Ball & R. B. Jones (Eds.), *Computer Aided Verification, 18th International Conference (CAV 2006)* (pp. 123–136). Seattle, Washington, USA: Springer.

[114] Moura, L. D. & Bjø rner, N. (2008). Z3: An Efficient SMT Solver. In C. R. Ramakrishnan & J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)* (pp. 337–340). Budapest, Hungary: Springer.

[115] Moura, L. D. & Bjø rner, N. (2011). Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9), 69–77.

[116] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4), 541–580.

[117] Namjoshi, K. & Kurshan, R. (2000). Syntactic program transformations for automatic abstraction. In E. A. Emerson & A. P. Sistla (Eds.), *Computer Aided Verification, 12th International Conference (CAV 2000)* (pp. 435–449). Chicago, Illinois: Springer.

[118] Navabi, Z. (1997). *VHDL: Analysis and Modeling of Digital Systems*. New York, NY, USA: McGraw-Hill, Inc., 2 edition.

[119] Necula, G. C. (2011). Proof-carrying code. In H. C. A. van Tilborg & S. Jajodia (Eds.), *Encyclopedia of Cryptography and Security*, number November (pp. 984—-986). Springer, 2 edition.

[120] Pandey, R. K. (2011). Object constraint language (OCL); past, present and future. *ACM SIGSOFT Software Engineering Notes*, 36(1), 1—-4.

[121] Prabhakar, P., Duggirala, P. S., Mitra, S., & Viswanathan, M. (2013). Hybrid automata-based cegar for rectangular hybrid systems. In R. Giacobazzi, J. Berdine, & I. Mastroeni (Eds.), *Verification, Model Checking, and Abstract Interpretation, 14th International Conference (VMCAI 2013)* (pp. 48–67). Rome, Italy: Springer.

[122] Priesterjahn, C., Steenken, D., & Tichy, M. (2013). Timed hazard analysis of self-healing systems. In J. Cámara, R. de Lemos, C. Ghezzi, & A. Lopes (Eds.), *Assurances for Self-Adaptive Systems – Principles, Models, and Techniques* (pp. 112–151). Springer.

[123] Rafe, V. & Rahmani, A. T. (2008). Formal analysis of workflows using UML 2.0 activities and graph transformation systems. In J. S. Fitzgerald, A. E. Haxthausen, & H. Yenigün (Eds.), *International Conference on Theoretical Aspects of Computing (ICTAC 2008)* (pp. 305–318). Istanbul, Turkey: Springer.

[124] Rensink, A. (2003). *A logic of local graph shapes*. Technical report, Formal Methods, and Tools, University of Twente, Twente, The Netherlands.

[125] Rensink, A. (2004). Canonical Graph Shapes. In D. A. Schmidt (Ed.), *Programming Languages and Systems, 13th European Symposium on Programming (ESOP 2014)* (pp. 401–415). Barcelona, Spain: Springer.

[126] Rensink, A. & Distefano, D. (2006). Abstract Graph Transformation. *Electronic Notes in Theoretical Computer Science*, 157(1), 39–59.

[127] Rensink, A. & Zambon, E. (2011). Neighbourhood Abstraction in GROOVE. *Electronic Communications of the European Association of Software Science and Technology*, 32.

[128] Rensink, A. & Zambon, E. (2012). Pattern-Based Graph Abstraction. In H. Ehrig, G. Engels, H.-J. Kreowski, & G. Rozenberg (Eds.), *Graph Transformations, 6th International Conference (ICGT 2012* (pp. 66–80). Bremen, Germany: Springer.

[129] Reps, T., Sagiv, M., & Loginov, A. (2010). Finite Differencing of Logical Formulas for Static Analysis. *ACM Transactions on Programming Languages and Systems*, 32(6), 24.

[130] Rieger, S. & Noll, T. (2008). Abstracting Complex Data Structures by Hyperedge Replacement. In H. Ehrig, R. Heckel, G. Rozenberg, & G. Taentzer (Eds.), *Graph Transformations, 4th International Conference (ICGT 2008)* (pp. 69–83). Leicester, United Kingdom: Springer.

[131] Robertson, N. & Seymour, P. (2004). Graph Minors. XX. Wagner's Conjecture. *Journal of Combinatorial Theory, Series B*, 92(2), 325–357.

[132] Robertson, N. & Seymour, P. D. (2010). Graph minors XXIII. Nash-Williams' Immersion Conjecture. *Journal of Combinatorial Theory, Series B*, 100(2), 181–205.

[133] Rozenberg, G. & Ehrig, H. (1997). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, volume 1. World Scientific.

[134] Sagiv, M., Reps, T., & Wilhelm, R. (2002). Parametric Shape Analysis via 3-valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3), 217–298.

[135] Saïdi, H. & Shankar, N. (1999). Abstract and model check while you prove. In N. Halbwachs & D. Peled (Eds.), *Computer Aided Verification, 11th International Conference (CAV 1999)* (pp. 443–454). Trento, Italy: Springer.

[136] Saksena, M., Wibling, O., & Jonsson, B. (2008). Graph Grammar Modeling and Verification of ad hoc Routing Protocols. In C. R. Ramakrishnan & J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)* (pp. 18–32). Budapest, Hungary: Springer-Verlag.

[137] Sangiorgi, D. & Walker, D. (2001). *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press.

[138] Sannella, D. & Tarlecki, A. (2012). *Foundations of algebraic specification and formal software development, Monographs in Theoretical Computer Science*. Springer.

[139] Schilling, D. (2006). *Kompositionale Softwareverifikation mechatronischer Systeme*. Phd thesis, Universität Paderborn.

[140] Schmidt, A. & Varró, D. (2003). CheckVML: A Tool for Model Checking Visual Modeling Languages. In P. Stevens, J. Whittle, & G. Booch (Eds.), *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference*, volume 2 (pp.92̄). San Francisco, California, USA: Springer.

[141] Schürr, A. (1997). Developing Graphical (Software Engineering) Tools with PROGRES. In W. R. Adrion, A. Fuggetta, R. N. Taylor, & A. I. Wasserman (Eds.), *Proceedings of the 19th International Conference on Software Engineering* (pp. 487–550). Boston, Massachusetts: ACM.

[142] Schürr, A. & Klar, F. (2008). 15 Years of Triple Graph Grammars. In H. Ehrig, R. Heckel, G. Rozenberg, & G. Taentzer (Eds.), *Graph Transformations, 4th International Conference (ICGT 2008)*, volume 5214 (pp. 411–425). Leicester, United Kingdom: Springer.

[143] Sistla, A. P. (1994). Safety, Liveness and Fairness in Temporal Logic. *Formal Aspects of Computing*, 6(5), 495–511.

[144] Steenken, D., Wehrheim, H., & Wonisch, D. (2011). Sound and Complete Abstract Graph Transformation. In A. da Silva Simao & C. Morgan (Eds.), *14th Brazilian Symposium on Formal Methods, Foundations and Applications (SBMF 2011)* (pp. 92–107). Sao Paulo, Brazil: Springer.

[145] Stenzel, K., Moebius, N., & Reif, W. (2011). Formal verification of QVT transformations for code generation. In J. Whittle, T. Clark, & T. Kühne (Eds.), *Model Driven Engineering Languages and Systems, 14th International Conference (MODELS 2011)* (pp. 533–547). Wellington, New Zealand: Springer.

[146] Taentzer, G. (2004). AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. L. Pfaltz, M. Nagl, & B. Böhlen (Eds.), *Applications of graph Transformations with Industrial Relevance, 2nd International Workshop (AGTIVE 2003)* (pp. 446–453). Charlottesville, Virginia, USA: Springer.

[147] Taentzer, G. & Ehrig, H. (1996). *Computing by Graph Transformation - A Survey and Annotated Bibliography*. Technical Report March, TU Berlin, Berlin, Germany.

[148] Tichy, M. & Klöpper, B. (2011). Planning Self-Adaption with Graph Transformations. In A. Schürr, D. Varró, & G. Varró (Eds.), *Applications of Graph Transformations with Industrial Relevance, 4th International Symposium (AGTIVE 2011)* (pp. 137–152). Budapest, Hungary: Springer.

[149] Trächtler, A. (2006). Railcab-mit innovativer Mechatronik zum Schienenverkehrssystem der Zukunft. In *VDE-Kongress 2006*: VDE VERLAG GmbH.

[150] Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E., & Weinell, E. (2008). Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In A. Schürr, M. Nagl, & A. Zündorf (Eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)* (pp. 540–565). Kassel, Germany: Springer.

[151] Vizel, Y., Grumberg, O., & Shoham, S. (2012). Lazy abstraction and SAT-based reachability in hardware model checking. In G. Cabodi & S. Singh (Eds.), *Formal Methods in Computer-Aided Design (FMCAD 2012)*, number 978 (pp. 173–181). Cambridge, UK: IEEE.

[152] Wang, J., Kim, S.-K., & Carrington, D. A. (2006). Verifying metamodel coverage of model transformations. In *17th Australian Software Engineering Conference (ASWEC 2006)* (pp. 10 pp.–282). Sydney, Australia: IEEE Computer Society.

[153] Ward, M. P. & Zedan, H. (2013). Provably correct derivation of algorithms using FermaT. *Formal Aspects of Computing*, 26(5), 993–1031.

[154] Warmer, J. B. & Kleppe, A. G. (1998). The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series).

[155] Wies, T. (2009). *Symbolic shape analysis*. Phd thesis, Universität Freiburg.

[156] Wimmel, H. & Wolf, K. (2011). Applying CEGAR to the Petri net state equation. In P. A. Abdulla, M. Rustan, & M. Leino (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 17th International Conference (TACAS 2011)* (pp. 224–238). Saarbrücken, Germany: Springer.

[157] Wimmer, M., Kappel, G., Schönböck, J., Kusel, A., Retschitzegger, W., & Schwinger, W. (2009). TROPIC: a framework for model transformations on petri nets in color. In S. Arora & G. T. Leavens (Eds.), *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (pp. 783–784). Orlando, Florida: ACM.

[158] Wing, J. M. (1990). A specifier's introduction to formal methods. *IEEE Computer*, 23(9), 8,10–22,24.

[159] Wonisch, D. (2010). *Increasing the Preciseness of Shape Analysis for Graph Transformation Systems*. Master thesis, University of Paderborn.

[160] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., & O'Hearn, P. W. (2008). Scalable shape analysis for systems code. In A. Gupta & S. Malik (Eds.), *Computer Aided Verification, 20th International Conference (CAV 2008)* (pp. 385–398). Princeton, New Jersey, USA: Springer.

[161] Zambon, E. (2013). *Abstract Graph Transformation - Theory and Practice*. Phd thesis, University of Twente.

[162] Zambon, E. & Rensink, A. (2012). Graph Subsumption in Abstract State Space Exploration. In A. Wijs, D. Bosnacki, & S. Edelkamp (Eds.), *Proceedings of the 1st Workshop on Graph Inspection and Traversal Engineering, (GRAPHITE 2012)*, volume 99 (pp. 35–49). Tallinn, Estonia.

[163] Zündorf, A. (2009). Model Checking the Leader Election Protocol with Fujaba. In *5th International Workshop on Graph-based Tools (GraBaTs 2009)* (pp. 1–11). Zürich, Switzerland: Springer.