7-2022

Proceedings of the 13th International Workshop on Graph Computation Models (GCM 2022)

Reiko HECKEL

Christopher M. POSKITT
Singapore Management University, cposkitt@smu.edu.sg

GCM 2022

Graph Computation Models

13th International Workshop

Proceedings

Reiko Heckel and Christopher M. Poskitt, Editors

Nantes, France, 6th July 2022

Table of Contents

Session 1: Foundations

Session 2: Applications

Session 3: Keynote & Panel

Preface

This volume contains the proceedings of the Thirteenth International Workshop on Graph Computation Models (GCM 2022)[1], which was held in Nantes, France on 6th July 2022 as part of the STAF federation of conferences.

Graphs are common mathematical structures that are visual and intuitive. They constitute a natural and seamless way for system modelling in science, engineering, and beyond, including computer science, biology, and business process modelling. Graph computation models constitute a class of very high-level models where graphs are first-class citizens. The aim of the International GCM Workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation. It promotes the cross-fertilising exchange of ideas and experiences among senior and young researchers from different communities interested in the foundations, applications, and implementations of graph computation models and related areas.

Previous editions of GCM series were held in Natal, Brazil (2006), Leicester, UK (2008), Enschede, The Netherlands (2010), Bremen, Germany (2012), York, UK (2014), L'Aquila, Italy (2015), Wien, Austria (2016), Marburg, Germany (2017), Toulouse, France (2018), Eindhoven, The Netherlands (2019), online (2020), and online (2021).

These proceedings contain six accepted papers. All submissions were subject to careful refereeing (at least three reviews per paper). The topics of accepted papers range over a wide spectrum, including theoretical aspects of graph transformation, model checking, proofs, as well as applications to program analysis and transformation. Selected revised papers from these proceedings will be published online by Electronic Proceedings in Theoretical Computer Science (EPTCS, `http://www.eptcs.org/`). We would like to thank everyone who contributed to the success of GCM 2022, especially the Programme Committee and the additional reviewers for their valuable contributions to the selection process, as well as the contributing authors without whom this volume would not exist.

6th July 2022 Reiko Heckel and Chris Poskitt

Programme Chairs of GCM 2022

[1] GCM 2022 web site: `https://gcm2022.github.io/`

Programme Committee of GCM 2022

Andrea Corradini, Universita di Pisa, Italy
Frank Drewes, Umeå universitet, Sweden
Rachid Echahed, CNRS and Univ. Grenoble Alpes, France
Annegret Habel, Universität Oldenburg, Germany
Reiko Heckel, University of Leicester, UK (co-Chair)
Berthold Hoffmann, Universität Bremen, Germany
Gabor Karsai, Vanderbilt University, USA
Barbara König, Universität Duisburg-Essen, Germany
Leen Lambers, Brandenburgische Technische Universität Cottbus-Senftenberg, Germany
Mark Minas, Universität der Bundeswehr München, Germany
Fernando Orejas, Universitat Politècnica de Catalunya, Spain
Detlef Plump, University of York, UK
Chris Poskitt, Singapore Management University, Singapore (co-Chair)

Additional Reviewers

Brian Courtehoute, University of York, UK
Okan Özkan, Universität Oldenburg, Germany
Sven Schneider, Universität Potsdam, Germany

Graph-Based Specification and Automated Construction of ILP Problems

Sebastian Ehmes Maximilian Kratz Andy Schürr
Technical University of Darmstadt, Real-Time Systems Lab, Germany

{sebastian.ehmes, maximilian.kratz, andy.schuerr}@es.tu-darmstadt.de

In the Model-Driven Software Engineering (MDSE) community, the combination of techniques operating on graph-based models (e.g., Pattern Matching (PM) and Graph Transformation (GT)) and Integer Linear Programming (ILP) is a common occurrence, since ILP solvers offer a powerful approach to solve linear optimization problems and help to enforce global constraints while delivering optimal solutions. However, designing and specifying complex optimization problems from more abstract problem descriptions can be a challenging task. A designer must be an expert in the specific problem domain as well as the ILP optimization domain to translate the given problem into a valid ILP problem. Typically, domain-specific ILP problem generators are hand-crafted by experts, to avoid specifying a new ILP problem by hand for each new instance of a problem domain. Unfortunately, the task of writing ILP problem generators is an exercise, which has to be repeated for each new scenario, tool, and approach. For this purpose, we introduce the GIPS (**G**raph-Based **I**LP **P**roblem **S**pecification Tool) framework[1] that simplifies the development of ILP problem generators for graph-based optimization problems and a new Domain-Specific Language (DSL) called GIPSL (**G**raph-Based **I**LP **P**roblem **S**pecification **L**anguage) that integrates GT and ILP problems on an abstract level. Our approach uses GIPSL specifications as a starting point to derive ILP problem generators for a specific application domain automatically. First experiments show that the derived ILP problem generators can compete with hand-crafted programs developed by ILP experts.

1 Introduction

In recent years Model-Driven Software Engineering (MDSE) techniques have been successfully used to tackle a plethora of interesting problems, such as model synchronization using Triple Graph Grammars (TGGs) [14], model checking [23], Model-driven Virtual Network Embedding (MdVNE) [20], or the task of solving resource allocation problems in general [16]. All these examples share two commonalities: (1) They operate on (typed and attributed) graphs. (2) They check whether these graphs satisfy certain constraints (i.e., model checking), or they modify said graphs in order to satisfy certain constraints and optimize a given objective function. In the latter case, the graph is typically transformed by a sequence of local graph transformations, described by a set of Graph Transformation (GT) rules. In the many cases, where graph-based models are employed, graph Pattern Matching (PM) can be used to search for local structural as well as local attribute constraint violations or to find valid locations for graph transformations. Sadly, optimization as well as defining and enforcing global constraints, e.g., a constraint on the aggregate value of attribute values of several nodes, is something that is very hard to do using GT rules or PM only, since most of the existing GT tools are (a) not expressive enough or (b) do not have sufficient tool support. On the other hand, using Integer Linear Programming (ILP) only to define and enforce structural constraints on graphs is a cumbersome and inefficient task [18]. Therefore,

[1] GIPS - https://github.com/Echtzeitsysteme/gips

a common approach is to combine GT frameworks (e.g., eMoflon [13], Henshin [3], VIATRA [21]) with ILP, to enable both, checking for global constraint violations and optimizing results for some objective function. MdVNE is an example where the combination of GT and ILP techniques is employed to solve a Virtual Network Embedding (VNE) problem. More precisely, the goal is to find an optimal mapping of virtual elements onto a physical network while upholding certain resource constraints. Besides the fact that this combination works well in specific scenarios, a major issue with this common approach remains, namely the fact that for each new tool tackling some new problem domain, an ILP problem generator has to be hand-crafted anew and, usually, connected to preexisting GT frameworks via hand-written code. This is a tedious and error-prone task and, consequently, the problem that we are tackling in our approach presented in this paper.

GIPS (**G**raph-Based **I**LP **P**roblem **S**pecification Tool), the tool proposed in this paper, enables users to generate domain-specific ILP problem generators given a metamodel (in form of a UML class diagram) and a problem specification in our new Domain-Specific Language (DSL) GIPSL (**G**raph-Based **I**LP **P**roblem **S**pecification **L**anguage), containing a set of GT rules, a set of constraints, and an optional objective function. This enables the usage of graph pattern matches within ILP constraints. Therefore, we directly encode the matches into binary ILP variables and automatically generate the necessary glue code, which connects off-the-shelf ILP solvers (e.g., Gurobi[2]) with a GT engine.

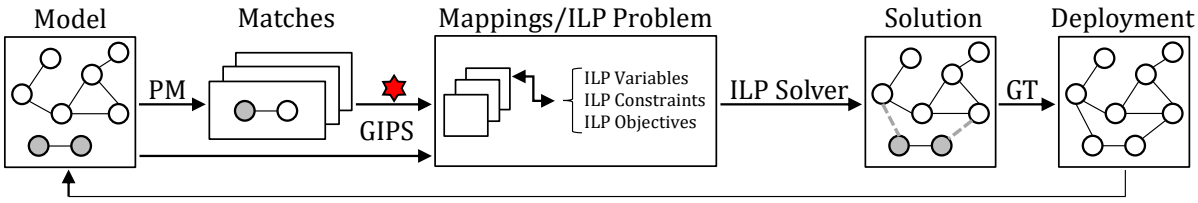


Figure 1: Workflow diagram of the GIPS framework (red star = generated during build time).

To give a brief overview, concept Figure 1 shows the workflow of our tool. The input consists of two parts: An instance graph and a set of pattern matches are obtained from the same instance graph as a result of pattern matching. Both of these inputs are used by our tool GIPS to construct an ILP problem according to user specifications given in GIPSL. After solving the problem, changes can be applied to the instance graph by applying GT rules, which, in turn, leads to a modified instance graph as output. If necessary, this output can be reused as an input for another GIPS run. To achieve this goal, our tool is tightly integrated into the state-of-the-art GT interpreter eMoflon::IBeX-GT[3], which does not only provide an expressive DSL to specify patterns and rules but also the massively parallelized incremental graph pattern matcher HiPE[4]. In addition, integration into the Eclipse development environment[5] ensures a high level of automation and developer support (e.g., code completion, syntax highlighting, semantic checking, type safety, etc.). This heavily model-driven type of software development leads to a potential reduction of the prototype engineering time as well as the maintenance effort.

In this paper we evaluate our approach w.r.t. the following two research questions: **(RQ1)** How does our approach compare itself to hand-crafted approaches with respect to problem solving runtime performance? **(RQ2)** How much effort (e.g., lines of code) can really be saved by using our approach to generate an ILP generator compared to implementing one by hand? In Section 2, we discuss publications

[2]Gurobi Optimizer - `https://www.gurobi.com/products/gurobi-optimizer/`
[3]eMoflon::IBeX-GT - `https://emoflon.org/#emoflonIbex`
[4]HiPE - `https://github.com/HiPE-DevOps/HiPE-Updatesite`
[5]Eclipse development environment - `https://www.eclipse.org/`

that are related to our approach, i.e., other works that are combining ILP and GT as well as works that are concerned with the specification and generation of ILP problems in general. In Section 3, we introduce an illustrative motivating example using the VNE problem, followed by background knowledge concerning ILP in Section 4.1 and GT in Section 4.2. Section 5 explains the overall concept of the approach presented in this paper. In Section 6, we present our newly developed DSL GIPSL together with the tool GIPS, which we use to evaluate our approach in Section 7 regarding the central research questions. Finally, Section 8 sums up our contribution and gives an overview of possible future enhancements.

2 Related Work

Our approach and our motivating example touch on several active fields of research within the MDSE community, such as (rule-based) Model Transformation (MT), graph mapping, resource allocation, etc., all in combination with ILP to various degrees. For this reason, we have identified a set of related works mostly centered around (a) model transformation, model synchronization, or resource allocation, with or without the help of ILP solving and (b) ILP problem construction from specifications with various degrees of abstraction. Regarding (a), MT and resource allocation, we have deemed the following works to be related to our approach:

Pohlmann et al. [16] present an approach to specify and solve resource allocations problems in general, using an example of the automotive system's domain. They present a DSL called ASL, with which a resource allocation problem can be specified in a concise manner. From this specification an ILP problem is constructed and, at runtime, the solution (if one does exist) is constructed automatically. Our idea is somewhat related to that since we aim to automatically generate ILP problems from an abstract specification and we do both provide graph patterns as a means to encode structural constraints into ILP variables. As a major difference, our approach is much more general in nature, since we do not aim to generate ILP problems that lead to solutions for one problem domain only, which is reflected in our own DSL. But we will show that one can solve such a class of problems with our approach using our motivating example. In addition to that, we allow for user-defined objective functions that are not supported by Pohlmann et al.

In their works, [9, 10] Götz et al. have developed an approach that, similar to our work, creates an ILP problem from a specification in a model-based and mostly automated fashion. In the presented use-case their approach is used to find an optimal placement of software modules on hardware resources, which is related to our motivating MdVNE example. As a major difference, Götz et al. do not use graph patterns and in extension, graph pattern matching to check structural constraints of a given model locally and, thus, cannot generate ILP variables automatically, which correspond to matches satisfying these constraints. Instead, they encode all possible binary combinations of model elements into ILP variables and enforce structural constraints by transforming them into regular algebraic constraints and adding those constraints to the ILP problem. This approach, compared to ours, has two downsides: Firstly, it increases the amount of ILP variables and, therefore, the search space substantially and, secondly, it increases the amount of necessary ILP constraints, increasing the problem size and usage of working memory.

Tomaszek et al. presented a model-driven approach to the VNE problem based on so-called Triple Graph Grammars [19], which also combines GT and ILP technology and also has a focus on resource allocation problems. Similar to our approach, but not to the same extent, it provides a somewhat automated ILP problem construction approach. TGG rules are automatically translated into binary ILP variables and ILP constraints can partly be extracted from the TGG rules automatically, while more complex con-

straints, e.g., constraints on aggregated attribute values must be implemented by hand. In contrast to our approach, this TGG-based approach does not allow for the specification of an objective (optimization) function. Instead, the overall optimization goal is fixed and aims to map a maximum number of virtual (communication) network elements onto a (physical communication) substrate network. To bypass the optimization goal limitations of TGGs, Tomaszek et al. modified the approach towards the one presented in [18], which uses GT rules and a hand-crafted ILP problem generator to specify a solution strategy for VNE problems with a specific optimization function. The drawback of this approach is the fact that the generation of ILP problems cannot be automated to the same degree as in the previous publication, is highly domain-specific, and is prone to manual coding errors.

Fleck et al. [6] tackle the rule orchestration problem, where they try to find an ordered set of rule-match pairs that modify a given model in such a way that it optimizes a given fitness function. They present a model-driven approach that also constructs an optimization problem automatically from a given specification, similar to our approach. In contrast, this will not result in a linear problem intended for an ILP solver but a multi-objective optimization problem intended to be solved by a search-based optimization algorithm (e.g., a genetic algorithm). As a downside, genetic algorithms do not guarantee optimal solutions and often have serious problems identifying promising sequences of rule applications in the huge search space of all possible sequences. In contrast, our approach can guarantee optimal solutions by using ILP solvers, given enough time.

Weidmann et al. [2] presented a hybrid approach to fault-tolerant consistency management in MDSE that combines TGGs and various optimization techniques (which include ILP). In a case study, Weidmann et al. used their approach to automate scheduling processes of assigning test tasks to test developers. For this purpose, they designed a DSL to allow head test engineers to specify test schedule problem instances. Unfortunately, the presented DSL is highly specialized for the task of scheduling test jobs for test engineers. In contrast, the presented approach in this paper can be used to specify and solve a variety of graph mapping problems that also include (generic) scheduling problems, as long as these are expressible as an ILP problem.

Regarding (b), ILP problem specification and automated construction, we have identified the following well-established approaches:

General Algebraic Modeling System (GAMS) [5] and A Mathematical Programming Language (AMPL) [8]. Both are abstract modeling languages and systems for the specification, maintenance, and solving of mathematical optimization problems. They were created to model and solve a variety of mathematical optimization problems like ILP and Mixed-Integer Linear Programming (MILP), but they also address non-linear problem domains. Therefore, they provide their own syntax to specify a problem that is close to the abstract mathematical problem description known from the literature. In contrast to that, the DSL presented in this paper is more object-oriented and combines aspects from the GT world with ILP, with the intention of making our DSL more accessible and intuitive to use. On the downside, most of the concepts of this paper are currently limited to ILP problems and, thus, cannot be used to solve non-linear problems, which GAMS and AMPL can.

Since most of the algebraic modeling languages (like GAMS and AMPL) were commercial products back in the 1980s, the authors of Zuse Institute Mathematical Programming Language (ZIMPL) [12] created their own mathematical programming language that was open-sourced and, hence, can easily be used by, e.g., students. The ZIMPL language is used to translate the mathematical model of a given problem into either a linear or a non-linear (M)ILP or mathematical programming system formulation, respectively. The output of this translation can be used to start an open-source or commercial solver to find the solution to the problem. As with GAMS and AMPL, the ZIMPL syntax is tightly related to the algebraic mathematical problem description, which GIPSL is not. Similarly, other algebraic modeling

languages that we have found mostly increase the abstraction level of the algebraic optimization problem description to varying degrees. In contrast to that, our work allows for the use of a tightly integrated specification language to generate ILP problems automatically from GT rules, mapping constraints, and objective functions for a given graph-based optimization model.

3 Motivating Example

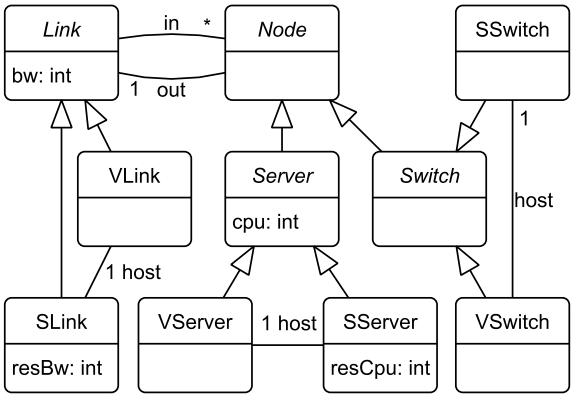


Figure 2: Simplified MdVNE metamodel.

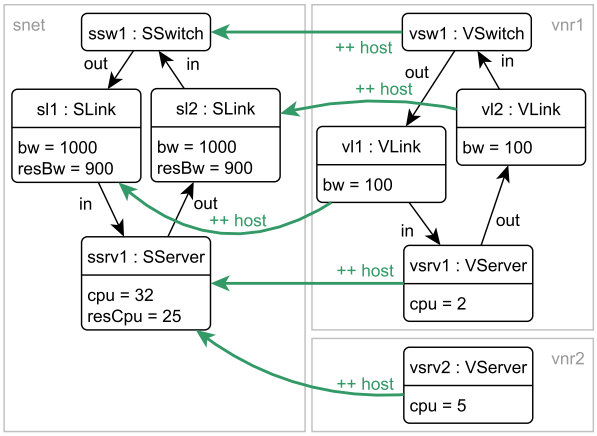


Figure 3: Example MdVNE model instance.

We present MdVNE [20] as a motivating example to explain and to evaluate our newly developed approach. On the one hand, this presents us with the unique opportunity to directly compare the ILP generator, which was automatically generated by our approach, with the hand-crafted ILP generator of Tomaszek's et al. work. On the other hand, MdVNE is a relevant problem that is subject to ongoing research and helps to illustrate the capabilities of our approach nicely.

MdVNE is a model-driven approach to VNE, which tackles the problem of mapping virtual network topologies onto a substrate network topology within the data center context. In MdVNE, both, the virtual network and the substrate network elements are modeled as nodes in a typed and attributed graph, corresponding to the metamodel[6] in Figure 2. Accordingly, references between these nodes are modeled as typed edges of the same graph. The central challenge of the VNE problem and, therefore, the MdVNE problem, is to find a valid mapping of virtual elements (of one or more Virtual Network Requests (VNRs)) onto substrate elements that does not exceed available substrate resources and is optimal according to a given objective function. This can be done individually for each VNR or for multiple VNRs at once (batch approach). For example, a suitable objective function could be the minimization of the aggregated communication costs of all virtual links [20]. Unfortunately, VNE is a resource allocation problem, which is known to be $\mathcal{NP}$-hard [1]. Therefore, the size of the search space has a significant impact on the runtime of algorithms that solve VNE problems. To reduce the search space, the methodology of MdVNE was introduced by Tomaszek et al. [20, 18]. The search space reduction is achieved by using Incremental Graph Pattern Matching (IGPM), which preselects only tuples (matches) of the substrate and virtual nodes that satisfy certain structural and attribute constraints according to a set of graph patterns. Figure 3 shows an example mapping that illustrates this, using a small graph whose

[6]The original metamodel in Tomaszek's et al. work [20] is quite large. Due to space constraints for this submission, we present a reduced version of the MdVNE metamodel.

components correspond to the metamodel of Figure 2. In Figure 3, the substrate link `sl2` with a total bandwidth of 1000 Mbit/s has an available bandwidth of 900 Mbit/s because the virtual link `vl2` with a bandwidth of 100 Mbit/s is mapped, by virtue of creating the host edge between `vl2` and `sl2`.

4 Preliminaries

4.1 Integer Linear Programming

The goal of an Integer Linear Programming (ILP) problem is the minimization (maximization) of an objective function $F : \mathbb{Z}^n \to \mathbb{R}$ by finding an integer target vector $\vec{x} \in \mathbb{Z}^n$ while adhering to various constraints $f_j(\vec{x}) \leq 0$ $(j = 1, \ldots, m)$ [4, 15]. Its canonical form is given in Eq. (1), where $\vec{b}, \vec{c} \in \mathbb{R}^n$ are vectors, $A \in \mathbb{Z}^{n \times m}$ is a coefficient matrix with integer entries only, and $\vec{x} \in \mathbb{Z}^n$ is the solution vector.

$$\text{minimize } \vec{c}^T \vec{x} \text{ subject to } A\vec{x} \leq \vec{b}, \ \vec{x} \geq 0, \text{ and } \vec{x} \in \mathbb{Z}^n \tag{1}$$

For problems with non-convex objective functions, there can be more than one locally optimal vector. The minimization of the target function can be converted to a maximization (or vice versa) if it is multiplied by the factor -1. One special case of an integer linear problem is the so-called bivalent linear problem, in which all entries of the vector $\vec{x}$ have to be either 0 or 1 [4].

The mapping of virtual networks can be written as a bivalent linear problem as proposed by Tomaszek et al. [18]. Therefore, we may describe the VNE problem as a set of unknown integer variables, linear constraints, and a linear target function [20] (see Section 3). The VNE problem can then be solved as an integer linear problem using highly sophisticated solvers like Gurobi[7] or CPLEX[8].

4.2 Graph Transformation

As indicated by the motivating example, models in our approach are represented by typed and attributed graphs with objects corresponding to typed nodes and references between objects corresponding to typed edges. Due to this circumstance, we make use of GTs, which provides the means to express model transformations on graphs in a declarative and rule-based fashion. A GT rule is composed of a so-called Left-Hand Side (LHS) and an Right-Hand Side (RHS). The LHS represents a precondition, which has to be met in the instance graph (e.g., a network model) before a GT rule can be applied. The RHS, on the other hand, defines a postcondition, which must be satisfied after the rule has been applied. Consequently, graph transformation heavily depends on graph pattern matching, which solves the problem of finding some subgraph in an instance graph that matches a graph pattern, i.e., the LHS of a rule. Hence, such a subgraph is called a match. Graph pattern matching tools will find all match occurrences in a graph similar to a predefined graph pattern, where subgraphs of a model that fulfill a graph pattern are called (graph pattern) matches and consist of a set of instance graph nodes successfully mapped to graph pattern nodes. Figure 4 shows two examples GT rules:

First, the LHS of the `server2server` rule is a pattern that will instruct a pattern matcher to find matches that contain a `SServer` and a `VServer`. In addition to that, the attribute constraint demands that a match is only valid if the `SServer` has enough CPU resources left to fully satisfy the CPU requirements of the `VServer`. If the `server2server` rule is applied, the RHS demands the creation of a new `host`

[7] Gurobi Optimizer - `https://www.gurobi.com/products/gurobi-optimizer/`

[8] ILOG CPLEX Optimization Studio - `https://www.ibm.com/products/ilog-cplex-optimization-studio`

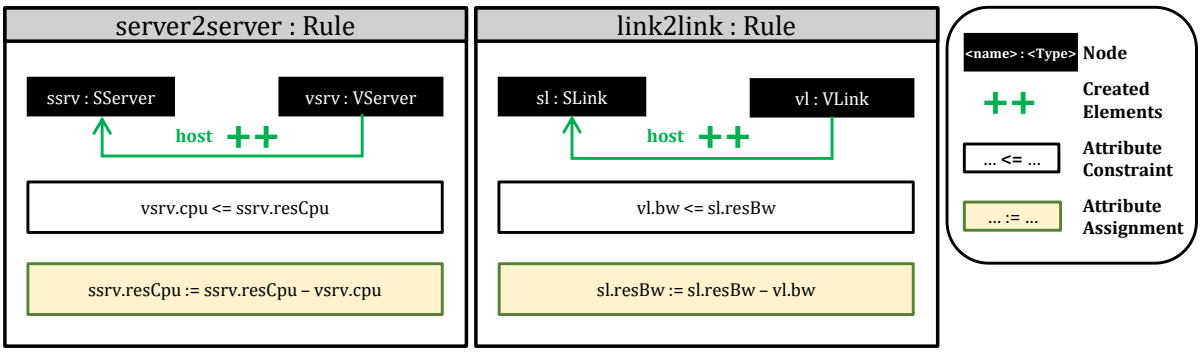


Figure 4: Rule for matching and mapping a virtual server to a substrate server (`server2server`) and rule for matching and mapping a virtual link to a substrate link (`link2link`).

edge and the reduction of the residual `SServer` CPU resource by the amount of the `VServer` CPU requirements, which leads to a mapping of a virtual server onto a substrate host server.

Second, the LHS of the `link2link` rule is a pattern to find matches that contain a `SLink` and a `VLink`. As with the resource constraint of the server rule, the link rule needs to ensure that a match is only found if the residual bandwidth attribute of `sl` is larger or equal to the bandwidth of `vl`. Similar to `server2server`, the `link2link` rule has a RHS that demands the creation of a `host` edge and the reduction of the residual attribute `sl.resBw` by the amount of bandwidth of the virtual link `vl`. Hence, applying the `link2link` rule to a match will lead to a mapping of a virtual link onto a substrate link, given a valid solution of the ILP problem generated by our approach (see Section 5).

On a side note, despite the example GT rules only showing the creation of new edges, the GT rules in our approach support the creation and deletion of arbitrary graph elements as well[9].

Pattern matching approaches can be divided into two categories, namely batch and incremental PM. In our work, we rely on the latter approach, since IGPM keeps track of individual model changes and, therefore, allows the sets of newly found or recently disappeared matches to be updated incrementally. This has the nice advantage that PM runtime after the initial model exploration does not increase proportionally to the total size of the model but is proportional to the magnitude of a model change. Recollecting all previously gathered knowledge about a network model every time a virtual network is mapped would be grossly inefficient and would lead to potentially increasing ILP problem construction times as a consequence. Therefore, IGPM-based tools seem to be the logical choice for our approach.

Most IGPM approaches implement some derivative of Forgy's original Rete algorithm [7], which is a widely known approach to incremental pattern matching. Viatra [21], Democles [22], and the recently developed HiPE[10], are examples of Rete inspired IGPM tools, which all are also employed as pattern matchers within eMoflon, our GT tool of choice. While Viatra, Democles, and other well-known IGPM tools are mostly single-threaded, HiPE was developed with the goal to perform graph pattern matching massively in parallel by reinterpreting Forgy's Rete-approach anew using an actor system approach [11] based on the Akka[11] framework.

[9] eMoflon (`https://emoflon.org/`), our GT tool of choice, is configured to use a single-pushout GT approach.
[10] HiPE - `https://github.com/HiPE-DevOps/HiPE-Updatesite`
[11] Akka - `https://www.akka.io`

5 From Mappings to ILP Variables

The motivating example in Section 3 is a typical example of a problem, where graphs (virtual networks) have to be mapped onto other graphs (substrate networks), while satisfying a set of structural constraints as well as a set of attribute constraints and possibly optimizing a given cost function at the same time. For this reason, generalizing the idea of Tomaszek et al. [20], which combines GT techniques and ILP to solve the VNE problem in a model-driven fashion, seems to be a promising approach to solve a whole class of problems. Therefore, our concept revolves around the combination of IGPM, GT rules, and ILP, with the goal to enable a concise and intuitive method to (1) specify mapping problems as optimization problems, using ILP, (2) generate an ILP generator automatically from the specification, and (3) solve the corresponding problem.

We define subgraphs (e.g., virtual networks) that have to be mapped onto other subgraphs (e.g., substrate networks) as mappings. These mappings are essentially binary ILP variables in $\vec{x}$ (see Section 4.1), which correspond to matches of a pattern such as the LHS of a rule, e.g., `server2server` in Figure 4. Therefore, any of these corresponding matches that are found in a given model (e.g., Figure 5) are encoded as binary ILP variables onto which we can place additional ILP constraints. By using an ILP solver, we can obtain solutions that consist of a set of binary variables, where non-zero variables represent valid mappings, which are optimal according to an additionally given objective function.

This integration has two distinct advantages when compared to the use of either graph PM or ILP by themselves. On the one hand, as stated in Section 4.2, graph patterns are perfectly suited to describe graph structures within a local scope, but they cannot ensure constraints on a global scale, e.g., constraints over aggregated attribute values, which is a necessary precondition in order to solve a problem similar to MdVNE. This shortcoming can be alleviated through the use of ILP, which enables the definition of constraints placed upon variables that are enforced on a global scale. On the other hand, specifying structural constraints between tuples of graph nodes, represented by variables, as ILP constraints is possible in principle, albeit tedious, but it enlarges the solution search space needlessly. Using patterns and in extension massively parallelized pattern matching to efficiently find tuples of graph nodes adhering to certain structural constraints enables the reduction of the ILP search space and offers a more concise means of specification.

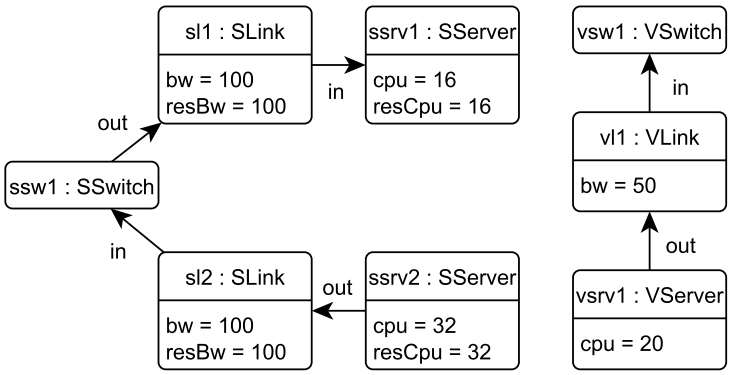


Figure 5: MdVNE model instance used as an example for the GIPS concept.

Figure 5 shows an example instance graph that consists of a substrate network and one virtual network. By using GIPS with the example rule `link2link` of Figure 4, the pattern matcher will find two valid matches each containing `vl1` since both substrate links have enough available resources to host

vl1. Therefore, the framework will generate two binary ILP variables to define the mapping of vl1 onto sl1 and sl2, respectively, which is shown in Eq. (2)[12].

$$x_{\text{vl1-sl1,link2link}},\ x_{\text{vl1-sl2,link2link}} \in \{0,1\} \tag{2}$$

Due to its limited local knowledge, the pattern matcher cannot verify that the available resources of the substrate link can host all of the found matches of different virtual elements at the same time. To compensate for this limitation, the resource constraints must each be formulated as ILP constraints. The ILP solver must ensure that it chooses mappings according to all constraints, i.e., it has to verify that no constraints are violated for a found solution. In the example of Figure 5, GIPS must generate a constraint for each substrate link, which ensures that the residual resources are always able to host the virtual link of the selected matches. This is shown in Eq. (3).

$$x_{\text{vl1-sl1,link2link}} \cdot \text{vl1}_{\text{bw}} \leq \text{sl1}_{\text{resBw}} \quad \wedge \quad x_{\text{vl1-sl2,link2link}} \cdot \text{vl1}_{\text{bw}} \leq \text{sl2}_{\text{resBw}} \tag{3}$$

Another important aspect of the generated ILP problem over a purely PM-based approach is the fact that each virtual element must be mapped exactly once. Eq. (2) shows that there are two possible ways to map vl1 in the example. As a result, an ILP constraint must be generated to ensure the unique mapping of vl1, which is shown in Eq. (4).

$$x_{\text{vl1-sl1,link2link}} + x_{\text{vl1-sl2,link2link}} = 1 \tag{4}$$

Additionally, the valid solution to the ILP problem must guarantee that the source and the target node of the virtual link are mapped onto the source and target node of the candidate substrate link. This requirement is necessary to achieve a contiguous network mapping. Simplified, the specification must lead to an ILP constraint of the type shown in Eq. (5), where $x_{i,\text{server2server}}, x_{j,\text{server2server}}, x_{k,\text{link2link}}$ are binary variables that correspond to mappings.

$$x_{i,\text{server2server}} + x_{j,\text{server2server}} \geq 2x_{k,\text{link2link}} \tag{5}$$

Using this expression, it can be ensured that the binary variable $x_{k,\text{link2link}}$ (the mapping of the virtual link) can only be true, if $x_{i,\text{server2server}}$ (the mapping of the source node) and $x_{j,\text{server2server}}$ (the mapping of the target node) are also true. A simplified specification of such a link constraint in GIPSL that is used to derive the ILP formula of Eq. (5) presented above can be found in Appendix A.1.

Finally, as a side effect of our graph patterns being LHSs of GT rules, a mapping can be implemented by applying a rule (e.g., creating a host edge) to a suitable match that corresponds to a non-zero ILP variable.

6 The GIPS Framework

As mentioned in the introduction, the connection of GT frameworks and ILP solvers is common practice in the MDSE community. At the same time this process, despite all its advantages, is a tedious and error--prone challenge. For example, in Tomaszek's et al. work [20] given a set of GT specifications, a Java-based ILP generator has to be written that transforms each match of a graph pattern into an ILP variable and creates corresponding constraints and objectives. Consequently, with increasing project complexity,

[12] We use the notation $x_{a,b}$ to refer to ILP variables corresponding to certain mappings, where subscript a denotes a match of the rule defined by subscript b.

the manual effort of specifying graph patterns as well as implementing some form of ILP generator becomes increasingly complex and time-consuming. To tackle this challenge, we developed the new approach as presented in Section 5 and implemented it as a Java-based framework, composed of two key components, namely: GIPS, which combines ILP solver as well as GT engine runtime components and GIPSL, which is our newly developed DSL that integrates the specification of graph patterns, GT rules, and ILP constraints as well as ILP objectives.

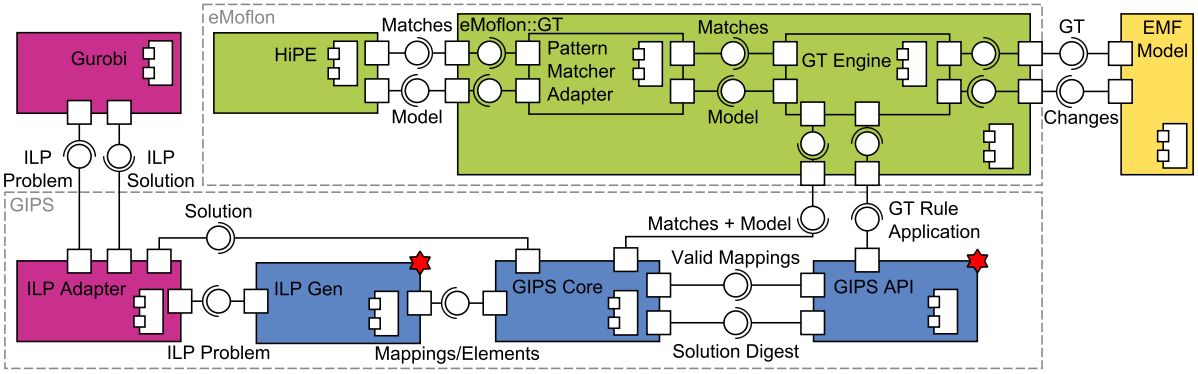


Figure 6: Component diagram of the GIPS framework (red star = generated during build time).

GIPS GIPS itself is composed of four components, as shown in Figure 6. During build time, the ILP generator component `ILP Gen` and the user API `GIPS API` is generated according to GIPSL specifications. At runtime, `GIPS Core` interfaces with eMoflon to request matches as well as access to the underlying model. The generated ILP generator component `ILP Gen` will construct a new ILP problem from the model and a given set of matches, while `ILP Adapter` interfaces with the ILP solver (here: `Gurobi`) to supply the solver with the ILP problem and, in turn, receive a solution if one exists. Finally, using the generated API `GIPS API`, GT rules can be applied to matches that correspond to valid mappings (i.e., non-zero binary variables).

Listing 1: GIPSL as simplified EBNF.

```
Specification = {Rule | Mapping | Constraint | Objective}, GlobalObjective;
Mapping = Rule;
Constraint = (Mapping | Pattern | Type), BooleanExpression;
Objective = (Mapping | Pattern | Type), ArithmeticExpression;
GlobalObjective = ('Min' | 'Max'), ArithmeticExpression;
BooleanExpression = BooleanExpression, BinaryBoolOperator,
    BooleanExpression | UnaryBoolOperator, BooleanExpression |
    RelationalExpression | AttributeExpression | 'True' | 'False';
RelationalExpression = ArithmeticExpression, RelationalOperator,
    ArithmeticExpression;
ArithmeticExpression = ArithmeticExpression, BinaryOperator,
    ArithmeticExpression | UnaryOperator, ArithmeticExpression |
    AttributeExpression | Number;
BinaryBoolOperator = '&' | '|';
UnaryBoolOperator = '!';
RelationalOperator = '>' | '>=' | '==' | '<=' | '<' | '!=';
BinaryOperator = '+' | '-' | '*' | '/';
UnaryOperator = 'sin' | 'cos' | 'sqrt';
```

GIPSL GIPSL was developed with the Xtext[13] framework, which allows for the specification of DSLs using a specification language based on the Extended Backus-Naur Form (EBNF). As a result, we can provide several common IDE support features for our language, such as code completion, syntax highlighting, as well as syntactic and semantic validation. Listing 1 shows a reduced version[14] of our DSL in EBNF, which we will explain using examples of the MdVNE scenario (see Section 3), beginning with `Mapping`. As explained in Section 5, mappings are the central element in our approach, since this allows us to map whole subgraphs, encoded as graph pattern matches, by encoding the matches into binary ILP variables. In Listing 2, we define such a mapping with the freely configurable name `srv2srv` for the rule `server2server`. Consequently, this will lead to the creation of a binary ILP variable $x_{i,\text{server2server}}$ for the i^{th} match of the LHS of `server2server` that is found in the graph.

Listing 2: GIPSL mapping example for the server to server mapping.

```
1 mapping srv2srv with server2server;
```

Constraints place additional requirements upon mappings that can go beyond simple structural or attribute requirements defined in their corresponding GT rules. Using the `constraint` keyword a constraint for mappings can be specified, which will be translated into ILP constraint(s) at runtime. The `constraint` keyword is always followed by the context specification after the "`->`" operator. This will have two consequences:

First, an ILP constraint will be created for each element within the given context. Currently there are 3 possible context types: `class`, `pattern` and `mapping`. In the case of the `class` keyword, a constraint will be created for each element of the model with the type specified after the keyword. In the case of both, the `pattern` and `mapping` keyword, a constraint will be created for each match of the corresponding pattern (e.g., LHS of a rule). For example, in Listing 3, a constraint within the context of `SubstrateServer` class is defined, as indicated by the statement in line 1 after the "`->`" operator, which will lead to the creation of an ILP constraint for each element of type `SubstrateServer` present in a given model.

Second, the `self` operator, which can be used within the constraint specification, offers access to attributes in case of the `class` context, or grants access to match nodes and their attributes in case of both, the `pattern` and `mapping` contexts.

Listing 3: GIPSL constraint example for the server to server mapping.

```
1 constraint -> class::SubstrateServer {
2     mappings.srv2srv->filter(m | m.nodes().ssrv == self)->sum(m |
          m.nodes().vsrv.cpu) <= self.resCpu
3 }
```

In our approach, constraints as a whole are inspired by OCL invariants, where constraints can be placed upon models by means of boolean expressions. A constraint must hold for every non-zero binary variable, for a found solution to be valid. Line 2 in Listing 3 shows an example of such a boolean OCL-like expression. Here, we demand that the sum of CPU requirements of all virtual servers that should be mapped to a single substrate server must not exceed its available CPU capacity. To achieve this, we fetch all `srv2srv` mappings, by accessing the set of all `mappings` using the dot-operator. These mappings are filtered using a `filter` expression, that only allows mappings corresponding to matches, where the `ssrv` node is equal to the current `SubstrateServer` node, as indicated by the `self` keyword. The filtered set

[13] Xtext - `https://www.eclipse.org/Xtext/`

[14] The full Xtext version is available in the appendix Listing 7.

of mappings is then used in the `sum` expression. It sums up the virtual server requirement values of the `cpu` attribute of all virtual server nodes that are contained within the corresponding `server2server` matches. The resulting sum must be smaller or equal to (`<=`) the residual CPU resources (`resCpu`) of the current `SubstrateServer` node, as indicated by the `self` keyword. Therefore, in this example, for each substrate server a constraint is generated, in which the following condition must hold true: The sum of all binary ILP variables, which correspond to `server2server` matches that contain the given substrate server, multiplied with the `vsrv`'s `cpu` requirement must not exceed the value of the `resCpu` attribute.

Objectives associate mappings, model elements, or attribute values with certain costs. Defining an objective using the keyword `objective` will lead to the creation of a linear cost function, which can be used in the global cost function to evaluate the score of a valid solution. The name of the objective is followed by the context specification after the "`->`" operator. Similar to constraints, the context has two effects:

First, a linear cost function is created for each element within the given context. Similar to constraints, contexts may be model elements of a certain type, as indicated by the `class` keyword, or whole matches, as indicated by the keywords `pattern` or `mapping`. In the example's case, the `srv2srv` mapping is selected as context, which will lead to the creation of a linear cost function for each match of `server2server` and whose value can only be non-zero, if the corresponding binary variable is non-zero.

Second, the `self` operator, which can be used within the objective function specification, grants access to attributes in case of the `class` context, or grants access to match nodes and their attributes in case of both, the `pattern` and `mapping` contexts. The example objective will associate the cost of each `srv2srv` mapping with the substrate server's residual CPU resource value divided by its maximum CPU resource value. Such a cost function will nudge virtual server mappings towards non-empty substrate servers since its value gets smaller the more packed a substrate server is.

Listing 4: GIPSL objective example for the server to server mapping.

```
1  objective srvObj -> mapping::srv2srv {
2      self.nodes().ssrv.resCpu / self.nodes().ssrv.cpu
3  }
```

The final language construct of GIPSL is the `global objective`, which has two purposes, the definition of an optimization goal and the linear combination of all other objective functions into one linear global objective function. To define the optimization goal, either the keyword `min` or the keyword `max` can be used. The former leads to the minimization of the global objective function by the ILP solver, the latter leads to the maximization. Within curly brackets, all previously defined objective functions can be linearly combined by addition or subtraction and weighted through multiplication with constant factors. In the example, the global objective consists of the `srvObj` objective. This will create an ILP objective function, where all linear functions created by virtue of the `srvObj` are summed up and minimized, according to the `min` keyword.

Listing 5: GIPSL global objective example for the MdVNE example.

```
1  global objective : min {
2      srvObj
3  }
```

7 Evaluation

Our model-driven approach to automatically generate ILP problems aims to offer a more productive and less error-prone development experience while delivering runtime performance on par with state-of-the-art hand-crafted solutions. Hence, we evaluated our approach w.r.t. the following two central research questions:

(RQ1) How does our approach compare itself to hand-crafted approaches with respect to problem solving runtime performance?

(RQ2) How much effort (e.g., lines of code) can really be saved by using our approach to generate an ILP generator compared to implementing one by hand?

To answer both research questions, we compare our GIPS-driven VNE algorithm implementation with the hand-crafted solution by Tomaszek et al. [20]. In the evaluation setup, the data center (substrate network) is a two-tier network with two core switches that are connected to all rack switches via 10 Gbit/s links. Each rack consists of ten servers with 32 CPU cores, 512 GiB[15] memory, and 1 TiB[16] storage resources each, whereby all servers are equipped in the same way. All servers are connected to the corresponding rack-switches via 1 Gbit/s links. All possible paths of the substrate network are generated before running the performance analysis. In total, the simulated (substrate) data center network consists of eight racks with ten servers each. The virtual networks use the star topology with one central switch and between 2 and 10 virtual servers. Virtual resources are sampled from distributions provided by the Bitbrains publication [17]. Each virtual server consists of 1 to 32 CPU cores, the amount of memory is sampled from the interval 1 GiB up to 511 GiB and the individual bandwidth of every link between the server and the switch has a value between 100 Mbit/s and 1 Gbit/s. As the Bitbrains distribution does not state explicit values for the storage demand per server, an equal distribution between 50 GiB and 300 GiB is used. For every experiment, both VNE implementations have to map all virtual networks from the same set of 40 VNRs one after another onto the substrate network[17]. Every plot in this section shows measurements that are the calculated mean of five results, which were performed using the same random seed. The error bars of every plot show the standard deviation. All experiments were run on a workstation equipped with an AMD Ryzen Threadripper 2990WX with 32 CPU cores and 96 GB of memory. The used operating system is Ubuntu 20.04.4 LTS, the used Java environment is OpenJDK Temurin (build 17.0.2+8), and the ILP solver is Gurobi Optimizer.

Regarding the runtime experiment, both algorithms were able to successfully map the whole set of virtual networks into the physical network, while achieving an approximately equal objective function value. Figure 7 shows the runtime of the mapping process of each virtual network for both, the hand-crafted as well as our generated VNE algorithm. Overall, the two plots show similar behavior. In contrast to the manually derived implementation, the plot of the GIPS-based algorithm includes two relatively high spikes up to around 100 s runtime. Even so, the mean runtime of each mapping is lower when comparing it to the hand-crafted implementation. In fact, the total runtime of our framework is ≈25 % lower (613.01 s, $\sigma = 25.56$ s vs 815.78 s, $\sigma = 7.2$ s) compared to the implementation that was hand-written by experts. In this scenario, GIPS is able to automatically generate an ILP problem generator that is able to keep up or even outperform the hand-written implementation, which provides a tentative but clearly positive answer to **RQ1**. These results also show that tight integration of GT and ILP can lead to a performance improvement compared to a loosely coupled hand-crafted approach.

[15] $1\,\text{GiB} = 2^{30}\,\text{B}$

[16] $1\,\text{TiB} = 2^{40}\,\text{B}$

[17] Both implementations are able to map individual VNRs as well as whole sets of VNRs at the same time.

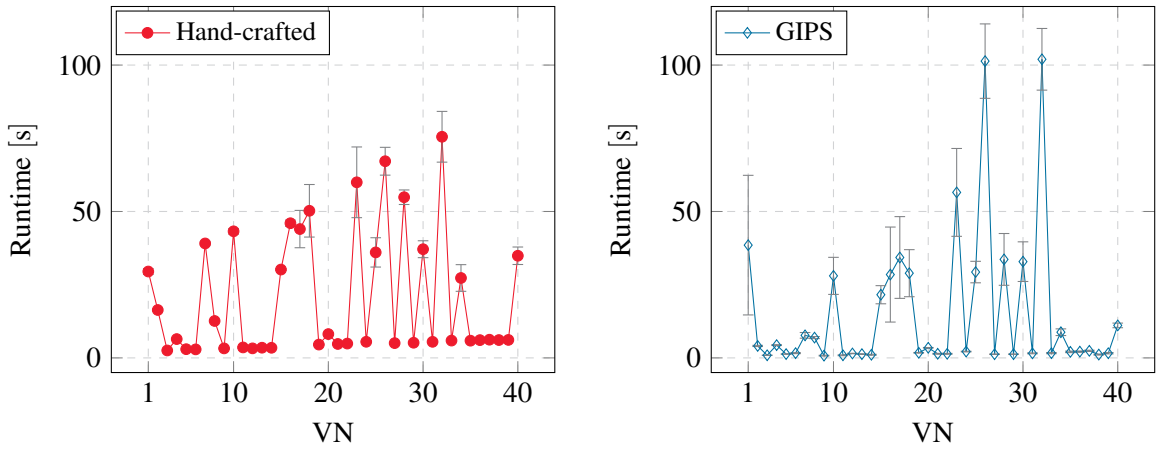


Figure 7: *Hand-crafted* implementation vs *GIPS*-based implementation: Algorithm runtime for the evaluation scenario per virtual network mapping.

Regarding **RQ2**, we used the metrics Lines Of Code (LOC) and Number Of Characters (NOC) as measures of the amount of effort that has to be invested to combine a GT framework with an ILP solver. Therefore, we counted all non-zero LOC and NOC of the GIPSL-based specification as well as the needed Java code to initialize and run the API to solve the MdVNE problem. For the hand-written MdVNE algorithm implementation from [20], we counted all non-zero LOC and NOC that are part of the ILP problem generator. This also includes the code needed to translate the matches of the pattern matcher into an ILP problem and to interpret the ILP solver's solution. Tomaszek's et al. implementation contains a large number of additional features compared to the MdVNE implementation in GIPS, which complicates the isolation and the line counting of the relevant code parts. Therefore, we only counted the LOC and NOC that are necessary to run the presented evaluation scenario. We found that the hand-written implementation consists of over 2000 LOC and 91000 NOC in total. In contrast to that, the problem specification in GIPSL consists of 56 LOC or 3884 NOC together with 29 LOC or 1251 NOC used to initialize and run the API. Comparing the abstract GIPSL specification with the implementation written by an expert leads to a LOC and NOC reduction of ≈ 95 % and ≈ 94 %, respectively. The example in this evaluation showed that the amount of source code/specification language can be reduced radically if the graph-based optimization problem is non-trivial. Consequently, these results can provide a positive answer to **RQ2**, as implementation effort can be clearly reduced by our approach.

Threats to Validity

The evaluation result of our prototype's implementation gives a first impression of the capabilities of the presented approach. However, the evaluation was only performed using the MdVNE scenario of Tomaszek et al. It would be interesting to use multiple examples from different problem domains (e.g., assigning test tasks to test developers [2]), which could be examined to validate that GIPSL's expressiveness can cover different problem domains other than MdVNE. Moreover, the used MdVNE scenario of the evaluation does not give any insights into the scalability of our approach in terms of runtime behavior. Hence, in future works with GIPS, a variety of different scenarios from different application domains on different levels of complexity should be used to evaluate the performance for even larger models to gain more detailed insights in terms of scalability.

Furthermore, the metric LOC is arguably not the most expressive measure for comparing different code bases. In this work, we have used LOC primarily as a measure of the amount of effort that has to be invested in order to combine GT frameworks and ILP solvers. In future evaluations of GIPS we could perform user studies or use different metrics to compare hand-crafted solutions with the automatically generated implementations.

Regarding correctness, we are looking at two different aspects: First, the correctness of our framework's implementation is currently verified by various unit tests and black box tests (e.g., using the MdVNE scenario) as well as using tests that compare results against established reference VNE implementations. Second, in future publications, we plan to present a detailed proof of the correctness of the presented approach.

8 Conclusion and Future Work

Designing a domain-specific ILP generator is a difficult and error-prone problem that repeatedly occurs for every domain or every newly created tool. Hence, we presented a novel approach to combine GT and ILP techniques in an integrated DSL to tackle this problem. GIPS, the tool proposed in this paper, enables users to generate domain-specific ILP problem generators given a metamodel and a problem specification in our new domain-specific language GIPSL. Therefore, we use GIPSL as a textual language combining GT and ILP to automatically generate software artifacts that integrate both approaches in order to benefit from their synergy. GIPS is integrated into eMoflon, a state-of-the-art GT tool, in which we implemented a model-driven approach to the VNE problem that was used as an example and as the evaluation scenario throughout this paper. We showed multiple benefits of using GIPSL in comparison to a hand-crafted version. First, GIPSL specifications promise to reduce the complexity as well as the number of errors of the implementation remarkably. Moreover, the automatic generation of code can decrease the effort needed to design an implementation. Second, we showed that our improvement of the implementation process is as efficient as an optimized but hand-crafted solution, in terms of runtime.

In the future, we would like to extend the presented approach by enlarging the language's capabilities to, for example, also include non-binary variable types like integer or even floating-point numbers. This would lead to an increased expressiveness including, but not limited to, the possibility to specify MILP problems. Another important aspect of future improvements is the extension of the GIPSL grammar to also include more complex logic expressions, e.g.: "*If mapping x is chosen, mapping y can be chosen*". Such language features would allow for compacter integrated specifications of complex problems, which fits well with the overall goal of the presented approach. Furthermore, it would be desirable to implement a translation mechanism that produces ILP problems in a more common representation, e.g., LP. This would enable the usage of a broad variety of available ILP solvers, without implementing GIPS interfaces.

Acknowledgements This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI.

References

[1] Edoardo Amaldi, Stefano Coniglio, Arie M.C.A. Koster & Martin Tieves (2016): *On the computational complexity of the virtual network embedding problem*. Electronic Notes in Discrete Mathematics, pp. 213–220, doi:10.1016/j.endm.2016.03.028.

[2] Anthony Anjorin, Nils Weidmann, Robin Oppermann, Lars Fritsche & Andy Schürr (2020): *Automating Test Schedule Generation with Domain-Specific Languages: A Configurable, Model-Driven Approach*. In: Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems, MODELS '20, ACM, p. 320–331, doi:10.1145/3365438.3410991.

[3] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause & Gabriele Taentzer (2010): *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*. In: Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems, Springer, pp. 121–135, doi:10.1007/978-3-642-16145-2_9.

[4] Stephen P. Bradley, Arnoldo C. Hax & Thomas L. Magnanti (1977): *Applied Mathematical Programming*. Addison-Wesley.

[5] Michael R. Bussieck & Alex Meeraus (2004): *General Algebraic Modeling System (GAMS)*. Springer, doi:10.1007/978-1-4613-0215-5_8.

[6] Martin Fleck, Javier Troya & Manuel Wimmer (2015): *Marrying search-based optimization and model transformation technology*. In: Proc. of the North American Symposium on Search Based Software Engineering, NasBASE '15, pp. 1–16.

[7] Charles L. Forgy (1982): *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, p. 17–37, doi:10.1016/0004-3702(82)90020-0.

[8] Robert. Fourer, David M. Gay & Brian W. Kernighan (2002): *AMPL: A Modeling Language for Mathematical Programming*. Cengage Learning.

[9] Sebastian Götz, Johannes Mey, René Schöne & Uwe Aßmann (2018): *A JastAdd- and ILP-based Solution to the Software-Selection and Hardware-Mapping-Problem at the TTC 2018*. In: Proc. of Transformation Tool Contest, TTC@STAF '18, CEUR-WS.org, pp. 31–36. Available at `http://ceur-ws.org/Vol-2310/paper4.pdf`.

[10] Sebastian Götz, Johannes Mey, René Schöne & Uwe Aßmann (2018): *Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem*. In: Proc. of Transformation Tool Contest, TTC@STAF '18, CEUR-WS.org, pp. 3–11. Available at `http://ceur-ws.org/Vol-2310/paper1.pdf`.

[11] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: Proc. of the Int. joint Conf. on Artificial Intelligence, IJCAI '73, Morgan Kaufmann Publishers Inc., p. 235–245.

[12] Thorsten Koch (2006): *Rapid Mathematical Programming or How to Solve Sudoku Puzzles in a Few Seconds*. In: Operations Research Proceedings, GOR '05, Springer, pp. 21–26, doi:10.1007/3-540-32539-5_4.

[13] Erhan Leblebici, Anthony Anjorin & Andy Schürr (2014): *Developing eMoflon with eMoflon*. In: Proc. of the Int. Conf. on Theory and Practice of Model Transformations, ICMT '14, Springer, pp. 138–145, doi:10.1007/978-3-319-08789-4_10.

[14] Erhan Leblebici, Anthony Anjorin & Andy Schürr (2017): *Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques*. In: Proc. of the Int. Conf. on Fundamental Approaches to Software Engineering, FASE '17, Springer, pp. 191–207, doi:10.1007/978-3-662-54494-5_11.

[15] David G. Luenberger & Yinyu Ye (1984): *Linear and Nonlinear Programming*. Springer, doi:10.1007/978-3-319-18842-3.

[16] Uwe Pohlmann & Marcus Hüwe (2019): *Model-driven allocation engineering: specifying and solving constraints based on the example of automotive systems*. Automated Software Engineering, pp. 315–378, doi:10.1007/s10515-018-0248-3.

[17] Siqi Shen, Vincent Van Beek & Alexandru Iosup (2015): *Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters*. In: *Proc. of the Int. Symposium on Cluster Computing and the Grid*, CCGrid '15, ACM, pp. 465–474, doi:10.1109/CCGrid.2015.60.

[18] Stefan Tomaszek, Erhan Leblebici, Lin Wang & Andy Schürr (2018): *Virtual Network Embedding: Reducing the Search Space by Model Transformation Techniques*. In: *Proc. of the Int. Conf. on Theory and Practice of Model Transformation*, ICMT '18, pp. 59–75, doi:10.1007/978-3-319-93317-7_2.

[19] Stefan Tomaszek, Erhan Leblebici, Lin Wang & Andy Schürr (2018): *Model-driven Development of Virtual Network Embedding Algorithms with Model Transformation and Linear Optimization Techniques*. In: *Modellierung 2018*, Gesellschaft für Informatik e.V., pp. 39–54, doi:20.500.12116/14957.

[20] Stefan Tomaszek, Roland Speith & Andy Schürr (2021): *Virtual network embedding: ensuring correctness and optimality by construction using model transformation and integer linear programming techniques*. *Software and Systems Modeling*, pp. 1299–1332, doi:10.1007/s10270-020-00852-z.

[21] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth & Zoltán Ujhelyi (2016): *Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework*. *Software and Systems Modeling*, p. 609–629, doi:10.1007/s10270-016-0530-4.

[22] Gergely Varró & Frederik Deckwerth (2013): *A Rete Network Construction Algorithm for Incremental Pattern Matching*. In: *Proc. of the Int. Conf. on Theory and Practice of Model Transformations*, ICMT '13, Springer, pp. 125–140, doi:10.1007/978-3-642-38883-5_13.

[23] Markus Weckesser, Malte Lochau, Michael Ries & Andy Schürr (2018): *Mathematical Programming for Anomaly Analysis of Clafer Models*. In: *Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems*, MODELS '18, ACM, p. 34–44, doi:10.1145/3239372.3239398.

A Appendix

A.1 Link Mapping Constraint

Listing 6 shows an example implementation of a constraint in GIPSL format, that would ensure the presence of valid source and target node mappings for a possible virtual to substrate link mapping. Note, that this example constraint does not solve the issue completely because it is only valid for links starting and ending at a virtual server. To ensure a valid link mapping in larger substrate networks, one has to define constraints that also take virtual switches into account because links might as well start or end at virtual switches. To make it clear, it is possible to define a constraint that addresses these issues completely and correctly using GIPSL. We decided not to include the full constraint version for the sake of brevity. A working version of the link mapping constraint can be found in the full example, including all other constraints in their respective full versions, on the GIPS examples repository[18] on Github.

Listing 6: Simplified GIPSL constraint example for the link mapping.

```
constraint -> pattern::link2link {
    mappings.srv2srv->filter(
        m | m.nodes().vsrv == self.nodes().vl.source
        & m.nodes().ssrv == self.nodes().sl.source)->count()
    + mappings.srv2srv->filter(
        m | m.nodes().vsrv == self.nodes().vl.target
        & m.nodes().ssrv == self.nodes().sl.target)->count()
    >= 2 * mappings.l2l->filter(
        m | m.nodes().sl == self.nodes().sl
        & m.nodes().vl == self.nodes().vl)->count()
}
```

A.2 GIPSL Grammar

Listing 7 shows the full version of the grammar of our GIPSL DSL in the EBNF-based Xtext specification language.

Listing 7: GIPSL as Xtext source file.

```
grammar org.emoflon.gips.gipsl.GIPSL with org.emoflon.ibex.gt.editor.GT
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.emoflon.org/ibex/gt/editor/GT" as GT
generate GipsL "http://www.emoflon.org/gips/gipsl/GIPSL"

@Override
EditorGTFile: {EditorGTFile}
    (imports+=EditorImport)*
    config = GipsConfig
    (patterns+=EditorPattern |
    conditions+=EditorCondition |
    mappings += GipsMapping |
    constraints += GipsConstraint |
    objectives += GipsObjective)*
    (globalObjective = GipsGlobalObjective)?;
GipsConfig : {GipsConfig}
    'config' '{'
```

[18] GIPS examples - https://github.com/Echtzeitsysteme/gips-examples

```
    'solver' ':=' solver=SolverType '[' 'home' ':=' home=GipsStringLiteral
        ',' 'license' ':=' license=GipsStringLiteral ']' ';'
    ('launchConfig' ':=' enableLaunchConfig = GipsBoolean '[' 'main' ':='
        mainLoc=GipsStringLiteral ']' ';')?
    ('timeLimit' ':=' enableLimit = GipsBoolean '[' 'value' ':='
        timeLimit=GipsDouble ']' ';')?
    ('randomSeed' ':=' enableSeed = GipsBoolean '[' 'value' ':='
        rndSeed=GipsInteger ']' ';')?
    ('presolve' ':=' enablePresolve = GipsBoolean ';')?
    ('debugOutput' ':=' enableDebugOutput = GipsBoolean ';')?
    '}';
enum SolverType:
    GUROBI='GUROBI';
GipsMapping :
    'mapping' name=ID 'with' rule=[GT::EditorPattern|ID] ';';
GipsConstraint :
    'constraint' '->' context=(GipsMappingContext | GipsTypeContext |
        GipsPatternContext) '{'
         expr = GipsBool
    '}';
GipsObjective :
    'objective' name=ID '->' context=(GipsMappingContext | GipsTypeContext
        | GipsPatternContext) '{'
         expr = GipsArithmeticExpr
    '}';
GipsMappingContext:
    'mapping::' mapping = [GipsMapping|ID];
GipsTypeContext:
    'class::' type = [ecore::EClassifier];
GipsPatternContext:
    'pattern::' pattern = [GT::EditorPattern|ID];
GipsObjectiveExpression:
    objective=[GipsObjective|ID];
GipsGlobalObjective :
    'global' 'objective' ':' objectiveGoal = GipsObjectiveGoal ('{'
        expr = GipsArithmeticExpr
    '}')?;
enum GipsObjectiveGoal:
    MIN = 'min' | MAX = 'max';
GipsBool:
    expr=GipsBoolExpr;
GipsBoolExpr:
    GipsBinaryBoolExpr;
GipsBinaryBoolExpr returns GipsBoolExpr:
    GipsUnaryBoolExpr({GipsBinaryBoolExpr.left = current}
        operator=GipsBoolBinaryOperator right=GipsUnaryBoolExpr)*;
GipsUnaryBoolExpr returns GipsBoolExpr: {GipsUnaryBoolExpr}
    (operator=GipsBoolUnaryOperator '(' operand=GipsBoolExpr ')' ) |
        GipsBooleanLiteral | GipsRelExpr;
GipsBooleanLiteral:
    literal = GipsBoolean;
enum GipsBoolBinaryOperator:
    AND='&'     |
    OR='|';
enum GipsBoolUnaryOperator:
    NOT = '!';
```

```
GipsRelExpr:
    left=GipsArithmeticExpr (operator=GipsRelOperator
        right=GipsArithmeticExpr)?;
enum GipsRelOperator:
  GREATER='>' |
  GREATER_OR_EQUAL='>=' |
  EQUAL='==' |
  UNEQUAL='!=' |
  SMALLER_OR_EQUAL='<=' |
  SMALLER='<';
GipsAttributeExpr:
    GipsMappingAttributeExpr |  GipsContextExpr |
        GipsLambdaAttributeExpression;
GipsMappingAttributeExpr returns GipsAttributeExpr:
    {GipsMappingAttributeExpr}
    'mappings.' mapping=[GipsMapping|ID] '->' expr = GipsStreamExpr;
GipsLambdaAttributeExpression returns GipsAttributeExpr:
    {GipsLambdaAttributeExpression}
    var=[GipsLambdaExpression|ID] '.' (expr = GipsNodeAttributeExpr | expr
        = GipsContextOperationExpression | expr = GipsFeatureExpr);
GipsContextExpr returns GipsAttributeExpr: {GipsContextExpr}
    'self' ('.' typeCast=GipsTypeCast)? ('.' (expr = GipsNodeAttributeExpr
        | expr = GipsContextOperationExpression | expr = GipsFeatureExpr)
        ('->' stream = GipsStreamExpr)?)?;
GipsContextOperationExpression:
    operation=GipsContextOperation;
enum GipsContextOperation:
    VALUE = 'value()' |
    MAPPED = 'isMapped()';
GipsTypeCast:
    'toType(' type=[ecore::EClass] ')';
GipsNodeAttributeExpr: {GipsNodeAttributeExpr}
    'nodes().' node=[GT::EditorNode|ID] ('.' typeCast=GipsTypeCast)? ('.'
        expr = GipsFeatureExpr)?;
GipsFeatureExpr:
    GipsFeatureNavigation;
GipsFeatureNavigation returns GipsFeatureExpr:
    GipsFeatureLit ({GipsFeatureNavigation.left=current} '.'
        right=GipsFeatureNavigation)?;
GipsFeatureLit returns GipsFeatureExpr: {GipsFeatureLit}
    feature = [ecore::EStructuralFeature] ('.' typeCast=GipsTypeCast)?;
GipsArithmeticExpr:
    GipsSumArithmeticExpr;
GipsSumArithmeticExpr returns GipsArithmeticExpr:
    GipsProductArithmeticExpr({GipsSumArithmeticExpr.left = current}
        operator=GipsSumOperator right=GipsProductArithmeticExpr)*;
GipsProductArithmeticExpr returns GipsArithmeticExpr:
    GipsExpArithmeticExpr({GipsProductArithmeticExpr.left = current}
        operator=GipsProductOperator right=GipsExpArithmeticExpr)*;
GipsExpArithmeticExpr returns GipsArithmeticExpr:
    GipsUnaryArithmeticExpr({GipsExpArithmeticExpr.left = current}
        operator=GipsExpOperator right=GipsUnaryArithmeticExpr)*;
GipsUnaryArithmeticExpr returns GipsArithmeticExpr:
    {GipsUnaryArithmeticExpr}
    operator=GipsArithmeticUnaryOperator  '(' operand=GipsBracketExpr ')'
        | GipsBracketExpr;
```

```
GipsBracketExpr returns GipsArithmeticExpr: {GipsBracketExpr}
    '(' operand=GipsArithmeticExpr ')' | GipsExpressionOperand;
GipsExpressionOperand:
    GipsAttributeExpr | GipsObjectiveExpression | GipsArithmeticLiteral;
GipsArithmeticLiteral:
    value = (GipsDoubleLiteral | GipsIntegerLiteral);
enum GipsProductOperator:
    MULT='*' |
    DIV='/'  ;
enum GipsExpOperator:
    POW='^' ;
enum GipsSumOperator:
    PLUS='+'    |
    MINUS='-'   ;
enum GipsArithmeticUnaryOperator:
    NEG='-' |
    ABS='abs'   |
    SQRT='sqrt' |
    sin='sin'   |
    cos='cos'   ;
GipsStreamExpr:
    GipsStreamNavigation;
GipsStreamNavigation returns GipsStreamExpr:
    GipsStreamLit({GipsStreamNavigation.left=current}'->'
       right=GipsStreamNavigation)?;
GipsStreamLit returns GipsStreamExpr:
    {GipsSelect} 'typesOf' '(' type=[ecore::EClassifier] ')' |
    {GipsStreamSet} operator=GipsStreamSetOperator lambda =
       GipsLambdaExpression |
    {GipsStreamArithmetic} operator=GipsStreamArithmeticOperator lambda =
       GipsLambdaExpression |
    {GipsStreamBoolExpr} operator=GipsStreamNoArgOperator;
GipsLambdaExpression:
    '(' name=ID '|' expr=GipsBoolExpr ')';
enum GipsStreamSetOperator:
    FILTER='filter';
enum GipsStreamArithmeticOperator:
    SUM = 'sum';
enum GipsStreamNoArgOperator:
    EXISTS = 'exists()' |
    NOTEXISTS = 'notExists()' |
    count = 'count()';
GipsDoubleLiteral:
    GipsDouble;
GipsIntegerLiteral:
    GipsInteger;
GipsDouble returns ecore::EDouble:
    ('-')?INT '.' INT;
GipsInteger returns ecore::EInt:
    (('-')? INT);
@Override
terminal INT returns ecore::EInt:
    '0'..'9' ('0'..'9')*;
GipsBoolean returns ecore::EBoolean:
    TRUE | FALSE;
terminal TRUE returns ecore::EBoolean:
```

```
        'true';
    terminal FALSE returns ecore::EBoolean:
        'false';
    GipsStringLiteral:
        STRING;
    @Override
    terminal STRING returns ecore::EString:
        '"' ( '""' | !('"') )* '"';
```

Time and Space Measures for a Complete Graph Computation Model

Brian Courtehoute and Detlef Plump
Department of Computer Science, University of York, York, UK
{bc956,detlef.plump}@york.ac.uk

We present a computation model based on a subclass of GP 2 graph programs which can simulate any off-line Turing machine of space complexity $O(s(n)\log s(n))$ in space $O(s(n))$. The simulation only requires a quadratic time overhead. Our model shares this property shown by van Emde-Boas with Schönhage's storage modification machines and Kolmogorov-Uspenskii machines. These machines use low-level pointer instructions whereas our GP 2-based model uses pattern-based transformation rules and high-level control constructs.

1 Introduction

Schönhage's storage modification machines (SMMs) and Kolmogorov-Uspenskii machines (KUMs) are graph-based computation models that do not use graph transformation rules. They have the remarkable feature of being able to simulate Turing machines using less space with only polynomial time overhead, using a uniform space measure [15]. Although these are graph-based models, the literature on rule-based graph transformation so far seems to have ignored the work of Schönhage [13] and Kolmogorov [9], and in particular the results of van Emde Boas [15] and Luginbuhl [10].

SMMs and KUMs differ from modern graph transformation languages in that they use low-level pointer instructions instead of pattern-based rules where the programmer can specify a subgraph and how it is transformed, allowing for a more natural implementation of graph algorithms. This is a higher-level problem-oriented approach.

The lack of "go to" statements is an advantage of structured programming languages like GP 2 since they are known to be harmful [6]. Furthermore, GP 2 is based on the double-pushout approach for graph transformation, which comes with a large amount of theoretical results. We are not aware of any comparable theory for SMMs and KUMs.

The fact that this space compression property applies to a high-level language is not obvious. Even so, we show in this paper that GP 2 exhibits the same space compression feature. Specifically, we show that a Turing machine using $O(s(n)\log s(n))$ space can be simulated in $O(s(n))$ space with quadratic time overhead, where $s(n)$ is an arbitrary function. The complexity class of the simulation is a strict subset of the original class (for non-constant $s(n)$).

Note that another way to show space compression is to efficiently simulate SMMs in general. This would also mean other properties are applicable to GP 2, such as the real-time simulation of Turing machines shown by Luginbuhl [10]. This paper's approach however comes with a concrete class of programs that show in much detail that the simulation is possible.

In Section 2, we describe how GP 2 uses rule-based graph transformation efficiently and show a new variant of a Theorem that allows for constant-time rule matching.

We show how we simulate Turing machines in Section 3. How configurations of Turing machines are encoded as graphs is described in Subsection 3.1. The simulating program is in Subsection 3.2. We then discuss correctness in Subsection 3.4.

In Section 4 we identify a class of efficient computation models based on GP 2, which come with time and space complexity measures. We then show that our simulation gives rise to such a model.

Finally, in Section 5, we show the aforementioned time and space complexities of the simulation, and discuss the use of logarithmic versus uniform space measures.

2 Rule-Based Graph Transformation

Throughout this paper, we use a subset of GP 2 called SGP 2 (Small GP 2). Instead of using expressions with variables as labels, we use rules with various constant labels rather than rule schemata as in full GP 2. Application conditions are not needed either.

We call the graphs to which rules are applied *host graphs*. Nodes and edges can be *marked* red, green or blue. In addition, nodes can be marked grey and edges can be dashed. For example, rules in `CacheNext` in Figure 12 contains red and unmarked nodes and blue edges. Marks are convenient, among other things, to record visited items during a graph traversal and to differentiate between similar structures.

The principal SGP 2 programming construct used in this paper are graph transformation rules according to the double-pushout approach with injective matching. For example, the rules in `CacheNext` in Figure 12 consist of a left-hand graph and a right-hand graph which are specified graphically. The small numbers attached to nodes are identifiers, all other black text in the graphs are labels.

The structural operational semantics of programs is defined in Figures 2 and 3, where inference rules inductively define a small-step transition relation $\rightarrow$ on *configurations*. In the setting of SGP 2, a configuration is either a command sequence together with a host graph, just a host graph or the special element fail:

$$\rightarrow \subseteq (\mathrm{ComSeq} \times \mathscr{G}) \times ((\mathrm{ComSeq} \times \mathscr{G}) \cup \mathscr{G} \cup \{\mathrm{fail}\})$$

where $\mathscr{G}$ is the set of SGP 2 host graphs. Configurations in $\mathrm{ComSeq} \times \mathscr{G}$, given by a rest program and a host graph, represent states of unfinished computations while graphs in $\mathscr{G}$ are final states or *results* of computations. The element fail represents a failure state. A configuration γ is said to be *terminal* if there is no configuration δ such that $\gamma \rightarrow \delta$.

Figure 2 shows the inference rules for the core commands of SGP 2. The rules contain meta-variables for command sequences and graphs, where R stands for a call of a rule set or of a rule, C, P, P', Q stand for command sequences, and G, H stand for host graphs. The transitive and reflexive-transitive closures of $\rightarrow$ are written $\rightarrow^+$ and $\rightarrow^*$, respectively. We write $G \Rightarrow_R H$ if H results from host graph G by applying the rule set R, while $G \not\Rightarrow_R$ means that there is no graph H such that $G \Rightarrow_R H$ (application of R fails).

An example of a SGP 2 program can be found in Figure 1.

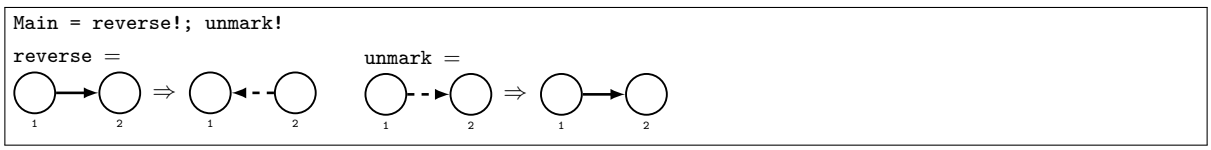


Figure 1: Small GP 2 program that reverses the edges of an `empty`-labelled graph.

$$[\text{call}_1]\ \frac{G \Rightarrow_R H}{\langle R, G\rangle \to H} \qquad [\text{call}_2]\ \frac{G \not\Rightarrow_R}{\langle R, G\rangle \to \text{fail}}$$

$$[\text{seq}_1]\ \frac{\langle P, G\rangle \to \langle P', H\rangle}{\langle P; Q, G\rangle \to \langle P'; Q, H\rangle} \qquad [\text{seq}_2]\ \frac{\langle P, G\rangle \to H}{\langle P; Q, G\rangle \to \langle Q, H\rangle}$$

$$[\text{seq}_3]\ \frac{\langle P, G\rangle \to \text{fail}}{\langle P; Q, G\rangle \to \text{fail}}$$

$$[\text{if}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle} \qquad [\text{if}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{try}_1]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, H\rangle} \qquad [\text{try}_2]\ \frac{\langle C, G\rangle \to^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{alap}_1]\ \frac{\langle P, G\rangle \to^+ H}{\langle P!, G\rangle \to \langle P!, H\rangle} \qquad [\text{alap}_2]\ \frac{\langle P, G\rangle \to^+ \text{fail}}{\langle P!, G\rangle \to G}$$

$$[\text{alap}_3]\ \frac{\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle}{\langle P!, G\rangle \to H} \qquad [\text{break}]\ \langle \texttt{break}; P, G\rangle \to \langle \texttt{break}, G\rangle$$

Figure 2: Inference rules for Small GP 2 core commands

The inference rules for the remaining SGP 2 commands are given in Figure 3. These commands are referred to as *derived* commands because they can be defined by the core commands.

$$[\text{skip}]\quad \langle \texttt{skip}, G\rangle \to G$$

$$[\text{if}_3]\quad \langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_3]\quad \langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$$

$$[\text{try}_4]\quad \langle \texttt{try}\ C\ \texttt{else}\ P, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else}\ P, G\rangle$$

$$[\text{try}_5]\quad \langle \texttt{try}\ C, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else skip}, G\rangle$$

Figure 3: Inference rules for Small GP 2 derived commands

When analysing the time and space complexity of programs, we assume that these are fixed. This is customary in algorithm analysis where programs are fixed and running time or space is measured in terms of input size [1, 14]. In our setting, the input size is the *size* of a host graph, which we define to be the total number of nodes and edges.

Since matching can be time-consuming we use rooted rules. They contain special nodes called *roots* that can match with other roots in the host graph in constant time, allowing rule matching to happen locally.

In order to achieve this efficient rule matching, we use a variant of the fast matching theorem first introduced in [3]. The trade-off with the original is that we can have a more general host graph, i.e. we allow the indegree to be unbounded, but there are more restrictions on rules, namely there needs to be a directed path from some root node to every node in the left-hand side.

For the sake of this paper, a rule $L \Rightarrow R$ is *fast* if each node in L is reachable from some root (respecting edge directions). Other requirements for a rule to be fast are omitted since they concern variable labels and application conditions, which none of the rules in this paper have.

Theorem 1 (Fast Rule Matching)**.** *Matching can be implemented to run in constant time for fast rules, using host graphs with a bounded outdegree that contain a bounded number of roots.*

Proof. Let G be such a host graph and `r` such a rule. The rule `r` also satisfies the original variant of the fast rule matching theorem since directed reachability implies undirected reachability. Let A be the matching algorithm as shown in [3] as part of the proof of the original theorem. Since G can have unbounded indegree, we cannot conclude that A is constant time. The algorithm A works by matching paths starting from roots edge by edge. Now if we modify A to build this path using outgoing edges only, the algorithm only needs to look for outgoing edges, of which there are constantly many due to G having bounded outdegree. The same arguments as in the proof of the original theorem then apply, meaning the modified algorithm A takes constant time. □

3 Simulating Turing Machines

In this paper, we consider deterministic off-line Turing machines, i.e. there is a read-only finite input tape, as well as an initially blank working tape that is one-way infinite. Acceptance happens when an accepting state is reached. We use off-line Turing machines in order to exhibit that space compression can happen with sublinear space complexities.

Without loss of generality, we consider off-line Turing machines whose state set Q is a finite set of integers. The input alphabet is $\Sigma = \{1,2\}$, and the tape alphabet is $\Gamma = \{0,1,2\}$, where 0 is the blank symbol. There is an initial state q_0, an accepting state h_a, and a transition function $\delta: Q \times \Sigma \times \Gamma \to Q \times \Gamma \times \{\mathrm{L},\mathrm{R},\mathrm{S}\} \times \{\mathrm{L},\mathrm{R},\mathrm{S}\}$.

The time complexity of a Turing machine is the function which, to an integer n, associates the maximum number of transitions starting from an input of size n. The space complexity is the function which associates to n the maximum number of used working tape squares for an input of size n.

We simulate a Turing machine M with an SGP 2 program called `Sim`(M), as specified in Figures 6 to 12. We call the class of these programs `Sim`. These programs are constructed in order to achieve the complexity results given in Section 5. For instance, rooted nodes are only essential for time complexity, otherwise they could be replaced by marks.

Input symbols, tape symbols, and state names are encoded as integers. For simplicity, the figures in this paper show the case where the tape alphabet has size 3, and we reserve 0 for the blank symbol.

3.1 Turing Machine Configurations as Graphs

We encode the working tape of a Turing machine by dividing it into b blocks, which we keep track of in a list called BLOCKSET. The content of a block can be considered a ternary number that we store as an index of BLOCKSET. Hence we want the size of a block to be $\log b$. Since the tape alphabet is assumed to be ternary, logarithms in this paper are of base 3. Note however that $\mathrm{O}(\log_3 n) = \mathrm{O}(\log_c n)$ for any constant $c > 1$. The content of the block that is currently being worked on is stored in CACHE as a list of tape squares. Given a Turing machine of space complexity $\mathrm{O}(s(n)\log s(n))$, that many working tape squares are needed. So we can divide our tape into $\mathrm{O}(s(n))$ blocks of size $\mathrm{O}(\log s(n))$. So to represent this in the simulation, the number of nodes needed is the sum of these complexities, i.e. $\mathrm{O}(s(n))$. The space complexity of the simulation will be proven in Theorem 4.

Figure 4 shows the initial configuration of a Turing machine with its input and working tapes and tape heads, as well as the initial state q_0. The corresponding graph can be seen in Figure 5.

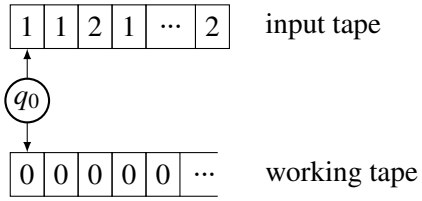


Figure 4: Initial configuration of a Turing machine

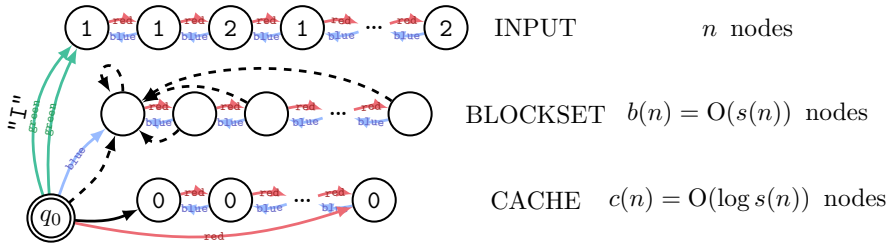


Figure 5: Graph representation of the initial configuration of a Turing machine of space complexity $\mathrm{O}(s(n)\log s(n))$.

That graph consists of the *central node*, labelled by the initial state q_0 as represented by some integer, and the subgraphs INPUT, BLOCKSET, and CACHE. We represent 'left' and 'right' positioning by blue and red edges respectively. These edges will not be modified by the program. The unmarked and unlabelled green edges mark the positions of the tape heads. Dashed edges serve to encode the state of BLOCKSET and how it relates to CACHE.

INPUT represents the input tape of the Turing machine, and the position of the tape head is represented by an unmarked green edge from the central node. The green edge labelled `"I"` always points to the leftmost square of the input tape. The working tape is represented by BLOCKSET and CACHE. These subgraphs are implemented as doubly linked lists in order to enable fast rule matching.

Each node in BLOCKSET represents a block of $c(n)$ nodes of the working tape. The content of a block is encoded as a non-negative integer $k \leq b(n)$, represented by a dashed edge towards the k^{th} node of BLOCKSET (i.e. k red edges away from the leftmost node). The blue edge from the central node points towards the leftmost node, and the dashed edge towards the block that contains the position of the tape head.

CACHE is the block of the working tape that contains the tape head. Its position within BLOCKSET is marked with a dashed edge from the central node. The red edge from the central node points towards the rightmost node of the cache, and the unmarked edge towards the current position of the tape head. The node labels represent the content of each square.

The operations of the Turing machine happen within CACHE. If the tape head moves out of bounds of CACHE, we encode the content of CACHE within BLOCKSET, move on to another block, and decode it into CACHE.

3.2 The Simulation

In this subsection, we assume we have a Turing machine M and describe $\texttt{Sim}(M)$, the SGP 2 program that simulates M. The program $\texttt{Sim}(M)$ takes inputs that consist of graphs such as INPUT together with the central node and green edges as represented in Figure 5. The overall behaviour of the program is that

it starts the simulation with a small BLOCKSET and CACHE. If we run out of tape squares, we reset to the initial configuration, extend BLOCKSET and CACHE, and restart the simulation.

Figure 6 contains the main control sequence of `Sim`(M). First the rule `setup` is called. It matches the root and constructs CACHE with two nodes and BLOCKSET with nine nodes, as seen in Figure 5. We omit the definition of that rule since it is straightforward.

Next we have the loop `Simulate!`, which means the procedure `Simulate` is applied until no longer possible. The loop body first calls `Transitions` (Figure 7), a procedure that applies the transition function to the current state of M. If `Transitions` cannot apply the transition function, the procedure results in failure, and the loop terminates. Next, `try MoveLeft` is called, which means we attempt to apply `MoveLeft` and if it fails, we skip this instruction. The rules in `MoveLeft` detect a label `"L"` on the unmarked edge adjacent to the central node, which is created in `Transitions` if the working tape head needs to move to the left. It then executes that move in CACHE. Analogously, `try MoveRight` attempts to move the working tape head to the right. These rule sets (which are nondeterministic calls of the rules they contain) will fail however if the tape head moves out of bounds of CACHE, which is detected by `Left` or `Right` since the continued presence of the `"L"` or `"R"` label indicates that tape head movement still needs to be done. In that case, `PrevBlock` or `NextBlock` (Figure 9) are called in order to move towards the relevant block.

Figure 7 contains `Transitions`, the rule set that encodes the transition functions of M. For each entry in the transition table, there is a corresponding rule in `Transitions`. The rules are divided into nine categories `transitionXY`, where `X` represents the movement of the input tape head, and `Y` that of the working tape head. Each rule implements the change of labels and the movement of the input tape head directly. For the movement of the working tape head, a label is left behind on the unmarked edge, and the actual movement is handled by `Simulate` in Figure 6.

Figure 8 shows the procedure `Restart` which, when the machine runs out of tape, resets the simulation and extends the tape. CACHE and hence the size of a block is extended by one square, and the size of BLOCKSET is tripled. This is done by first rewinding all tape heads to the initial square, then both tapes are erased and extended.

In Figure 9, we show the procedures `NextBlock` and `PrevBlock`, which handle movement to another block. We call the node pointed at by a dashed edge from the central node the *active block node*. The procedure `Encode` (Figure 10) saves the information from CACHE into BLOCKSET. This process is further described in the example in Subsection 3.3.

The procedure `Decode` in Figure 11 decodes the active block in BLOCKSET and stores the information in CACHE. It is analogous to `Encode`.

The procedure `CacheInc` in Figure 12 increments CACHE as a ternary number. First the rightmost digit is turned into a red root. Then the digit is attempted to be incremented with the rule set `Inc`. If the digit is a `0` or a `1`, incrementation succeeds, the process is marked as finished by turning the red root into an unmarked node, and `CacheInc` terminates. If the digit is a `2` however, `Inc` fails, the `2` is turned into a `0`, and the red root is moved along to the next digit which is attempted to be incremented. If `Increment!` ends without `Inc` ever succeeding, it means that all digits were `2`, and that the red root is still present. In that case, `Reset!` turns all digits into `0`.

Similarly, `CacheDec` decrements CACHE as a ternary number if that number is not 0, and keeps it as 0 if it already is.

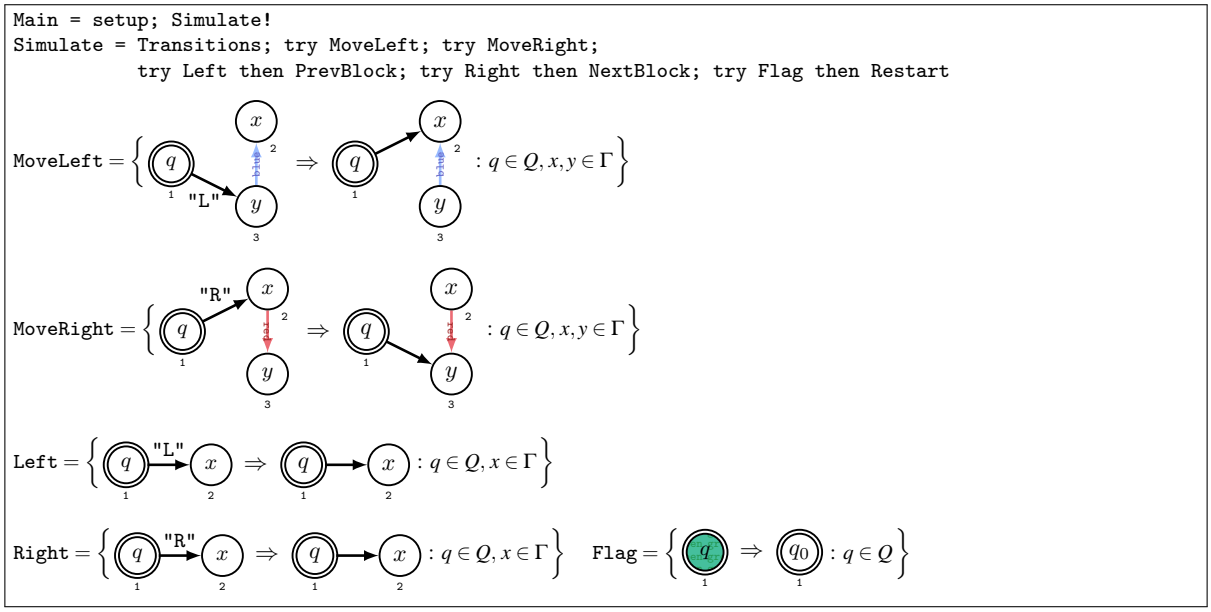


Figure 6: SGP 2 program $\texttt{Sim}(M)$ that simulates Turing machine M.

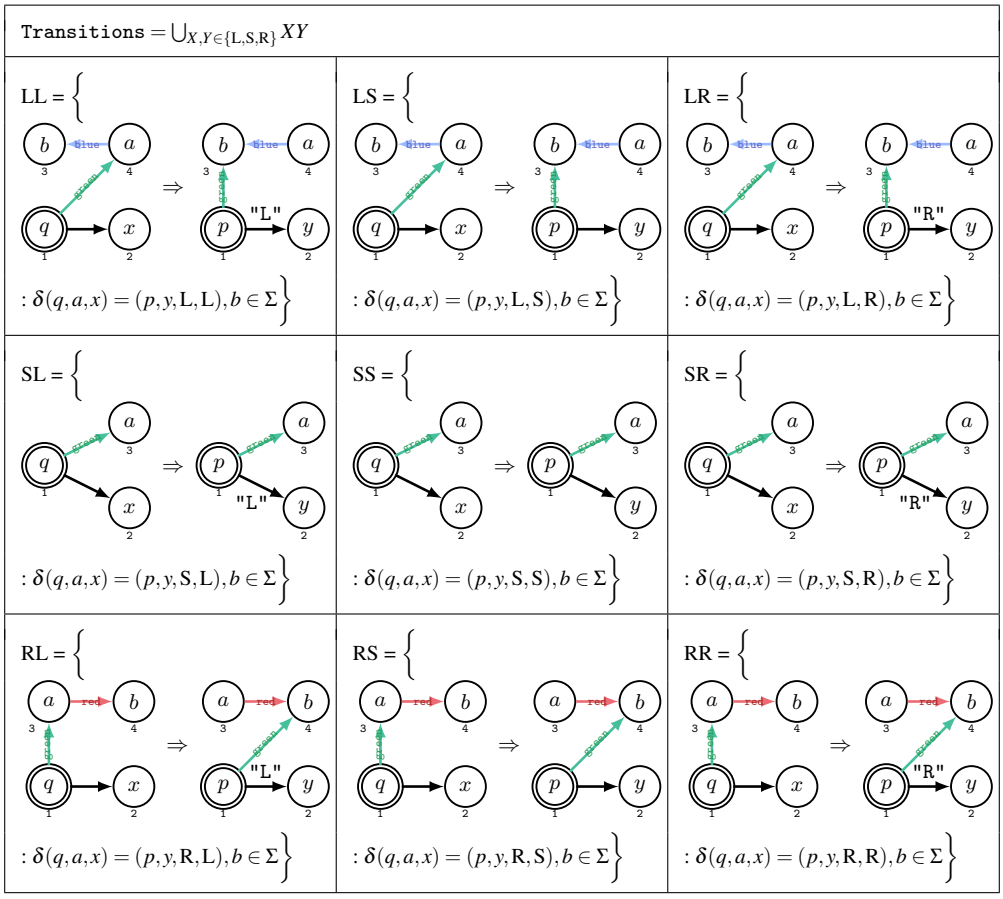


Figure 7: Rule set `Transitions` that models the transitions of a Turing Machine.

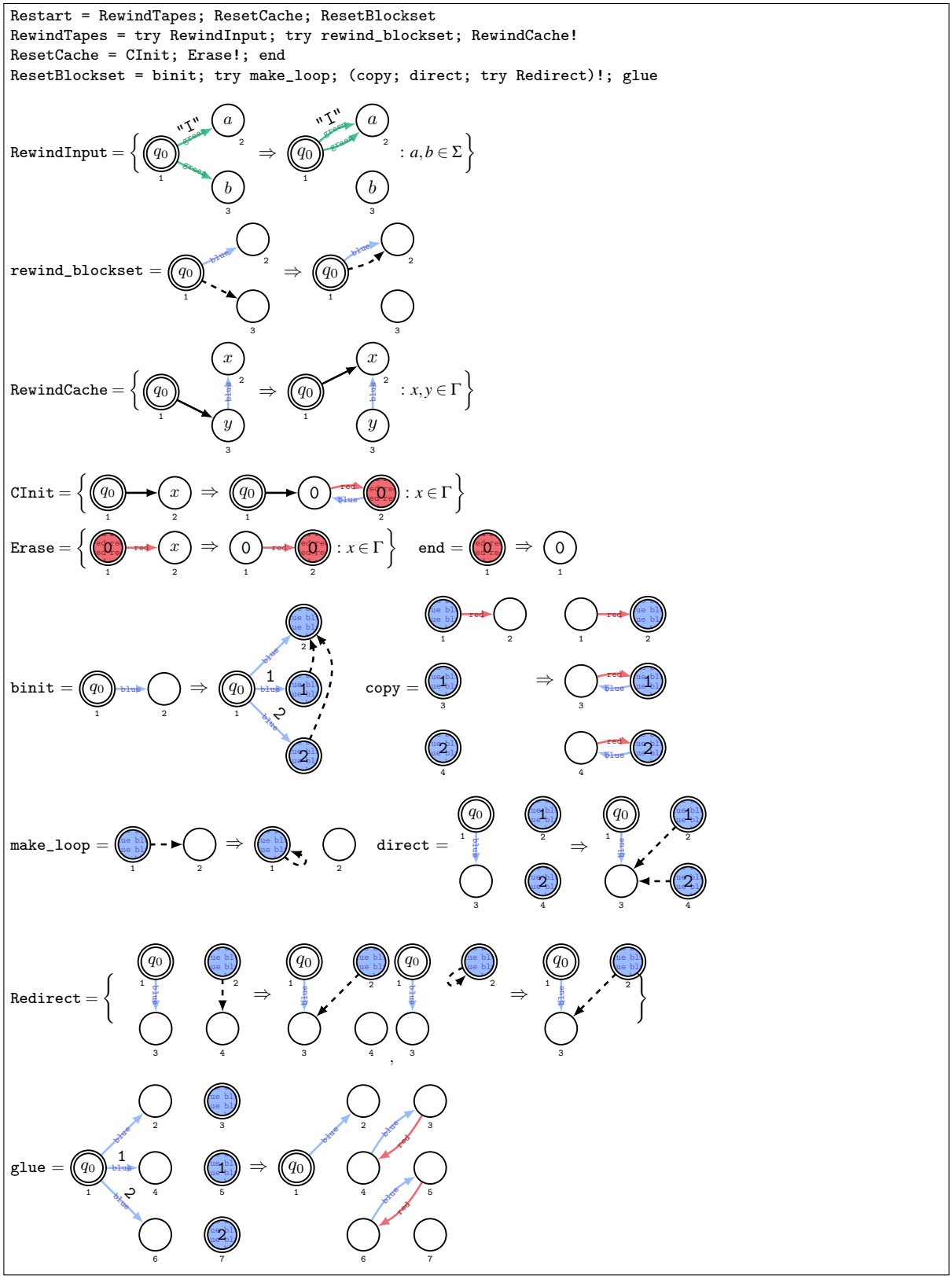


Figure 8: Procedure `Restart` that resets the simulation and enlarges the tape.

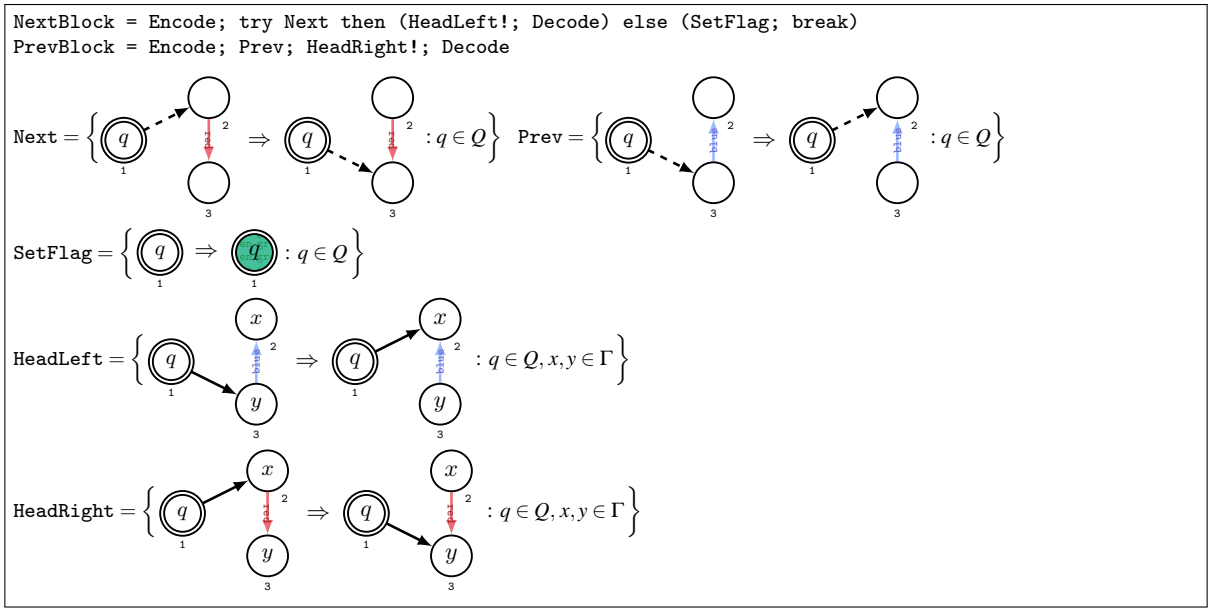


Figure 9: Procedures `NextBlock` and `PrevBlock` that change the active block.

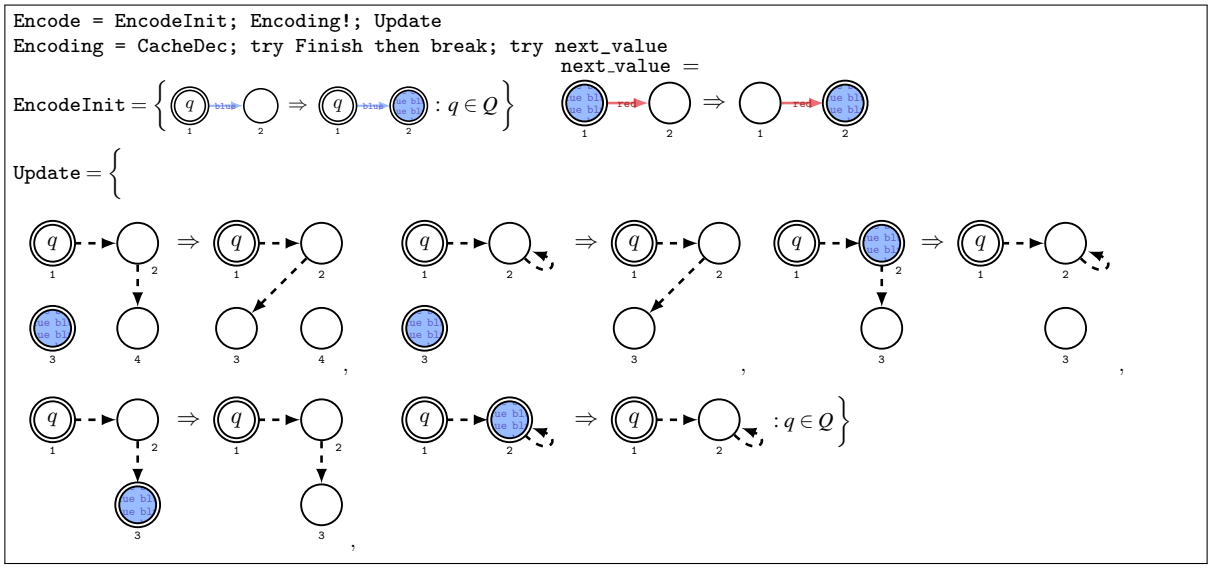


Figure 10: Procedure `Encode` that encodes the current block into BLOCKSET.

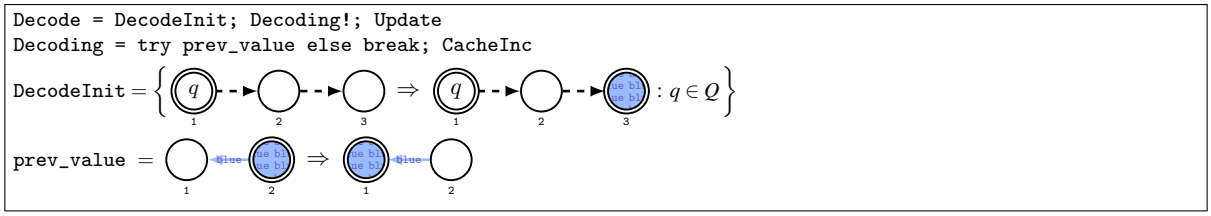


Figure 11: Procedure `Decode` that decodes the current block into CACHE.

```
CacheInc = CacheInit; Increment!; if Unfinished then (Reset!; Finish)
Increment = try Inc then (Finish; break)
                  else (overflow; try CacheNext else break)
CacheDec = CacheInit; Decrement!; if Unfinished then Reset!
Decrement = try Dec then (Finish; break)
                  else (underflow; try CacheNext else break)
Reset = overflow; try CachePrev else break
```

CacheInit = $\left\{ q \xrightarrow{\text{red}} x \Rightarrow q \xrightarrow{\text{red}} x : q \in Q, x \in \Gamma \right\}$

Inc = $\left\{ 0 \Rightarrow 1 , 1 \Rightarrow 2 \right\}$ Dec = Inc^{-1} overflow = $2 \Rightarrow 0$ underflow = $0 \Rightarrow 2$

Finish = $\left\{ x \Rightarrow x : x \in \Gamma \right\}$ Unfinished = $\left\{ x \Rightarrow x : x \in \Gamma \right\}$

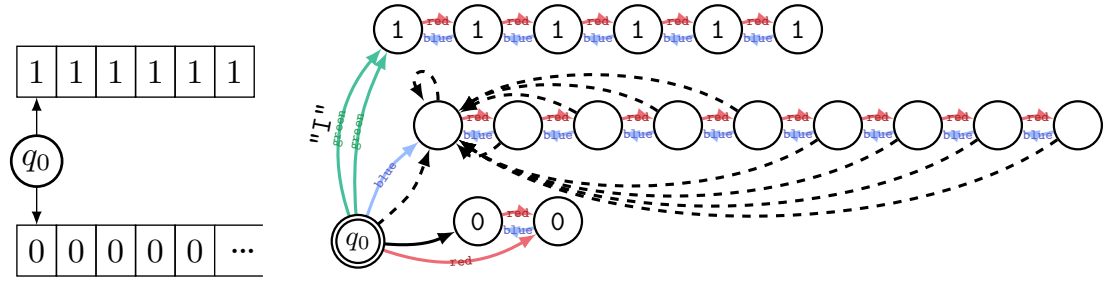
CacheNext = $\left\{ x \xrightarrow{\text{blue}} y \Rightarrow x \xrightarrow{\text{blue}} y : x, y \in \Gamma \right\}$

CachePrev = $\left\{ x \xrightarrow{\text{red}} y \Rightarrow x \xrightarrow{\text{red}} y : x, y \in \Gamma \right\}$

Figure 12: Procedures `CacheInc` and `CacheDec` that increment/decrement the ternary number stored in CACHE.

3.3 Example

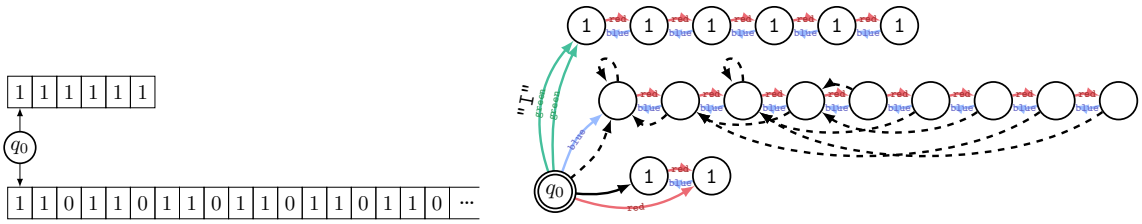
In this subsection, we give an example of a Turing machine simulation, and show how we move from one block to another. Consider a Turing machine that takes as input the number n represented in unary, and writes n in binary on its working tape n times. It is reasonable to assume the machine has a space complexity of $O(n \log n)$. If $n = 6$, the machine uses 18 squares, which are filled with 6 copies of the string `110` (6 in binary). A CACHE size of 2 and a BLOCKSET size of 9 are enough to represent $2 \cdot 9 = 18$ tape squares. Their representation in the simulation has only $2 + 9 = 11$ nodes, which is $O(n + \log n) = O(n)$. The initial state of the machine on input 6 is shown in Figure 13, and the final one in Figure 14.



(a) Turing machine (b) Simulation

Figure 13: Initial state of the example machine

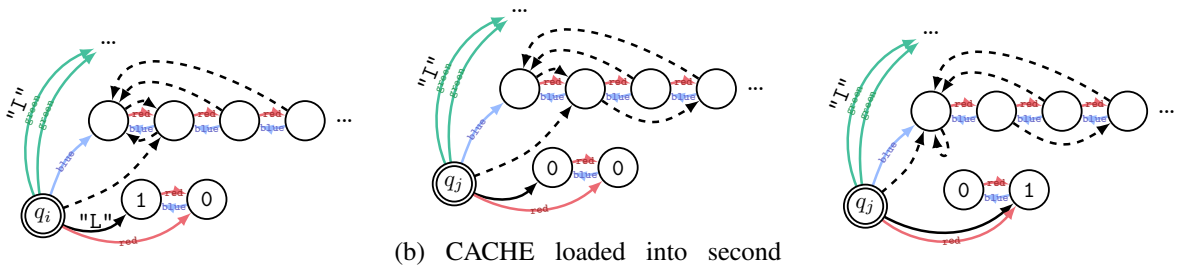
Let us sketch the behaviour of the machine. First, the input is copied in unary onto the working tape for use as a counter. Then, a binary number to the right of the counter is incremented while traversing the input. The previous step is repeated while decrementing the counter until it reaches 0. Tape contents need to be shifted. The symbol 2 can be used for marking tape squares.



(a) Turing machine (b) Simulation

Figure 14: Final state of the example machine

Now let us show what happens when the block has to be changed. Consider the situation where `010` are the first 3 squares of the working tape and the tape head is on the third square. Let the next transition write symbol `1` and move the tape head to the left, and change state q_i to state q_j.



(a) Symbol "1" written into CACHE. (b) CACHE loaded into second block. (c) First block loaded into CACHE.

Figure 15: Changing blocks

Figure 15 shows the process of changing the active block. In Subfigure 15a, a rule from `Transitions` has just been applied, labelling the left node of CACHE `1`, and the unmarked edge "`L`". Next, in `Simulate`, `Left` matches and `PrevBlock` is called. The procedure `Encode` loads the content of CACHE into a dashed edge within BLOCKSET. While CACHE is decremented as a ternary number, the outgoing dashed edge of the current (second) BLOCKSET node is shifted to the right. This produces the graph from Subfigure 15b. Next we move one block to the left with `Prev`, and reposition the tape head in CACHE with `HeadRight!`. We then load the new block into CACHE. While moving the outgoing dashed edge of the current (first) BLOCKSET node to the left, the ternary number in CACHE is incremented. This results in the graph from Subfigure 15c.

3.4 Correctness

We define the *configuration* of a Turing machine to consist of the input and working tapes, the position of the tape heads on those tapes, and the current state.

Let us define what graphs programs in `Sim` operate on. We call graphs as in Figure 5 *configuration graphs*. These graphs can vary from the depicted graph in the following ways. The targets of the unlabelled green edge, dashed edges, and the unmarked edge can be any node in INPUT, BLOCKSET, and CACHE respectively. An exception to that is the outgoing dashed edge of the active node in BLOCKSET (target of dashed edge from the root), which is targeted towards the leftmost node in BLOCKSET since the content of that block is currently stored in CACHE. This is a convention that allows the graph representation of a configuration to be unique. The labels of nodes in INPUT, CACHE, and the root node

can be any element from the input alphabet, tape alphabet, or set of states respectively, encoded as SGP 2 labels.

Note that the graphs' capacity to represent tape squares is limited by the sizes of BLOCKSET and CACHE. Hence for a non-negative integer k, we define a function enc_k that encodes the configuration S of a Turing machine as the configuration graph $\mathrm{enc}_k(S)$ with 3^{k+2} nodes in BLOCKSET and $k+2$ in CACHE. The integer k corresponds to the number of times the tape is extended by the procedure `Restart`. For a given value of k that is large enough, $\mathrm{enc}_k(S)$ exists and is unique.

For a Turing machine M we denote a single transition from configuration S to S' by $S \Rightarrow_M S'$, and the transitive reflexive closure of that relation using $\Rightarrow_M^*$. Similarly, for an SGP 2 rule `r` we use $S \Rightarrow_{\mathtt{r}} S'$ to denote a change of configuration, and $S \Rightarrow_{\mathtt{P}}^* S'$ for a program `P`.

During the execution of $\mathtt{Sim}(M)$, graphs that are not configuration graphs are generated. They only differ in terms of edge labels, node marks, roots, and dashed edge targets. However, these variations are temporary. Once the procedure `Simulate` has successfully terminated, the host graph is once again a configuration graph.

We say an SGP 2 program `P!` (i.e. a loop where procedure `P` is applied as long as possible) *simulates* a Turing machine M of initial configuration I if, for each two configurations S and S' of M such that $I \Rightarrow_M^* S \Rightarrow_M S'$, we have $\mathrm{enc}_0(I) \Rightarrow_{\mathtt{P!}}^* \mathrm{enc}_k(S) \Rightarrow_{\mathtt{P}}^* \mathrm{enc}_k(S')$ for some integer k.

The following theorem shows that this is indeed the case for the class of programs presented in this paper.

Theorem 2 (Correctness)**.** *Let M be a Turing machine and $\mathtt{Sim}(M)$ the corresponding SGP 2 program. Then the subprocedure `Simulate!` simulates M.*

Proof. Let S and S' be configurations of M such that $I \Rightarrow_M^* S \Rightarrow_M S'$. Then some transition $\delta(q,a,x) = (p,y,X,Y)$ happened, where $p,q \in Q$, $a \in \Sigma$, $x,y \in \Gamma$, and $X,Y \in \{\mathrm{L},\mathrm{S},\mathrm{R}\}$. So the only changes from S to S' are that q and x have been updated to p and y, and that the input and working tape heads have been shifted into direction X and Y respectively.

We proceed by induction on the length of $I \Rightarrow_M^* S$. For the base case we only need to show $\mathrm{enc}_k(S) \Rightarrow_{\mathtt{P}}^* \mathrm{enc}_k(S')$ for some integer k. This also happens to be what we need to show for the induction step since $\mathrm{enc}_0(I) \Rightarrow_{\mathtt{P!}}^* \mathrm{enc}_k(S)$ is given by the induction hypothesis, and the existence of a large enough k by part of the proof of Theorem 4.

Now it remains to show that $\mathrm{enc}_k(S) \Rightarrow_{\mathtt{P}}^* \mathrm{enc}_k(S')$. The procedure `Reset` can be ignored because it is never called since we already have a large enough k.

When running `Simulate` on the graph $\mathrm{enc}_k(S)$, the first call is the rule set `Transitions`. A rule in that set that is guaranteed to be applicable is the one corresponding to the aforementioned Turing machine transition. It correctly updates the q and x to be p and y to match $\mathrm{enc}_k(S')$. It also moves the input tape head to the correct position. For the rest of this proof, we will show that the working tape head moves to the correct position.

If $Y = \mathrm{S}$, the working tape head is not moved, corresponding to $\mathrm{enc}_k(S)'$. Due to lack of an unmarked edge label, none of the `try` conditions in `Simulate` are applicable, and we terminate with $\mathrm{enc}_k(S')$.

The cases for $Y = \mathrm{L}$ and $Y = \mathrm{R}$ are analogous, so we will only argue for the former. When attempting to apply `MoveLeft`, if the target of the unmarked edge is not the leftmost node of CACHE, the rule set succeeds, the unmarked edge is in the correct position, and no other conditions of `try` statements in `Simulate` are applicable. Hence we terminate with $\mathrm{enc}_k(S')$.

If target of the unmarked edge is the leftmost node of CACHE however, the only `try` condition that can match is `Left`. This removes the edge label and calls `PrevBlock`. So it remains to argue that `PrevBlock` turns the host graph into $\mathrm{enc}_k(S')$.

The commands `Prev; HeadRight!` correctly position the working tape head on the rightmost node of the previous block. It remains to show that `Encode` and `Decode` preserve the working tape content according to the encoding enc_k.

The procedure `Encode` contains a loop that, whenever it decrements the content of CACHE, it increments the content of the active block (represented by an outgoing dashed edge). This happens until the CACHE content reaches 0, meaning the content of CACHE is correctly store in BLOCKSET. After moving to the new block, the procedure `Decode` is called. It contains a loop that, whenever it decrements the content of the active block, increments the content of CACHE. This terminates once the content of the active block has reached 0, meaning the content of the active block has been correctly stored in CACHE. Hence the host graph is now $\text{enc}_k(S')$. □

4 From the Simulation to Computation Models

Based on the `Sim` programs, we can generalise to computation models that are subsets of GP 2, for which we can define a time measure. One obstacle is time-intensive rule matching. A second one is that when *critical subprograms* (bodies of loops and conditions of `if` and `try`) fail, they need to be undone, which can be handled in different ways. The current implementation of GP 2 reverses the relevant rule applications. The formal semantics uses a stack of graphs to keep track of the state of the host graph before entering a critical subprogram. Reversion can then be done by a simple pop operation. The downside to this is that the host graph has to be duplicated every time a loop or a branching statement is entered, which can add a polynomial factor to the time complexity of the overall program. To avoid undoing, one wants failed critical subprograms to be *null*, i.e. they do not change the graph state, meaning they do not need to be undone.

Models that overcome these obstacles can be implemented to match rules in constant time, which allows us to assign unit time cost to that operation. They also make undoing redundant, allowing us to charge no time cost for that.

Another obstacle is that matching a set of rules is nondeterministic, and backtracking that nondeterminism can be costly in time or space. This can be addressed at the implementation level. The current GP 2 compiler for instance picks the first match it finds and does not backtrack. A more general approach however is to show that within the scope of running a program on an input graph, every rule and rule set (nondeterministic call of a list of rules) can only have at most one match, effectively making the program deterministic. This property applies to the class of programs in this paper and is shown in Proposition 2.

We define an *efficient model* as $\mathscr{M} = \langle \mathscr{P}, \mathscr{I} \rangle$, where $\mathscr{P}$ is a set of GP 2 programs and $\mathscr{I}$ a set of GP 2 graphs such that the following two properties are satisfied within the scope of each derivation sequence starting with some $\langle P, I \rangle \in \mathscr{M}$.

(1) *Constant Matching*: Every rule matches in constant time.

(2) *Critical Subprogram*: Every critical subprogram that fails is null.

Let us define the *graph space* measure of a graph as the number of nodes plus the number of edges. We do not consider the size of labels in this paper since all labels are of constant size. This measure is uniform in that it gives unit cost to each node and edge. A discussion about this uniformity can be found at the end of Section 5.

For an efficient model $\mathscr{M}$, we define the time complexity of a program P of $\mathscr{M}$ as the maximum number of rule calls in terminating derivation sequences starting with P on graphs of a given size.

We show in this section that $\langle P, I \rangle$, where $P = \{\texttt{Sim}(M) \mid M \text{ is a Turing machine}\}$ and $I = \{\text{CG} \mid \text{CG is a configuration graph}\}$, is an efficient model. The constant matching property is given by Theorem 1,

which holds because all rules are fast, input graphs have bounded outdegree and a bounded number of roots, and the programs preserve this boundedness. The critical subprogram property is shown by Proposition 1.

The remainder of this section consists of aforementioned propositions and a lemma needed to prove one of them.

Lemma 1. *Let `P` be a critical subprogram in a graph state S. Assume either `P` cannot fail from state S, or `P` can only fail from state S due its first component being a rule or rule set call that fails to match. Then if `P` fails from state S, `P` is null.*

Proof. If `P` cannot fail from state S, the lemma is trivially satisfied. Now assume `P` fails from S due to its first component being a rule or rule set failing to match. Since it failed to match, the first component cannot have changed the host graph, and since it is the first, no other component can have changed the host graph. □

Proposition 1 (Critical Subprogram Property)**.** *In $\mathtt{Sim}(M)$, given configuration graphs as inputs, every critical subprogram that fails is null.*

Proof. We will argue for each critical subprogram of `Sim` that Lemma 1 applies.

The conditions of all `if` and `try` statements are either rules or rule set calls, satisfying Lemma 1.

Let us now argue for the loop bodies. The procedures `Erase`, `RewindCache`, `HeadLeft` as well as `HeadRight` are rule sets, and `rewind_blockset` is a rule, satisfying Lemma 1.

`Reset`: The rule `overflow` can fail, but the other component cannot since it is a `try` statement whose branches only contain `break` which cannot fail.

`Increment`: This consists of a `try` statement, so only the branches can fail. `Finish` cannot fail because `CacheInit` is always called before this loop, providing a match. The rule `overflow` cannot fail because it is only called when `Inc` fails and the labels of nodes in CACHE are either `0`, `1`, or `2`. The `try` statement cannot fail because its branches cannot fail.

`Decrement`: The reasoning is analogous to that in the previous paragraph.

`Decoding`: The `try` statement cannot fail because its branches cannot fail. For the remainder of this paragraph, we argue that `CacheInc` cannot fail either. `Increment!` cannot fail because it is a loop. `CacheInit` always succeeds because the target of the red edge originating from the unmarked root is unmarked because of the structure of the input graph, and because previous calls of `CacheInc` and `CacheDec` turn the only marked rooted node in this part of the graph unmarked and unrooted with `Finish`.

`Encoding`: The reasoning is analogous to that in the previous paragraph.

The body of the loop contained in `ResetBlockset`: The rule `copy` is allowed to fail, and `direct` cannot fail since the roots are present due to `binit`, and the unmarked node is present due to at least one application of `copy`.

`Simulate`: `Transition` is a rule set at the start of the body and hence allowed to fail. The rest consists of `try` statements whose branches call `PrevBlock` and `NextBlock`. `Prev` and `Next` cannot fail because we assume that BLOCKSET in the input graph is large enough to accommodate the execution of the Turing machine. `Encode` and `Decode` do not fail, which we will argue for the rest of this paragraph. `EncodeInit` and `DecodeInit` find a match because of the structure of the input graph and because previous calls of `Encode` and `Decode` leave BLOCKSET unmarked and unrooted. `Encoding!` and `Decoding!` are loops and hence cannot fail. `Update` has a match regardless of the blue root's location within BLOCKSET. □

Proposition 2 (Unique Matches)**.** *In $\mathtt{Sim}(M)$, given configuration graphs as inputs, whenever a rule or rule set is called, there is at most one match.*

Proof. In each rule set, if the left-hand side of two rules have the same structure, their labels are different. In `Transitions` in particular, no two rules have the same left-hand side since we consider deterministic Turing machines. Among rules with different labels but the same structure, at most one can match, namely the one that has the same labels as the host graph. Among rules with different structures and the right labels, at most one can match because they differ by an edge or node that is unique in host graphs. Furthermore, it is easy to check that each rule can only have at most one match in the host graph. Each rule contains a root with a unique combination of mark and label. And for each node, outgoing edges can be distinguished in the same way. □

5 Results on Time and Space Complexity

In this section, we present theorems on the time and space complexities of the simulation.

Theorem 3 (Time Complexity)**.** *Every Turing machine M of time complexity $t(n)$ is simulated by $\mathtt{Sim}(M)$ in time $\mathrm{O}(t^2(n))$, where n is the size of the input.*

Proof. Given the discussion in Section 4, we can assign unit time to rule and rule set calls and argue time complexity to be the number of such calls.

First, we will show that simulating one step of M (not counting restarting the simulation) takes $\mathrm{sim}(n) = \mathrm{O}(s(n))$ time. We consider the size of CACHE from the final simulation since it provides an upper bound. The only sources of non-constant time are `PrevBlock` and `NextBlock`. Their complexity is the worst of the loops `Encoding!`, `Decoding!`, `HeadLeft!`, and `HeadRight!`. The latter two simply traverse CACHE, which takes $\mathrm{O}(s(n))$ time. The former two decrement/increment CACHE as a ternary counter. Their time complexity is proportional to the number of digit operations it takes to decrement the counter from the number $s(n)$ all the way to 0. The rightmost digit is modified $s(n)$ times, the next one $\frac{1}{3}s(n)$ times, the one after $\frac{1}{3}\cdot\frac{1}{3}s(n)$ times, and so on. So the total number of digit operations is $\sum_{k=0}^{\log s(n)} s(n)\left(\frac{1}{3}\right)^k$. Using properties of the geometric series, one can see that this is $\mathrm{O}(s(n))$.

In this paragraph, we will show that resetting the simulation and extending the tape takes $r(n) = \mathrm{O}(s(n))$ time. Consider the loops of `Restart`. Both `RewindCache!` and `Erase!` traverse CACHE and hence take time $\mathrm{O}(\log s(n))$. The loop in `ResetBlockset` traverses BLOCKSET and thus takes $\mathrm{O}(s(n))$ time.

Next, we will show that the number of times the simulation is restarted is $l(n) = \mathrm{O}(\log s(n))$. The final size of the tape of M is $\mathrm{O}(s(n)\log s(n))$. Since the number of represented squares is tripled in each step, the number of steps is $\mathrm{O}(\log(s(n)\log s(n)))$. Using the formula for the logarithm of a product, one can simplify this to $\mathrm{O}(\log s(n))$.

The total time complexity can be bounded by $\mathrm{reset}(n) + \mathrm{simulation}(n)$, where $\mathrm{reset}(n) = l(n)\cdot r(n)$ is the total cost of all resets, and $\mathrm{simulation}(n) = l(n)\cdot t(n)\cdot \mathrm{sim}(n)$ the total cost of simulating M across all resets. Using results from previous paragraphs, we get $\mathrm{reset}(n) = \mathrm{O}(s(n)\log s(n))$ and $\mathrm{simulation}(n) = \mathrm{O}(t(n)\cdot s(n)\log s(n))$. Since space complexity $s(n)\log s(n)$ can be bounded by time complexity $t(n)$, the entire simulation takes $\mathrm{O}(t^2(n))$ time.

□

Theorem 4 (Space Complexity)**.** *Every Turing machine M of space complexity $\mathrm{O}(s(n)\log s(n))$ is simulated by $\mathtt{Sim}(M)$ in graph space $O(s(n))$, where n is the size of the input.*

Proof. During the execution of $\texttt{Sim}(M)$, nodes and edges are only created by `setup` and `Restart`, and none are ever deleted. The numbers of nodes and edges only differ by a constant factor since the outdegree is bounded, so we will argue for space complexity using number of nodes only.

Initially, after application of `setup`, BLOCKSET has $b(n) = 3^2$ nodes, and CACHE $c(n) = 2$. Then, each application of `Restart` adds one to $c(n)$ and triples $b(n)$. So after k iterations, we have $b(n) = 3^{2+k}$ and $c(n) = 2+k$.

The Turing machine needs $S(n) = \mathrm{O}(s(n)\log s(n))$ tape squares. This means that there are positive integers n_0 and c such that $S(n) \leq c\,s(n)\log s(n)$ for all $n \geq n_0$. So for all n we can say $S(n) \leq c\,s(n)\log s(n) + m$, where $m = \max_{n\in\{0,\ldots,n_0\}} S(n)$, a constant.

Assume `Restart` is called $k = \log s(n) - 2 + d$ times, where $d = \max(m, \log c)$. For that value of k, we have $c(n)b(n) = 3^d s(n)\log s(n) + 3^d s(n)\,d$. By definition we have $d \geq m$ and $3^d \geq c$. Furthermore, we have $3^d s(n) \geq 1$. Hence we get $c(n)\cdot b(n) \geq c\,s(n)\log s(n) + m \geq S(n)$. So for this value of k, the graph can store enough tape squares to execute M.

With the aforementioned value of k, the number of nodes in this graph is $c(n) + b(n) = 3^d s(n) + \log s(n) + d$, which is $\mathrm{O}(\log s(n) + s(n)) = \mathrm{O}(s(n))$.

Hence the number of nodes of the graph that is created is bounded by $c(n) + b(n)$, and hence in $\mathrm{O}(s(n))$. □

One might wonder why this space compression is not possible on random access machines. GP 2 does have a C implementation after all. The reason for this is related to how graphs are represented in random access machines (RAMs), and how much space that takes. Graph edges are usually implemented as pointers. However the size of pointer addresses grows logarithmically with the number of nodes, since these addresses are usually stored as binary numbers. So in the context of RAMs, it does not seem very accurate to assign unit cost to edges. To take this into account, one can use *logarithmic space*, in which graphs of $s(n)$ nodes are assigned a cost of $s(n)\log s(n)$.

If we use logarithmic space on our model, space compression is nullified since the simulation then has the same asymptotic space complexity as the machine it simulates. This puts into question whether uniform or logarithmic space should be used, which is discussed by Van Emde Boas [15]. One may want to charge more than unit space since in RAMs, edges are represented by pointers whose size grows with the number of nodes. A related issue can be found in the time measure of RAM models when programs have to deal with large integers.

6 Conclusion

In GP 2, we have found the same space compression phenomenon that SMMs and KUMs exhibit. Unlike the other computation models, GP 2 uses rule-based graph transformation. The compression happens due to use of a graph-based data structure to encode the tape of a Turing machine, which is more space-efficient with respect to uniform measures.

Using this simulation, we have identified efficient computation models that are subsets of GP 2. Requiring these models to have certain properties allows us to define rigorous complexity measures.

In future work, the simulation could be refined in the way Luginbuhl [10] refines Van Emde Boas [15], namely with a more complex data structure that allows for real-time simulation with some form of space compression.

Furthermore, the simulation could be extended to nondeterministic Turing machines. Extending the simulation would be straightforward, but the complexity results would need further proofs.

References

[1] Alfred Aho, John Hopcroft & Jeffrey Ullman (1974): *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[2] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York, UK. Available at `https://etheses.whiterose.ac.uk/12586/`.

[3] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs*. In: *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54, doi:10.14279/tuj.eceasst.54.780.

[4] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C*. In: *Proc. International Conference on Graph Transformation (ICGT 2016)*, *Lecture Notes in Computer Science* 9761, Springer, pp. 102–117, doi:10.1007/978-3-319-40530-8_7.

[5] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. *Science of Computer Programming* 214, doi:10.1016/j.scico.2021.102727. 32 pages.

[6] Edsger W Dijkstra (1968): *Letters to the editor: go to statement considered harmful*. *Communications of the ACM* 11(3), pp. 147–148.

[7] Joseph Y. Halpern, Michael C. Loui, Albert R. Meyer & Daniel Weise (1986): *On time versus space III*. *Mathematical systems theory* 19, pp. 13–28, doi:10.1007/BF01704903.

[8] Ivaylo Hristakiev & Detlef Plump (2016): *Attributed Graph Transformation via Rule Schemata: Church-Rosser Theorem*. In: *Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers*, *Lecture Notes in Computer Science* 9946, Springer, pp. 145–160, doi:10.1007/978-3-319-50230-4_11.

[9] A. N. Kolmogorov & V. A. Uspenskii (217–245): *On the definition of an algorithm*. *American Mathematical Society Translations: Series 2* 29, pp. 217–245, doi:10.1090/trans2/029.

[10] David R. Luginbuhl & Michael C. Loui (1993): *Hierarchies and Space Measures for Pointer Machines*. *Information and Computation* 104(2), pp. 253–270, doi:10.1006/inco.1993.1032.

[11] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *Electronic Proceedings in Theoretical Computer Science* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[12] Detlef Plump (2017): *From Imperative to Rule-based Graph Programs*. *Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173, doi:10.1016/j.jlamp.2016.12.001.

[13] A. Schönhage (1980): *Storage Modification Machines*. *SIAM Journal on Computing* 9(3), pp. 490–508, doi:10.1137/0209036.

[14] Steven Skiena (2020): *The Algorithm Design Manual*, 3rd ed. edition. Texts in Computer Science, Springer, doi:10.1007/978-3-030-54256-6.

[15] Peter van Emde Boas (1989): *Space measures for storage modification machines*. *Information Processing Letters* 30(2), pp. 103–110, doi:10.1016/0020-0190(89)90117-8.

A Foundation for Functional Graph Programs: The *Graph Transformation Control Algebra* (GTA)

Jens H. Weber
Department of Computer Science
University of Victoria
Victoria, Canada, BC
jens@acm.org

Many applications of graph transformation (GT) systems require or benefit from control structures that can be used to restrict and direct GT processes. Most existing GT tools follow a stateful computational model, where a single graph is repeatedly modified *in-place* when GT rules are applied. The implementation of control structures in such tools is not trivial. Common challenges include dealing with the non-determinism inherent to rule application and transactional constraints when executing compositions of GTs, in particular atomicity and isolation. The complexity of associated transaction mechanisms and rule application search algorithms (e.g., backtracking) complicates the definition of a formal foundation for these control structures. Compared to these stateful approaches, *functional* graph rewriting presents a simpler (stateless) computational model, which simplifies the definition of a formal basis for (functional) GT control structures. In this work-in-progress paper, we propose the *Graph Transformation control Algebra* (GTA) as such a foundation. The GTA has been used as the formal basis for implementing the control structures in the (functional) GT tool *GrapeVine*.

1 Introduction

There are diverse practical applications of graph transformations (GT) in software engineering, computer science and beyond [7]. They often require or benefit from *control structures* that can be used to restrict and direct the application of GTs. Heckel and Taentzer categorize four approaches for controlling the application of GTs, including non-terminals, dedicated control expressions, integrity constraints, and procedural abstraction [6]. Real-world applications often use a combination of these approaches. Most existing GT tools follow a stateful model of computation where rule applications and control expressions mutate the program state. Multiple rule applications may need to be grouped into *atomic* units that are executed entirely or not at all and in isolation of other rule applications. Finding solutions with non-deterministic rule applications may require search mechanisms like backtracking and heuristic optimization. Implementing control structures that provide such sophisticated mechanisms is not trivial and their semantics is often not formalized, which impedes formal reasoning about GT programs.

It is well known that stateless computation provides for a simpler semantic model when compared to stateful computation. We recently presented a functional GT tool (called *GrapeVine*) that implements a stateless computational model [20]. This paper presents the formal foundation for the control structures provided in this tool. We use an algebraic approach to define a set of operations referred to as the *Graph Transformation control Algebra* (GTA). In addition to providing a precise semantics for the control structures provided by *GrapeVine*, we hope that the GTA will enable future research on functional GT program analysis and optimization, analogous to the use of the relational algebra for database query optimization.

The rest of this work-in-progress paper is structured as follows. We start with a brief introduction to graphs, graph transformations and graph constraints. Sec. 3 provides a description of related work

on control structures for GT programs. We introduce the *Graph Transformation control Algebra* (GTA) and our notion of programmed graph transformation systems in Sec. 4. Sec. 5 provides a description of an implementation of the control structures in the tool *GrapeVine*, which are based on the GTA. Finally, we offer conclusions and discuss current work in Sec. 6.

2 Preliminaries

For the purpose of this paper, we limit ourselves to a definition of basic forms of graphs and graph transformations (GTs). While many GT tools (including *GrapeVine*) provide more advanced concepts (e.g., labels, attributes, typed graphs), the definitions below can easily be extended accordingly. The reader is referred to [3] for a more complete introduction to graph transformations.

Definition 1 (Graph)**.** *A graph is a tuple $G:(N,E,s,t)$ where N is a finite set of* nodes*, E is a finite set of* edges*, and $s,t:E\to N$ are total* source *and* target *functions, respectively.*

Definition 2 (Rule)**.** *A (GT)* rule *is defined as a pair of graph morphisms $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$, which map elements of a so-called* interface *graph I to elements in graph L (called the* left-hand side*) and graph R (called the* right-hand side*), respectively.*

Note that while the above definition of rules is more general, GT rules in our tool *GrapeVine* require φ_L and φ_R to be injective. Given this restriction, a rule's interface I is a subgraph of its left- and right-hand side. For simplicity, we (ab)use set notation to express this property, i.e., $I\subseteq R, I\subseteq L$.

Definition 3 (Transformation)**.** *An application of a rule $r:L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ to a given host graph G requires a* match *of the left-hand side in G, i.e., the existence of a graph morphism $L \xrightarrow{m} G$. The application deletes all elements from G that are matched to elements that only appear on the rule's left-hand side ($L-I$), creates elements for any elements that appear only on the rule's right-hand side ($R-I$), and embeds these new elements in the preserved context ($m(I)$). A more formal definition of a rule application based on category theory (double-pushout approach) is provided by Corradini et al. [2]. To avoid problems that may occur when deleting graph elements, a match has to satisfy the so-called* gluing condition*, which consists of two parts: (1) the embedding context (attached edges) of any deleted nodes must also be identified and deleted by the rule (*dangling condition*), and (2) deleted element must only have one pre-image in L (*identification condition*).*

The transformation of a given graph G into a graph G' by applying rule r at a valid match m is denoted as $G \overset{r,m}{\rightsquigarrow} G'$.

As mentioned before, we opted against adding more advanced concepts to our formalization of rules and transformations, e.g., (negative) application conditions and rule parameters. While *GrapeVine* provides these concepts, they are not the focus of this paper. We do, however, introduce the notion of *graph constraints*,[11] as constraints are one of the mechanisms used to control the application of GT rules [6].

Definition 4 (Atomic Constraint)**.** *An atomic (graph) constraint $I \xrightarrow{c} T$ is defined as a graph monomorphism between two graphs I and T. A graph G satisfies c (denoted as $G\vDash c$) if for every monomorphism $I \xrightarrow{h} G$ there is a monomorphism $T \xrightarrow{f} G$ such that $h = f\circ c$.*

Definition 5 (Constraint)**.** *A (graph) constraint is inductively defined as either an atomic graph constraint $c:I\to T$, a negation of a graph constraint (i.e., if κ is a graph constraint, so is $\neg\kappa$), or the disjunction of two graph constraints (i.e., if κ_1 and κ_2 are graph constraints, so is $\kappa_1\vee\kappa_2$).*

Now that we have introduced constraints, we extend our definition of graphs and transformations to constrained graphs and transformations on constraint graphs.

Definition 6 (Constrained Graph)**.** *A constrained graph is a tuple* (G,K) *where G is a graph and K is a finite set of graph constraints satisfied by G, i.e,* $\forall \kappa \in K : G \vDash \kappa$.

Definition 7 (Transformation (of a constrained graph))**.** *A transformation of a constrained graph* $(G,K) \overset{r,m}{\rightsquigarrow} (G',K)$ *exists, if there exists a corresponding transformation for the unconstraint graph* $G \overset{r,m}{\rightsquigarrow} G'$*, where the resulting graph satisfies all constraints, i.e.,* $\forall \kappa \in K : G' \vDash \kappa$.

From here on, we only consider constrained graphs and transformations on constrained graphs. The reader should therefore assume that we are referring to a "constrained graph" whenever we use the term *graph* without explicitly mentioning otherwise.

A transformation $G \overset{r,m}{\rightsquigarrow} G'$ is also called a *direct linear derivation*. In general, there are many possible matches for a rule in a given graph, i.e., multiple direct linear derivations are possible. We may just write $G \overset{r}{\rightsquigarrow} G'$ to denote any direct linear derivation, if we do not care about the match.

3 Related work

Most GT tools offer some constructs to control the application of GT rules. The inherent complexity related to stateful computation and non-deterministic choice between possible derivations complicates the implementation of GT control structures. To the best of our knowledge, *PROGRES*[16], *Grape* [18], and *GP* [12] appear to be the only GT tools that implement backtracking mechanisms, capable of reconstructing graphs at non-deterministic choice points. The PROGRES project has since been discontinued since its platform is no longer available. *Grape* is still available but no longer under further development, in part due to the complexity of maintaining the implemented backtracking system. Work on *GP* appears to continue but the tool is described as experimental and not publicly available (to our knowledge). The execution of *GP* programs requires the York Abstract Machine (YAM) which handles a sophisticated data structure to keep track of non-determinism, choice points and environment frames.

PROGRES and *GP* provide a formalized semantics for their control structures. The semantics definition for *PROGRES* is complex and spans over three hundred pages. The formalization of *GP* is comparably simple. The original language (*GP1*) consist of only four types of commands: non-deterministic rule application, sequential composition, branching, and iteration [12]. Its successor (*GP2*) provides additional constructs, including a second kind of branching statement and an explicit operator for non-deterministic choice between sub-programs [13]. Moreover, GP2 changed the semantics control structures in order to allow for efficient implementation of branching and looping. In particular, failures in conditions of branching statements or loop bodies no longer enforce backtracking behaviour. While *GP* programs theoretically associate all reachable output graphs to a given input graphs, the program computation may not succeed. This is because the GP search mechanism uses depth-first search and the program may diverge [12].

AGG is another GT tool with a formally defined execution semantics [17, 15]. However, AGG avoids backtracking altogether and uses a random selection when making a non-deterministic choice during rule application. *FUJABA* has control structures based on a combination of UML activity diagrams and GT rules [10]. The semantics has not been defined formally and there is no backtracking. *GrGen* provides a textual language for controlling GT rule applications but also without a formal semantics and backtracking [4].

GReAT is a GT tool with dedicated support for model transformations [1]. It is notably different from other tools in that it support arbitrarily many input and output graphs. Its control structures use a combination of control-flow diagrams, structuring constructs and OCL-based model constraints.

4 Programmed Graph Transformation Systems with GTA

This section is divided in two parts. We begin by defining our notion of a programmed graph transformation system (GTS) and then discuss our rational and reasoning behind the choices we made.

Definition 8 ((programmed) Graph Transformation System (GTS))**.** *We define a (programmed)* Graph Transformation System *(GTS) as a tuple* (R,C,P)*, where R is a set of* rules*, C is a set of* constraints*, and P is a set of graph* p*rograms.*

In this paper, we are mainly concerned with defining a formal basis for the programs (P) in a GTS. We define the *graph transformation control algebra* (GTA) for that purpose. GTA operators work on a common data type, a sequence of graph sets, referred to as a **grap***h set* **e***numeration* (or "*grape*" for short) in the rest of this paper. We will define this data type first. Let $\mathbb{G}$ denote the domain of graphs.

Definition 9 (Graph set enumeration (grape))**.** *A* graph set enumeration *(or* grape *for short) is a non-empty sequence* $\ddot{G} : \langle \bar{G}_1, \bar{G}_2, .., \bar{G}_n \rangle$ *where each* $\bar{G}_i$ *is a finite set of graphs, i.e.,* $\bar{G}_i \subset \mathbb{G}$*. Let* $\ddot{\mathbb{G}}$ *denote the domain (data type) of* grapes*.*

The GTA has twelve operators representing binary relations on *grapes*. We first introduce their syntax and then proceed to defining their semantics.

Definition 10 (Graph Program (syntax))**.** *Given a GTS* (R,C,P)*,* $c \in C$ *and* $r \in R$*, each program* $p \in P$ *is a GTA expression, which is defined as one of the following:*

- $\triangle(c)$ *and* $\boxtimes(c)$ *are GTA expressions, called* constrain *and* unconstrain*, respectively ;*
- $\twoheadrightarrow(r)$ *and* $\rightsquigarrow(r)$ *are GTA expressions, called* derive *and* derive-choice*, respectively;*
- $\circledcirc(n, \lesssim)$ *with* $n \in \mathbb{N}$ *and a total order* $\lesssim$ *on graphs is a GTA expression, called* select*;*
- $\cdot|\cdot(e_1,e_2)$, $\div(e_1,e_2)$ *and* $\dot{\div}(e_1,e_2)$ *are GTA expressions, if* e_1 *and* e_2 *are GTA expressions; They are called* sequence*,* alternative*, and* alternative-choice *respectively;*
- $\circlearrowleft(e)$ *is a GTA expression called* loop *if e is a GTA expression;*
- $\looparrowright(c,e)$ *is a GTA expression called* search *if e is a GTA expression;*
- $)($ *and* $\not\equiv$ *are GTA expressions called* new *and* distinct*, respectively.*

Graph programs that make use of any of the two "choice" expressions ($\rightsquigarrow$ *and* $\dot{\div}$*) are called* non-deterministic*, and* deterministic*, otherwise.*

Definition 11 (Deterministic Graph Program (semantics))**.** *A deterministic graph program p is interpreted as a function mapping* grapes *to* grapes*, i.e.,* $[\![p]\!] : \ddot{\mathbb{G}} \to \ddot{\mathbb{G}}$*. The program semantics is based on the interpretation of the individual GTA operators as functions* $\ddot{\mathbb{G}} \to \ddot{\mathbb{G}}$*. Most of them are interpreted as total functions, with the exception of* loop *and* search *(*$\circlearrowleft, \looparrowright$*) which are not guaranteed to terminate.*

- **Constrain and Unconstrain (**$\triangle$ **and** $\boxtimes$**)**
  The purpose of these functions is to declare and undeclare graph constraints, respectively. $\triangle(c)$ *declares constraint c on all the graphs in the last element of a given* grape *that satisfy c. All other graphs are removed.*
  $[\![\triangle(c)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}'_n \rangle$ *with* $\bar{G}'_n = \{(G, K + \{c\}) \,|\, (G,K) \in \bar{G}_n \wedge G \vDash c\}$
  $\boxtimes(c)$ *removes constraint c from the graphs in the last element of a grape:*
  $[\![\boxtimes(c)]\!](\langle .., \bar{G}_n \rangle) = \langle .., \bar{G}'_n \rangle$ *and* $\bar{G}'_n = \{(G, K - \{c\}) \,|\, (G,K) \in \bar{G}_n\}$
  Please note that whenever we use the notation of $\langle .., \rangle$ *on two sides of a definition (like above), we require that ".." stands for the same sequence of elements. We use the notation "..a" and "..b" to refer to different sequences of elements.*

- **Derive ($\twoheadrightarrow$)**
  Function derive *computes all direct linear derivation of each graph in the last element of a given* grape *and extends the given input* grape *with an element that contains* all *resulting graphs, i.e.,* $[\![\twoheadrightarrow(r)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}_n,\bar{G}_{n+1}\rangle$ *where* $\bar{G}_{n+1} = \{G' | \exists G \in \bar{G}_n : G \overset{r}{\rightsquigarrow} G'\}$
- **Select ($\circledcirc$)**
  Function Select $(\circledcirc(k,\lesssim))$ *reduces the last element of a given* grape *to at most k elements. The selection is determined by a total order on graphs* $\lesssim$*. Formally,* $[\![\circledcirc(k,\lesssim)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}'_n\rangle$*, with* $\bar{G}'_n \subseteq \bar{G}_n \wedge |\bar{G}'_n| \leq k \wedge |\bar{G}'_n| \leq |\bar{G}_n| \wedge \nexists G \in \bar{G}_n - \bar{G}'_n, G' \in \bar{G}'_n : G' \lesssim G$
- **Sequence ($\cdot|\cdot$)**
  $\cdot|\cdot(a,b)$ *composes two GTA expressions sequentially by using relational composition, i.e.,* $[\![\cdot|\cdot(a,b)]\!] = \{(\dddot{G},\dddot{K}) \in \dddot{\mathbb{G}} | (\dddot{G},\dddot{H}) \in [\![a]\!] \wedge (\dddot{H},\dddot{K}) \in [\![b]\!]\}$.
- **Alternative ($\div$)**
  $\div(a,b)$ *composes two GTA expressions (a and b) as alternatives by extending a given* grape *with a new element that is the union of the last elements of the* grapes *produced by interpreting the two expressions, i.e.,* $[\![\div(a,b)]\!](\dddot{G} : \langle..x\rangle) = \langle..x,\bar{O}_1 \cup \bar{O}_2\rangle$ *with* $[\![a]\!](\dddot{G}) = \langle..y,\bar{O}_1\rangle$ *and* $[\![b]\!](\dddot{G}) = \langle..z,\bar{O}_2\rangle$.
- **Distinct $\not\equiv$**
  Graph exploration may produce graphs that are identical (up to isomorphism). The distinct *operator ($\not\equiv$) removes all graphs from the last element of a given* grape*, if they are identical to any other graph in the* grape*. Formally, $\not\equiv$ is defined as*

$$[\![\not\equiv]\!](\langle\bar{G}_1,..,\bar{G}_n\rangle) = \begin{cases} \langle\bar{G}_1,..,[\![\not\equiv]\!](\bar{G}_n - \{D\})\rangle, & \text{if } \exists D,J \in \bigcup_{1\leq i\leq n}\bar{G}_i : D \neq J \wedge D \cong J \\ \langle\bar{G}_1,..,\bar{G}_n\rangle, & \text{otherwise} \end{cases}$$

- **New ($\rangle($)**
  The new *operator is used to start a new* grape*. It takes a* grape *as input but "forgets" all but the last element in the sequence. This is useful to restrict the distinct's operators ability to "look back" in the derivation history to find identical graphs.* new *is defined as* $[\![\rangle(]\!](\langle..,\bar{G}_n\rangle) = \langle\bar{G}_n\rangle$.
- **Loop ($\circlearrowleft$)**
  $\circlearrowleft\ (e)$ *is interpreted as a function that recursively interprets GTA expression e on the most recently computed* grape *while the last element is not empty, i.e,*

$$[\![\circlearrowleft\ (e)]\!](\dddot{G}) = \begin{cases} \dddot{G}, & \text{if } [\![e]\!](\dddot{G}) = \langle..,\varnothing\rangle \\ [\![\circlearrowleft\ (e)]\!] \circ [\![e]\!](\dddot{G}) & \text{otherwise} \end{cases}$$

- **Search ($\looparrowright$)**
  $\looparrowright\ (c,o)$ *is interpreted as a (recursive) function that repeatedly interprets a GTA expression o on the most recently computed* grape *while none of the graphs in the last element of the current* grape *satisfy constraint c and the last element is not empty, i.e,*

$$[\![\looparrowright\ (c,o)]\!](\dddot{G} : \langle..,\bar{G}_n\rangle) = \begin{cases} \dddot{G}, & \text{if } \bar{G}_n = \varnothing \vee \exists G \in \bar{G}_n : G \vDash c \\ [\![\looparrowright\ (c,o)]\!] \circ [\![o]\!](\ddot{G}) & \text{otherwise} \end{cases}$$

Definition 12 (Non-Deterministic Graph Program (semantics))**.** *A non-deterministic graph program p defines a binary relation between grapes, i.e.,* $[\![p]\!] \subset \dddot{\mathbb{G}} \times \dddot{\mathbb{G}}$*. The two* additional *GTA operators available to non-deterministic programs are $\rightsquigarrow$ and $\dot{\div}$, which are also interpreted as binary relations on* grapes*.*

- **Derive-choice ($\rightsquigarrow$)**
  $\rightsquigarrow$ is interpreted as a relation that extends the classical notion of non-deterministic rule application to the data type of grapes*. From an operational perspective, this operation computes a direct linear derivation of each graph in the last element of a given* grape *and returns a copy of the input* grape *with an added element that contains all resulting graphs.*
  $[\![\rightsquigarrow(r)]\!] = \{(\langle .., \bar{G}_n\rangle, \langle .., \bar{G}_n, \bar{G}_{n+1}\rangle) \in \ddot{\mathbb{G}} \times \ddot{\mathbb{G}} | \forall G \in \bar{G}_n : ((\exists! X \in \bar{G}_{n+1} : G \overset{r}{\rightsquigarrow} X) \vee (\nexists Y \in \mathbb{G} : G \overset{r}{\rightsquigarrow} Y)) \wedge |\bar{G}_{n+1}| \leq |\bar{G}_n|\}$.

- **Alternative-choice ($\dot{\div}$)**
  $\dot{\div}$ is interpreted as a relation that makes a non-deterministic choice between the relations implied by the two GTA expressions, i.e., $[\![\dot{\div}(a,b)]\!] = [\![a]\!] \vee [\![\dot{\div}(a,b)]\!] = [\![b]\!]$.

Properties of the GTA and Rationale

Habel and Plump have shown that any graph programming language is computationally complete, which is capable of nondeterministic rule application, sequential composition of program statements and iteration over program statements [5]. They also showed that this set of operators is minimal. The rule application operator ($\Rightarrow_R$) in their minimal list incorporates both forms of non-determinism that are usually associated with GT systems, i.e., non-deterministic rule selection and non-deterministic rule matching. The GTA has separate operators for rule choice ($\div$, respectively $\dot{\div}$) and rule matching ($\twoheadrightarrow$, respectively $\rightsquigarrow$). However, it is easy to see that the semantics of operator $\Rightarrow_R$ can be achieved by combining the two non-deterministic GTA operators $\dot{\div}$ and $\rightsquigarrow$.

Since GTA has operators for sequential composition and iteration, it follows that the set of operators $\{\rightsquigarrow, \dot{\div}, \cdot|\cdot, \circlearrowleft\}$ is computationally complete (and minimal). Substituting the non-deterministic GTA operators with their deterministic counterparts will obviously also yield a computationally complete (and minimal) set of operators $\{\twoheadrightarrow, \div, \cdot|\cdot, \circlearrowleft\}$.

The reason for including both non-deterministic and deterministic operators for rule application and choice in the GTA is that we want to provide the programmer with control over and assurances about the GT process. While the notion of non-determinism is theoretically simple and appealing, since it defers decisions about concrete execution to the design of the underlying engine, it also becomes problematic for practical applications. This is because programmers cannot make assumptions on how non-determinism is resolved. Non-deterministic choice must be implemented with deterministic algorithms (unless true randomness is used). Choices made during program execution may prevent computation of a desired result, unless the execution engine has mechanisms to "track back" and explore other options. As mentioned earlier, few GT tools have that capability due to complexity of the associated execution engine. Even with backtracking, graph programs may diverge during depth-first exploration even if a solution is theoretically computable. Moreover, backtracking is expensive and may lead to efficiency problems if performed by default [13]. This is the reason why GP allowed programmers to disable backtracking and why GP2 has changed the semantics of its branching and looping operators to no longer enforce backtracking by default. To some degree, including the two "choice" operators in the GTA serves a similar purpose. They can be used when it is sufficient to find *a* possible choice rather than needing to explore the range of possible choices.

Of course, breadth-first exploration has its own challenges due to the well known "state-space explosion problem". To make this feasible, the GTA has operators for reducing the search space. In particular the *Distinct* operator ($\not\equiv$) was introduced to detect "collisions". This operator is the main reason to consider *grapes* rather than simple graph sets as the common data type for computation. The *Select* operator

(◎) provides a way to limit the search to an upper number of ranked graphs to explore (based on a given total order). For similar reasons (to restrict the search space), we included graph constraints in addition to GT rules. *Constrain* (△) and *Unconstrain* (⧖) expressions provide programs with ways to (temporarily) restrict the exploration of possible computation.

5 Implementing the GTA within *GrapeVine*

An implementation of GTA-based graph programs requires a functional GT tool in which graphs are immutable first-class objects rather than treating *the* graph as "singleton" global variable for stateful computation. We have recently proposed *GrapeVine* as a tool to meet this criterion [20]. We start this section with a short introduction to this tool, before discussing graph programs in *GrapeVine*.

5.1 *GrapeVine* - A short overview

GrapeVine is implemented on top of the Neo4J graph database and derives much of its scalability from that architecture. The tool is a fundamentally new revision of the earlier tools *Grape* [18] and *Grape Press* [19]. Compared to these earlier tools, the main novelty in *GrapeVine* is its functional computational model, where graphs are considered immutable objects.

GrapeVine uses a textual language for defining GT rules, constraints and programs. That language is provided as an internal DSL (domain-specific language) to the general-purpose programming language Clojure. *GrapeVine* has been integrated with the *Gorilla*. Gorilla is a browser-based computational notebook for Clojure, similar to other computational notebook technologies, like for example Jupyter. Using this platform, *GrapeVine* supports visualization of graphs, rules, and constraints (among other things) [19]. *GrapeVine* can also be used independently of the computational notebook UI; *GrapeVine* programs are regular Clojure programs, which run on the Java Virtual Machine (JVM) and can thus integrate with any other JVM language, such as Java, Kotlin, Scala, Groovy, etc.

5.2 *GrapeVine* control structures

When implementing GTA-based control structures for *GrapeVine* we used the constructs provided by the host language (Clojure) as much as possible and avoided the creation of unnecessary "syntactic sugar". For example, rule (derivations) can simply be called by the name of the rule, i.e., declaring a rule creates rule application derivation functions with that name. Similarly, declaring a constraint automatically creates functions to check that constraint (positively and negatively).

Tab. 1 provides a correspondence between the syntax used in *GrapeVine* (Clojure) graph programs and the abstract syntax of GTA expressions. As mentioned above, rule application is simply a call to a function with the rule's name (with an added tilde if non-deterministic rule application is sufficient). Similarly, constraint checks are done by calling a constraint by name (with an added minus sign for negation). Graphs are constrained and unconstrained using the `schema/schema-drop` forms. In addition to a simple "While possible" loop, *GrapeVine* has two variations of an "Until" loop (based on the ↬ operator). The behaviour of these variants is similar but they differ in one detail, namely whether or not collisions are avoided (using the GTA ≢ operator). The reason for including an "Until" operator without collision check is to accommodate conditional looping behaviour in programs with side-effects, e.g., programs that wait for input events.

Description	GTA expression	*GrapeVine* Syntax
Rule application (deterministic)	$\twoheadrightarrow(r)$	`r`
Rule application (non-deterministic)	$\rightsquigarrow(r)$	`r~`
Add constraint c	$\triangle(c)$	`(schema c ..)`
Add constraint negated c	$\triangle(\neg c)$	`(schema c- ..)`
Remove constraint c	$\boxtimes_c$	`(schema-drop c ..)`
Remove negated constraint c	$\boxtimes(\neg c)$	`(schema-drop c- ..)`
Check constraint c	$\cdot\shortmid\cdot(\triangle(c),\boxtimes(c))$	`c`
Check negated constraint c	$\cdot\shortmid\cdot(\triangle(\neg c),\boxtimes(\neg c))$	`c-`
Sequence	$\cdot\shortmid\cdot(e_1,e_2)$	`(-> e_1 e_2 ..)`
Alternative (deterministic)	$\div(e_1,e_2)$	`(|| e_1 e_2 ..)`
Alternative (non-deterministic)	$\dot{\div}(e_1,e_2..)$	`(||~ e_1 e_2 ..)`
Loop (while possible)	$\circlearrowleft(e)$	`(->* e ..)`
Until (without collisions check)	$\curlyvee(c,e)$	`(->?* c e ..)`
Until (with collision check)	$\curlyvee(c,\cdot\shortmid\cdot(e,\not\equiv))\circ\,)\!($	`(->?+ c e ..)`
New	$)\!($	`newgrape`
(creates a *grape* with a single element containing an empty graph if called without argument)	$)\!(\,(\langle(\varnothing,\varnothing)\rangle)$	`(newgrape)`
Distinct	$\not\equiv$	`dist`
Select	$\circledcirc(k,v)$	`(select k v)`

Table 1: Correspondences between *GrapeVine* control structures and GTA expressions

While the *Distinct* operator is automatically used in one of the looping constructs (`->+`), *GrapeVine* also provides a `dist` control structure, which can be used to remove duplicate graphs in longer sequences of operations.

The `newgrape` function is overloaded (for convenience). If called without any parameter (without a *grape*), it returns a new *grape* with a single element that contains a single, empty, unconstrained graph.

Note that all *GrapeVine* control structures are variadic, i.e., they allow an arbitrary number of parameters. This is indicated by the ellipses in right column of Tab. 1.

5.3 *GrapeVine* control structures - a simple example

Let us look at a simple example of a graph program in *GrapeVine* to get an impression. We use the example of the well-known ferryman problem [21]. In that problem, a ferryman needs to (safely) transport three things (a goat, a wolf, and a cabbage) from one side to the other side of a river. He can only transport one thing at a time. The wolf will eat the goat and the goat will eat the cabbage, respectively, if left unattended.

Fig.1 shows three rules and three constraints modelled within *GrapeVine* in support of solving the ferryman problem. (For simplicity, we omit the textual definition of these rules and only show their generated graphical view. Also, please note that for artistic freedom, the cabbage was exchanged by a grape.) Rule *setup-ferryman* creates the initial graph with the two river sides and all items on one side. Created graph elements are shown in green. The other two rules *ferry_one_over* and *cross_empty* specify the two possible actions, namely to cross with or without an item. Deleted graph elements are shown in

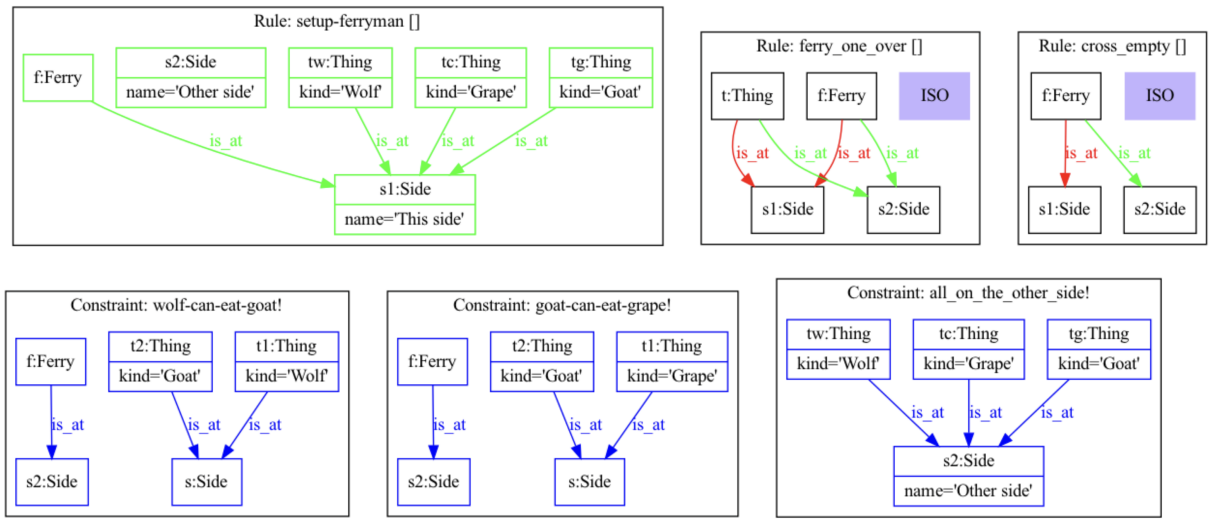


Figure 1: Ferryman problem (rules and constraints)

red.

The shaded isomorphism ("ISO") box in the two rules specify that these rules must use *isomorphic* subgraph matching. (By default, *GrapeVine* does not enforce isomorphic matches, i.e., multiple elements on a rule's left-hand side do not need to match to separate elements in the host graph.)

The bottom half of Fig.1 shows three (atomic) graph constraints. In *GrapeVine*, basic graph constraints $I \xrightarrow{c} T$ are visualized by using black colour for elements in I ("if" pattern) and blue colour for elements in T ("then" pattern). In the case of this example, the three graph constraints have the form $\varnothing \xrightarrow{c} T$, i.e., they represent simple existence constraints $\exists T$. Such constraints are also called *basic constraints* [11]. Note that pattern matching for constraints always uses isomorphic matching semantics.

To better distinguish between the names of rules and constraints in graph programs, it is a convention in *GrapeVine* to append an exclamation mark to the names of constraints.

Fig.2 shows a *GrapeVine* program for solving the ferryman problem. The program should be understandable based on the above definition of control structures and the definition of the rules and constraints. The call to `newgrape` without a parameter creates a new *grape* with an empty graph. After setting up the initial state (applying rule `setup-ferryman`), we enter a loop that recurs until one of the graphs in the last element of the current *grape* satisfies the constraint `all_on_the_other_side!`.

Note that the program in Fig. 2 is guaranteed to find a solution if there exists one. If we had used non-deterministic operators, the execution mechanism may or may not find a solution, depending on how it was implemented. A mechanism that resolves non-deterministic choices by a fair process (e.g., by using true randomness) will eventually find a solution, but without bounded time. A mechanism that uses depth-first search could cause the program execution to diverge.

Fig. 3 shows an alternative, yet equivalent program for solving the ferryman problem. This program declares the two constraints we checked manually in Fig. 2 as invariant constraints on the graphs in the grape. Once declared, *GrapeVine* will ensure that only valid derivations are possible.

```
(-> (newgrape) setup-ferryman
    (->?+ all_on_the_other_side!
          (|| ferry_one_over cross_empty)
          wolf-can-eat-goat!-
          goat-can-eat-grape!-)))
```

Figure 2: A program to solve the ferryman problem

```
(-> (newgrape)
    (schema wolf-can-eat-goat!- goat-can-eat-grape!-)
    setup-ferryman
    (->?+ all_on_the_other_side
          (|| ferry_one_over cross_empty)))
```

Figure 3: An alternative program, using schema constraints

5.4 Efficiency considerations

An important factor for making the above described approach to GT programming feasible is the ability to avoid exploring graph states that have been seen before (by means of the ≠ operator). This operator requires that previous versions of graphs are efficiently accessible (and comparable). Moreover, parallel exploration of possible choices generate non-linear version histories of graphs that must be maintained efficiently. *GrapeVine* uses a fully-persistent data structure for maintaining graphs in a graph database (Neo4J). Details on that persistent data structure have been reported in [20] and interested readers are referred to that publication. However, one new aspect that we want to comment on is the implementation of the ≠ operator. While graph isomorphism checks are expensive, Rensink has demonstrated that hashed graph "certificates" can be used to speed up similarity checking in practice based on the idea of bisimulation [14]. *GrapeVine* uses a similar approach for comparing graphs for collisions. Moreover, since graphs are immutable, their hash certificates are computed only once, at creation time and then stored in the database using an efficient indexed search structure. This means that the run-time for similarity checks only grows logarithmically in practice (in relation to the graphs in the *grape*).

When compared to some other GT tools, specifically those that perform all computation in main memory, *GrapeVine* has a higher latency when performing graph transformations. This is due to its client-server architecture between the Clojure (JVM) client and the database server. However, *GrapeVine's* performance does not suffer much with increasing number and size of graphs, however.

While a comprehensive performance analysis is out of the scope of this paper, we have conducted a preliminary experiment with the program in Fig. 1. The platform was a 2017 iMac Pro with a 3Ghz Intel Xeon processor running both the database server (Neo4J) and *GrapeVine* in separate Docker containers connected by a bridged network.

Running the Ferryman program in Fig. 1 on an empty graph database takes approximately 7 seconds and creates 27 graphs in the process. The execution time of the program remains the same (approximately 7 seconds) even after running the program 1000 times and creating 27,000 graphs in the database. (Note: *GrapeVine* does not automatically "garbage collect" graphs, but the user can trigger a garbage collector manually [20].)

If we modify the program in Fig. 1 to use the iteration without a collision check (`->?*` operator

instead of `->?+`), the program creates 216 graphs and takes 52 seconds.

Currently, *GrapeVine* does not support backtracking when using its non-deterministic operators (based on *derive-choice* and *alternative-choice*). We can therefore not directly compare its breadth-first (deterministic) operators with a depth-first search strategy within exactly the same tool. However, *GrapeVine's* predecessor (*Grape*) supports backtracking-based execution of non-deterministic graph programs. It has the same architecture (Clojure run-time with Neo4J database back-end.) It is therefore a suitable candidate for comparison. *Grape* is actually not able to compute a solution for the ferryman problem as stated in Fig. 1. This is because it would always explore non-deterministic alternatives in the order they are stated. In this case, this would result in a diverging program run that would take the goat back and forth forever. Note that this problem would not go away if the order of the "alternative" statement was reverse. In that case, the goat would be moved and the ferryman would travel empty forever. [18] provides a modified program that uses bounded depth-first search (by giving the ferryman a budget of seven moves). That program takes approximately 50 seconds to find a solution on the same machine as above. Of course, we note that *Grape* does not check for collisions when exploring solutions depth-first. Doing so would certainly increase efficiency. However, the comparison shows that the deterministic (breadth-first) operators in *GrapeVine* have a performance that is competitive with traditional (backtracking-based) solutions to resolve non-determinism.

6 Conclusions and current work

While graph transformations have a well-defined semantics, the control structures used for programming with GTs are often either of limited expressiveness (but formalized) or they are expressive but lack a formal semantics. Implementing non-deterministic control structures in GT tools that use stateful computation requires complex and often inefficient processing such as backtracking on graph exploration. Depth-first search strategies do not guarantee that a solution is found, as graph programs may diverge. For example, a solution for the above-mentioned ferryman problem may not be found in GT tools that implement non-deterministic rule application based on depth-first search and backtracking unless all previously visited graphs are kept and compared to the current execution state for collisions.

Functional GT tools avoid many of the pitfalls of stateful GT tools, since graphs are immutable, i.e., derivations do not destroy (modify) the input graph. In this paper, we have defined a formal foundation for control structures used for functional GT, called the *Graph Transformation control Algebra* (GTA). GTA-based programs can be written as deterministic functions, i.e., they guarantee that all graphs that can be derived via a particular program can be returned. To make this feasible, GTA operators provide means to limit the exploration of the search space, for example by detecting collisions, checking graph constraints and selecting graphs based on a total ordering heuristic. The GTA also provides the traditional non-deterministic operators, which leave decisions on how to perform choices up to the implementer of the execution mechanism.

We have shown the implementation of GTA-based control structures within the functional GT tool *GrapeVine*. Our current work is on performing a more thorough performance evaluation with more realistic problems to provide more evidence for the feasibility of the proposed approach.

References

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi & Attila Vizhanyo (2006): *The design of a language for model transformations. Software & Systems Modeling* 5(3), pp. 261–288, doi:10.1007/s10270-

006-0027-7.

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel & M. Löwe (1997): *Algebraic Approaches to Graph Transformations – Part I: Basic Concepts and Double Pushout Approach*. In G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, pp. 163–245, doi:10.1142/9789812384720_0003.

[3] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, doi:10.1007/3-540-31188-2.

[4] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack & Adam Szalkowski (2006): *GrGen: A Fast SPO-Based Graph Rewriting Tool*. In: *Graph Transformations - Third International Conference, ICGT 2006, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, *LNCS* 4178, Springer, Berlin, pp. 383–397, doi:10.1007/11841883_27.

[5] Annegret Habel & Detlef Plump (2001): *Computational Completeness of Programming Languages Based on Graph Transformation*. In Furio Honsell & Marino Miculan, editors: *Foundations of Software Science and Computation Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 230–245, doi:10.1007/3-540-45315-6_15.

[6] Reiko Heckel & Gabriele Taentzer (2020): *Beyond Individual Rules: Usage Scenarios and Control Structures*. In: *Graph Transformation for Software Engineers*, Springer International Publishing, pp. 67–85, doi:10.1007/978-3-030-43916-3_3.

[7] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers*. Springer International Publishing, doi:10.1007/978-3-030-43916-3.

[8] Barbara König, Dennis Nolte, Julia Padberg & Arend Rensink (2018): *A Tutorial on Graph Transformation*. In: *Graph Transformation, Specifications, and Nets*, *LNCS* 10800, Springer International Publishing, pp. 83–104, doi:10.1007/978-3-319-75396-6_5.

[9] Hans-Jörg Kreowski, Sabine Kuske & Aaron Lye (2018): *A Simple Notion of Parallel Graph Transformation and Its Perspectives*. In: *Graph Transformation, Specifications, and Nets*, *LNCS* 10800, Springer International Publishing, pp. 61–82, doi:10.1007/978-3-319-75396-6_4.

[10] Ulrich Nickel, Jörg Niere & Albert Zündorf (2000): *The FUJABA environment*. In: *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, ACM Press, doi:10.1145/337180.337620.

[11] Fernando Orejas, Hartmut Ehrig & Ulrike Prange (2008): *A Logic of Graph Constraints*. In: *Fundamental Approaches to Software Engineering*, *LNCS* 4961, Springer Berlin Heidelberg, pp. 179–198, doi:10.1007/978-3-540-78743-3_14.

[12] Detlef Plump (2009): *The Graph Programming Language GP*. In: *Algebraic Informatics*, *LNCS* 5725, Springer Berlin Heidelberg, pp. 99–122, doi:10.1007/978-3-642-03564-7_6.

[13] Detlef Plump (2012): *The Design of GP 2*. *EPTCS 82, 2012, pp. 1-16*, doi:10.4204/EPTCS.82.1. Available at `https://arxiv.org/pdf/1204.5541.pdf`.

[14] Arend Rensink (2007): *Isomorphism Checking in GROOVE*. *Electronic Communications of the EASST* 1, doi:10.14279/TUJ.ECEASST.1.77.

[15] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations*. In: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 7233, Springer Berlin Heidelberg, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[16] Andy Schürr, Andreas J Winter & Albert Zündorf (1999): *The PROGRES approach: Language and environment*. In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*, World Scientific, pp. 487–550, doi:10.1142/9789812815149_0013.

[17] Gabriele Taentzer (2004): *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. In: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 3062, Springer Berlin Heidelberg, pp. 446–453, doi:10.1007/978-3-540-25959-6_35.

[18] Jens H. Weber (2017): *GRAPE – A Graph Rewriting and Persistence Engine*. In: *Graph Transformation*, *LNCS* 10373, Springer International Publishing, pp. 209–220, doi:10.1007/978-3-319-61470-0_13.

[19] Jens H. Weber (2021): *GrapePress - A Computational Notebook for Graph Transformations*. In: *Graph Transformation*, *LNCS* 12741, Springer International Publishing, pp. 294–302, doi:10.1007/978-3-030-78946-6_16.

[20] Jens H. Weber (2022): *Tool support for Fully-Persistent Graph Rewriting - GrapeVine*. In: *Graph Transformation*, *LNCS* 13349, Springer International Publishing.

[21] Albert Zündorf & Andy Schürr (1992): *Nondeterministic control structures for graph rewriting systems*. In: *Graph-Theoretic Concepts in Computer Science*, *LNCS* 570, Springer Berlin Heidelberg, pp. 48–62, doi:10.1007/3-540-55121-2_5.

A Lightweight Approach for Model Checking Variability-Based Graph Transformations

Mitchell Albers
Radboud University
Nijmegen, The Netherlands
mitchell.albers@ru.nl

Carlos Diego N. Damasceno
Radboud University
Nijmegen, The Netherlands
d.damasceno@cs.ru.nl

Daniel Strüber
Chalmers | University of Gothenburg, SE
Radboud University Nijmegen, NL
danstru@chalmers.se

Graph transformation systems often contain large numbers of similar rules, leading to maintenance issues as well as performance bottlenecks during rule applications. Previous work introduced variability-based graph transformations as a paradigm for explicitly managing variability in rules, successfully addressing these issues. However, no previous work investigated whether variability-based graph transformations can also lead to benefits during the automated analysis of graph transformations, particularly during model checking, in which the main performance bottleneck is the combinatorial explosion arising during state space exploration.

In this paper, as an initial approach for model checking of variability-based graph transformations, we present an extension of an existing symbolic model checking technique. The existing technique, called Gryphon, converts the graph transformation system into a symbolic encoding and, from there, into the input format of a hardware model checker. We adapt Gryphon's encoding to incorporate information on variability, which reduces the size and complexity of the overall encoding since it is now derived from a smaller set of rules (some of them being variability-based rules that represent several similar rules). In a preliminary evaluation, we show that our extension leads to performance benefits in a standard model checking scenario.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm that promotes the use of models and transformations as primary artefacts, allowing for the abstraction of non-essential aspects [23]. MDE promotes model transformations as a key enabling technique for automatically generating models to reduce errors and save implementation efforts. One of the main paradigms in model transformation is algebraic graph transformation (AGT [16]) which allows the specification of transformation rules in a high-level, declarative manner using model transformation languages. Sometimes, one wants to specify many similar transformation rules that have many commonalities but differ in some parts. In that case, these similarly-structured rules can be merged while preserving their structure by specifying variability points on diverging elements, introducing variability on transformation rules. Therefore, by expressing variability points on diverging graph elements in transformation rules, we can merge rules and keep the number of total rules minimal. Doing so, one obtains a new concept of graph transformation rules with variability, called *variability-based graph transformation* in previous work [41]. As a further benefit, one can improve the performance of such graph transformations by providing a variability-aware execution mode that can make the rule application procedure faster and more scalable [41].

When verifying the correctness of regular AGT rules, model checking has proven to be a powerful tool of increasing interest [36]. However, current state-of-the-art model checking tools do not support variability-based graph transformation. But why would it be desirable if they did so? Model checking tools currently do not consider variability, implying that each rule variant has to be made explicit in

a separate rule. From a model checking perspective, this leads to redundant computational effort when dealing with rules with many shared actions, resulting in a sub-optimal model checking approach in terms of runtime behaviour. A model checking approach that addresses variability as part of the model checking procedure without considering each rule variant as a separate rule could lead to runtime improvements.

To our knowledge, no previous work has addressed variability in model checking of graph transformation systems. Related work (discussed in Section 6) addresses variability in model checking for other formalisms, such as *labelled transition systems* [14, 13] and *Markov decision processes* [11].

This paper introduces an initial approach for model checking graph transformations with variability. We propose an extension of the open-source, lightweight symbolic model verification technique Gryphon [19]. Gryphon uses a symbolic encoding to represent graph transformation systems to bounded first-order relational logic. Gryphon assumes a bounded universe, implying that it does not support arbitrary creation and deletion of nodes. However, it does support negative application conditions (NACs) as well as arbitrary creation and deletion of edges. We study the working assumption that addressing variability explicitly, as part of the model checking procedure, may help to increase the performance of the model checker, compared to enumerating all rule variants explicitly and feeding them as input to the model checker. Thereby, we will be answering the following research question:

RQ: *Does the direct support for variability-based rules in Gryphon decrease the execution time for model checking of graph transformations of the kind supported by Gryphon?*

To answer this research question, we propose an extension of Gryphon's symbolic encoding that supports variability-based graph transformations. We further present an implementation of the resulting encoding, leading to the first implementation of a variability-based model checking approach for graph transformations. Like Gryphon, the resulting approach is restricted to a certain type of graph transformations: double-pushout graph transformations, restricted to rules that can create and delete edges and can have NACs. The considered transformation kind is expressive enough to support previous scenarios from the literature on model checking of graph transformations, such as *dining philosophers*, *interlocking railway systems*, and *pacman* (provided as examples in Gryphon's presentation [18, 19]). However, several other examples from the literature cannot be handled by our approach, such as *circular buffers* [35], *malaria surveillance* [8], *knowledge graph management* [9], and *health information system* [7] that require creation, deletion, cloning and/or merging of nodes.

Using the encoding, we answer the research question in an empirical performance evaluation. This evaluation compares the execution time of model checking with standard Gryphon and a Gryphon version using the modified encoding.

2 Preliminaries

Graph transformations

We consider the paradigm of algebraic graph transformations (AGT), in particular the formalization of graph morphisms and graph transformation rules based on the *double-pushout approach* [16]. A graph morphism $m : G \to H$ is a structure-preserving mapping between two graphs, in the sense that edges are mapped in a way that respects their source and target. A transformation rule t is defined as $t = (L \xleftarrow{l} I \xrightarrow{r} R,\ Nac)$, consisting of the following elements: a left-hand side graph L and a right-hand side graph R, along with two graph morphisms l and r, both being inclusions, an interface graph I, satisfying $I \subseteq L$ and $I \subseteq R$, and a set Nac (negative application conditions) of graph morphisms of the

form $n : L \to N$. In the rest of this paper, we consider a restricted notion of rules in which the node sets of L, I, and R are identical.

Rules are applied to a given host graph in order to obtain a result graph. The application of a transformation rule t on a given start graph G assumes an available injective graph morphism $m : L \to G$ that matches L to G. To support node deletions in a consistent way, m needs to fulfill a *dangling edge condition*: if a node n is to be deleted by the rule application, the rule application has to delete all adjacent edges of the node as well. For the restricted kind of rules we consider (which cannot delete or create nodes), this condition is always fulfilled. Furthermore, m needs to satisfy the NACs. Intuitively, each NAC graph N is interpreted as a pattern whose existence in the host graph is forbidden. More formally, it has to be checked that for none of the NACs there exists morphism $n' : N \to G$ s.t. $m = n' \circ n$. If m fulfills these conditions, it is called a match.

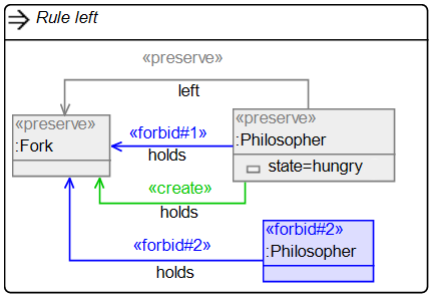
Given a match, a rule application is executed as follows: First, for each graph element that is part of L and that is not part of the interface graph I, the element identified by m is deleted. Second, for each graph element that is part of R and that is not part of interface graph I, a graph element is created; new graph elements are "glued" to existing ones as specified by m and r.

The graphs supported by our technique can have types and attributes. We omit providing details of the formalization of these concepts; the interested reader is referred to Ehrig et al.'s [16] formalization.

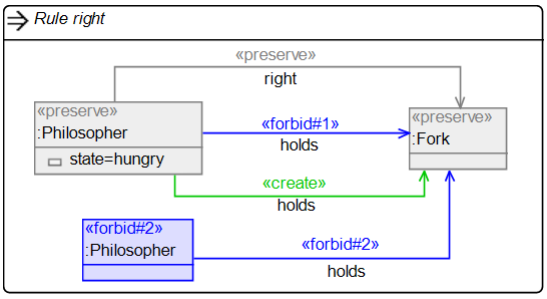
Example. We consider the Dining Philosopher's problem as presented in [19], which was originally designed to illustrate challenges concerning deadlocks, making it a well-suited problem for model checking. The Dining Philosopher's problem starts out with a group of n philosophers around a table on which there are n plates and n forks (i.e., one fork on the left, and one fork on the right of each philosopher). A philosopher can either be *hungry*, *thinking*, or *eating*. Whenever a philosopher transitions from *thinking* to *hungry*, it wants to eat. In order for a philosopher to be able to eat, it is required to have 2 forks assigned to that philosopher. Whenever a philosopher is done with eating, it releases both forks so that another philosopher can take them. Figure 1 captures a standard specification of the Dining Philosopher's problem in terms of graph transformation rules, inspired by [19]. In this example, the following five graph transformation rules are considered:

- Rule *hungry* transitions a philosopher's state from *thinking* to *hungry*.
- Rule *left* leads to a philosopher picking up their left fork (indicated by *holds*), provided that no philosopher (including themselves) already holds that fork.
- Rule *right* leads to a philosopher picking up their right fork (indicated by *holds*), provided that no philosopher (including themselves) already holds that fork.
- Whenever a philosopher is assigned to both a left and right fork, they can start eating. This behaviour is specified in rule *eating*.
- Whenever a philosopher is done eating, they release both forks, which is specified in rule *release*.

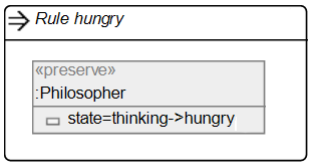
Note that these transformation rules follow an integrated syntax with respect to the formal definition; graph elements are annotated with labels. The labels *delete*, *preserve*, and *create* represent the sets $L \setminus I$, I, and $R \setminus I$, respectively. The label *forbid*, together with a hash symbol and the index of a NAC, represents the set $N \setminus L$ for the given NAC. In the example, rule *left* has two NACs, the first of them (*forbid#1*) ensuring that philosophers cannot pick up a fork that they already hold.



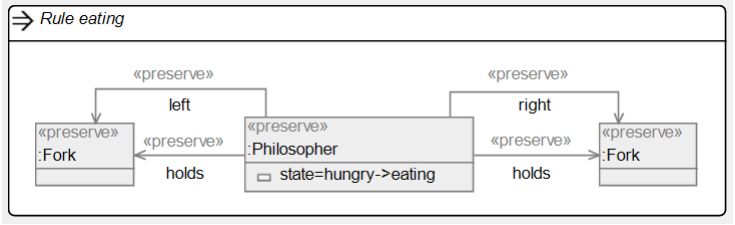
(a) Rule *left*



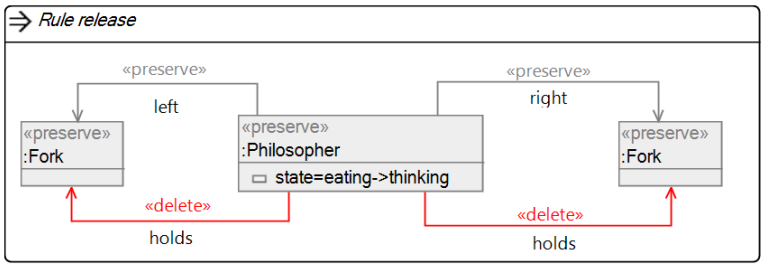
(b) Rule *right*



(c) Rule *hungry*



(d) Rule *eating*



(e) Rule *release*

Figure 1: Graph transformation rules of the Dining Philosopher's problem

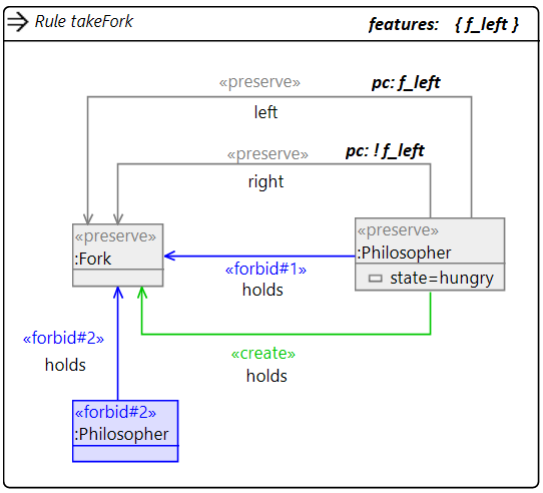


Figure 2: Variability-based rule *takeFork*

Variability-Based Graph Transformations

Variability-based graph transformations are an extension of standard graph transformations, which allows to express several similar rules in a "single-copy" representation. Individual, diverging elements (in the scope of this paper, edges) that are specific to some, but not all of the rules are annotated with *presence conditions*. A presence conditions is a boolean formula over a given set of features $\mathscr{F}$ (a.k.a. variation points), specifying a condition under which an annotated element is present. Given a variability-based rule, flat *rules* can be derived by binding each feature from $\mathscr{F}$ to either *true* or *false*, and removing those elements whose presence condition evaluates to *false*.

A variability-based transformation rule $\hat{t}$ is formalized as $\hat{t} = (t,\ \mathscr{F},\ pc)$, where t is a standard graph transformation rule (the "maximal rule" with all elements), and $pc : (L_t \cup R_t) \rightarrow Bool(\mathscr{F})$ is a *presence condition* function which maps rule elements to a propositional formula over the set of features $\mathscr{F}$ (defaulting to *true*, unless explicitly stated). To apply a variability-based graph transformation rule, intuitively, the first step is to *configure* it, that is, bind each of its features to either *true* or *false*, and remove all elements from t whose presence conditions evaluate to false. Configuring a variability-based graph transformation rule that way yields a flat rule that can be applied in the standard way (described above). The configuration can be either done manually by the user, or automatically by the configuration engine, where the latter leads to the notion of a variability-aware execution engine described in [41, 43]. Editing variability-based rules in a user-friendly way is supported by a dedicated extension of Henshin [42].
Example. When taking a closer look at rules *left* and *right* from Figure 1, we observe that these rules share a similar structure with an exception on the edges that are annotated with '*left*' and '*right*'. Therefore, we can merge these similar-structured rules and introduce a variability-based rule $takeFork = (t,\ \mathscr{F},\ pc)$ to represent the designation of forks to a philosopher. Transformation rule t consists of all the rule elements from rule *left* and *right*, such that it is "maximal" in the sense that it contains all rule elements. We can then annotate the diverging elements from the other rules with presence conditions over the set of features $\mathscr{F} = \{f_{left}\}$. Thereby, we annotate edges *left* and *right* with presence conditions $pc(left) = f_{left}$ and $pc(right) = \neg f_{left}$, respectively as shown in Figure 2. To be able to produce the original rules, both presence conditions may not evaluate to false or true at the same time—which is the case here, because they are mutual exclusive. For more complicated examples, one would need a *configuration constraint* to avoid illegal configurations (a.k.a. feature selections). Nonetheless, the concept of configuration constraints is out of scope of this paper.

Gryphon

The verification technique Gryphon [19] provides a model checking approach for model-driven software systems, whose static structure is defined by models built with the Eclipse Modeling Framework and whose behaviour is defined by graph transformation rules built with the Henshin API [2, 39]. From this static structure and behaviour models, Gryphon constructs a relational transition system (an encoding of a transition system using first-order logic formulas) which then can be checked by hardware model checkers to verify both safety and reachability properties. Gryphon does not support for the specification of custom LTL/CTL properties, but considers only one CTL reachability property in the form of $\exists\Diamond\phi$, were ϕ is a graph state to be reached. By duality, we may negate this formula and check for the unreachability or safety of ϕ, which can be expressed by $\forall\Box\neg\phi$.

Gryphon uses a symbolic encoding to encode an input graph, along with graph transformation rules, into relational formulas (i.e., first-order logic) that describe the model transformations. In essence, the encoding consists of two steps.

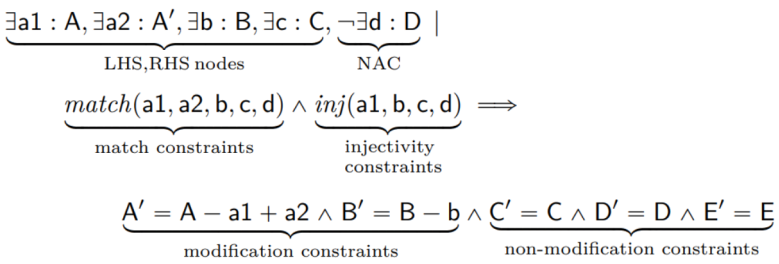


Figure 3: Scheme of a relational formula produced from a graph transformation rule (based on [19])

First, relational variables are generated. This entails assumes that a type graph $G_T = (V_T,\ E_T)$ is given with nodes V_T representing node types and edges E_T representing edge types. The creation of these relational variables is done by using the function $relgen : V_T \cup E_T \to Rel$ which generates for each node type a unary relational variable, and for each edge type a binary relational variable. The unary relational variables are included as atoms in a fixed universe, which consists of a sequence of uninterpreted atoms $\mathbb{A}$. This universe is initially derived from the initial model, such that for every object in the initial model, there exists a corresponding atom in the universe. The Gryphon technique considers a bounded, first-order logic, meaning that each relational variable is assigned an upper bound, and optionally a lower bound. Bounds are specified over the set of atoms in the universe [19]. In order to assign upper and lower bounds to unary relational variables, we use the function $\sqcup : Rel \to \mathscr{P}(\mathbb{A})$; upper bounds of binary relational variables arise from the product of edge source's and target's upper bounds.

Then, from these relational variables, graph transformation rules are translated into first-order, relational formulas. These relational formulas are generated by deriving a formula[1]

$$\mathrm{F}_t := Pre(L, Nac, R) \Longrightarrow Post(L, R)$$

for each graph transformation rule $t : (L \leftarrow I \rightarrow R,\ Nac)$. The function $Pre : G \times G \times G \times G \to \mathbb{F}$ takes a quadruple of graphs and produces a conjunction of relational formulas that mimic the match conditions of the transformation's left-hand side and NACs. Moreover, *Pre* also takes injectivity constraints into account. The function $Post : G \times G \to \mathbb{F}$ takes a pair of graphs and produces a conjunction of relational formulas that mimic how the rule application changes the assumed host graph via deletion and addition. The resulting formulas can be modification or non-modification constraints, in the sense that specify what needs to be changed in the host graph or what needs to remain unchanged. The formula F_t can then be expressed by the scheme shown in Figure 3, which is based on an example with the relational variables $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$. A more comprehensive introduction of the relational encoding is found in [19].

Internally, Gryphon constructs the first-order relational formula from Figure 3 by using the KODKOD API [30], which is then used to translate it to a propositional formula (exploiting the assumption that the universe is bounded, which makes this translation possible). After that, this propositional formula is rewritten into an and-inverter graph (AIG)—a boolean circuit consisting of only 'and' and 'not' gates—and stored in the AIGER format (see Figure 4, [19]). This step is aimed to foster interoperability as several model checkers can read this format. For the scope of this research, we consider the Incremental Inductive model checker (IIMC, [24]). Properties are specified at the graph level, as Henshin "check rules" (non-modifying rules), and translated to the same format.

[1]This formula originates from Gryphon's encoding [19], which, in addition, considers positive application conditions–out of the scope of this research.

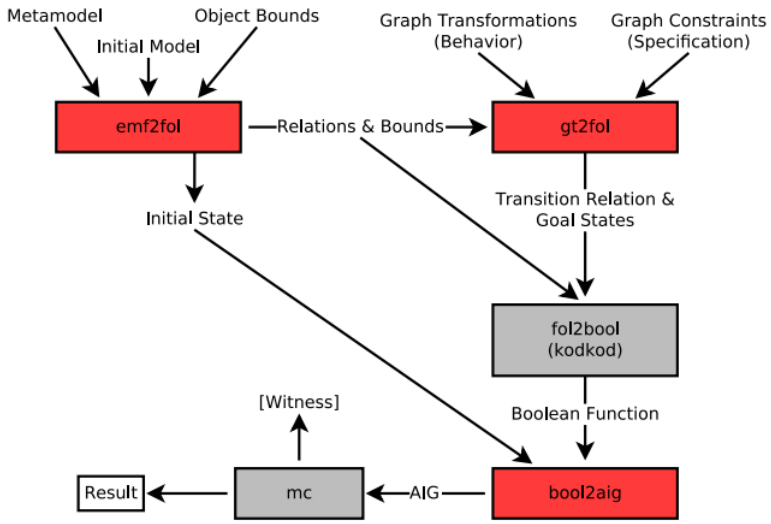


Figure 4: Gryphon's workflow architecture (from [19]). Grayed components represent external tools

3 Motivating Example

In this section, we will give a motivating example to illustrate how model checking of graph transformations with variability can be improved by explicitly using variability-related information. For this example, we consider the Dining Philosopher's problem with variability as presented in Section 2.

For model checking, we consider a very simple reachability property in which we want to check if it is possible whether a philosopher can be in the state *eating*, expressed as the following CTL property: $\exists \Diamond eating$. When considering the variability-based rule *takeFork* from Figure 2 in which we have specified presence conditions $pc(left) = f_{left}$ and $pc(right) = \neg f_{left}$, we observe that this can be easily done by applying the rule variants for picking up the left and the right fork. This can be achieved by applying rule *takeFork* twice; once in which f_{left} is set to true, and once by applying *takeFork* in which f_{left} is set to false. This evaluates the corresponding presence conditions to true, making it possible for a philosopher to take the corresponding fork(s). Consequently, given this information, the model checker is able to conclude that it is now possible for a philosopher to eat as both forks can be assigned to it, which means that rule *eating* can be applied, satisfying the reachability property. However, when taking a closer look at the rule *takeFork*, we observe that this rule tries to match a single fork as a left and a right fork to a philosopher. Therefore, we consider presence conditions to preserve the semantics relative to having two separate rules (i.e., one for picking a left fork and one for a right fork). These considerations would lead to a *variability-aware model checking algorithm* that explicitly addresses the rule variants expressed by a variability-based rule as part of its internal workings.

Without dedicated support for variability-based rules as part of the model checking algorithm, one could still supporting *takeFork*, by enumerating all possible configurations for the features $\mathscr{F}$, and creating a rule for each of them, and feeding the rules as input to a standard model checker. In general, this leads to an exponential increase in the number of rules, depending on the number of features and presence conditions that are annotated inside the rules. Feeding more rules with lots of redundant information into the model checker is likely to make the model checking procedure less efficient.

Therefore, by instead using the information contained in the variability-based rules explicitly during

the model checking procedure, we expect to obtain efficient support for variability in graph transformation rules, as we can keep the number of rules minimal and avoid expressing information redundantly.

4 Variability Encoding

Gryphon's encoding encompasses the use of relational formulas to mimic the behaviour of graph transformations. This section describes an extension of this encoding to support graph transformation rules with variability.

Recall from Section 2 that a variability-based graph transformation rule consists of a "maximal rule" t, a presence condition function pc, and a set of features $\mathscr{F}$ that can be used to annotate graph elements with presence conditions in the form of a boolean formula over the set of features $\mathscr{F}$. The intention of annotations on a graph transformation rule with presence conditions is to specify an additional constraint on the rule's matching behaviour. Thereby, the evaluation of a rule's set of presence conditions, determines which rule variant is applied.

To specify this variability-based matching behaviour within Gryphon's encoding, we first introduce a relational variable for each feature. Similarly to the function *relgen* as described in Section 2, we introduce the function $f_{relgen} : \mathscr{F} \rightarrow Rel$, which maps features to unary relational variables. These relational variables are then also included in the set of atoms $\mathbb{A}$. This step is required for constructing relational formulas, in which we can refer to the atoms in the universe that correspond to the features as relational variables.

Recall from Section 2 that relational variables must be bounded. Since feature variables are used to construct presence conditions in the form of propositional formulas, they are upper bounded by boolean values (i.e., true and false). As we are now able to generate and bound relational feature variables from graph annotations, we consider the function $pc_{relgen} : Bool(\mathscr{F}) \rightarrow Bool(Rel)$ that maps a boolean formula of features to a relational formula, representing a presence condition.

Gryphon's encoding as discussed in Section 2, considers the formula F_t, following the formula generation scheme as exemplified in Figure 3. This formula uses the function *Pre* to create a relational formula that mimics the matching conditions of the left-hand side of a graph transformation rule t. The function *Pre* consists of an explicit formula *match* that contains the matching constraint of a transformation rule. Since we are interested in embedding the presence conditions into the matching constraints, we imply that we want our presence condition to be a dependent variable of this matching. More concretely, we want to make the matching constraints of graph elements dependent on these presence conditions such that: whenever the presence conditions can be met, then the actual matching can also be done. This can be easily specified as an implication between the presence condition and the matching constraints of the corresponding graph element. For the scope of this research, we only consider edges to be annotated with presence conditions. Therefore, we translate every edge e with $src(e) = c$ and $trg(e) = d$ to the following matching constraint: $(c \rightarrow d) \subseteq C_e$, where C_e is a binary relational variable that represents the edge. However, whenever an edge annotated with a presence condition, we want to embed this information in this edge. Therefore, we add our presence condition pc to the matching constraint of this edge as follows: $pc \rightarrow ((c \rightarrow d) \subseteq C_e)$. By doing so, the matching constraint is dependent on the presence condition; if the presence condition is met, then this reference can be matched.

We argue for the soundness of our encoding informally, leaving a rigorous soundness proof to future work: In [41], we introduced a notion of *variability-based rule applications*, which supports the application of a variability-based rule to a graph. We showed the soundness of variability-based rule application, i.e., its equivalence to applying each of the rule variants expressed by the variability-based rule

to the graph, using traditional rule applications. We argue that Gryphon's standard encoding captures the notion of traditional rule application, whereas our modified encoding captures variability-based rule application. More specifically, including presence conditions of edges into the encoding as we do allows *variability-based matches* to be identified. Hence, we can benefit from our earlier soundness proof.

Example. Consider the dining philosopher problem with variability-based rules, as discussed in the motivating example in Section 3. In this example, we consider $\mathscr{F} = \{f_{left}\}$ as the set of features that are present in rule *takeFork*. These features are also added as uninterpreted atoms in the universe. From this, we can construct the following relational feature variable $F_{left} = f_{relgen}(f_{left})$ and bind it by $\sqcup(F_{left}) = \{true,\ false\}$. In this case, rule *takeFork* is annotated with f_{left} on edges *left* and *right*. Therefore, we have presence conditions $pc_{relgen}(pc(left)) = pc_{relgen}(f_{left}) = F_{left}$ and $pc_{relgen}(pc(right)) = pc_{relgen}(\neg f_{left}) = \neg F_{left}$ for edge *left* and *right*, respectively. The references for edge $left$ and *right* are translated as the following matching constraints:

$$F_{left} \rightarrow \big((p \rightarrow f) \subseteq Phil_{left}\big),\ src(left) = p \in Phil,\ trg(left) = f \in Fork$$

$$\neg F_{left} \rightarrow \big((p \rightarrow f) \subseteq Phil_{right}\big),\ src(right) = p \in Phil,\ trg(right) = f \in Fork$$

Note that *Phil* corresponds to the class '*Philosopher*', and *Fork* to the class '*Fork*' in the transformation rule *takeFork*, where p is a philosopher object and f is a fork object in the matching host graph.

5 Evaluation

This section concerns an evaluation in the form of a runtime comparison between the model checking procedure with variability encoding and the standard Gryphon approach without variability-based graph transformation rules. For this evaluation, we again consider the dining philosopher's problem as described in Section 2. Thereby, we consider the system of transformation rules from Figure 1, consisting of rules: *left*, *right eating*, *hungry*, *release*. For fair comparison, we compare this system of transformation rules with the same system, but instead of using the rules *left* and *right*, we consider the variability-based rule *takeFork* from Figure 2. Moreover, we consider one single reachability property that checks whether it is possible for a philosopher to be in an *eating* state (i.e., $\exists\Diamond eating$). We reuse the provided specification of this property from the original Gryphon implementation of the case, using a non-changing rule to specify the graph pattern part of the constraint.

For this runtime comparison, we are interested in the following two metrics: (1) the time of the actual model checking to conclude the property and (2) the total execution time of the tool (i.e., including rule translation, etc.). We can then gain insight into how much time was spent on solving and how much time was spent on translating the models. During this comparison, we consider five different input graphs, consisting of 10, 20, 30, 40, and 50 philosophers. For accuracy, we have executed the model checking tool several times over each input model and for each model type; 30 times with variability-based graph transformation rules (VAR), and 30 times without variability-based rules (STD).

The runtime comparison between both model checking approaches is depicted in Figures 5, 6, and 7, covering total runtime, solving time, and translation time, respectively. From these figures, we generally observe that model checking with support for variability outperforms model checking without that support on larger input models in terms of solving time, but not on small input models. For translation time, we observe that a variability-based rules increases performance for every input model. Moreover, the main contribution to total runtime comes from solving, whereas rule translation affects total runtime performances non-significantly.

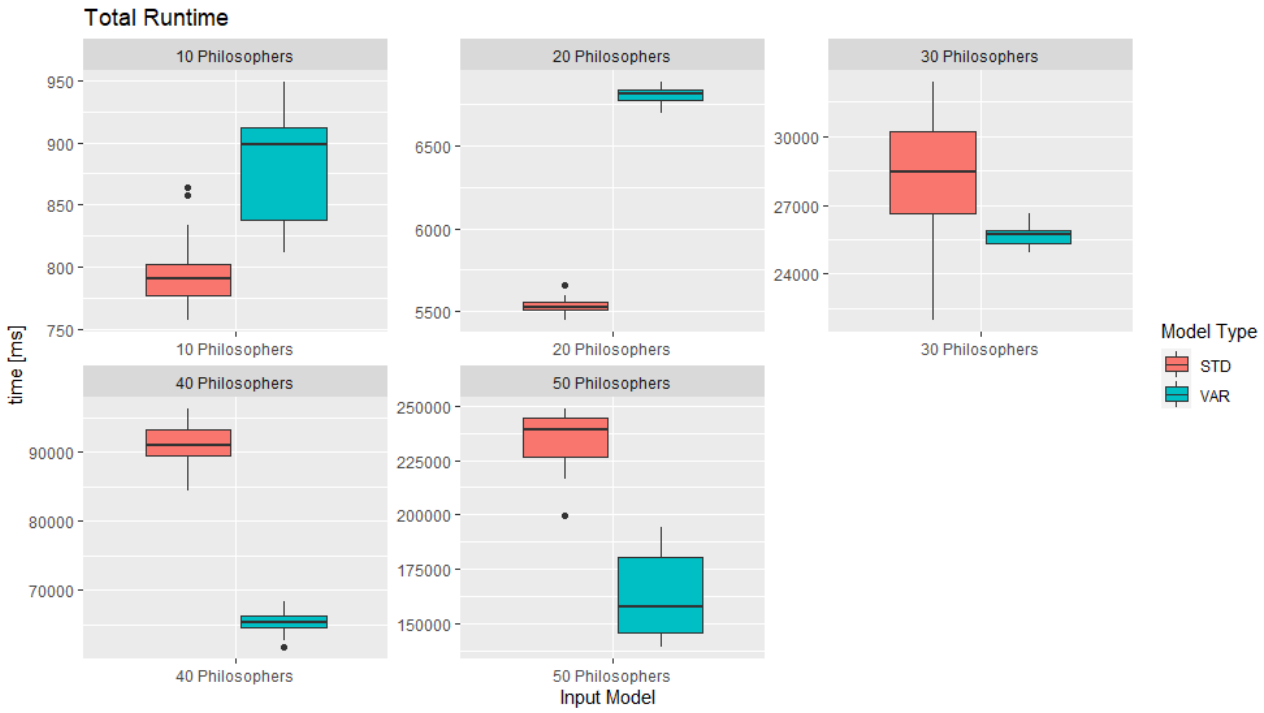


Figure 5: Total runtime for standard (STD) and variability-aware (VAR) model checking.

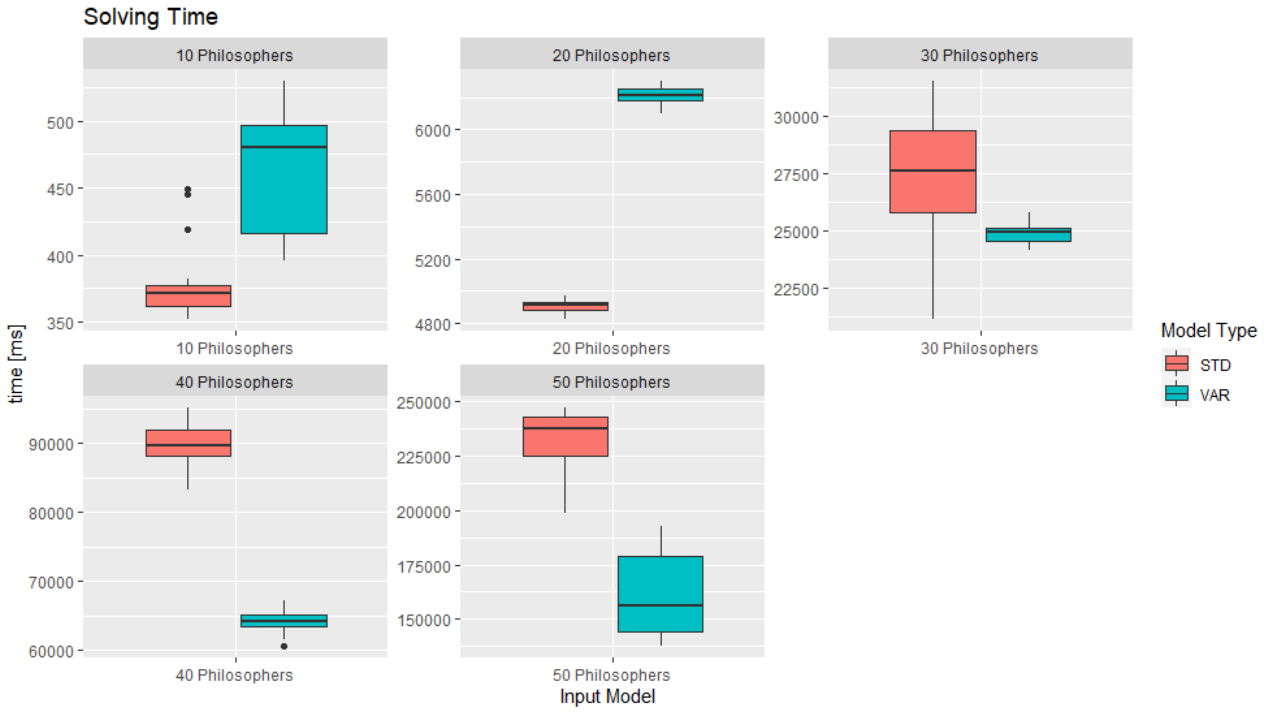


Figure 6: Solving time for standard (STD) and variability-aware (VAR) model checking.

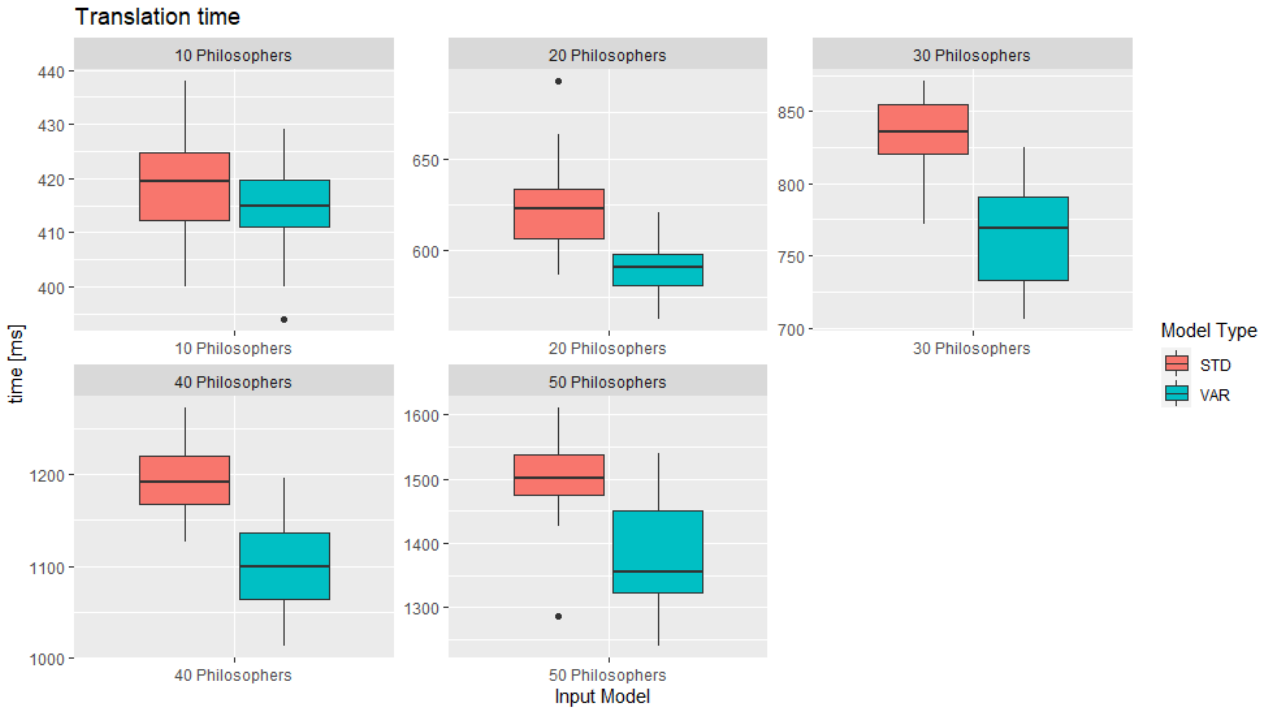


Figure 7: Translation time for standard (STD) and variability-aware (VAR) model checking.

When taking a closer look at the metrics of our runtime comparison from Figures 5, 6, and 7, we consider the mean μ and standard deviation σ to give insightful information regarding our obtained evaluation result (see Table 1). Note that lines that are written in bold represent the metrics that had the best overall performance for a given model type (i.e., STD or VAR) and the corresponding input model. Based on the mean average total runtime, we observe a slowdown of 11% on VAR relative to STD for an input model of 10 philosophers. This slowdown increases to 23% for an input model that considers 20 philosophers. At 30 philosophers, we start to observe a turning point as we see a speedup of 11%. This speedup increases substantially for an input model that considers 40 philosophers towards 39% and, for an input model with 50 philosophers, to 45%.

Regarding our research question **RQ**, we can conclude that variability-aware model checking can significantly improve performance, up to a speedup of 45%, even for a seemingly simple case with only two similar rules that were represented as one variability-based rule. These savings can be explained from the specifics of the case that, however, might be transfer to other similar cases as well: The variability is included in the most complicated (because NAC-including) pair of rules (i.e., by merging rules *left* and *right* into *takeFork*), which also leads to a particular complicated encoding of these rules as part of the overall encoding. By representing these two rules using one variability-based rule, we get a shorter and less complex overall encoding.

While a formal performance argumentation based on the encoding's complexity would need to address the internal workings of the underlying hardware model checker, which is outside our scope, we empirically observe that the less complex encoding leads to performance improvements for larger instances. Most of the observed savings are made in the *solving* time, which, especially for larger instances, is the computational bottleneck of the overall process. We expect these observed savings to increase even

		Total Runtime (ms)		*Solving Time (ms)*		*Translation time (ms)*	
Input Model	*Model Type*	μ	σ	μ	σ	μ	σ
10 Philosophers	**STD**	**795**	**25**	**371**	**23**	**385**	**10**
	VAR	879	43	477	43	376	8
20 Philosophers	**STD**	**5534**	**44**	**4901**	**35**	**539**	**23**
	VAR	6803	50	6691	54	489	15
30 Philosophers	STD	28385	2423	27396	2426	686	25
	VAR	**25662**	**459**	**24368**	**451**	**654**	**34**
40 Philosophers	STD	91031	2653	89880	2646	939	35
	VAR	**65291**	**1472**	**59473**	**1472**	**812**	**48**
50 Philosophers	STD	234842	11868	232038	11830	1377	63
	VAR	**162486**	**18771**	**142951**	**18774**	**1213**	**80**

Table 1: Performance evaluation results: runtime comparison between STD and VAR.

more for cases with a greater amount of variability.

6 Related Work

Studies related to ours are on model checking for software product lines [46], in particular product-based [29, 28, 15, 1] and family-based analysis [22, 12, 6]; and model checking [25, 37, 36, 35] and analysis [21, 26, 34] of graph transformations.

Product-Based Model Checking. In product-based model checking, product-specific models of a product line are generated as separate entities and individually verified, each using a standard verification technique which may be optimized or not. In unoptimized approaches, product variants are verified each time they are derived [29]. To improve scalability and reduce redundant computations, model checking techniques can be optimized for incremental verification [46] with richer notions of features, e.g. *conservative features* that do not remove behaviour [15]; or aspects, e.g., spectative, regulative, and invasive [28]. Sampling-based model checking aims at reducing the verification problem by selecting a subset of valid products [47]. They are likely to find defects quickly, but may miss defects due to its incompleteness [1]. Instead, our technique for model checking graph transformations is family-based.

Family-Based Model Checking. The main problem with product-based analyses are redundant computations over shared assets. Then, to achieve a more efficient verification, family-based model checking incorporates domain artifacts, such as feature models, to analyze a family model with respect to the variability model and one or more properties. For a given property, a family-based model checker analyzes whether the property is fulfilled by all products. If not, the model checker provides a propositional formula specifying those products that violate the property [22]. As standard model checking, their family-based variants can operate directly on source code or on an abstraction of a system [6], such as featured transition systems [12]. Our work differs from the aforementioned literature on family-based analysis by operating on graph transformations as our abstraction.

Model Checking Graph Transformations. The basics of verifying graph transformation systems by model checking have been studied thoroughly by Heckel in [25]. More recently, approaches for model checking graph transformation systems have been also extended to more complex scenarios, such as compositionality, and probabilistic, timed behaviour [33, 34]. Typically, graphs can be interpreted as states and rule applications as transitions in a transition system [36]. From a graph transformation perspective,

there are two main approaches: CheckVML [37] and Groove [35]. The main idea of CheckVML [37] is to exploit off-the-shelf model checker tools, like SPIN [5], by translating a graph transformation system and property graphs into their Promela and temporal logic equivalents to carry out the formal analysis. In contrast, in the GROOVE approach [35], the core concepts of graphs and graph transformations are used all the way through model checking by explicitly representing and storing states as graphs, and transitions as applications of transformation rules. Also, since properties are specified in a graph-based logic, the theory and tool support of standard model checkers [3] may not be applied immediately and graph-specific model checking algorithms should be developed. We contribute the first approach to address variability in the model checking of graph transformations.

Analyzing Graph Transformations. In addition to model checking, there are various other techniques to analyze graph transformation systems. For an overview in this topic, we refer the interested reader to [26]. According to Heckel and Taentzer [26], techniques to analyze graph transformation systems may address *conflict and dependency analysis* – to determine the possibility of conflicts or dependencies between rules, *termination analysis* – to establish the absence of infinite transformation sequences, *constraint verification and enforcement* – to derive and check weakest preconditions, and *graph parsing* – to construct a derivation for a graph based on grammar rules. One of the severe issues in analyzing graph transformations is that graphs are typically specified monolithically and, for large models, it can quickly undermine the advantage of visualisation, and lead to state space explosion [21]. To address this issue, the analysis of graph transformation systems can also be enriched with notions of compositionality [21, 34]. Our work addresses this problem by matching and merging similar-structured transformation rules and annotating their divergences with presence conditions. Within the scope of infinite-state or potentially-large systems, description logic [8], symbolic execution [38], and over/under-approximation [4, 20] contribute to the design of push-button technologies for verifying graph transformation systems in a timely manner. Our technique is suitable for cases where the explicit representation of states is practical, in which our preliminary evaluation shows reduced computational effort. It can be seen as a complement to verification techniques suitable for infinite-state spaces [27].

7 Conclusion

In this paper, we have presented a variability encoding as an extension of the lightweight symbolic model checking technique Gryphon. We have introduced a minimal model checking approach that can use variability information when it is explicitly encoded in graph transformations rules, introducing an initial approach to model checking of variability-based graph transformations. This variability encoding tries to mimic the matching behaviour of multiple rule variants. Moreover, we have given a motivating example that shows how including features as part of the model-checking procedure can result to increase performance relative to enumerating all rule variants of a given transformation rule.

Considering our research question, we have empirically evaluated the resulting model checking approach for variability-based graph transformations. This evaluation was performed by a runtime comparison with a standard model checking approach without variability-based rules (see Table 1). From this evaluation, we observe moderate improvements in terms of performance on solving time on larger input models. Therefore, we may conclude that a variability-based execution mode shows potential in model checking of graph transformations. We expect that this improvement will become more significant when including more variability-based graph transformation rules with more features and presence conditions for a given system of transformation rules.

8 Limitations and Future Work

Our presented model checking approach is far from complete and is to be considered a proof of concept. Important future directions for extending our present, Gryphon-based, approach include:

- A rigorous argumentation for the soundness and performance of our approach. For soundness, this entails proving that our variability-based execution mode leads to the same results as the classical one—a conjecture currently supported by our informal argumentation and automated testing in our evaluation scenario. A performance proof addressing the internal workings of the underlying hardware model checker could be used to explain our empirically observed performance benefits.
- Increasing the expressiveness of supported variability-based transformations, to support configuration constraints as well as arbitrary propositional formulas as presence conditions. While the current implementation does not support constraints and only supports simple presence conditions consisting of a singular feature or its negation, we do not foresee any particular limitations towards improving the offered support as described.
- A more exhaustive empirical performance evaluation. The present evaluation considered a small graph transformation system, including a limited number of features and presence conditions. While we have shown that including variability in transformation rules can bring performance improvements even in this small setting, we foresee that the improvement can be more significant when considering more rule variability and larger input models.

The abovementioned directions could be addressed by further extensions of and experimentation with the Gryphon model checker. However, any future extensions of Gryphon are likely to retain the same limitations regarding the supported kinds of graph transformations that we currently have. In particular, they would be focused on graph transformation rules that create and delete edges.

To lift this limitation, a promising direction is to extend other graph-based model checking techniques that may benefit from introducing variability as well. As a starting point, one might want to introduce a variability-based execution mode in the GROOVE [35] model verification technique. As described in Section 6, GROOVE explicitly represents states as graphs and transitions as applications of transformation rules. Therefore, a promising idea is to include a variability-based rule application mode to make GROOVE variability-aware. To this end, one could use some of the concepts of dealing with variability as presented in this paper.

Finally, while the current work is focused on addressing variability in rules, graphs could be affected by variability as well. In particular, this is the case when modeling and analyzing a *software product line* [46] using graphs, which during model checking would lead to a further dimension of combinatorial explosion. Graph transformation of software product lines has been addressed in previous work [45, 40, 10], motivating future work on model checking as a means of validating such transformations.

References

[1] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger & Dirk Beyer (2013): *Strategies for product-line verification: Case studies and experiments*. In: *2013 35th International Conference on Software Engineering (ICSE)*, pp. 482–491.

[2] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause & Gabriele Taentzer (2010): *Henshin: advanced concepts and tools for in-place EMF model transformations*. In: *International Conference on Model Driven Engineering Languages and Systems*, Springer, pp. 121–135.

[3] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. MIT Press, Cambridge, MA, USA.

[4] Paolo Baldan, Andrea Corradini & Barbara König (2008): *A framework for the verification of infinite-state graph transformation systems*. Information and Computation 206(7), pp. 869–907.

[5] Mordechai Ben-Ari (2008): *Principles of the Spin Model Checker*. Springer, London.

[6] Harsh Beohar, Mahsa Varshosaz & Mohammad Reza Mousavi (2016): *Basic behavioral models for software product lines: Expressiveness and testing pre-orders*. Science of Computer Programming 123, pp. 42–60.

[7] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2016): *Ensuring Correctness of Model Transformations While Remaining Decidable*. In Augusto Sampaio & Farn Wang, editors: Theoretical Aspects of Computing – ICTAC 2016, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 315–332.

[8] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2018): *Verifying Graph Transformation Systems with Description Logics*. In Leen Lambers & Jens Weber, editors: Graph Transformation, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 155–170.

[9] Jon Haël Brenas & Arash Shaban-Nejad (2021): *Proving the Correctness of Knowledge Graph Update: A Scenario From Surveillance of Adverse Childhood Experiences*. Frontiers in Big Data 4. Available at `https://www.frontiersin.org/article/10.3389/fdata.2021.660101`.

[10] Marsha Chechik, Michalis Famelis, Rick Salay & Daniel Strüber (2016): *Perspectives of model transformation reuse*. In: International Conference on Integrated Formal Methods, Springer, pp. 28–44.

[11] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz & Christel Baier (2018): *ProFeat: feature-oriented engineering for family-based probabilistic model checking*. Formal Aspects of Computing 30(1), pp. 45–75.

[12] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay & Jean-Francois Raskin (2013): *Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking*. IEEE Transactions on Software Engineering 39(8), pp. 1069–1089.

[13] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens & Axel Legay (2011): *Symbolic model checking of software product lines*. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 321–330.

[14] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay & Jean-François Raskin (2010): *Model checking lots of systems: efficient verification of temporal properties in software product lines*. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10, 1, ACM Press, p. 335.

[15] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans & Axel Legay (2012): *Towards an incremental automata-based approach for software product-line model checking*. In: Proceedings of the 16th International Software Product Line Conference - Volume 2, SPLC '12, Association for Computing Machinery, New York, NY, USA, pp. 74–81.

[16] Hartmut Ehrig, Karsten Ehrig, Ulrike Golas & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. XIV, Springer.

[17] Andre W.B. Furtado, Andre L.M. Santos, Geber L. Ramalho & Eduardo Santana de Almeida (2011): *Improving Digital Game Development with Software Product Lines*. IEEE Software 28(5), pp. 30–37. Conference Name: IEEE Software.

[18] Sebastian Gabmeyer (2015): *New model checking techniques for software systems modeled with graphs and graph transformations*. Ph.D. thesis.

[19] Sebastian Gabmeyer & Martina Seidl (2016): *Lightweight Symbolic Verification of Graph Transformation Systems with Off-the-Shelf Hardware Model Checkers*. In Bernhard K. Aichernig & Carlo A. Furia, editors: Tests and Proofs, 9762, Springer International Publishing, pp. 94–111.

[20] Fabio Gadducci, Alberto Lluch Lafuente & Andrea Vandin (2012): *Exploiting Over- and Under-Approximations for Infinite-State Counterpart Models*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 51–65.

[21] Amir Hossein Ghamarian & Arend Rensink (2012): *Generalised Compositionality in Graph Transformation*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 234–248.

[22] Alexander Gruler, Martin Leucker & Kathrin Scheidemann (2008): *Modeling and Model Checking Software Product Lines*. In Gilles Barthe & Frank S. de Boer, editors: *Formal Methods for Open Object-Based Distributed Systems: 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 113–131.

[23] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp & Dániel Varró (2005): *Using Graph Transformation for Practical Model-Driven Software Engineering*. In Sami Beydeda, Matthias Book & Volker Gruhn, editors: *Model-Driven Software Development*, Springer Berlin Heidelberg, pp. 91–117.

[24] Matthias Güdemann (2022): *mgudemann/iimc*. Available at `https://github.com/mgudemann/iimc`.

[25] Reiko Heckel (1998): *Compositional verification of reactive systems specified by graph transformation*. In Egidio Astesiano, editor: *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 138–153.

[26] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham.

[27] Tobias Isenberg, Dominik Steenken & Heike Wehrheim (2013): *Bounded Model Checking of Graph Transformation Systems via SMT Solving*. In Dirk Beyer & Michele Boreale, editors: *Formal Techniques for Distributed Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 178–192.

[28] Shmuel Katz (2006): *Aspect categories and classes of temporal properties*. In: *Transactions on Aspect-Oriented Software Development I*, Springer-Verlag, Berlin, Heidelberg, pp. 106–134.

[29] Tomoji Kishi & Natsuko Noda (2006): *Formal verification and software product lines*. *Communications of the ACM* 49(12), pp. 73–77.

[30] Kodkod (2017): *Kodkod: About*. Available at `https://emina.github.io/kodkod/`.

[31] Christian Krause (2021): *Henshin | The Eclipse Foundation*. Available at `https://www.eclipse.org/henshin/`.

[32] Sonja Maier & Daniel Volk (2008): *Facilitating language-oriented game development by the help of language workbenches*. In: *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, Association for Computing Machinery, New York, NY, USA, pp. 224–227.

[33] Maria Maximova, Holger Giese & Christian Krause (2018): *Probabilistic timed graph transformation systems*. *Journal of Logical and Algebraic Methods in Programming* 101, pp. 110–131.

[34] Maria Maximova, Sven Schneider & Holger Giese (2021): *Compositional Analysis of Probabilistic Timed Graph Transformation Systems*. In Esther Guerra & Mariëlle Stoelinga, editors: *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 196–217.

[35] Arend Rensink (2004): *The GROOVE Simulator: A Tool for State Space Generation*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 479–485.

[36] Arend Rensink, Ákos Schmidt & Dániel Varró (2004): *Model Checking Graph Transformations: A Comparison of Two Approaches*. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce & Grzegorz Rozenberg, editors: *Graph Transformations*, 3256, Springer Berlin Heidelberg, pp. 226–241.

[37] Ákos Schmidt & Dániel Varró (2003): *CheckVML: A Tool for Model Checking Visual Modeling Languages*. In Perdita Stevens, Jon Whittle & Grady Booch, editors: *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 92–95.

[38] Sven Schneider, Johannes Dyck & Holger Giese (2020): *Formal Verification of Invariants for Attributed Graph Transformation Systems Based on Nested Attributed Graph Conditions*. In Fabio Gadducci & Timo Kehrer, editors: *Graph Transformation*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 257–275.

[39] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf & Matthias Tichy (2017): *Henshin: A usability-focused framework for EMF model transformation development*. In: *International Conference on Graph Transformation*, Springer, pp. 196–208.

[40] Daniel Strüber, Sven Peldszus & Jan Jürjens (2018): *Taming Multi-Variability of Software Product Line Transformations*. In: *FASE*, pp. 337–355.

[41] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer & Jennifer Plöger (2018): *Variability-based model transformation: formal foundation and application*. *Formal Aspects of Computing* 30(1), pp. 133–162.

[42] Daniel Strüber & Stefan Schulz (2016): *A tool environment for managing families of model transformation rules*. In: *International Conference on Graph Transformation*, Springer, pp. 89–101.

[43] Daniel Strüber, Julia Rubin, Marsha Chechik & Gabriele Taentzer (2015): *A Variability-Based Approach to Reusable and Efficient Model Transformations*. In Alexander Egyed & Ina Schaefer, editors: *Fundamental Approaches to Software Engineering*, 9033, Springer Berlin Heidelberg, pp. 283–298.

[44] Eugene Syriani & Hans Vangheluwe (2008): *Programmed Graph Rewriting with Time for Simulation-Based Design*. In Antonio Vallecillo, Jeff Gray & Alfonso Pierantonio, editors: *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 91–106.

[45] Gabriele Taentzer, Rick Salay, Daniel Strüber & Marsha Chechik (2017): *Transformations of software product lines: A generalizing framework based on category theory*. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, pp. 101–111.

[46] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer & Gunter Saake (2014): *A Classification and Survey of Analysis Strategies for Software Product Lines*. *ACM Comput. Surv.* 47(1), pp. 6:1–6:45.

[47] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi & Ina Schaefer (2018): *A Classification of Product Sampling for Software Product Lines*. In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, Association for Computing Machinery, New York, NY, USA, pp. 1–13.

[48] Meng Zhu & Alf Inge Wang (2019): *Model-driven Game Development: A Literature Review*. *ACM Computing Surveys* 52(6), pp. 123:1–123:32.

Towards Mechanised Proofs in Double-Pushout Graph Transformation

Robert Söldner Detlef Plump
Department of Computer Science, University of York, York, UK
{rs2040,detlef.plump}@york.ac.uk

We formalise the basics of the double-pushout approach to graph transformation in the proof assistant Isabelle/HOL and provide associated machine-checked proofs. Specifically, we formalise graphs, graph morphisms and rules, and a definition of direct derivations based on deletion and gluing. We then formalise graph pushouts and prove with Isabelle's help that both deletions and gluings are pushouts. We also prove that pushouts are unique up to isomorphism. The formalisation comprises around 4000 lines of source text. Our motivation is to pave the way for rigorous, machine-checked proofs in the theory of the double-pushout approach, and to lay the foundations for verifying graph transformation systems and rule-based graph programs by interactive theorem proving.

1 Introduction

Software faults may lead to unexpected system's behaviour with a significant loss of goods or even personal harm. Documented examples of system failures range from medical devices [15] over space launch vehicles [7] to hardware design [13]. To prevent software faults, formal methods such as static analysis or program verification continue to attract a considerable amount of research.

Computing by rule-based graph transformation provides an intuitive and visual approach to specification and programming. Here, the main formal concepts for ensuring correctness are model checking [22, 25, 2, 21, 10] and proof-based verification [12, 14, 20, 24, 6, 23, 4, 27]. One of the oldest and most established approaches to graph transformation is the double-pushout (DPO) approach, where rule applications are defined by a pair of pushouts in the category of graphs [9]. Formal proofs in the DPO approach come in two flavours, they either establish results in the DPO theory (such as the commutativity of independent rule applications) or they show the correctness of concrete graph transformation systems and graph programs.

While main stream formal methods increasingly employ proof assistants such as Coq [3] or Isabelle [16] to obtain rigorous, machine-checked proofs, to the best of our knowledge such tools have not yet been used in the area of DPO graph transformation. In this paper, we report on first steps towards the formalisation of the DPO theory in the Isabelle proof assistant. Specifically, we focus on linear rules with injective matching and show how to formalise (labelled, directed) graphs, morphisms, and rules. (Note that injective matching is more expressive than unrestricted matching because each rule can be replaced by the set of its quotient rules, and selected quotients can be omitted [11]). We give an operational definition of direct derivations based on deletion and gluing. We then formalise graph pushouts and prove with Isabelle's help that both deletion and gluing are pushouts. We also prove that pushouts are unique up to isomorphism.

We stress that we do not intend to formalise an abstract theoretical framework such as adhesive categories [9], nor do we aim at covering all kinds of graphs that one can find in the DPO literature such as infinite graphs, hypergraphs, typed graphs, etc. Rather, we are interested in concrete constructions on

graphs such as deletion and gluing, and how they relate to the double-pushout formulation. Our long-term goal is to provide interactive and automatic proof support for formal reasoning on programs in a graph transformation language such as GP 2 [5]. The underlying formalisation in Isabelle will inevitably have to deal with the concrete graphs, labels, rules, etc. that are the ingredients of such programs.

To summarize, this paper makes the following contributions:

- We formalise in Isabelle the basics of the DPO approach with injective matching.
- We prove that the operational construction of direct derivations by deletion and gluing gives rise to a double-pushout diagram.
- We prove that graph pushouts are unique up to isomorphism.

We believe that this is the first formalisation of DPO-based graph transformation in a theorem prover. The formalisations and proofs were developed using the Isabelle 2021 proof assistant. The entire formalisation comprises around 4000 lines of source text and can be accessed from GitHub[1].

Currently, we are revising our design in order to increase readability and facilitate certain automated proofs (see the comments in the conclusion). The rest of the paper is structured as follows: Section 2 briefly reviews the theoretical background required in this research. Section 3 will provide selected examples of our formalisation using the proof assistant Isabelle. Finally, in Section 4, the paper is summarised, some design decisions are discussed, and future work is stated.

2 Graphs, Rules and Derivations

This section reviews basic terminology and results regarding graphs, rules, and derivations in the double-pushout approach with injective matching; see for example [9, 11]. In Section 3, we formalise these definitions and results in Isabelle.

Definition 1 (Label alphabet)**.** A *label alphabet* $\mathscr{L} = (\mathscr{L}_V, \mathscr{L}_E)$ consists of a set $\mathscr{L}_V$ of node labels and a set $\mathscr{L}_E$ of edge labels. □

We define directed and labelled graphs and allow parallel edges and loops. We do not consider variables as labels.

Definition 2 (Graph)**.** A *graph* $G = (V, E, s, t, l, m)$ over the alphabet $\mathscr{L}$ is a system where V is the finite set of nodes, E is the finite set of edges, $s, t\colon E \to V$ functions assigning the source and target to each edge, $l\colon V \to \mathscr{L}_V$ and $m\colon E \to \mathscr{L}_E$ are functions assigning a label to each node and edge. □

Next we review graph morphisms which are structure-preserving mappings between graphs. We describe our Isabelle formalisation in Subsection 3.1.

Definition 3 (Graph morphism)**.** A *graph morphism* $f\colon G \to H$ is a pair of mappings $f = (f_V\colon V_G \to V_H, f_E\colon E_G \to E_H)$, such that for all $e \in E_G$ and $v \in V_G$:

1. $f_V(s_G(e)) = s_H(f_E(e))$ (sources are preserved)
2. $f_V(t_G(e)) = t_H(f_E(e))$ (targets are preserved)
3. $l_G(v) = l_H(f_V(v))$ (node labels are preserved)
4. $m_G(e) = m_H(f_E(e))$ (edge labels are preserved) □

We also define some special forms of morphisms.

[1] https://github.com/UoYCS-plasma/DPO-Formalisation

Definition 4 (Special morphisms and isomorphic graphs)**.** A morphism f is *injective* (*surjective*, *bijective*) if f_V and f_E are injective (surjective, bijective). Morphism f is an *inclusion* if for all $v \in V_G$ and $e \in V_E$, $f_V(v) = v$ and $f_E(e) = e$. A bijective morphism is an *isomorphism*. In this case, G and H are *isomorphic*, which is denoted by $G \cong H$. □

The composition of two morphisms yields a well-defined morphism, which we prove in Subsection 3.1.

Definition 5 (Morphism composition)**.** Let $f\colon F \to G$ and $g\colon G \to H$ be graph morphisms. The *morphism composition* $g \circ f\colon F \to H$ is defined by $g \circ f = (g_V \circ f_V, g_E \circ f_E)$. □

In DPO-based graph transformation, rules are the atomic units of computation. We describe the formalisation of rules in Subsection 3.5.

Definition 6 (Rule)**.** A *rule* $(L \leftarrow K \rightarrow R)$ consists of graphs L, K and R over $\mathscr{L}$ together with inclusions $K \to L$ and $K \to R$. □

The addition of graph components along a common subgraph is called *gluing*. We present our Isabelle formalisation in Subsection 3.3. The gluing construction below uses the disjoint union of sets A and B defined by $A + B = (A \times \{1\}) \cup (B \times \{2\})$. It comes with injective functions $i_A\colon A \to A + B$ and $i_B\colon B \to A + B$ such that $i_A(A) \cup i_B(B) = A + B$ and $i_A \cap i_B = \emptyset$. To keep the rest of this section readable, we tacitly assume that the injections i_A and i_B are inclusions. Only in Subsection 3 we will be dealing explicitly with the injections. We prove the correspondence between the gluing construction and pushouts in Subsection 3.3.

Lemma 1 (Gluing [8])**.** *Let $(L \leftarrow K \rightarrow R)$ be a rule and $d\colon K \to D$ an injective graph morphism. Then the following defines a graph H (see Fig. 1a), the gluing of D and R according to d:*

1. $V_H = V_D + (V_R - V_K)$
2. $E_H = E_D + (E_R - E_K)$
3. $s_H(e) = \begin{cases} s_G(e) & \text{if } e \in E_D \\ d_V(s_R(e)) & \text{if } e \in E_R - E_K \text{ and } s_R(e) \in V_K \\ s_R(e) & \text{otherwise} \end{cases}$
4. t_H *analogous to* s_H
5. $l_H = \begin{cases} l_D(v) & \text{if } v \in V_D \\ l_R(v) & \text{otherwise} \end{cases}$
6. m_H *analogous to* l_H

Moreover, the morphism $D \to H$ is an inclusion and the injective morphism h is defined for all items x in R by $h(x) =$ ***if*** *$x \in R - K$* ***then*** *x* ***else*** *$d(x)$.*

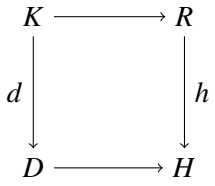
The dangling condition ensures that deletion results in a well-defined graph.

Definition 7 (Dangling condition)**.** Let $(L \leftarrow K \rightarrow R)$ be a rule. An injective graph morphism $g\colon L \to G$ satisfies the *dangling condition* if no edge in $E_G - g_E(E_L)$ is incident to a node in $g_V(V_L - V_K)$. □

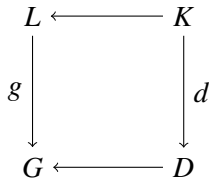
The following *deletion* of graph components is formalised in Subsection 3.4.

Lemma 2 (Deletion [8])**.** *Let $(L \leftarrow K \rightarrow R)$ be a rule and $g\colon L \to G$ an injective graph morphism satisfying the dangling condition (see Fig. 1b). Then:*

1. *$V_D = V_G - g_V(V_L - V_K)$ and $E_D = E_G - g_E(E_L - E_K)$ induce the inclusion $D \to G$, and*
2. *there is an injective graph morphism $d\colon K \to D$, defined by $d(x) = g(x)$ for all items x in K.*



(a) Gluing diagram



(b) Deletion diagram

Figure 1: Gluing and deletion diagram

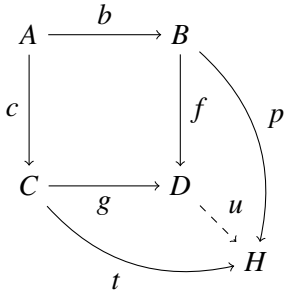


Figure 2: A Pushout Diagram

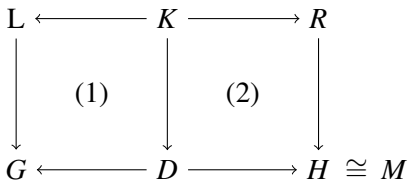


Figure 3: A Direct Derivation

Definition 8 (Pushout)**.** Given graph morphisms $b\colon A \to B$ and $c\colon A \to C$, a graph D together with graph morphisms $f\colon B \to D$ and $g\colon C \to D$ is a *pushout* of $A \to B$ and $A \to C$ if the following holds (see Fig. 2):

1. Commutativity: $f \circ b = g \circ c$
2. Universal property: For all graph morphisms $p\colon B \to H$ and $t\colon C \to H$ such that $p \circ b = t \circ c$, there is a unique morphism $u\colon D \to H$ such that $u \circ f = p$ and $u \circ g = t$. □

The formalisation of pushouts and the proof that pushouts are unique up to isomorphism is presented in Subsection 3.2.

Theorem 1 (Uniqueness of pushouts [1])**.** *Let $A \to B$ and $A \to C$ together with D induce a pushout as depicted in Fig. 2. A graph H together with morphisms $B \to H$ and $C \to H$ is a pushout of b and c if and only if there is an isomorphism $u\colon D \to H$ such that $u \circ f = p$ and $u \circ g = t$.*

Theorem 2 (Gluings are pushouts [8])**.** *Let $K \to R$ be an inclusion, $d\colon K \to D$ be an injective graph morphism, and H be the gluing of D and R according to d, as defined in Lemma 1. Then, the square in Fig. 1a is a pushout diagram where $D \to H$ is an inclusion and h is defined by $h(x) =$ **if** $x \in R - K$ **then** x **else** $d(x)$. We call H the pushout object.*

The deletion construction of Lemma 2 and the following theorem are formalised and proved in Subsection 3.4.

Theorem 3 (Deletions are pushouts [8])**.** *Let $K \to L$ be an inclusion, $g\colon L \to G$ be an injective graph morphism satisfying the dangling condition, and D be subgraph of G as defined in Lemma. 2. Then, the square in Fig. 1b is a pushout diagram where g is an inclusion and $d(x) = g(x)$ for all items x in K. We call D the pushout complement.*

The following definition of rule application is formalised in Subsection 3.5.

Definition 9 (Direct derivation)**.** Let $r = (L \leftarrow K \rightarrow R)$ be a rule and $g\colon L \rightarrow G$ be an injective graph morphism satisfying the dangling condition. Then G directly derives (see Fig. 3) M by r and g, denoted by $G \Rightarrow_{r,g} M$, if $H \cong M$, where H is constructed from G by:

1. (Deletion) D is the subgraph $G - g(L - K)$.
2. $d\colon K \rightarrow D$ is the restriction of g to K and D.
3. (Gluing) H is the gluing $H = D + (R - K)$. □

Note that by Theorem 2 and Theorem 3, square (1) and (2) in Fig. 3 are pushouts. The next section provides a general introduction to the Isabelle proof assistant and highlights selected parts of our formalisation.

3 DPO Formalisation in Isabelle/HOL

Isabelle is a generic, interactive theorem prover based on the so-called *LCF* approach. It is based on a small (meta-logical) proof kernel, which is responsible for checking all proofs. This concept provides high confidence in the prover's soundness. Isabelle/HOL refers to the higher-order logic instantiation which is considered to be the most established calculus within the Isabelle distribution [19].

Type variables are denoted by a leading apostrophe. For example, a term `f` of type `'a` is denoted by `f :: 'a`. The HOL library provides a large collection of definitions, including the option data type `'a option = Some 'a | None` which is used to denote the absence of a value (`None`) or the presence (`Some`). Partial functions (called *maps*) are modelled using the option type `'a ⇒ 'b option` with the infix notation `'a ⇀ 'b`.

Our formalisation is based on Isabelle's *locales*, a mechanism for writing parametric specifications. Furthermore, we use *intelligible semi-automated reasoning* (Isar) which is Isabelle's language of writing structured proofs [26]. In contrast to *apply-scripts*, which execute deduction rules in a linear manner, Isar follows a structured approach resulting in increased readability and maintainability [17].

A general introduction to Isabelle/HOL can be found in [16]. The main components of our formalisation and their interdependencies are depicted in Fig.4. The simple arrow ($\rightarrow$) can be read as "depends on", i.e., the definition of morphisms depends on the definition of graphs allowing the inheritance of properties. The blue arrow $\Longrightarrow$ highlights main theorems proven in this study, viz. that the gluing and deletion constructions correspond to pushouts and that pushout objects are unique up to isomorphism.

3.1 Graphs and Morphisms

Our definition of graphs (Def. 2) is different from Strecker's [23] where a graph is a set of nodes together with a binary relation of nodes. A consequence of Strecker's definition is the absence of parallel edges and the absence of edge labels. Moreover, the graphs in [23] do not have edge labels.

We represent the node and edge sets of graphs by two arbitrary but fixed types `'v` and `'e`. Similarly, `'`$\mathscr{L}_V$ and `'`$\mathscr{L}_E$ are types representing the universes of possible node and edge labels.

Since functions in Isabelle/HOL are defined over complete types (e.g., `nat`), we usually have to restrict our reasoning to a subset of the values of a type. There are two main approaches for this, the use of partial functions (maps) or the explicit specification of the defined reasoning subset. In this paper, we use partial functions but discuss the alternative to rely on total functions in Section 4.

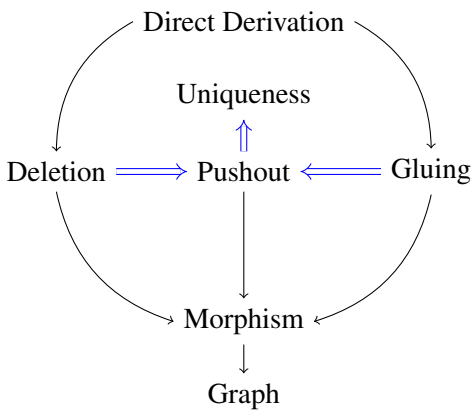


Figure 4: Overview of component dependencies (→) and major theorems (⇒)

Notation In the following, a partial function with a trailing apostrophe indicates the equivalent total function over the partial function's domain of definition. For example, the total function corresponding to the (partial) source mapping `s :: "'e ⇀ 'v"` is denoted by `s'`.

The locale graph declares all parameters using the `fixes` keyword, the corresponding types are given explicitly. With the `defines` keyword, we declare our total functions to be equivalent (with respect to extensional equality) to the transformer function `totalize` applied to the partial functions. The `totalize` function is defined by using the `abbreviation` keyword, which introduces a syntactic constant that is associated with the term behind the ≡ symbol.

```
abbreviation totalize :: "('a ⇀ 'b) ⇒ 'a ⇒ 'b" where
"totalize f a ≡ the (f a)"
```

The `simp` attribute, indicated by the brackets, tells the simplifier to consider this rule automatically. Premises for the locale context are specified using the `assumes` keyword. The premises for this formalisation are, the set of nodes and edges are finite, the source and target domains are the set of edges (e.g., `dom s = E`) while the range is a subset of the nodes (e.g., `ran s ⊆ V`), and the node and edge labelling function are defined for all nodes (edges) respectively, i.e., `dom l = V` and `dom m = E`.

```
locale graph =
  fixes
    V :: "'v set" and E :: "'e set" and
    s :: "'e ⇀ 'v" and s':: "'e ⇒ 'v" and
    t :: "'e ⇀ 'v" and t':: "'e ⇒ 'v" and
    l :: "'v ⇀ '𝒱" and l':: "'v ⇒ '𝒱" and
    m :: "'e ⇀ '𝓔" and m':: "'e ⇒ '𝓔"
  defines
    s_def[simp]: "s' ≡ totalize s" and t_def[simp]: "t' ≡ totalize t" and
    l_def[simp]: "l' ≡ totalize l" and m_def[simp]: "m' ≡ totalize m"
assumes
  finite_nodes: "finite V" and finite_edges: "finite E" and
  integrity_s: "dom s = E ∧ ran s ⊆ V" and
  integrity_t: "dom t = E ∧ ran t ⊆ V" and
  integrity_l: "dom l = V" and integrity_m: "dom m = E"
```

The built-in (defined in the standard library of Isabelle/HOL) functions `dom` and `ran` are defined for partial functions only and their definition is as follows.

```
definition dom :: "('a ⇀ 'b) ⇒ 'a set" where "dom m = {a. m a ≠ None}"
```

The domain of a map `m` is the set of values which applied to `m` result in some value, i.e., does not result in `None`. The range of a map `m` is the set of elements, `m` projects to.

```
definition ran :: "('a ⇀ 'b) ⇒ 'b set" where
"ran m = {b. ∃ a. m a = Some b}"
```

In contrast to the formalisation of graphs by Noschinski [18], we do not use an additional data structure to represent graphs in this paper. (Meanwhile, we have started to revise this design; see the discussion in the Conclusion.) Moreover, we allow nodes and edges to carry labels. The use of partial functions allows to express function equality directly, without the need to specify the domain. That is, we may write $f = g$ without a quantifier over the domain as in $\forall x \in A.\ fx = gx$.

The definition of morphisms (see Def. 3) follows a similar approach. The `locale morphism` makes use of so-called *locale instances*, which allows the usage of quantifiers to reference locale instances which are imported in the current context. A morphism carries its domain (`G: graph ...`), the codomain (`H: graph ...`), and two mappings between node and edge sets (g_V and g_E). The `for` keyword is used to define instance parameters explicitly, including the optional type attribute. In our case, the types of the domain (V_G `:: 'v`$_G$ `set` and E_G `:: 'e`$_G$ `set`) and codomain (V_H `:: 'v`$_H$ `set` and E_H `:: 'e`$_H$ `set`) graph are used to define the node (edge) mappings between both graphs.

```
locale morphism =
  G: graph V_G E_G s_G s'_G t_G t'_G l_G l'_G m_G m'_G +
  H: graph V_H E_H s_H s'_H t_H t'_H l_H l'_H m_H m'_H for
  V_G::"'v_G set" and E_G::"'e_G set" and s_G s'_G t_G t'_G l_G l'_G m_G m'_G and
  V_H::"'v_H set" and E_H::"'e_H set" and s_H s'_H t_H t'_H l_H l'_H m_H m'_H +
  fixes
    g_V :: "'v_G ⇀ 'v_H" and g'_V :: "'v_G ⇒ 'v_H" and
    g_E :: "'e_G ⇀ 'e_H" and g'_E :: "'e_G ⇒ 'e_H"
  defines
    g_V_def[simp]: "g'_V ≡ totalize g_V" and
    g_E_def[simp]: "g'_E ≡ totalize g_E"
  assumes
    integrity_v: "dom g_V = V_G ∧ ran g_V ⊆ V_H" and
    integrity_e: "dom g_E = E_G ∧ ran g_E ⊆ E_H" and
    source_preserve: "g_V ∘_m s_G = s_H ∘_m g_E" and
    target_preserve: "g_V ∘_m t_G = t_H ∘_m g_E" and
    node_label_preserve: "l_G = l_H ∘_m g_V" and
    edge_label_preserve: "m_G = m_H ∘_m g_E"
```

The morphism axioms include that mapping preserves the integrity of sources (3.1), targets (3.2) and labels (3.3 and 3.4). The built-in composition for maps is defined as follows:

```
definition map_comp :: "('b ⇀ 'c) ⇒ ('a ⇀ 'b) ⇒ ('a ⇀ 'c)" where
"f ∘_m g = (λk. case g k of None ⇒ None | Some v ⇒ f v)"
```

The evaluation of `f ∘`$_m$ `g` will case-split on the result of evaluating `g`. The `None` case will terminate the computation by returning `None`, while the `Some v` case will evaluate `f v` as its result. With this definition, we can prove that given two graph morphisms $g = (g_V, g_E)\colon G \to H$ and $k = (k_V, k_E)\colon H \to K$, the pairwise composition $k \circ g = (k_V \circ g_V, k_E \circ g_E)$ is a well-defined graph morphism. This lemma is expressed in Isabelle/HOL as follows:

```
lemma
  assumes
    g:"morphism V_G E_G s_G t_G l_G m_G V_H E_H s_H t_H l_H m_H g_V g_E" and
    k:"morphism V_H E_H s_H t_H l_H m_H V_L E_L s_L t_L l_L m_L k_V k_E"
  shows "morphism V_G E_G s_G t_G l_G m_G V_L E_L s_L t_L l_L m_L
    (k_V ∘_m g_V) (k_E∘_m g_E)"
proof intro_locales
```

A proof is initiated by the `proof` statement followed by an optional proof method. Here, we use the `intro_locales` proof method. This method will expand all introduction rules generated by the usage of `locales`. Isabelle generates the following output:

```
proof (state)
goal (3 subgoals):
 1. graph V_G E_G s_G t_G l_G m_G
 2. graph V_L E_L s_L t_L l_L m_L
 3. morphism_axioms V_G E_G s_G t_G l_G m_G V_L E_L s_L t_L l_L m_L
        (k_V ∘_m g_V) (k_E ∘_m g_E)
```

It highlights three subgoals the user has to discharge. The proof obligations (1) and (2) follow directly from the assumptions `g` and `k`, respectively. In Isar, we explicitly state the obligations using the `show` keyword. The proof of (1) is as follows:

```
show "graph V_G E_G s_G t_G l_G m_G"
  using g morphism_def by blast
```

Here, we are telling Isabelle to use the fact `g`, expressing that $g\colon G \to H$ is graph morphism, and our definition of morphisms (the fact `morphism_def` is automatically generated by Isabelle) using the `using` keyword. Ultimately, we propose to Isabelle to discharge the goal by using the supplied facts with the tableau prover `blast` (indicated by `by blast`). This goal is automatically proved by Isabelle using the supplied facts. The proof of (2) follows analogously. Goal (3) makes use of the automatically generated `morphism_axioms` predicate which contains all the assumptions stated in the morphism locale. A direct proof using a standard proof method, such as `blast`, fails. This proof obligation requires manual support from the user and we initiated the proof by the `standard` proof method (which is the default and can thus be omitted):

```
show "morphism_axioms V_G E_G s_G t_G l_G m_G V_L E_L s_L t_L l_L m_L (k_V ∘_m g_V)"
proof
```

Isabelle generated the upcoming six subgoals to discharge the goal:

```
proof (state)
goal (6 subgoals):
 1. dom (k_V ∘_m g_V) = V_G ∧ ran (k_V ∘_m g_V) ⊆ V_L
 2. dom (k_E ∘_m g_E) = E_G ∧ ran (k_E ∘_m g_E) ⊆ E_L
 3. k_V ∘_m g_V ∘_m s_G = s_L ∘_m (k_E ∘_m g_E)
 4. k_V ∘_m g_V ∘_m t_G = t_L ∘_m (k_E ∘_m g_E)
 5. l_G = l_L ∘_m (k_V ∘_m g_V)
 6. m_G = m_L ∘_m (k_E ∘_m g_E)
```

Whereby subgoals (1) and (2) reflect the domain and range restrictions of the composed morphisms, (3)-(6) correspond to the structure preserving property of source (Def. 3.1), target (Def. 3.2) and labels (Def. 3.3 and Def. 3.4). The proof of the domain of subgoal (1) and structure preserving under the composed mapping of (3) is shown below:

```
show "dom (k_V ∘_m g_V) = V_G"
  using morphism.dom_gv[OF k] morphism.dom_gv[OF g]
    morphism.integrity_v'[OF g]
  by fastforce
```

We prove the lemma by stating the facts manually, indicated by the usage of the `using` keyword. The `fastforce` proof method is able to discharge the obligation. The intuition behind `x[OF ...]` is the application of the theorem `x` to the argument ..., similar to function application but for theorems. We can thus specialise the `morphism.dom_gv` lemma to the graph `g` and `k`. In fact, the Sledgehammer tool is able to find a proof directly using the *verit* smt solver. Sledgehammer applies automatic theorem provers

(ATMs) and satisfiability-modulo-theories (SMT) solvers to the current (sub) goal and reconstructs each proof. The subgoal (3), the structure preserving under the source function, follows the textbook proof of [9] whereby in Isabelle we have to justify each proof step in more detail.

```
show "(k_V ∘m g_V) ∘m s_G = s_L ∘m (k_E ∘m g_E)"
  proof -
    have "(k_V ∘m g_V) ∘m s_G = (k_V ∘m s_H) ∘m g_E"
      by (simp add: map_comp_assoc morphism.source_preserve[OF g])
    also have "... = s_L ∘m (k_E ∘m g_E)"
      by (simp add: map_comp_assoc morphism.source_preserve[OF k])
    finally show ?thesis .
  qed
```

The proof requires the associativity of partial function composition, which was not part of Isabelle's standard library and thus proved previously by us in a separate lemma. The `source_preserve` fact is specialised (using `[OF g]`) to the morphism g. The remaining subgoals work similar. Isar provides a variaty of predefined abbreviations such as `?thesis` which is bound to the proof goal, "..." which refer to the right hand side of the previous stated expression, or the short proof method "." which is a synonym for `by this`. The interested reader will find additional abbreviations and explanations in the Isar reference.

3.2 Pushouts

This subsection formalises pushouts in Isabelle and proves their uniqueness up to isomorpism (cf. 1). The pushout characterisation comprises four commuting morphisms ($A \to B$, $A \to C$, $B \to D$, and $C \to D$) which satisfy the universal property (cf. Def. 8). The commuting property is expressed using the node ($f_V \circ_m b_V = g_V \circ_m c_V$) and edge ($f_E \circ_m b_E = g_E \circ_m c_E$) composition of morphisms. Our formalisation is as follows:

```
locale pushout_diagram =
  b: morphism V_A ... V_B ... b_V b'_V b_E b'_E +
  c: morphism V_A ... V_C ... c_V c'_V c_E c'_E +
  f: morphism V_B ... V_D ... f_V f'_V f_E f'_E +
  g: morphism V_C ... V_D ... g_V g'_V g_E g'_E for ... +
  assumes
    node_commutativity: "f_V ∘m b_V = g_V ∘m c_V" and
    edge_commutativity: "f_E ∘m b_E = g_E ∘m c_E" and
    universal_property:
      "⟦ graph (V_H :: 'v set) (E_H :: 'e set) s_H t_H l_H m_H;
        morphism V_B E_B s_B t_B l_B m_B V_H E_H s_H t_H l_H m_H p_V p_E;
        morphism V_C E_C s_C t_C l_C m_C V_H E_H s_H t_H l_H m_H t_V t_E;
        p_V ∘m b_V = t_V ∘m c_V; p_E ∘m b_E = t_E ∘m c_E ⟧
        ⟹ ∃! (u_V, u_E).
            morphism V_D E_D s_D t_D l_D m_D V_H E_H s_H t_H l_H m_H u_V u_E
              ∧ u_V ∘m f_V = p_V ∧ u_E ∘m f_E = p_E
              ∧ u_V ∘m g_V = t_V ∧ u_E ∘m g_E = t_E"
```

In Isabelle, unbound variables are implicitly bound using the (meta) allquantor indicated by $\bigwedge$, i.e. `lemma "P x"` will be interpreted as `lemma "⋀x. P x"`. This fact is used to express the universal property "for all graphs H ...". Here, our approach of using partial functions also paid off. The built-in unique existential quantifier is defined as follows:

```
definition Ex1 :: "('a ⇒ bool) ⇒ bool"
  where "Ex1 P ≡ ∃x. P x ∧ (∀y. P y → y = x)"
```

Since we quantify over partial functions, we can reason about equality (i.e. `x = y`) without being restricted to existential equality. Within the context of the `pushout_diagram` locale, we prove the uniqueness of the pushout object, expressed by the following lemma.

```
lemma uniqueness_po:
  fixes V_D' :: "'v set" and E_D' :: "'e set"
  assumes
    D': "graph V_D' E_D' s_D' t_D' l_D' m_D'" and
    p': "morphism V_B E_B s_B t_B l_B m_B V_D' E_D' s_D' t_D' l_D' m_D' p_V p_E" and
    d': "morphism V_C E_C s_C t_C l_C m_C V_D' E_D' s_D' t_D' l_D' m_D' q_V q_E"
  shows "pushout_diagram V_A E_A s_A t_A l_A m_A V_B E_B s_B t_B l_B m_B
    V_C E_C s_C t_C l_C m_C V_D' E_D' s_D' t_D' l_D' m_D' b_V b_E c_V c_E p_V p_E q_V q_E
  ⟷ ∃ (u_V u_E).
    bijective_morphism V_D E_D s_D t_D l_D m_D V_D' E_D' s_D' t_D' l_D' m_D' u_V u_E
  ∧ u_V ∘_m g_V = q_V ∧ u_E ∘_m g_E = q_E ∧ u_V ∘_m f_V = p_V ∧ u_E ∘_m f_E = p_E"
```

The proof is around 350 lines of code and follows mostly from the universal property. Subsequently, the gluing and deletion constructions are given and their pushout correspondence is shown.

3.3 Gluings are Pushouts

The gluing locale is used as an environment with the required preconditions as described in Lemma 1. In this context, we first define the gluing construction depicted in Fig. 1a. In the special case where k is an inclusion, the gluing graph `D` can be constructed using the disjoint union of the nodes and edges, denoted by $+$, i.e. $V = V_D + (V_R - V_K)$. In Isabelle/HOL we use the built-in `sum` type which implements the disjoint sum of two types. The `Inl` and `Inr` definitions can be used to lift arbitrary values into the sum encoding and correspond to the definition if i_A and i_B of Section 3.1. The set of nodes can be constructed as follows:

```
abbreviation V where "V ≡ Inl ` V_D ∪ Inr ` (V_R - V_K)"
```

The infix operator ` (backtick) corresponds to the image function of sets, which applies the given function to all elements of the second argument, e.g. the expression `Inl ` {1::nat} ∪ Inr ` {1::nat}` will evaluate to `{Inl 1, Inr 1}`. The edge construction `E` follows analogously. The source (target) function is defined using the `fun` keyword, which automatically tries to prove important properties such as termination and completeness. In case the automation fails, the user has to prove these properties by hand. In our case, Isabelle was able to discharge all the required goals automatically. The source function is given by:

```
fun s where
  "s (Inl e') = map_option Inl (s_D e')"
| "s (Inr e') = (if e' ∈ E_R - E_K ∧ s'_R e' ∈ V_K
    then map_option Inl ((f_V ∘_m s_R) e')
    else map_option Inr ((s_R|`(E_R-E_K)) e'))"
```

The intuition here is to first case-split on the sum encoding to derive the potential edge origin (`E_D` or `E_R`), strongly following Lemma 1. The target function is defined analogously. The labelling function is defined using the `case_sum` keyword which will case-split on the sum-encoded argument (the edge) and applies the corresponding argument (first or second).

```
abbreviation l where "l ≡ case_sum l_D (l_R |` (V_R-V_K))"
```

The intuition here is, if the edge belongs to D, we can directly use l_D, if not the edge might belong to $R-K$ and therefore we have to restrict l_R. The edge labelling function is analogously. With all components defined, we can prove the components form a graph by *interpreting* the graph locale and discharge all proof goals generated by Isabelle.

```
interpretation object:
  graph V E s "totalize s" t "totalize t" l "totalize l" m "totalize m"
proof unfold_locales
```

The proof is mechanic and follows case splitting of the sum type. For the pushout construction, we define the morphisms $j\colon D \to H$ and $h\colon R \to H$ (see Fig. 1a). Since h is an inclusion the node mapping by:

```
definition j_V where "j_V ≡ λv. if v ∈ V_D then Some (Inl v) else None"
```

The node mapping returns `Some` in case `v` is in V_D, otherwise `None` is returned, the edge definition j_E is analogously. To prove this definition is indeed a valid inclusion, we interpret the `inclusion_morphism` locale. Defining h_V with the argument v will return `Some (Inr v)` in case, v is a node to be added, i.e. if $v \in V_R$ - V_K. Otherwise, the node (if in the graph) must belong to `K` and using the fact that `k` is an inclusion, we can use f_V.

```
definition h_V where
"h_V ≡ λv. if v ∈ V_R - V_K then Some (Inr v) else map_option Inl (f_V v)"
```

The `map_option` functor allows to apply the `Inl` projection in case f_V `v` evaluates to `Some`. We follow by interpretation of the `injective_morphism` locale, which restricts the morphism to be injective.

Ultimately, we show that this construction corresponds to the pushout object,see Theorem 2, also by interpretation of the `pushout_diagram` locale, which automatically generates the upcoming three subgoals:

```
proof (state)
goal (3 subgoals):
 1. h_V ∘_m k_V = j_V ∘_m f_V
 2. h_E ∘_m k_E = j_E ∘_m f_E
 3. ⋀ V_H E_H s_H t_H l_H m_H p_V p_E t_V t_E.
       ⟦ graph V_H E_H s_H t_H l_H m_H;
        morphism V_R E_R s_R t_R l_R m_R V_H E_H s_H t_H l_H m_H p_V p_E;
        morphism V_D E_D s_D t_D l_D m_D V_H E_H s_H t_H l_H m_H t_V t_E;
        p_V ∘_m k_V = t_V ∘_m f_V; p_E ∘_m k_E = t_E ∘_m f_E ⟧
       ⟹ ∃!(u_V, u_E).
             morphism V E s t l m V_H E_H s_H t_H l_H m_H u_V u_E ∧
             u_V ∘_m h_V = p_V ∧ u_E ∘_m h_E = p_E ∧
             u_V ∘_m j_V = t_V ∧ u_E ∘_m j_E = t_E
```

The first two subgoals (1) and (2) are analogous by case-splitting, here we show the proof of (1):

```
 show "h_V ∘_m k_V = j_V ∘_m f_V" (is "?f = ?g")
 proof
   show "?f x = ?g x" for x
   proof (cases "x ∈ V_K")
     case True
     then show ?thesis
      unfolding j_V_def h_V_def
      using i.inclusion_nodes_alt i.v_morph_not_none
        m.integrity_v' m.v_morph_not_none by fastforce
   next
     case False
     then show ?thesis
      using i.dom_gv m.dom_gv by auto
   qed
 qed
```

The proof method states `cases "x ∈ V_K"` which will introduce two subgoals: (1) `x` is in V_K and (2) it is not. The proof of (1) follows from the definition of both mappings, the integrity of the injective morphism, and the inclusion and subgoal (2) follows from the morphisms domain.

3.4 Deletions are Pushouts

The deletion locale is used as an environment with the required preconditions as stated in Lemma 2. The dangling condition (cf. Def. 7) is stated as:

```
"e ∈ E_G - f'_E ‘ E_L ⟹ s'_G e ∉ f'_V ‘ (V_L-V_K) ∧ t'_G e ∉ f'_V ‘ (V_L-V_K)"
```

With our introduction of the total equivalent functions (denoted with a trailing apostrophe), we can directly express the condition without the need to care about the `option` datatype. The construction of the graph D removes all nodes and edges from G which belong to the subgraph L without K under the morphism f. The set of nodes is constructed by:

```
abbreviation V where "V ≡ V_G - f'_V ‘ (V_L - V_K)"
```

The edge set is defined analogous. The `|‘` infix operator (`restrict_map`) does restrict the partial function `m` to the domain `A`. The source function is the restriction of s_G to the edge set E and can be defined as:

```
abbreviation s where "s ≡ s_G |‘ E"
```

The target and both labelling functions are defined analogous. We followed by proving this construction corresponds to a well-founded graph by interpretation the `graph` locale, as described in Sec. 3.3. Within this setup, we define two additional graph morphisms to make the diagram, depicted in Fig. 1b, commute. The injective morphism $d\colon K \to D$ which is the restriction of the match f is defined by:

```
definition d_V where "d_V ≡ f_V |‘ V_K"
```

The edge mapping is analogous. The inclusion $g\colon D \to G$ is defined by:

```
definition g_V where "g_V ≡ λv. if v ∈ V then Some v else None"
```

The edge mapping is analogous. The proofs (injectivity and inclusion) are given by interpreting the corresponding locale. Ultimately, we prove that the constructed pushout complement together with the two morphisms form a pushout diagram, see Theorem 3. The proof is by interpretation and the details are omitted to conserve space. The subsequent section introduces our formalisation of rules and direct derivations.

3.5 Rules and Derivations

A rule accordingly to Def. 6 can be characterised by the two inclusions $K \to L$ and $K \to R$. We model this in Isabelle using the `locale rule` as follows:

```
locale rule =
  l: inclusion_morphism V_K ... V_L ... +
  r: inclusion_morphism V_K ... V_R ...
  for ...
```

As we have explicitly included the domain and codomain in our definition of morphisms, we inherit all properties and Isabelle‘s locales mechanism allows this dense definition. A direct derivation (cf. Def. 9) is built on top of existing constructs by using the `locale` import capability:

```
locale direct_derivation =
  r: rule V_K ... V_L ... V_R ... +
  k: injective_morphism V_L ... V_G ... +
  d: deletion V_L ... V_K ... V_G ... +
  g: gluing
    V_K E_K s_K s'_K t_K t'_K l_K l'_K m_K m'_K
    V_R E_R s_R s'_R t_R t'_R l_R l'_R m_R m'_R
    d.V d.E d.s _ d.t _ d.l _ d.m _
  for ...
```

This setup is depicted in Fig. 3. We calculate the output graph by sequential execution of the deletion and gluing construction, if the assumptions (valid rule, injective match, and dangling condition) are met.

4 Conclusion and Future Work

Formal verification increases the trustworthiness of software. In this paper, we present a formalisation of the double-pushout approach with injective matching over node and edge labelled directed graphs, in the proof assistant Isabelle/HOL. The formalisation uses the extensible locale mechanism, which allows us to combine theories and to structure our work.

We first formalise graphs and morphisms and prove several properties, such as the well-definedness of morphism composition. Direct derivations are introduced in terms of deletion and gluing. We prove their correspondence to double-pushouts. We also prove that pushouts are unique up to isomorphism. Although Isabelle is not able to discharge most of the generated proof obligations automatically, the available proof methods support the discharging process.

We are currently revising our design in order to increase readability and facilitate certain automated proofs. The use of *locales* in conjunction with dedicated record types to represent graphs and morphisms, as implemented by Noschinski [18], substantially decreases the amount of parameters used in a locale. This increases clarity since not all components of a graph (e.g. set of nodes/edges and source/target mappings) or the components of a morphism have to be stated explicitly. Additionally, we started to rely on total functions exclusively. This change removes the option type and thereby allows simpler proof obligations which in turn facilitate automated proofs.

Future developments will include the proof of classical DPO results such as the Church-Rosser theorem. We also plan the extension towards attributed DPO graph transformation. Our long-term goal is the development of a practical Isabelle-based proof assistant for the verification of individual programs in the graph programming language GP 2 [5].

References

[1] Jiří Adámek, Horst Herrlich & George Strecker (1990): *Abstract and Concrete Categories*. Wiley.

[2] Luciano Baresi, Vahid Rafe, Adel T. Rahmani & Paola Spoletini (2008): *An Efficient Solution for Model Checking Graph Transformation Systems*. Electronic Notes in Theoretical Computer Science 213(1), pp. 3–21, doi:10.1016/j.entcs.2008.04.071.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, Springer, doi:10.1007/978-3-662-07964-5.

[4] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2018): *Verifying graph transformations with guarded logics*. In: *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, IEEE, pp. 124–131, doi:10.1109/TASE.2018.00024.

[5] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. Science of Computer Programming 214, doi:10.1016/j.scico.2021.102727.

[6] Simone André da Costa Cavalheiro, Luciana Foss & Leila Ribeiro (2017): *Theorem proving graph grammars with attributes and negative application conditions*. Theoretical computer science 686, pp. 25–77, doi:10.1016/j.tcs.2017.04.010.

[7] Mark Dowson (1997): *The Ariane 5 software failure*. ACM SIGSOFT Software Engineering Notes 22(2), p. 84, doi:10.1145/251880.251992.

[8] Hartmut Ehrig (1979): *Introduction to the Algebraic Theory of Graph Grammars*. In: *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, *Lecture Notes in Computer Science* 73, Springer-Verlag, pp. 1–69, doi:10.1007/BFb0025714.

[9] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, doi:10.1007/3-540-31188-2. Available at `https://link.springer.com/book/10.1007%2F3-540-31188-2`.

[10] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. *International Journal on Software Tools for Technology Transfer* 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[11] Annegret Habel, Jürgen Müller & Detlef Plump (2001): *Double-Pushout Graph Transformation Revisited*. *Mathematical Structures in Computer Science* 11(5), pp. 637–688, doi:10.17/S0960129501003425.

[12] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Mathematical Structures in Computer Science* 19(2), pp. 245–296, doi:10.1017/S0960129508007202.

[13] J. Harrison (2003): *Formal verification at Intel*. In: *Proc. 18th Annual IEEE Symposium of Logic in Computer Science*, IEEE Computer Society, doi:10.1109/lics.2003.1210044.

[14] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato & Keisuke Nakano (2011): *Graph-Transformation Verification Using Monadic Second-Order Logic*. In: *Proc. of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11)*, ACM Press, p. 17–28, doi:10.1145/2003476.2003482.

[15] N.G. Leveson & C.S. Turner (1993): *An investigation of the Therac-25 accidents*. *Computer* 26(7), pp. 18–41, doi:10.1109/mc.1993.274940.

[16] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics — With Isabelle/HOL*. Springer, doi:10.1007/978-3-319-10542-0.

[17] Tobias Nipkow & Gerwin Klein (2014): *Concrete semantics: with Isabelle/HOL*. doi:10.1007/978-3-319-10542-0. Available at `http://concrete-semantics.org/`.

[18] Lars Noschinski (2015): *A graph library for Isabelle*. *Mathematics in Computer Science* 9(1), pp. 23–39.

[19] Lawrence C. Paulson, Tobias Nipkow & Makarius Wenzel (2019): *From LCF to Isabelle/HOL*. *Formal Aspects of Computing* 31(6), pp. 675–698, doi:10.1007/s00165-019-00492-1.

[20] Christopher M. Poskitt & Detlef Plump (2014): *Verifying Monadic Second-Order Properties of Graph Programs*. In: *Proc. International Conference on Graph Transformation (ICGT 2014)*, *Lecture Notes in Computer Science* 8571, Springer, pp. 33–48, doi:10.1007/978-3-319-09108-2_3.

[21] Arend Rensink (2008): *Explicit State Model Checking for Graph Grammars*. In: *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 5065, Springer, pp. 114–132, doi:10.1007/978-3-540-68679-8_8.

[22] Arend Rensink, Ákos Schmidt & Dániel Varró (2004): *Model checking graph transformations: A comparison of two approaches*. In: *Proc. International Conference on Graph Transformation (ICGT 2004)*, Springer, pp. 226–241, doi:10.1007/978-3-540-30203-2_17.

[23] Martin Strecker (2018): *Interactive and automated proofs for graph transformations*. *Mathematical Structures in Computer Science* 28(8), pp. 1333–1362, doi:10.1017/S096012951800021X.

[24] Jan Stückrath (2016): *Verification of Well-Structured Graph Transformation Systems*. Ph.D. thesis, Universität Duisburg-Essen.

[25] Dániel Varró (2004): *Automated formal verification of visual modeling languages by model checking*. *Software & Systems Modeling* 3(2), pp. 85–113, doi:10.1007/s10270-003-0050-x.

[26] Markus Wenzel (1999): *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*. In: *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Springer, pp. 167–183.

[27] Gia S. Wulandari & Detlef Plump (2021): *Verifying Graph Programs with Monadic Second-Order Logic*. In: *Proc. International Conference on Graph Transformation (ICGT 2021)*, *Lecture Notes in Computer Science* 12741, Springer, pp. 240–261, doi:10.1007/978-3-030-78946-6_13.

A Graph-Transformational Approach for Proving the Correctness of Reductions between NP-Problems

Hans-Jörg Kreowski, Sabine Kuske, Aaron Lye, Aljoscha Windhorst
University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
{kreo,kuske,lye,windhorst}@uni-bremen.de

The complexity class NP of decision problems that can be solved nondeterministically in polynomial time is of great theoretical and practical importance where the notion of polynomial-time reductions between NP-problems is a key concept for the study of NP. As many typical NP-problems are naturally described as graph problems, they and their reductions are obvious candidates to be investigated by graph-transformational means. In this paper, we propose such a graph-transformational approach for proving the correctness of reductions between NP-problems.

1 Introduction

The complexity class NP of decision problems that can be solved nondeterministically in polynomial time is of great theoretical and practical importance, with the notion of polynomial-time reductions between NP-problems being a key concept for the study of NP and, in particular, of NP-completeness. Since the early analysis of the class NP it is known that many graph-theoretic problems are NP-complete (cf. e.g. [1, 2]). Examples include finding a clique, a Hamiltonian cycle, a vertex cover, an independent set, a vertex coloring, etc. Moreover, many typical NP-problems are naturally described as graph problems, such as routing and scheduling problems as well as various further optimization and planning problems. Usually the complexity class NP as well as the notion of complexity-theoretic reductions are defined by means of (polynomial-time) Turing machines. However, explicit problems and constructions are described on some higher level with much more abstraction. Polynomial graph transformation units [3, 4] may be helpful not only for specifying and understanding decision problems and reductions, but also for obtaining correctness proofs in a systematic way. In [5], it has been shown that polynomial graph transformation units are a formal computational model for decision problems in NP. In [6], the problems of finding a clique, an independent set, a vertex cover and a Hamiltonian cycle are modeled as graph transformation units, and reductions are characterized by deadlock-free and confluent polynomial graph transformation units. Moreover, a proof principle is proposed for proving the correctness of reductions. It is based on a certain interaction of the unit of an NP problem and the unit of a reduction requiring a family of auxiliary variants of the reduction unit. In this paper, we continue this research by proposing a novel graph-transformation-based framework for proving the correctness of reductions between NP-problems. As correctness requires to construct certain derivations of one kind from certain derivations of another kind the initial graphs of which are connected by a third kind of derivations, we provide a toolbox of operations on derivations that allows such constructions. In particular, we employ operations based on parallel and sequential independence of rule applications.

The paper is organized as follows. In Section 2 and 3, we recall the notions of graph transformation and graph transformation units. In Section 4, we present the notion of graph-transformational modeling of NP-problems. In Section 5, we introduce the notion of graph-transformational modeling of reductions. Section 6 discusses the correctness proofs of such reductions. Section 7 concludes the paper.

2 Preliminaries

In this section, we recall the basic notions and notations of graphs and rule-based graph transformation as far as needed in this paper.

Graphs Let Σ be a set of labels with $* \in \Sigma$. A (directed edge-labeled) *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a finite set of *vertices*, E is a finite set of *edges*, $s,t\colon E \to V$ and $l\colon E \to \Sigma$ are mappings assigning a *source*, a *target* and a *label* to every edge $e \in E$. An edge e with $s(e) = t(e)$ is called a *loop*. An edge with label $*$ is called an *unlabeled edge*. In drawings, the label $*$ is omitted. *Undirected edges* are pairs of edges between the same vertices in opposite directions. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The empty graph is denoted by $\emptyset$. The class of all directed edge-labeled graphs is denoted by $\mathscr{G}_\Sigma$.

For graphs $G, H \in \mathscr{G}_\Sigma$, a *graph morphism* $g\colon G \to H$ is a pair of mappings $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that are structure-preserving, i.e., $g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_G(e) = l_H(g_E(e))$ for all $e \in E_G$. If the mappings g_V and g_E are bijective, then G and H are *isomorphic*, denoted by $G \cong H$. If they are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. For a graph morphism $g\colon G \to H$, the image of G in H is called a *match* of G in H, i.e., the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$.

Rules and rule application A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathscr{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively.

The application of r to a graph G consists of the following three steps. (1) Choose a match $g(L)$ of L in G. (2) Remove the vertices of $g_V(V_L) \setminus g_V(V_K)$ and the edges of $g_E(E_L) \setminus g_E(E_K)$ yielding Z, i.e., $Z = G - (g(L) - g(K))$. (3) Add R to Z by gluing Z with R in $g(K)$ (up to isomorphism) yielding H. The construction is subject to the *dangling condition* ensuring that Z becomes a subgraph of G so that H becomes a graph automatically. Moreover, we require the *identification condition*, i.e., if different items of L are mapped to the same item in $g(L)$, then they are items of K. Sometimes any identification may be forbidden. Then we add the postfix *(inj)* to the rule. The construction produces a right match $h(R)$ that extends $g(K)$ by the identity on $R - K$ (up to isomorphism).

The application of r to G w.r.t. g is called *direct derivation* and is denoted by $G \underset{r}{\Longrightarrow} H$ (where g is kept implicit). A *derivation* from G to H is a sequence of direct derivations $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \cdots \underset{r_n}{\Longrightarrow} G_n$ with $G_0 = G$, $G_n \cong H$ and $n \geq 0$. $r_1 \cdots r_n$ is called *application sequence*. If $r_1, \cdots, r_n \in P$, then the derivation is also denoted by $G \overset{n}{\underset{P}{\Longrightarrow}} H$. If the length of the derivation does not matter, we write $G \overset{*}{\underset{P}{\Longrightarrow}} H$.

A rule $r = (L \supseteq K \subseteq R)$ may be equipped with a *negative application condition* (NAC) N with $L \subseteq N$. It prevents the application of r to a graph G if the match $g(L)$ can be extended to a match $\overline{g}(N)$.

Let $|S|$ denote the cardinality of a finite set S and $size(G) = |V_G| + |E_G|$ the size of a graph G. It is worth noting that the application of a given (fixed) rule to G can be performed in polynomial time provided that the equality of labels can be checked in polynomial time. This is due to the fact that the number of graph morphisms from L to G is bounded by $size(G)^{size(L)}$ so that a match can be found in polynomial time. The further steps of the rule application can be done in linear time.

It may be noted that the chosen notion of rule application fits into the DPO framework as introduced in [7] (see, e.g., [8] for a comprehensive survey). For details on NACs, we refer to [9].

Extension and independence Let $d = (G \overset{*}{\underset{P}{\Longrightarrow}} H)$ and G be extended to $\hat{G}$ by adding some vertices and some edges with new or old vertices as sources and targets. Then it is well-known that d can be extended to $\hat{G} \overset{*}{\underset{P}{\Longrightarrow}} \hat{H}$ if none of the direct derivations of d removes a source or target of a new edge and if, in the case of a negative application condition, the added vertices and edges do not contradict it.

Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ for $i = 1,2$ be rules. Two direct derivations $G \underset{r_i}{\Longrightarrow} H_i$ with matches $g_i(L_i)$ are *parallel independent* if $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$. Successive direct derivations $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$ with the right match $h_1(R_1)$ and the (left) match $g_2'(L_2)$ are *sequentially independent* if $h_1(R_1) \cap g_2'(L_2) \subseteq h_1(K_1) \cap g_2'(K_2)$. It is well-known that parallel independence induces the direct derivations $H_1 \underset{r_2}{\Longrightarrow} X$ and $H_2 \underset{r_1}{\Longrightarrow} X$ with matches $g_2'(L_2) \cong g_2(L_2)$ and $g_1'(L_1) \cong g_1(L_1)$ respectively and that sequential independence induces a derivation $G \underset{r_2}{\Longrightarrow} H_2 \underset{r_1}{\Longrightarrow} X$ with matches $g_2(L_2) \cong g_2'(L_2)$ and $g_1'(L_1) \cong g_1(L_1)$ (cf., e.g., [8, 10, 11]).

The two constructions can be extended to derivations by simple inductions. Sets of rules P_1 and P_2 are called *independent* if each two applications of a rule of P_1 and a rule of P_2 applied to the same graph are parallel independent and, applied one after the other, sequentially independent. In the special case of $P = P_1 = P_2$, we say that P is independent. Then the following hold: (1) $G \overset{n}{\underset{P_1}{\Longrightarrow}} H_1$ and $G \overset{m}{\underset{P_2}{\Longrightarrow}} H_2$ induce $H_1 \overset{m}{\underset{P_2}{\Longrightarrow}} X$ and $H_2 \overset{n}{\underset{P_1}{\Longrightarrow}} X$ for some X, and (2) $G \overset{n}{\underset{P_1}{\Longrightarrow}} H_1 \overset{m}{\underset{P_2}{\Longrightarrow}} X$ induces $G \overset{m}{\underset{P_2}{\Longrightarrow}} H_2 \overset{n}{\underset{P_1}{\Longrightarrow}} X$ for some H_2. The derivations are obtained by repeating the respective constructions for direct derivations of the given derivations as long as possible. One way to look at the situation is that the P_2-derivation is moved either forward or backward along the P_1-derivation. Therefore, we refer to the resulting P_2-derivation as *moved variant* of the given P_2-derivation.

3 Graph Transformation Units

In this section, the basic notions and notations of graph transformation units (see, e.g., [3, 12]) are recalled using graphs, rules and rule application as introduced in the previous section. Besides a set of rules, a graph transformation unit provides two graph class expressions to specify initial and terminal graphs and a control condition to regulate the derivation process.

We restrict the consideration to graph class expressions e that specify graph classes $SEM(e) \subseteq \mathscr{G}_\Sigma$ such that membership can be decided in polynomial time. In examples, we use (1) $forbidden(\mathscr{F})$ for some finite $\mathscr{F} \subseteq \mathscr{G}_\Sigma$ with $SEM(forbidden(\mathscr{F})) = \{G \mid \text{there is no match } f\colon F \to G \text{ for any } F \in \mathscr{F}\}$ and (2) some constant terms like *undirected*, *unlabeled*, *simple*, and *loop-free* referring to the class of undirected graphs, unlabeled graphs, simple graphs without parallel edges, and loop-free graphs respectively. The term *standard* specifies the intersection of these four graph classes. Moreover, we use the term $bound(\mathbb{N})$ that specifies the set of all graphs with a single vertex, a single *bound*-loop and an arbitrary number of unlabeled loops. Another useful graph class expression is $reduced(P)$ for some finite set of rules P that specifies the class of graphs to which none of the rules of P can be applied. Finally, we use the binary operator $+$ for graph class expressions specifying the disjoint union of the graphs of the corresponding graph classes. The membership problems of all explicitly used graph class expressions can be checked in polynomial time. *forbidden* and *reduced* are variants of the matching problem. The properties of the *standard* expressions follow from an inspection of the edges which applies also to $bound(\mathbb{N})$. The disjoint union symbol is only used in $standard + bound(\mathbb{N})$. Given a graph, one can identify the component $bound(k)$ for some k by inspecting all edges. The rest of the graph must be checked with

respect to *standard*.

As control conditions, we use extended regular expressions over sets of rules. Let C be a regular expression over some set of rules P and $d = (G \overset{*}{\underset{P}{\Longrightarrow}} H)$. Then d *satisfies* C if the application sequence of d is in $L(C)$ where $L(C)$ is the regular language specified by C. This is denoted by $G \overset{*}{\underset{P,C}{\Longrightarrow}} H$. Employing a finite automaton corresponding to the regular expression, one can control such derivations stepwise, meaning that the allowed and applicable rules can be determined. In this way, satisfaction can be checked in polynomial time for derivations of polynomial lengths. In an *extended regular expression*, the Kleene star * behind a rule may be replaced by !, where $r!$ requires that r be applied as long as possible (in contrast to r^* that allows to apply r arbitrarily often). As the stepwise control provides all currently applicable rules, the applicability of r is included so that the control of $r!$ does not require any extra time. In examples, the rules are listed so that their numbers in this order can be used to refer to them in control conditions.

A *graph transformation unit* is a system $gtu = (I,P,C,T)$ where I is an *initial* graph class expression, P is a finite set of rules, C is an extended regular expression over P and T is a *terminal* graph class expression. The components I, P, C and T of gtu are also denoted by I_{gtu}, P_{gtu}, C_{gtu} and T_{gtu}, respectively. The *semantics* of gtu is the binary relation $SEM(gtu) = (SEM(I) \times SEM(T)) \cap \overset{*}{\underset{P,C}{\Longrightarrow}}$. A derivation $G \overset{*}{\underset{P,C}{\Longrightarrow}} H$ with $G \in SEM(I)$ and $H \in SEM(T)$ is called *successful*, denoted by $G \overset{*}{\underset{gtu}{\Longrightarrow}} H$. As long as a derivation $G \overset{*}{\underset{P}{\Longrightarrow}} H$ with $G \in SEM(I)$ follows the stepwise control of C, it is called permitted and denoted by $G \overset{*}{\underset{P,C?}{\Longrightarrow}} H$.

In examples, a graph transformation unit is presented schematically where the components I, P, C, and T are listed after respective keywords initial, rules, cond, and terminal.

A graph transformation unit is *polynomial* if there is a polynomial p for each permitted derivation $G \overset{n}{\underset{P,C?}{\Longrightarrow}} H$ such that $n \leq p(size(G))$. A polynomial graph transformation unit gtu is *functional* if (1) every permitted derivation can be prolonged into a successful one and (2) for every initial graph G every successful derivation from G yields the same terminal graph (up to isomorphism). In this case, the resulting graph is denoted by $gtu(G)$.

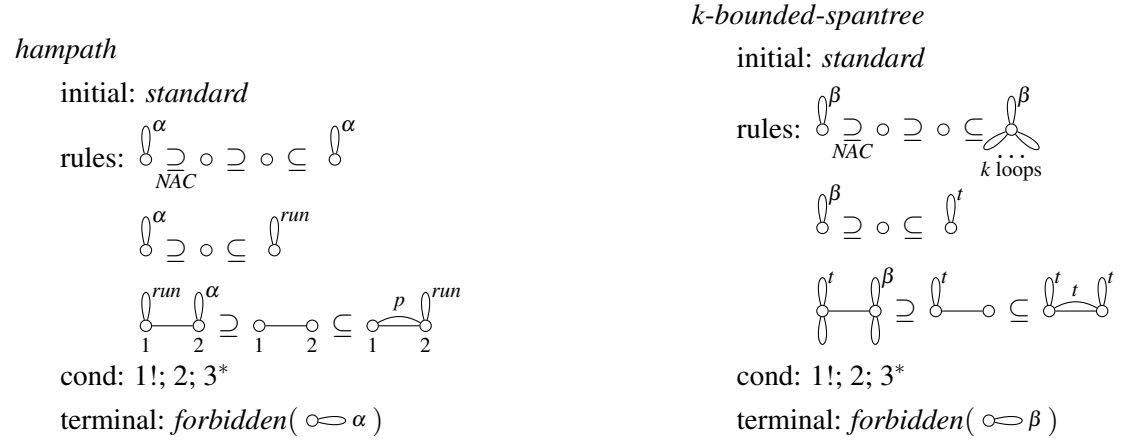
Note that we restrict ourselves to graph class expressions with polynomial membership problems for specifying initial and terminal graphs and to extended regular expressions as control conditions that can be checked stepwise in polynomial time. Therefore, our definition of polynomial graph transformation units is a special case of the one given in [6] where a more general kind of control conditions is allowed.

4 Graph-Transformational Modeling of NP-Problems

Based on its input-output semantics, a graph transformation unit can also be interpreted as a model of a decision problem. This problem belongs to the class NP if the length of each derivation of the unit is polynomially bounded and the proper nondeterminism is provided by the general functioning of units. This means that the usual technique to prove polynomial termination can be applied to show that the decision problem of a unit is in NP. The four examples are further considered in Sections 5 and 6.

Definition 1. *Let $gtu = (I,P,C,T)$ be a graph transformation unit. Then the decision problem of gtu, denoted by $DEC(gtu)$: $SEM(I) \to BOOL$, yields TRUE for $G \in SEM(I)$ if $(G,H) \in SEM(gtu)$ for some $H \in SEM(T)$ and FALSE otherwise.*

Observation 1. Let gtu be a graph transformation unit. Then $DEC(gtu) \in$ NP if gtu is polynomial.



(a) Hamiltonian path problem (b) k-bounded spanning tree problem

Figure 1: Graph transformation units *hampath* and *k-bounded-spantree*

Proof. By definition, the membership problems of the classes of initial and terminal graphs, and the stepwise control of rule application are polynomial and the lengths of permitted derivations are polynomially bounded. Moreover, the rule application needs polynomial time and its nondeterminism is polynomially bounded so that $DEC(gtu) \in \text{NP}$. □

Example 1. The Hamiltonian path problem asks whether a graph contains a simple path that visits all vertices. The problem is modeled by the unit in Figure 1a. The length of each of its derivations is obviously bounded by twice the number of vertices of the initial graph such that $DEC(hampath) \in \text{NP}$.

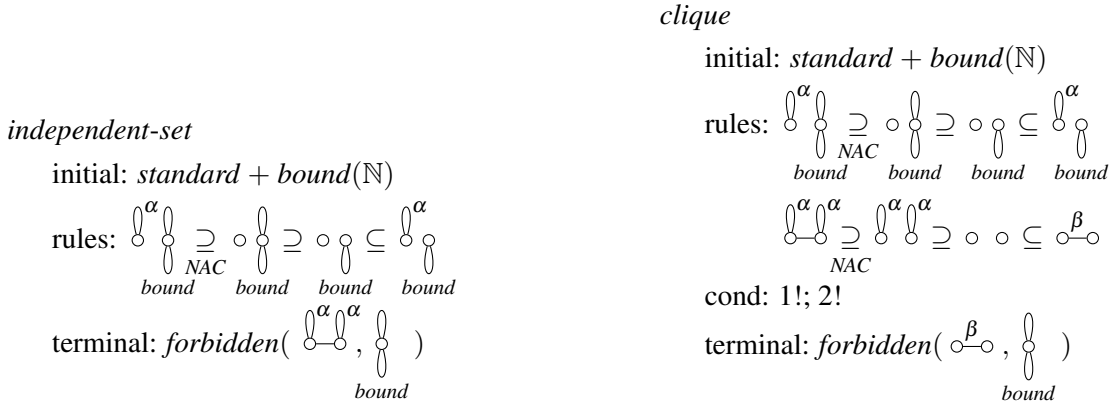
Example 2. The k-bounded spanning tree problem asks whether, for a given bound $k \in \mathbb{N}$, a graph contains a spanning tree with a vertex degree not greater than k. The problem is modeled by the unit in Figure 1b. The length of each of its derivations is obviously bounded by twice the number of vertices of the initial graph such that $DEC(k\text{-}bounded\text{-}spantree) \in \text{NP}$.

An often easy way to show the polynomiality of a unit *gtu* and as a consequence that $DEC(gtu) \in \text{NP}$ is by finding a natural variable the value of which is polynomially bounded and decreases whenever a rule of *gtu* is applied. This is a well-known fact.

Fact 1 (Polynomial Termination)**.** Let $gtu = (I,P,C,T)$ be a graph transformation unit, $f\colon \mathscr{G}_\Sigma \to \mathbb{N}$ be a function and p a polynomial such that $f(x) \leq (p \circ size)(x)$ for each $x \in \mathscr{G}_\Sigma$ and $f(G) > f(H)$ for each direct derivation $G \underset{P}{\Longrightarrow} H$. Then the length k of every successful derivation $G \overset{k}{\underset{P}{\Longrightarrow}} H$ is polynomially bounded in the size of the input graph G such that $DEC(gtu) \in \text{NP}$.

Example 3. The polynomial termination of *hampath* may be shown using Fact 1 by separating the unit into two units. The first one with rule 1 as its only rule terminates in a linear number of steps as the number of vertices without α-loop decreases if a rule is applied. The second unit with rules 2 and 3 terminates in a linear number of steps as the number of vertices with α-loops decreases if a rule is applied. The polynomial termination of *k-bounded-spantree* can be shown analogously.

Example 4. The independent set problem asks whether a graph contains a set of k vertices, no two of which are connected by an edge. The problem is modeled by the unit in Figure 2a. As the number of loops decreases whenever the rule is applied, one gets $DEC(independent\text{-}set) \in \text{NP}$.



(a) Independent set problem

(b) Clique problem

Figure 2: Graph transformation units *independent-set* and *clique*

Example 5. The clique problem asks whether a graph contains a set of k vertices every two of which are connected by an edge, constituting a *complete* subgraph. The problem is modeled by the unit in Figure 2b. As the number of unlabeled loops decreases whenever the first rule is applied and the number of α-loops decreases whenever the second rule is applied, one gets $DEC(clique) \in$ NP.

5 Graph-Transformational Modeling of Reductions

Reductions between NP-problems are a key concept for the study of the class NP. Given two NP-problems DEC and DEC', a reduction RED is a function from the inputs of DEC to the inputs of DEC' that can be computed in polynomial time and is subject to the following correctness condition for each input x of DEC: $DEC(x) = TRUE$ if and only if $DEC'(RED(x)) = TRUE$. If the NP-problems are given by polynomial graph transformation units, then a reduction can be modeled as a functional and polynomial graph transformation unit.

Definition 2 (Reduction)**.** *Let gtu and gtu' be polynomial graph transformation units. Let $red = (I_{gtu}, P_{red}, C_{red}, I_{gtu'})$ be a functional and polynomial graph transformation unit. Then red is a* reduction *of gtu to gtu' if the following holds for all $G \in SEM(I_{gtu})$: $(G,H) \in SEM(gtu)$ for some $H \in SEM(T_{gtu})$ if and only if $(red(G),H') \in SEM(gtu')$ for some $H' \in SEM(T_{gtu'})$.*

Observation 2. Let red be a reduction of gtu to gtu'. Then the function $RED = SEM(red) : SEM(I_{gtu}) \to SEM(I_{gtu'})$, defined by $RED(G) = red(G)$, is a reduction of $DEC(gtu)$ to $DEC(gtu')$, denoted by $DEC(gtu) \leq DEC(gtu')$.

Proof. As red is functional and polynomial, RED is a function that is computed in polynomial time. Its correctness can be shown as follows: Let $G \in SEM(I_{gtu})$ with $DEC(gtu)(G) = TRUE$ meaning that there is an $H \in SEM(T_{gtu})$ with $(G,H) \in SEM(gtu)$. As red is a reduction of gtu to gtu', this is equivalent to $(red(G),H') \in SEM(gtu')$ for some $H' \in SEM(T_{gtu'})$, i.e., $DEC(gtu')(red(G)) = TRUE$. As $red(G) = RED(G)$, RED turns out to be a reduction of $DEC(gtu)$ to $DEC(gtu')$. □

The observation means that $DEC(gtu) \leq DEC(gtu')$ can be established by means of a reduction of gtu to gtu'. The polynomiality of such a reduction can be shown by using Fact 1. The functionality can be shown using the following well-known fact.

hampath-to-2-bounded-spantree

initial: $I_{hampath}$

rules:

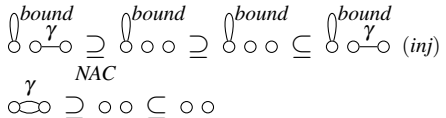
cond: ε

terminal: $I_{2\text{-}bounded\text{-}spantree}$

(a) Hamiltonian path to 2-bounded spantree

clique-to-independent-set

initial: I_{clique}

rules: 

cond: 1!; 2!; 3!

terminal: $I_{independent\text{-}set}$

(b) Clique to independent set

Figure 3: Graph transformation units for reductions

Fact 2. Let $gtu = (I,P,C,T)$ be a polynomial graph transformation unit. Then *gtu* is functional if the following hold:

1. P is independent,
2. $(G,H) \in SEM(gtu)$ implies $H \in reduced(P)$, and
3. $G \overset{*}{\underset{P,C}{\Longrightarrow}} H$ for $G \in SEM(I)$ and $H \in reduced(P)$ implies $(G,H) \in SEM(gtu)$.

Proof. As *gtu* is polynomial, the lengths of its derivations are finitely bounded. The required independence implies the confluence of P-derivations. Both together means that for each $G \in SEM(I)$, there is – up to isomorphism – exactly one $H \in reduced(P)$ with $G \overset{*}{\underset{P}{\Longrightarrow}} H$. Properties 2 and 3 make sure that this functionality holds for $SEM(gtu)$, too. □

Example 6. According to the considerations in Examples 1 and 2, we have $DEC(hampath)(G) = TRUE$ if and only if G has a Hamiltonian path, and we have $DEC(k\text{-}bounded\text{-}spantree)(G) = TRUE$ if and only if G has a spanning tree with a vertex degree not greater than k. A Hamiltonian path is connected and cycle-free and visits all vertices such that it is a spanning tree with vertex degree 2. Conversely, a spanning tree with vertex degree 2 or smaller must be a simple path and, as it covers all vertices, therefore, a Hamiltonian path. In other words, the identity on the initial graphs of *hampath* and *2-bounded-spantree* is a reduction of *hampath* to *2-bounded-spantree*, as specified in the unit in Figure 3a.

As there are no rules to be applied and as the regular expression ε specifies the language with the empty string as its sole element allowing derivations of length 0 only, each initial graph derives itself exclusively, which is terminal at the same time because the initial graphs of *hampath* and *2-bounded-spantree* coincide. The unit is obviously polynomial and functional.

Example 7. The reduction from *clique* to *independent-set* is modeled by the unit in Figure 3b. Each initial graph has the form $G + bound(k)$ for some $k \in \mathbb{N}$. The first rule can be applied to two vertices of G that are not connected by a γ-edge and neither of them has a *bound*-loop, connecting them by such an edge. Hence the rule can be applied $|V_G| \cdot (|V_G| - 1)/2$ times and must be applied as often due to the control condition. Then, any two vertices of G are connected by γ-edges. The second rule can be applied to an unlabeled edge of G with a parallel γ-edge, removing both. Due to the control condition, this must be done for all such pairs of edges in G. The number of steps is $|E_G|$, and γ-edges remain between all pairs of vertices of G that are not connected in G. By applying the third rule as long as possible, all γ-edges are replaced by (unlabeled) edges such that the resulting graph is isomorphic to the dual one of

the initial graph. In particular, the unit turns out to be functional and polynomial. The overall lengths of derivations is $|V_G| \cdot (|V_G| - 1)$ for $G \in SEM(I)$. The rule i for $i = 1,2,3$ is independent of itself, so that its application as long as possible is confluent in every case. And as all intermediate graphs of derivations contain γ-edges, the terminal graphs are only reached at the end. Finally, the unit has the correctness property of a reduction. If two vertices in a graph G are connected, then they are not connected in the dual graph and the other way round. Consequently, a clique in G becomes an independent set in the dual graph and the other way round.

The correctness proofs in Example 6 can be found in Karp [1] and in Garey and Johnson [2]. This traditional graph theoretic way of proving the correctness of reductions can always be employed for proving the correctness of a graph-transformational reduction if the reduction as well as the two NP-problems modeled by the two units are characterized in graph-theoretic terms. In the next section, we investigate an alternative graph-transformational way of proving the correctness of reductions.

6 Proving the Correctness of Reductions

In this section, we investigate possibilities to prove the correctness of reductions by graph-transformational means.

Let *gtu* and *gtu*$'$ be two polynomial graph transformation units and $red = (I_{gtu}, P_{red}, C_{red}, I_{gtu'})$ be a polynomial and functional graph transformation unit. Then *red* is a reduction of $DEC(gtu)$ to $DEC(gtu')$ if the following correctness conditions hold:

Forward If $G \overset{*}{\underset{gtu}{\Longrightarrow}} H$ is a successful derivation of *gtu*, then there is a successful derivation $red(G) \overset{*}{\underset{gtu'}{\Longrightarrow}} H'$ for some $H' \in SEM(T_{gtu'})$.

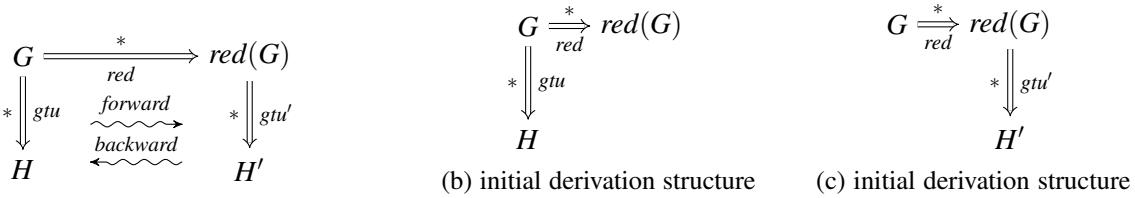
Backward If $red(G) \overset{*}{\underset{gtu'}{\Longrightarrow}} H'$ is a successful derivation of *gtu*$'$, then there is a successful derivation $G \overset{*}{\underset{gtu}{\Longrightarrow}} H$ for some $H \in SEM(T_{gtu})$.

The diagram in Figure 4a illustrates the task. Given a *gtu*- and a *red*-derivation, one must find an appropriate *gtu*$'$-derivation, and conversely given a *gtu*$'$-derivation and the same *red*-derivation, one must find an appropriate *gtu*-derivation. To achieve this, we propose a kind of toolbox that provides operations on derivation structures where the latter are defined in Definition 3 and the operations on them in Definition 4.

A derivation structure is a finite directed graph, the vertices of which are graphs and the edges are direct derivations.

Definition 3. *A* derivation structure *is a finite unlabeled directed graph DS, such that* $V_{DS} \subseteq \mathscr{G}_\Sigma$ *and every edge* $e \in E_{DS}$ *is a direct derivation* $e = (G \underset{r}{\Longrightarrow} H)$ *with* $s_{DS}(e) = G$ *and* $t_{DS}(e) = H$. *The class of all derivation structures is denoted by* $\mathscr{DS}$.

The operations add derivations to given derivation structures so that the results are derivation structures. The idea is (1) to start with the derivation structure in Figure 4b and to apply operations until the derivation $red(G) \overset{*}{\underset{gtu'}{\Longrightarrow}} H'$ appears as a substructure and (2) to start conversely with the derivation structure in Figure 4c and apply operations until the derivation $G \overset{*}{\underset{gtu}{\Longrightarrow}} H$ appears as a substructure.



(a) Proof schema

(b) initial derivation structure (forward)

(c) initial derivation structure (backward)

Figure 4: Correctness proof schema and initial derivation structures

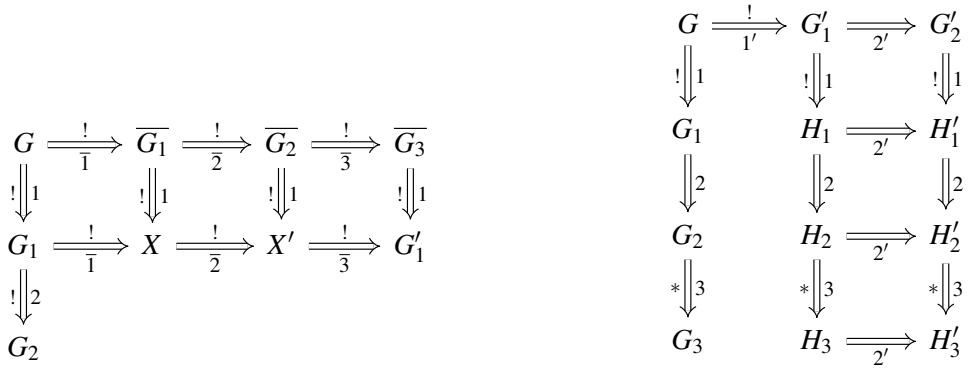
We provide five operations which are defined as binary relations on derivation structures. (1) *conflux* of two parallel independent rule applications adding two corresponding rule applications (cf. Section 2) to the derivation structure; (2) *interchange* of two sequential independent rule applications adding one corresponding derivation consisting of the interchanged rule applications (cf. Section 2) to the derivation structure; (3) *sprout* with a functional graph transformation unit as parameter attaching a successful derivation of the parameter starting in a graph of the given derivation structure; (4) *couple* with a span of parallel independent rule applications as parameter adding a corresponding extended right part of the span to the derivation structure if an extension of the left part of the span is present; and (5) *associate* with a pair of particular derivations with the same first graph and the same last graph as parameter adding an extension of the second derivation to the derivation structure if a corresponding extension of the first one is present. All these enlargements of the derivation structures are formally defined by unions of derivation structures. Concerning *conflux* and *interchange*, the rule applications corresponding to the given independent ones are unique up to isomorphism. One of the choices is taken. Concerning *sprout*, there are many derivations $G \stackrel{*}{\Longrightarrow} funct(G)$ with a result unique up to isomorphism. Again one of them is chosen. Concerning *couple* and *associate*, the extensions are unique up to isomorphism provided they exist at all. Then one of the extensions is added. The extensions may not exist because the second component of the span or the pair may remove vertices that are needed for the extensions. In such a case, the respective operation is undefined.

Definition 4. 1. *conflux: Let $DS \in \mathscr{DS}$ with two parallel independent direct derivations $G \underset{r_i}{\Longrightarrow} H_i$ for $i = 1,2$ as substructure. Let DS' be the enlargement of DS by two corresponding direct derivations $d_1 = (H_1 \underset{r_2}{\Longrightarrow} X)$ and $d_2 = (H_2 \underset{r_1}{\Longrightarrow} X)$ provided by the parallel independence, i.e., $DS' = DS \cup d_1 \cup d_2$. Then $(DS, DS') \in$ conflux.*

2. *interchange: Let $DS \in \mathscr{DS}$ with two sequential independent direct derivations $G \underset{r_1}{\Longrightarrow} H_1 \underset{r_2}{\Longrightarrow} X$ as substructure. Let DS' be the enlargement of DS by one corresponding derivation $d = (G \underset{r_2}{\Longrightarrow} H_2 \underset{r_1}{\Longrightarrow} X)$ provided by the parallel independence, i.e., $DS' = DS \cup d$. Then $(DS, DS') \in$ interchange.*

3. *sprout(funct): Let funct be a functional graph transformation unit, $DS \in \mathscr{DS}$ and $G \in V_{DS}$. Let DS' be the enlargement of DS by attaching one of the derivations $d = (G \stackrel{*}{\Longrightarrow} funct(G))$ at G, i.e., $DS' = DS \cup d$. Then $(DS, DS') \in sprout(funct)$.*

4. *couple(span): Let span be a pair of parallel independent direct derivations $\hat{d}_i = (\hat{G} \underset{r_i}{\Longrightarrow} \hat{H}_i)$ for $i = 1,2$, $DS \in \mathscr{DS}$, and $G \stackrel{*}{\Longrightarrow} H_1$ be a substructure of DS extending $\hat{d}_1$. Let DS' be the enlargement of DS by adding one corresponding extension $d_2 = (G \underset{r_2}{\Longrightarrow} H_2)$ of $\hat{d}_2$, i.e., $DS' = DS \cup d_2$, provided it exists. Then $(DS, DS') \in couple(span)$.*



(a) Derivation structure in Example 8

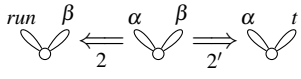
(b) Derivation structure in Example 9

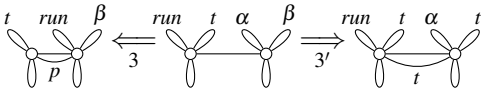
Figure 5: Derivation structures in Example 8 and 9

5. *associate*(*pair*): *Let pair be a pair of derivations* $\hat{d}_i = (\hat{G} \overset{*}{\Longrightarrow} \hat{H})$ *for* $i = 1,2$, $DS \in \mathscr{DS}$, $d_1 = (G \overset{*}{\Longrightarrow} H)$ *be a substructure of DS extending* $\hat{d}_1$. *Let* DS' *be the enlargement of DS by adding one extension* d_2 *of* $\hat{d}_2$ *from G to H, i.e.,* $DS' = DS \cup d_2$, *provided it exists. Then* $(DS, DS') \in$ *associate*(*pair*).

In the following examples, we demonstrate how the operations may be employed to prove the correctness of reductions.

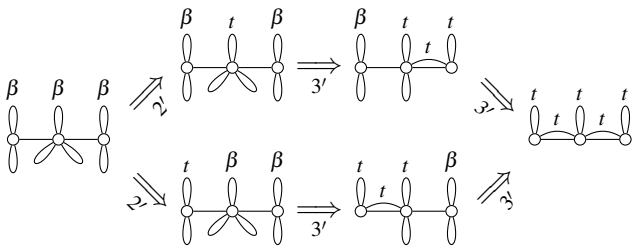
Example 8. Using the operations *conflux*, *interchange* and *sprout*, the correctness of *clique-to-independent-set* can be shown as follows. Let $G \overset{!}{\underset{1}{\Longrightarrow}} G_1 \overset{!}{\underset{2}{\Longrightarrow}} G_2$ be a successful derivation in *clique*. As G_2 is terminal, it has no β-edges so that $G_1 \overset{!}{\underset{2}{\Longrightarrow}} G_2$ has length 0 and can be omitted. Let $G \overset{!}{\underset{\bar{1}}{\Longrightarrow}} \overline{G_1} \overset{!}{\underset{\bar{2}}{\Longrightarrow}} \overline{G_2} \overset{!}{\underset{\bar{3}}{\Longrightarrow}} \overline{G_3}$ be a successful derivation of *clique-to-independent-set* reducing G where the numbers of rules are overlined to distinguish them from the rules 1 and 2 of *clique*. Each rule application of rule 1 is parallel independent of each rule application of rule $\bar{i}$ for $i = 1,2,3$. Therefore, the application of *conflux* as long as possible to the rule applications of the two given derivations yields - among others - a derivation $\overline{G_3} \overset{!}{\underset{1}{\Longrightarrow}} G_1'$ being a moved variant of $G \overset{!}{\underset{1}{\Longrightarrow}} G_1$ as defined at the end of Section 2. Figure 5a illustrates the derivation structure. The involved rule applications keep the set of vertices invariant. Moreover, the independence of all rule applications makes sure that the sets of vertices with α-loops in G_1 and G_1' are equal. Let v and v' be two such vertices which are connected by an edge e in G according to the choice of $G \overset{!}{\underset{1}{\Longrightarrow}} G_1$. Consequently, e gets a parallel γ-edge in $\overline{G_1}$, both are removed between $\overline{G_1}$ and $\overline{G_2}$, and v and v' are not connected in $\overline{G_3}$. This means that G_1' is a terminal graph of *independent-set*. Altogether, the reduction is forward correct. The backwards correctness follows analogously, starting from the derivation reducing G and a successful derivation $\overline{G_3} \overset{!}{\underset{1}{\Longrightarrow}} G_1'$ in *independent-set* and applying *interchange* (instead of *conflux*) to the sequential independent rule applications of rule $\bar{i}$ for $i = 1,2,3$, followed by one application of rule 1. This yields a derivation $G \overset{!}{\underset{1}{\Longrightarrow}} G_1$ as a moved variant of $\overline{G_3} \overset{!}{\underset{1}{\Longrightarrow}} G_1'$. As each two vertices with α-loops in G_1' are not connected, the definition of the reduction implies that they are connected in G_1. In particular, rule 2 of *clique* cannot be applied and the derivation $G_1 \overset{!}{\underset{2}{\Longrightarrow}} G_2$ has length 0. Adding it with *sprout* at G_1, one gets a successful derivation in *clique*. This completes the proof.

Example 9. In proving the correctness of *hampath-to-2-bounded-spantree*, all five operations are used. Let $G \overset{!}{\underset{1}{\Longrightarrow}} G_1 \underset{2}{\Longrightarrow} G_2 \overset{*}{\underset{3}{\Longrightarrow}} G_3$ be a successful derivation of *hampath*. This is a start structure as the reduction is the identity and one can begin to construct the corresponding derivation immediately. Its first section is $G \overset{!}{\underset{1'}{\Longrightarrow}} G'_1$ necessarily which can be added with *sprout* at G using the functional unit with the rule $1'$ and $1'!$ as control condition. The rules of *2-bounded-spantree* are primed to distinguish them from the rules of *hampath*. The rule sets $\{1,2,3\}$ and $\{1',2',3'\}$ are independent. Therefore, *conflux* applied as long as possible yields the subderivation $G \overset{!}{\underset{1'}{\Longrightarrow}} G'_1 \overset{!}{\underset{1}{\Longrightarrow}} H_1 \underset{2}{\Longrightarrow} H_2 \overset{*}{\underset{3}{\Longrightarrow}} H_3$ where the last three sections are a moved variant of the given derivation so that H_1 contains G and an α-loop, a β-loop and two $*$-loops at each vertex and H_3 contains G_3 and a β-loop and two $*$-loops at each vertex. In particular, $H_1 \underset{2}{\Longrightarrow} H_2$ is an extension of the left part of the span 

. Therefore, *couple* can be applied adding the direct derivation $H_1 \underset{2'}{\Longrightarrow} H'_1$. Using the known independence, the latter can be interchanged with the rule applications of $G'_1 \overset{!}{\underset{1}{\Longrightarrow}} H_1$ yielding, in particular, $G'_1 \underset{2'}{\Longrightarrow} G'_2$ and $G'_2 \overset{!}{\underset{1}{\Longrightarrow}} H'_1$. Moreover, *conflux* can be applied to $H_1 \underset{2'}{\Longrightarrow} H'_1$ and the rule applications of $H_1 \underset{2}{\Longrightarrow} H_2 \overset{*}{\underset{3}{\Longrightarrow}} H_3$ yielding the moved variant $H'_1 \underset{2}{\Longrightarrow} H'_2 \overset{*}{\underset{3}{\Longrightarrow}} H'_3$. Figure 5b shows the main parts of the derivation structure that is constructed so far. If $H'_2 \overset{*}{\underset{3}{\Longrightarrow}} H'_3$ has length 0, then we are done. The uppermost horizontal derivation is the result we are looking for. Otherwise, one must repeat the construction done for $H_1 \underset{2}{\Longrightarrow} H_2$, i.e. couple, interchange, and then conflux as long as possible, for each of the applications of rule 3 in $H'_2 \overset{*}{\underset{3}{\Longrightarrow}} H'_3$ one after the other using the span 

. In the end, this yields a derivation $G'_2 \overset{*}{\underset{3'}{\Longrightarrow}} G'_3$ with the same length as $G_2 \overset{*}{\underset{3}{\Longrightarrow}} G_3$, i.e., $n-1$ if G has n vertices. Summarizing, the resulting derivation $G \overset{!}{\underset{1'}{\Longrightarrow}} G'_1 \underset{2'}{\Longrightarrow} G'_2 \overset{*}{\underset{3'}{\Longrightarrow}} G'_3$ is successful as the derivation picks $n-1$ edges which always is a spanning tree, therefore, the reduction turns out to be forward correct.

Conversely, let $G \overset{!}{\underset{1'}{\Longrightarrow}} G'_1 \underset{2'}{\Longrightarrow} G'_2 \overset{*}{\underset{3'}{\Longrightarrow}} G'_3$ be a successful derivation of *2-bounded-spantree*. Let v_0 be the vertex matched by rule $2'$ and $e_1,\ldots,e_{n-1}$ be the edges matched by the following applications of rule $3'$ in this order, where e_1 is attached to v_0. If none of the further edges is attached to v_0, then the arguments proving forward correctness can be converted so that one gets backwards correctness, too. Otherwise, let e_i with $i \geq 2$ be attached to v_0. Without loss of generality, one can assume $i = 2$ because this case can always be obtained by interchanges. Then the derivation $G'_1 \underset{2'}{\Longrightarrow} G'_2 \underset{3'}{\Longrightarrow} \overline{G} \underset{3'}{\Longrightarrow} \overline{\overline{G}}$ is an extension of the upper derivation in the pair



so that *associate* can be applied. This means that each successful derivation can be rearranged by repeated interchanges and associations in such a way that the conversion of the forward correctness proof works.

There are several further reductions the correctness of which can be shown in a similar line of consideration, but their documentation here is beyond the space limit of the paper. Examples are *independent-set* $\leq$ *clique*, *k-bounded-spantree* $\leq$ *l-bounded-spantree* for $2 \leq k < l$ and *hampath* $\leq$ *hamcycle* $\leq$ *TSP* where *TSP* is the famous traveling salesperson problem. The latter example works very similar to *clique* $\leq$ *independent-set* with the exception that the successful derivations of *hamcycle* are not moved to the end of the reduction derivation, but only to some intermediate graph where the corresponding *TSP*-derivation is constructed, which is further moved along the reduction. The examples indicate that the following proof procedure is quite promising to result in the correctness of reductions between NP-problems.

Proof procedure. Let *gtu* and *gtu'* be two polynomial graph transformation units and $red = (I_{gtu}, \overline{P_1} \cup \overline{P_2}, \overline{C}, I_{gtu'})$ with $\overline{P_1} \cap \overline{P_2} = \emptyset$ be a functional graph transformation unit such that each of its successful derivations has the form $G \overset{*}{\underset{\overline{P_1}}{\Longrightarrow}} \overline{G} \overset{*}{\underset{\overline{P_2}}{\Longrightarrow}} red(G)$. Let P_{gtu} and $\overline{P_1}$, $\overline{P_2}$ and $P_{gtu'}$, as well as P_{gtu} and $P_{gtu'}$ be disjoint and independent. Then the correctness of *red* may be proved as follows:

Forward Consider the start structure of Figure 4b.

(f1) Apply *conflux* as long as possible using the independence of P_{gtu} and $\overline{P_1}$. The moved variant of the given *gtu*-derivation starting at $\overline{G}$ is called *guide* and $\overline{G} = AS$ *active spot* being the start graph of a permitted *gtu'*-derivation that is built further on.

(f2) Repeat the following operations as long as possible depending on the next rule application permitted by the stepwise control condition $C_{gtu'}$.

- (f21) If the next rule to be applied is the first one of a functional section of a permitted derivation given by *funct*, then apply *sprout*(*funct*) to *AS* where the graph *funct*(*AS*) is the new *AS*. Then apply *conflux* as long as possible using the independence of P_{gtu} and $P_{gtu'}$. The moved variant of the guide is the new guide.
- (f22) Otherwise, apply *couple*(*span*) for some given *span* to the first direct derivation of the guide where this is possible, provided that the prolongation yields a permitted derivation. Then apply *conflux* and *interchange* as long as possible. In particular, the added direct derivation is moved to *AS* prolonging the permitted *gtu'*-derivation. The result graph is the new *AS* and the following *gtu*-derivation is the new guide.

(f3) Afterwards, apply *conflux* as long as possible using the independence of $\overline{P_2}$ and $P_{gtu'}$ moving the constructed permitted *gtu'*-derivation to $red(G)$. Then continue the derivation if $C_{gtu'}$ requires further rule application.

(f4) Check whether the derived graph is in $SEM(T_{gtu'})$.

Backward Reverse the procedure of the forward proof by starting with the structure of Figure 4c and changing the roles of *gtu* and *gtu'*.

(b1) Apply *interchange* instead of *conflux* as long as possible using the independence of $\overline{P_2}$ and $P_{gtu'}$.

(b2) Repeat the operation as in (f2), but now using the spans conversely.

(b3) Afterwards, apply *interchange* as long as possible using the independence of P_{gtu} and $\overline{P_1}$.

(b4) Check whether the derived graph is in $SEM(T_{gtu})$.

Preprocessing As the backward proof of *hampath-to-2-bounded-spantree* shows, a preprocessing may be necessary before the backward part of the proof procedure can work. The reason is that a graph may have several successful derivations, but only some of them may be suitable for forward and backward processing. In such a case, one may apply *associate*(*pair*) for appropriate pairs of derivations together with *interchange*.

It may be noted that our proof for *clique* $\leq$ *independent-set* in Example 8 must be modified to be covered by the proof procedure in the following way: α is replaced by α' in *independent-set* and the corresponding derivation is constructed by applying *couple* for each rule application where the span $\overset{\alpha}{\circlearrowleft}\!\circ \underset{1}{\Longleftarrow} \circ \underset{1'}{\Longrightarrow} \overset{\alpha'}{\circlearrowleft}\!\circ$ is used.

The proof procedure works very well for several examples, but is far from perfect. For instance, we have failed to prove the correctness of the reduction from independent set to vertex cover and from vertex cover to Hamiltonian cycles. The reason seems to be that one needs further operations. But there are various further reasons why the proof procedure may fail: (1) The chosen units for the involved NP problems and the reduction may not be suitable. (2) The preprocessing may not yield a successful derivation for further processing. (3) The processing may get stuck because the proper spans are missing. (4) The processing may result in a permitted derivation that is not successful. Such problems are not unusual if the employed proof technique is not complete as it is often the case. Fortunately, each of the problems indicate how one may find a way out: Try further operations, other modeling units, other spans for coupling, other pairs for association, new ideas for preprocessing. Confer the conclusion where some of these points are a bit more elaborated.

7 Conclusion

We have made a proposal on how the correctness of reductions between NP problems may be proved by graph-transformational means. For this purpose, we have provided a toolbox that allows to construct successful derivations of one graph transformation unit from successful derivations of another graph transformation unit that represent positive decisions and are connected by a reduction derivation in their initial graphs. The approach is an attempt in the early stage of development and needs further investigation, to shed more light on its usefulness.

1. The proof procedure has turned out to be suitable in several cases, but it seems to fail in other cases like, for example, for the sophisticated reduction from the vertex cover problem to the Hamiltonian cycle problem (cf. [1, 2]).
2. To cover more cases or to simplify proofs, one may look for further operations. Candidates are operations like *conflux* and *interchange* that are based on independence including parallelization, sequentialization, and shift (cf. [13]).
3. Another possibility is to generalize the coupling by considering spans that consist of derivations rather than direct derivations.
4. The preprocessing needs more attention. An initial graph may have a large class of successful derivations of which only a few particular ones may be suitable for the further proof procedure. Therefore, operations that preserve successfulness are of interest in addition to the association.
5. The toolbox relies heavily on independence. Can this be always achieved by respective units? Can it be relaxed?

6. Is there any chance of tool support for such correctness proofs? For particular initial graphs, the proof procedure is of an algorithmic nature. But what about arbitrary successful derivations and reduction derivations?
7. A reduction between NP problems is a kind of model transformation. Do results of the theory of model transformations and their correctness proofs like triple graph grammars (cf., e.g., [14]) apply to reductions?

Acknowledgment. We are grateful to the anonymous reviewers for their valuable comments.

References

[1] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W.A. Freeman and Company, New York, 1979.

[3] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – an overview. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2008.

[4] Hans-Jörg Kreowski and Sabine Kuske. Polynomial graph transformability. *Theoretical Computer Science*, 429:193–201, 2012.

[5] Hans-Jörg Kreowski and Sabine Kuske. Graph multiset transformation – A new framework for massively parallel computation inspired by DNA computing. *Natural Computing*, 10(2):961–986, 2011. DOI:10.1007/s11047-010-9245-6.

[6] Marcus Ermler, Sabine Kuske, Melanie Luderer, and Caro von Totth. A graph transformational view on reductions in NP. *Electronic Communications of the EASST*, 61, 2013.

[7] Hartmut Ehrig, Michael Pfender, and Hans-Jürgen Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.

[8] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, pages 163–245. World Scientific, Singapore, 1997.

[9] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[10] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In *Proc. Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 284–293, 1976.

[11] Hartmut Ehrig and Barry K. Rosen. Commutativity of independent transformations on complex objects. IBM Research Report RC 6251, Thomas J. Watson Research Center, Yorktown Height, New York, USA, 1976.

[12] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[13] Hans-Jörg Kreowski. Transformations of derivation sequences in graph grammars. In *Proc. Conference on Fundamentals of Computation Theory*, volume 56 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 1977.

[14] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *International Conference on Graph Transformation*, pages 411–425. Springer, 2008.

Graph Representations in Traditional and Neural Program Analysis

Elizabeth Dinella
University of Pennsylvania, USA
edinella@seas.upenn.edu

Traditional program analysis techniques often represent programs as graphs where nodes are program entities and edges represent various relations (data flow, control flow, call graphs, etc). Graphs are a standard representation for programs as they can capture syntactic and semantic structure with long range dependencies. Recently, "neural" program analysis has been proposed for traditional analysis tasks. Representing a program as input to a machine learning model is an active area of research. Popular representations include sequences of tokens, relational databases, and graph neural networks. In many cases, a graph representation is preferable. Program repair is an example of such a task. In this talk I will discuss the advantages and challenges of representing programs as graphs in both traditional and neural program repair.

Speaker Biography. Elizabeth is a PhD student at the University of Pennsylvania advised by Mayur Naik. She works on models of code for program analysis tasks such as program repair (Hoppity[1]). As an intern at Microsoft Research, Elizabeth worked on the DeepMerge[2] project contributing an edit-aware neural approach to merge programs. More recently, her interests lie in neural test suite generation (TOGA[3]), and web3 security. In her free time, Elizabeth enjoys hot coffee and her chow chow puppy Cinnabon.

[1] https://openreview.net/pdf?id=SJeqs6EFvB
[2] https://www.microsoft.com/en-us/research/publication/deepmerge-learning-to-merge-programs/
[3] https://www.seas.upenn.edu/~edinella/icse-camera-ready.pdf

Panel Discussion: Learning Graph Transformation Rules

Reiko Heckel (chair)
School of Computing and Mathematical Sciences
University of Leicester, UK, rh122@leicester.ac.uk

As with any detailed model of a complex problem, creating a graph transformation system is hard. An alternative in some situations is to learn models from historical or observed data or derive them by optimising a basic model based on some objective function using feedback from simulations or real-world processes. The session is dedicated to discussing application scenarios and solutions for learning, in particular, the rules of a graph transformation system with the aim of inspiring collaborative research on this topic. A preliminary classification of rule learning problems could distinguish these two aspects.

Prescriptive vs descriptive model A model is *prescriptive* if it is intended as a design of a system to be built, *descriptive* if it represents an existing system to be analysed.

In the prescriptive case, creating a model that best suits certain requirements is a form of optimisation based on a given objective function. In order to assess the fitness of a model, we have to either run the model directly (e.g. in a simulation) or use it to control a real-world process we can measure. For example, we may want to optimise the topology management for a decentralised network, maximising connectivity while minimising the number of links to be maintained. Such requirements should be expressed by the objective function and the aim should be to optimise the application of a set of basic rules, e.g. for nodes joining or leaving, making or breaking links.

In the descriptive case, we infer the model from existing data observed from a real process, e.g., the version history of a software project. This can be a precursor to a prescriptive use of the model in a re-engineering scenario, e.g., inferring a model of how faults are detected and fixed in software projects in order to recommend typical fixes. Rules in this case should represent typical fault patterns and their repair actions.

Online vs offline learning In either of the above scenarios, learning can be *online* (at runtime, using a form of reinforcement learning) or *offline* (based on historical data). In the online case we make continuous observations and use them to modify the model, e.g., in a stochastic or probabilistic model, adding weight to successful rules (in the prescriptive scenario) or frequently observed rules (in the descriptive case). In the offline case, there is no direct feedback or continuous update, but for prescriptive models learning can happen in longer cycles of model inference, usage and feedback.

Based on the above we would like to collect, describe and classify application scenarios for rule learning, such as network topology management or fault detection and repair, and discuss possible solutions. The result could be a position paper summarising our findings.

Acknowledgements The topic for this panel arose from a series of discussions between the chair, Nicolas Behr, Bello Shehu Bello, and Sebastian Ehmes, with further input from Marc Santolini and Ashiq Anjum.

