Pre-Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

September 24th, 2012, University of Bremen, Germany

A satellite event of ICGT’12
Preface

This volume contains the pre-proceedings of GraBaTs 2012, the 7th International Workshop on Graph Based Tools. GraBaTs 2012 is held as a workshop co-located with ICGT 2012, the 6th International Conference on Graph Transformation. This workshop continues the GraBaTs series of workshops that serve as a forum for researchers and practitioners interested in the development and application of practical graph-based tools. Previous workshops were held in 2002 (Barcelona, Spain), 2004 (Rome, Italy), 2006 (Natal, Brazil), 2008 (Leicester, UK), 2009 (Zurich, Switzerland), and 2010 (Enschede, The Netherlands). The format of the workshop varied over the years. In most issues (as in the current one), papers were presented and collected in workshop proceedings. In 2008/2009, GraBaTs was held as a tool transformation contest, which took place in 2010/11 as a separate event (TTC) co-located with the TOOLS conference.

A wide variety of graph-based tools have emerged, including e.g. tools for visual languages, model driven development (editing, analysis, simulation, and transformation of models), data analysis, pattern recognition, software evolution, and efficient graph algorithms. In all of these areas, tools are developed that retrieve, store, transform, and display graphs. It is the purpose of this workshop to survey the state of the art of graph-based tools, bring together developers of graph-based tools in different application fields and to encourage new tool development cooperations.

This year we received 17 submissions (15 regular and 2 short papers). From these, 8 submissions (7 regular papers and 1 short paper) were accepted for being presented at the workshop and for being included into the GraBaTs proceedings. Most of the accepted papers deal with different aspects of model transformations such as graph pattern matching, code generation, domain-specific model transformations, and triple graph grammars.

We would like to thank the members of the Program Committee and the external reviewers for their excellent work. We are also indebted to the members of the ICGT organization committee who provided outstanding support for running the GraBaTs workshop smoothly. Last but not least, we would like to express our thanks to Arend Rensink, who enriched the GraBaTs program by his invited talk.

September 2012

Christian Krause, Bernhard Westfechtel
PC chairs of GraBaTs 2012
Program Committee

- Artur Boronat, University of Leicester (UK)
- Claudia Ermel, Technische Universität Berlin (Germany)
- Esther Guerra, Universidad Autónoma de Madrid (Spain)
- Ethan Jackson, Microsoft Research (USA)
- Dimitris Kolovos, University of York (UK)
- Christian Krause, Hasso Plattner Institut, Universität Potsdam (Germany)
- Tihamér Levendovszky, Vanderbilt University (USA)
- Mark Minas, Universität der Bundeswehr München (Germany)
- Arend Rensink, Universiteit Twente (The Netherlands)
- Gabriele Taentzer, Philipps-Universität Marburg (Germany)
- Pieter Van Gorp, Technische Universiteit Eindhoven (The Netherlands)
- Hans Vangheluwe, University of Antwerp (Belgium) / McGill University (Canada)
- Gergely Varró, Budapest University of Technology and Economics (Hungary) / Technische Universität Darmstadt (Germany)
- Bernhard Westfechtel, Universität Bayreuth (Germany)
- Andreas Winter, Carl von Ossietzky Universität, Oldenburg (Germany)
- Albert Zündorf, Universität Kassel (Germany)

External Reviewers

Anthony Anjorin, Thorsten Arendt, Regina Hebig, Andreas Koch, Marius Lauder, Stefan Lindel, Sebastian Wätzoldt
# Table of Contents

Invited Talk: Graph Transformation, Check; And Now What?  
*Arend Rensink*  
4

Gray Box Coverage Criteria for Testing Graph Pattern Matching  
*Martin Wieber, Andy Schürr*  
6

Adding Rule-Based Model Transformation to Modelling Languages in MetaEdit+  
*Simon Van Mierlo and Hans Vanghelauwe*  
19

ModGraph – Generating Executable EMF Models  
*Sabine Winetzhammer*  
32

Integration of Triple Graph Grammars and Constraints  
*Stephan Hildebrandt, Leen Lambers, Basil Becker and Holger Giese*  
45

Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars  
*Claudia Ermel, Frank Hermann, Jürgen Gall and Daniel Binanzer*  
58

EMorF – A tool for model transformations  
*Lilija Klassen and Robert Wagner*  
71

ReConNet – A Tool for Modeling and Simulating with Reconfigurable Place/Transition Nets  
*Marvin Ede, Kathrin Hoffmann, Gerhard Oelker, Julia Padberg*  
78

Rooted Graph Programs  
*Christopher Bak and Detlef Plump*  
90
Pre-Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

Graph Transformation, Check; And Now What?

Arend Rensink

1 pages
Graph Transformation, Check; And Now What?

Arend Rensink

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl

Abstract: The graph transformation community can certainly claim some success in turning its concepts into usable tooling, for instance in the context of domain-specific modelling and model transformation. However, how can we further extend our scope? What are necessary steps in cementing these successes and reaching into new application areas?

In this talk I will indicate some potential directions for graph transformation-related tool development to go: quantitative aspects, reactive components, planning. Mostly this involves hooking up into other tool sets, with different strengths. Part of the issue also involves standardisation: but what to standardise on? Do we revive GXL, should we look to Eclipse instead, or is there another answer?

Keywords: graph transformation, tools, quantitative aspects, reactive components, planning
Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

Gray Box Coverage Criteria for Testing Graph Pattern Matching

Martin Wieber, Andy Schürr

12 pages
Gray Box Coverage Criteria for Testing Graph Pattern Matching

Martin Wieber\textsuperscript{1}, Andy Schürr\textsuperscript{2}

\textsuperscript{1}martin.wieber@es.tu-darmstadt.de, \textsuperscript{2}andy.schuerr@es.tu-darmstadt.de
Technische Universität Darmstadt, Real-Time Systems Lab, D-64283 Merckstraße 25, Darmstadt, Germany

Abstract: Model transformations (MT) are a core building block of Model-Driven Engineering. The quality of MT specifications and implementations is vital to their success. The well-researched formal underpinning of graph transformation (GT) theory allows for proving quality-relevant properties and enables stringent implementations. Yet, in practice, MT implementations often depend on verification/validation techniques based on dynamic testing. This work presents a new gray box coverage approach for systematic testing of GT-based MT implementations and pattern specifications. The approach uses GT specifics and enforces systematic testing by examining variable binding and unbinding steps, thereby not making further assumptions about the underlying pattern matching algorithm. A family of coverage criteria is defined as temporal logic (LTL) formulae, and the effectiveness of concrete criteria in limiting the testing effort is examined by an example.

Keywords: gray box testing, adequacy criterion, coverage, graph pattern matching

1 Introduction

Model transformations (MT) are a core concept of Model-Driven Engineering (MDE), and like any other software, MT programs should be examined thoroughly w.r.t. quality and correctness. MTs are usually written in specialized languages, often rule-based, following a declarative approach (cf. \cite{CH06}). Languages and tools (code generators, interpreters) are still frequently under construction, the number of users is limited, and esp. tools are commonly considered immature to some extent. Formal underpinnings like graph transformation theory \cite{EEPT06} temper the negative effects of this situation, but application of formal methods to concrete realizations is often tedious and infeasible in practice due to the imposed effort and inherent complexity.

Testing seems to be a promising solution to both the problem of ensuring the quality of MT specifications as well as of assessing the quality of MT implementations. In contrast to general-purpose programming languages, for which implementation-based (white or gray-box) testing techniques exist, there is no such well-established equivalent for typical MT languages. A fact that is partly founded on the challenges inherent to model transformation testing \cite{BGF+10}. In the rest of the paper, we restrict ourselves to the important class of graph pattern based MT languages. For this class of MT languages we introduce a novel concept of coverage which provides an adequacy criterion that is independent of the source code of a pattern matching (PM) implementation but still utilizes knowledge of the PM process. It limits the testing effort by defining min. level of coverage that a test suite should satisfy to be considered sufficient (construction of
tests – in form of concrete input models – is beyond the scope of this work). Several application scenarios are conceivable: (1) testing a concrete PM implementation based on the pattern (black-box) (2) testing an immediately executable MT specification (white-box) (3) rough figure of merit for existing test suite’s completeness. For the sake of clarity, compare a graph transformation (GT) tool to the Java compiler: MT specification plus metamodel correspond to source code plus Java types. Testing a Java method requires test inputs that exercise the method’s code sufficiently (w.r.t. statement coverage etc.). The same is true for good graph pattern tests exercising a PM implementation, except that other and new coverage notions are required.

The remainder of the paper is organized as follows: Section 2 introduces terminology and concepts based on an example. Section 3 describes the PM process (esp. its operational semantics) and necessary testing infrastructure. Section 4 presents our novel coverage notion and core contribution. Section 5 reports on first results drawn from an application of the concepts to the given example based on a prototype framework implementation and manually derived test suites. Section 6 provides related work, and in Section 7 we draw conclusions and outline future work.

2 Terminology and Running Example

Metamodels and Models. A metamodel (MM) reflects the core concepts of the modeled domain. It features a graph-like structure and defines the (abstract) syntax of corresponding instance models (M). The MM comprises classes as its nodes and associations as its edges, the former define valid types of the models, the latter define the structuring/relations among them.

Figure 1a shows an exemplary metamodel. It represents a minimalist description of Simulink (a tool often used in engineering). A System contains Blocks and Lines, Blocks comprise Outports and Inports. Blocks can further be interconnected via Lines originating and ending at Ports.

Model- and Graph Transformations. A MT takes models, conforming to one input MM, and transforms them to models, conforming to another MM. Here, we do not distinguish between the in-place modification of models and the translation of models from one language to another. The specifics of the considered MT are defined by graph transformation theory (c.f. [EEPT06]), where models are represented by typed graphs and transformations are formalized by graph rewriting rules. Several tools implementing a GT approach are available, although the formalizations/implementations can differ (cf. [Tae04, JBK10, LV02] for examples).
Graph Patterns  The aforementioned tools all specify a MT via declarative, rule-based descriptions of graph rewriting step. They all use some sort of graph pattern (GP) notion, which is referred to as the left-hand side (LHS) of a GT rule. During the processing of one rule, the transformation engine tries to find a match for the LHS graph in the model (or host graph), where a match is a sub-graph of the host graph that is isomorphic to the LHS. This process is commonly implemented based on some elaborated search algorithm, instead of relying on a trivial “generate and test”-approach. The former usually combines depth-first-search and backtracking, one exception being RETE-like pattern matching algorithms which compute (sub)matches in parallel, sacrificing memory consumption for run-time performance [BÖR+08]. In the following we will restrict ourselves to depth-first-search algorithms (for more details see below).

Figure 1b depicts an exemplary GP. This GP comprises six variables: $S$, $I_1$, $I_2$, $B$, $O$, $L$ of the respective types State, Inport and so forth. The specially outlined node $S$ is initially bound, meaning that is initially set to some fixed node. During PM, the engine tries to assign objects (of compatible types; links constrain the number of valid variable bindings further) to the remaining variables. The process of assigning a concrete value to one of the variables is called binding operation/step. To reassign a new value to the variable, this binding step needs to be reverted first, resulting in an unbound (intermediate) state. We refer to this as unbinding step.

Pattern Matching  The goal of PM is to find a match of the LHS of the GP in the host graph. As mentioned before, PM is usually done stepwise by a form of local search: a partial match (defined by a set of bound variables) is extended incrementally by binding additional variables, starting with a set of initially bound variables (in our example: $\{S\}$). If all variables can eventually be bound without violating constraints, a match will be found. If, during the process, the next to-be-assigned variable can not be bound, already bound variables get unbound and reassigned, when possible. This procedure is repeated until either a complete match can be determined or all possible permutations have been checked and discarded. In case a match has been found, it is replaced by a copy of the right-hand side (RHS) of the GT rule (rewrite step). This modifying step is out of scope and neglected here, since the focus lies solely on testing the PM.

In addition to structural and type constraints, and depending on the underlying graph model, the pattern might further be constrained by attribute conditions. Another common extension to basic GP is the concept of negative application conditions (NAC). NACs are specially marked elements of a pattern that, if their identification is possible, prevent a match from being valid. Other possible extensions to GP include optional elements (nodes or edges that can optionally be included in a match) and set-nodes (results in matches of variable and unknown size; comparable to the *-operator in regular expressions). In the remainder, only basic patterns are considered; NACs, optional elements, and set-nodes are omitted and left for future work.

3 Operational Semantics and Tracing

In this chapter we derive an abstract description of the run-time behavior of typical pattern matchers. This forms a (MT) language-neutral basis for the definition our coverage concepts in the following sections.
3.1 Operational Semantics of Pattern Matching

The process of determining a valid match for a given pattern in an input graph is crucial for the entire model transformation. Our experience in developing code-generator-based pattern matching engines shows that implementing/optimizing this task is complex and error-prone.

As already mentioned, a pattern is usually searched for iteratively by partial match extension and relying on backtracking. The overall process can be visualized in form of a decision tree as in Figure 2, where each decision/branching corresponds to an atomic operation like “extend partial match by navigating along link $l$ binding previously free variable $X$”, “check existence of link $l$ between two bound nodes $X, Y$” or “ensure variable $X$ and variable $Y$ are not bound to the same node”. The PM process is essentially characterized by such a sequence of operations (of types $extend$ and $check$), and one (not necessarily unique) concrete ordering of such operations is called a search plan (SP). The derivation of all possible, viable SPs for a given pattern, in order to determine the optimal one, is often computationally infeasible, and optimality also depends partially on the model (cf. [VDWS12] for details on this issue).

Failed check operations eventually lead to backtracking steps and indirectly result either in variable reassignments or incomplete matches. The pattern matching process thus is characterized solely by a sequence of binding and unbinding steps, whereby, by definition, the unbinding steps can only occur in inverse order of the corresponding binding steps.

The motivation for this abstraction lies in the possibility to describe the PM process in as much detail as possible without depending on a concrete implementation (e.g. in the sense of source-code). When using a compiled, generator-based approach, where code is derived from the pattern specification, one might design test suites based on that code using classical techniques. Unfortunately, even small changes to the pattern or a non-deterministic code-generator would result in different code, deprecating existing tests. The situation even worsens when using an interpreter-based approaches. Testing the PM process based on the source-code of an interpreter seems futile, because the exercised fraction of code can not be expected to change much with different SPs or models as the authors of [HLG+12] indicate.

3.1.1 Generated Search Plans

We generated several code variants with our customized CodeGen2 fork (from Fujaba), resulting in equivalent but different SPs. We considered two SPs, termed SP$1$ and SP$2$, for further investigation. Figure 2 depicts them. In the figure, each node represents a state, characterized by a mapping of the $n$ ($n = 6$) variables of the pattern $(S, I_1, \ldots)$ to the domain $\{0, 1\}^n$, where a value of 0 indicates that the resp. variable is not bound, and a value of 1 indicates the opposite. Transitions are labeled with the operations that are either used to structurally extend the partial match or to check pattern constraints. We use grey highlighting to indicate no change in the binding vector. From our previous work [VDWS12], we borrow the notions of adornments and masks to express the application conditions of the operations, distinguish check from extend operations, and indicate navigation direction. Basically speaking, the adornments define which variables should be bound ‘$B$’ or unbound ‘$F$’; masks define an ordering of all variables and indicate which variables are non-restricted ‘$*$’ for applicability.
3.2 Tracing the Pattern Matching Process

Observability is one prerequisite for testability, as Binder states in [Bin00]. So, when examining the dynamic behavior of the pattern matcher, one needs information on the engine during the algorithm’s running. One can imagine that (almost) any pattern matching implementation can be easily extended or instrumented to output traces of states comprising the binding status of the pattern’s variables. Consequently, we restrict ourselves to the information obtained through simple (offline) tracing, where status information is collected/stored during run-time. In our case, we instrumented the PM code manually, but adapting our generator templates is considered to be a straightforward task and left for future work.

4 Coverage Metrics

We continue by defining the gray box coverage metrics and motivate their usefulness to PM testing. As far as our experience goes, implementation bugs are likely to manifest in incorrect backtracking behavior during the search (e.g. premature, too late, etc.). On the other hand, bugs that only manifest in “unusual” situations are more likely to be overlooked in comparison to others. Consequently, for thorough testing, one needs to ensure that the test suite exercises the search plan in a systematic and sufficient manner. Our coverage metrics define minimal requirements so that when met, the test suite can be reasonably considered sufficient w.r.t. the examined patterns. Consequently, the metrics serve the same purpose as (code) coverage in traditional testing, namely “[. . . ] to mitigate unavoidable blind spots [. . . ]” [Bin00] during test suite design, and to help the tester in performing the task of path sensitization¹. If one tries to test the general functioning and fitness of a PM engine, regardless of any concrete MT, one needs to provide metamodels plus patterns (together they form the test cases) that are “complex enough” to cover all supported language features – a task that is out of scope here. This suggests, that the presented method is MT specific.

Coverage Items The basic idea of our testing approach is to stimulate the PM engine in such a way that certain combinations of variable binding and unbinding steps actually do occur. This is achieved either by extending a partial match or by performing a backtracking step. One single

¹ “Process of determining [. . . ] variable values that will cause a particular path to be taken”. ([Bin00], p. 399)
sequence of such steps represents a coverage item. If a trace log indicates that a test run led to
the occurrence of the demanded sequence, we say that the test covers the sequence and thus the
coverage item. To be able to precisely control the testing effort, we developed different metrics
and ensured them to form a hierarchy, where more elaborated metrics subsume more basic ones,
so that coverage of the subsuming criterion implies coverage of the subsumed one.

In the following, Linear-time Temporal Logic \(^2\) (LTL) is used to define the cov. items. Some
basic definitions are required first (\(x\) refers to one variable, \(X\) to the set of variables defined by
the pattern, \(\mathcal{X} := (X, \prec)\) is a strict totally ordered set, and \(\mathcal{X}' := \mathcal{X} \setminus \{\text{initially bound variables}\}\).
Additionally, for this example \(\mathcal{X} = (\{S, I_1, I_2, B, O, L\}, \text{order of appearance})\):

\[
b : \mathcal{X} \to \{0, 1\}, \quad b(x) := \begin{cases} 0, & \text{if variable } x \text{ is unbound} \\ 1, & \text{if variable } x \text{ is bound} \end{cases} \quad (1)
\]

\[
m \in \{-1, ?, +1\}, \quad m \triangleq \begin{cases} -1, & \text{matching failed} \\ ?, & \text{match still incomplete} \\ +1, & \text{match found} \end{cases} \quad (2)
\]

Equation 1 defines a function that assigns binding information to variables, and in Equation 2 \(m\)
is introduced, which represents knowledge about the outcome of the PM run. Figures 2 and 3 use
these concepts. Both definitions are used to formulate temporal predicates, whereby the pattern,
SP, and traces are kept implicitly fixed. In this regard, actual values depend on an implicit notion
of time/state (corresponding indices/arguments are omitted here, as in [HR04]). For example, to
state that a run should result in a complete match, one would define a predicate \(\Phi^+\) such as the
one in Equation 3 using LTL-operators. It states that eventually \((\mathsf{F}(.))\) variable \(m\) should remain
‘+1’ “forever” or globally \((\mathsf{G}(.))\). Until this happens \((\mathsf{U}(.))\) variable \(m\) should have the value
‘?’, indicating an incomplete match. Equation 4 considers the opposite outcome.

\[
\Phi^+ := (m = ?) \mathsf{U}(\mathsf{G}(m = +1)) \quad (3)
\]

\[
\Phi^- := (m = ?) \mathsf{U}(\mathsf{G}(m = -1)) \quad (4)
\]

\[4.1 \text{ PMC}_0\]

We now define the simplest of our coverage criteria, called \(\text{PMC}_0\), whereby PMC stands for
pattern matching coverage. It comprises the two conditions \(\text{PMC}_0^+\) and \(\text{PMC}_0^-\) which need to be
fulfilled separately. Informally, the first states that for every variable \(x\) there has to be (at least)
one test case that eventually binds \(x\) to some node in the model (of compatible type) and results
in a complete match (positive test). The second one states the same but excludes a complete
match in the end (negative test). More formally, the following proposition has to hold:

\[
\text{PMC}_0(\mathcal{T}) := (\text{PMC}_0^+(\mathcal{T}) \land \text{PMC}_0^-(\mathcal{T})) \quad (5)
\]

whereby \(\mathcal{T}\) denotes a set of captured traces, and one single trace (= sequence of trace entries,
whereby an entry is a bit vector, cf. columns in Figure 3) is referred to as \(t\). We also define a

\[2\] For a general introduction see, e.g., [HR04].
Theoretical extended trace \( t^\infty \) being an infinite sequence that repeats the last item of \( t \) in positive direction along the time axis. The aforementioned proposition holds iff the following is true:

\[
(\forall x \in X' \exists \tau \in \mathcal{T} : \text{PMC}_0^+(t,x)) \land (\forall x \in X' \exists \tau \in \mathcal{T} : \text{PMC}_0^-(t,x))
\] (6)

\[
(\text{PMC}_0^+(t,x)) \iff (\Phi_0^+(x) \text{ holds for the extended trace } t^\infty)
\] (7)

Equation 8 and Equation 9 state that \( x \) has to be initially unbound and has to become eventually bound. At the end, a steady state has to be reached where either a match has been found or not.

Figure 3a visualizes the concept for the case where \( x \) is set to \( B \). A minimal test suite meeting the requirements of PMC\(_0^0\) is presented later on in Section 5. Note that the initial, one intermediate, and the final state are restricted w.r.t. variable states. Any number of arbitrary intermediate states (indicated by the dots in the resp. fig.) are allowed to occur. Also note, that no distinct matching order, as predetermined by a concrete SP, is required for the definitions. Note further that one can expect the variables to be bound in the final state, at least in the case where there is a complete match. Nevertheless, the presented formulation is open to adaption (potential extensions could supports optional nodes etc.). Summarized, PMC\(_0^0\) only ensures a very rudimentary coverage, since it does not necessarily imply or require any form of complex branching during the PM process, but introduces the concepts and forms the basis for refinements.

### 4.2 PMC\(_1\)

The first extension is PMC\(_1\). It subsumes PMC\(_0^0\) and extends PMC\(_0^0\) in that distinct variables \( x \) need to consecutively take on the bound, the unbound, and the bound state again, before reaching the steady state with either a match (PMC\(_1^+\)) or no match (PMC\(_1^-\)) as outcome (intermediate states are permitted as well). This ensures a more thorough testing due to additionally required binding and unbinding steps and is likely to require larger (element count) and more complex test models. When met, it guarantees that for each variable at least two (not necessarily distinct) options were evaluated within one captured test run. Figure 3b underlines the concept. For formalization, we can reuse Eqs. 5, 6, 7, and 9 directly (by changing index 0 to 1). Equation 10...
Gray Box Coverage for Testing Graph PM

is the adapted version of Equation 8. Compared to the latter case, there are additional nested statements requiring variable \( x \) to be bound first (outermost \( F \) in the disjunction), afterwards unbound (intermediate \( F \)), and ultimately bound thereafter (innermost \( F \)).

\[
\Phi_1^\pm := \Phi_0^\pm \land \left( G(b(x) = 0) \lor F\left( (b(x) = 1) \land F\left( (b(x) = 0) \land F\left( G(b(x) = 1) \right) \right) \right) \right) \quad (10)
\]

Figure 3b and the previous remarks suggest that one could also demand more than one “cycle” of unbind/bind operations for a variable. This motivates the generalization of \( \text{PMC}_1 \), or \( \text{PMC}_{1,1} \) from now on (the second index stands for one cycle), to the case of \( \text{PMC}_{n} \), requiring \( n \), \( \{n \in \mathbb{N} \mid n \geq 1\} \), of such cycles. A higher value of \( n \) will likely lead to increased test model sizes, but this does not necessarily imply coverage of subsuming metrics as explained later.

Right now, there is ongoing work spent on the extension of the concept to \( \text{PMC}_{2,n} \) (pairs of variables), \( \text{PMC}_{3,n} \) (triples of variables) and so forth by considering variable tuples instead of single variables. This would help in testing the interplay of binding/unbinding steps for different variables. Further details are omitted here, due to space limitations.

5 Application

Now we come to the discussion of preliminary results obtained during evaluation of our prototype coverage framework. We modeled the pattern of Figure 1b in our eMoflon [ALPS11] tool suite and generated two Java realizations featuring a distinct SP from it (cf. Figure 2). Tracing commands were added manually to the code, and the tracing information was collected and analyzed by a prototypic tool.

Basic Test Suites We used the setup to evaluate whether it is possible to construct input data that leads to complete coverage for certain metrics, given the previously fixed SPs. With prior knowledge of an actual SP, it turned out to require not much of an effort to find small test models meeting the requirements. Figure 4 depicts test suites for \( \text{PMC}_0 \) and \( \text{PMC}_{1,1} \) respectively. The results indicate that one can effectively limit oneself to a pair of input models to ensure coverage of one (or several, in case of subsumption) metric for all relevant pattern variables at the same time, although this seems not very favorable when thinking of maintainability and traceability. Nevertheless, metamodels with additional constraints might restrict input models (e.g. by upper multiplicity bounds) so that one can not achieve the required coverage level with a minimum of two models. Chained patterns (as in programmed GT) would complicate things considerably.

The results also support the working assumption that our stronger metric leads to more complex test data in comparison to the basic one. Additionally, it should be noted that there exists a distinction between relevant input complexity and input complexity without influence on the search process. For example, when relying only on the metamodel to construct input models, one could construct very diverse looking models with (superficial) complexity (e.g. high element count), which is all but relevant to thorough testing the pattern. Think, for example, of “early decisions” during the PM process, which might skip further examination of the alleged complex parts. If the search plan changes, previously “irrelevant” complexity could turn out beneficial, though, and the test data might lead to sufficient coverage. Our coverage notion enables us to construct tests with relevant complexity systematically, without relying on chance.
Comparison with Code Coverage Deriving imperative code from a MT specification generally enables us to compare our coverage concept to well-accepted code coverage approaches. We used Cobertura\(^3\), which supports basic block coverage and branch coverage, to measure code-based coverage figures for the example.

Results show that even the test data for PMC\(_{0}\) already lead to complete statement coverage of the code. We presume that this is always the case, but this claim needs to be supported by further investigation (ongoing work). Also, there is no such thing as a canonical implementation, and it would be interesting to compare code coverage measures for functionally equivalent (even down to the search plan) code representations which are sufficiently different.

We also investigated whether code statement coverage implies coverage in our sense. Think of the situation, where one tries to test the code sections that process a complete match. One needs an input model comprising a complete match as test case, which shall be the only test in an initial test suite. We could extend the model to cover as many additional statements as possible, but due to the code fraction that processes incomplete matches as ultimate result, it is not possible to cover all statements with only just one test. A second, rather small model without complete matches is required. In general, this model can be expected to be rather small so that it does not ensure PMC\(_{0}\) coverage (cf. negative test). For the running example, Figure 4b depicts a test suite that achieves statement coverage but fails in achieving PMC\(_{0}\) coverage. For our concrete example, statement coverage is implied by PMC\(_{0}\) coverage, so here even our weakest coverage criterion subsumes statement coverage (unnecessary/dead code is neglected). This is obviously not a prove, though, and more experiments are required to investigate the interplay with code-based criteria.

\(^{3}\) http://cobertura.sourceforge.net/
6 Related Work

Several groups published results related to MT testing. A wide range of articles examine test data generation, but this only partially relates to test adequacy evaluation. By far the most coverage related work involves specification-based black box testing. A good example is [FBMT09] by Fleurey et al., where they motivate their decision for a black box approach with its independence from the underlying transformation language. They introduce the notions of class, attribute and association coverage. Bauer et al. extend these concepts in [BKE11] by introducing feature and transformation contract coverage. Their overall goal is to assess and optimize the test suite quality for model transformation chains. Both works do not anticipate pattern definition and/or implementation related bugs directly.

In [GV11], Gogolla et al. present a testing approach based on OCL constraints on source and target metamodel called Tracts (MT contracts). They describe how their USE tool can check contract adherence on the outcome and how corresponding test models can be derived. Cabot et al. describe a similar approach in [CCGL10] where they use OCL in conjunction with their UMLtoCSP tool whereby focusing on the analysis of MT properties rather than on testing. Both works do not examine test data adequacy aspects and esp. no coverage notion.

Darabos et al. take a different view on testing in [DPV08]. They present a fault model, which condenses their knowledge of typical programmer faults during implementation of PM engines. They use a hardware verification technique called Boolean difference method to derive test data which is sensitive to such faults. In some sense, the faults captured by their model could be interpreted as coverage items. A drawback of this approach is that it depends on the quality of the fault model. Unconsidered faults are likely to be overlooked.

Hildebrandt et al. use TGG rules to derive pairs of input models and expected output models for testing a TGG implementation in [HLG+12]. Although this represents an elegant approach to test oracle construction, the approach can only be applied if a TGG specification exists. Unfortunately, TGGs are not (yet) as expressive as most MT languages. The authors also evaluate code coverage of their interpreter-based MT engine with different derived test suites, and state that it remains virtually unchanged.

In [KA07], Küster et al. report on what they call a white box approach to validation. In addition to the input metamodel they use the “design and implementation of the model transformation” by building on so-called meta model templates plus constraints. A basic fault model is provided and the interplay of rules is examined. Other white box approaches are presented by Ciancone et al. in [CFM10], where unit testing of MT specified in QVT Operational is applied, and in [MP09] by McQuillan et al., where standard code coverage metrics are used for ATL specifications. Such white box approaches have the draw-back of being language specific. They also require more insight in the machine-runnable implementation, which contravenes the declarative paradigm.

Steel et al. describe the test driven development of the Tefkat engine in [SL04]. Geiger et al. derive JUnit tests from pattern specifications in [GZ05] which are used to test transformation code conformance. A different approach is presented by Baldan et al. [BKS04] where GT specifications are used to model the behavior of code generators, and tests are derived from those specifications. All those works focus on testing code generators rather than on testing MTs, the former task being more comparable to compiler testing than to program testing.

There also exist quite some work on the combination of temporal logic and GT, mostly with
a focus on analyzing transformation properties. The work of Baresi et al. [BRRS08] is one example where LTL is used, but there exist several other texts on that topic.

7 Conclusion

Testing model transformations is vital for their future success in practice. Coverage concepts are an important aspect of a full-fledged testing process, but there is no silver bullet to this problem as the number of approaches show.

We introduced a new coverage concept for gray box testing of GT and esp. graph pattern matching, which is an error-prone sub-task. Our coverage criteria are based on the operational semantics defined by search plans (and traces), and we motivate its usefulness with first experimental results and a comparison of its properties and performance to that of statement coverage.

Future work comprises a comparison with more elaborated code-based coverage criteria (like branch or path coverage), and an extension of the approach to treat optional, set and negative nodes in patterns. On the implementation side remains the task of developing an integrated testing framework including the new coverage concept plus other functionality like test generation and oracle functions. On theory side, other/additional temporal constraints are conceivable.

Bibliography


Gray Box Coverage for Testing Graph PM


Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

Adding Rule-Based Model Transformation
to Modelling Languages in MetaEdit+

†Simon Van Mierlo and †‡Hans Vangheluwe

12 pages
Adding Rule-Based Model Transformation
to Modelling Languages in MetaEdit+

†Simon Van Mierlo and †,‡Hans Vangheluwe

†University of Antwerp, Belgium
‡McGill University, Montréal, Canada
simon.vanmierlo@student.ua.ac.be, hv@cs.mcgill.ca

Abstract: MetaEdit+ is a commercial tool by MetaCase for creating domain-specific, syntax-directed visual modelling environments. MetaEdit+ synthesizes such environments from user-provided metamodels and contains a Generator Editor for code/report generation. An API is provided to allow external manipulation of models through SOAP. Currently, the MetaEdit+ tool does not natively support rule-based model-to-model transformation. Such transformations are useful as they allow domain experts to intuitively (using domain-specific notations) model either operational semantics (a simulator) or denotational semantics (through model-to-model transformation onto a model in a known formalism) of a modelling language. We will demonstrate how to add rule-based operational semantics to modelling languages in MetaEdit+. In our approach, transformation rules are visually created in MetaEdit+. The rule editor is synthesized using modified versions of the original language’s metamodel. This modification is performed in a structured fashion using a process called RAMification. Both the model and the rules are exported from MetaEdit+ to Python code. This code is combined with Py-T-Core, our library of transformation language primitives, to apply the rules on the model. Our demonstration has a client-server architecture, with the MetaEdit+ visual modelling environment as the client and the transformation engine as the server. After each transformation step, in-place changes to the model are propagated to MetaEdit+ for visualization using the SOAP API. A simple (manufacturing) Production System modelling language is used as an example.

Keywords: Model-Driven Engineering, Modelling Languages, MetaEdit+, Rule-Based Model Transformation

1 Introduction

Model-to-model transformations are an essential part of the Model Driven Engineering (MDE) approach. They allow the modeller to, for example, define semantics for a modelling language, either by simulating the model (operational semantics) or by mapping onto a known formalism (denotational semantics). These transformations can be defined in a number of ways. One of the most common and intuitive approaches is to use transformation rules. In ATL [4], the modeller textually defines transformation rules. This approach closely resembles programming, and may not be desirable for some purposes. The ability to use domain-specific notations in visual
modelling languages is one of their main advantages, and this advantage is partly lost when using generic specifications of transformations. Hence, using domain-specific notations (concrete syntax) of a visual modelling language to create models of transformations (and in particular, of transformation rules) lowers the threshold to the effective use of transformation. In addition to specifying individual transformation rules, the order in which the rules are tried may also be specified using a scheduling language. In Fujaba [7] and AToM3 [1], amongst others, visual modelling of transformations is supported.

In this paper, we show how to add the ability to visually model and execute model transformations for modelling languages created in MetaEdit+\(^1\). The particular example we will focus on is the description of operational semantics of modelling languages. Operational semantics are a concrete application of inplace transformations which are used to give an operational meaning to models created in the modelling language. MetaEdit+ currently does not natively provide support for model-to-model transformation. It does provide an API for externally manipulating models through SOAP, and a generator editor for code/text generation. We will use these two facilities to (1) enable the modelling of transformation rules inside MetaEdit+, to (2) export a rule-based model transformation model constructed in MetaEdit+ to Python and to (3) subsequently execute it, using our library of graph transformation primitives T-Core [8]. After executing the graph transformation, the results of rewriting the model (which is in essence a typed, attributed graph), are propagated back to MetaEdit+ for visual feedback. A similar method was used in constructing the tool booggie [2] for rule-based design-space exploration. There, a visual environment was constructed around the efficient graph rewriting kernel GrGen.NET [3]. Our demonstrates the practical applicability of RAMification [5] and is structured in a way which makes it easy to reproduce the results in MetaEdit+ or any other tool that provides similar functionality.

The paper is structured as follows. Section 2 explains the running example of the paper, which is a domain-specific modelling language for Production Systems we would like to build a simulator for. Section 3 explains the process of RAMification and how it will be used throughout the paper. Section 4 explains how the rule editor in MetaEdit+ is constructed using RAMification, starting from the original metamodel of a (domain-specific) language. Section 5 presents the architecture of our solution. Section 6 takes a closer look at the execution flow when a rule is executed and Section 7 concludes and suggests future work.

2 Running Example

To demonstrate the approach used in this paper we will define a domain-specific language which is easy to understand but nevertheless non-trivial. The language we create is used to model production systems of Armoured Personnel Carriers (APCs). The APCs are constructed using different parts which are generated by machines at the start of the production line. There are five different kinds of parts: wheels, bodies, tracks, water cannons and machine guns. Different amounts of these parts are needed to assemble two types of APC: war APCs and riot APCs.

\(^1\) http://www.metacase.com/
There are four types of machines: generators, assemblers, quality control machines and a sink. A machine has to be operated by an operator in order to perform its task. An operator can only operate one machine at any point in time. Machines are connected to each other by conveyor belts. A generator has one outgoing conveyor belt on which generated parts are placed. Assemblers have an incoming conveyor belt which supplies parts and an outgoing conveyor belt on which finished products are placed. Quality control machines have an incoming conveyor belt which supplies finished products that can either be broken or functioning correctly and two outgoing conveyor belts which are used to distinguish between these two types. Repair machines have one incoming conveyor belt, which supplies broken finished products and one outgoing conveyor belt, where repaired products are placed. Sinks have one incoming conveyor belt which supplies finished products. Conveyor belts can have multiple incoming connections (either from conveyor belts or from machines) and one outgoing connection (either to a conveyor belt or to a machine).

This explains, informally, the syntax and static semantics (well-formedness rules) of the example language. These rules can be formalized in a user-defined metamodel in MetaEdit+ and models can be visually created in the synthetized environment.

Once models can be created, we typically want to give them meaning. In this case, the meaning will allow for simulation of the dynamics of a model. A set of semantic rules will be created for our modelling language, defining how parts are generated, how they move from one conveyor belt to the other, which parts are taken off from which conveyor belt and at what time to create APCs. Constraints could be added to the language, enforcing that an operator is present at a machine in order for it to work. Using these semantic rules, simulation experiments can be created. Running these experiments allow a modeller to discover interesting facts about particular production system models. Subsequently, performance metrics such as throughput of model variants can be evaluated to design an optimal production system.

The creation of this simulator for our production system modelling language is done by defining a sequence of endogenous transformations that describe how the state of a running production system gets updated as time progresses. This simulator corresponds to the operational semantics of our language. This is different from denotational semantics, where a model in one language is mapped by an exogenous model-to-model transformation onto a formalism with known semantics (such as Petri Nets or code). The techniques we describe in this paper can also be used to define denotational semantics.

Defining our production system simulator can be done in a number of ways. An external simulator could be written in a general-purpose programming language and production system models are exported by a MetaEdit+ exporter to be simulated by that program. This is, however, not in line with the MDE approach, where as much as possible should be modelled explicitly, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). In the next section, we take a look at the possibilities to explicitly model transformations using a systematic approach.
3 RAMification

In [5], Kühne et al. advocate the explicit modelling of abstract and concrete syntax of transformation languages. They state that the advantages of metamodelling in general apply to the modelling of transformation language in particular as well: (1) the specification is not hidden in the code of a tool, making it easier to understand and correct, (2) one can reason about the specifications and the instance models they describe, (3) one may synthesise modelling environments from the specification and (4) this makes it easy for users to alter the specification instead of requiring a new tool release. That is why Kühne et al. explore the possibility of explicitly modelling (visual) rule-based model transformation languages for (visual) modelling languages described by a metamodel. The rules consist of a Left-Hand Side (transformation pre-condition) pattern (LHS, describing the part of the model that should be matched for the rule to be applied), zero or more Negative Application (transformation pre-)Conditions (NACs, specifying the patterns that, when found, should stop the rule from being applied), and a Right-Hand Side (post-condition) pattern (RHS, specifying how the matched part of the model should be rewritten).

The patterns that can appear in the LHS, RHS and NACs are, of course, very similar to the models we can create in the original modelling language. It is therefore logical to try to reuse the metamodel that defines the original language for the pattern specification language, instead of creating one from scratch. Above all, starting from the (domain-specific) modelling language’s metamodel allows for a highly specific transformation language which only permits transformation rules with patterns specific to the modelling language, including language-specific (visual) pattern notations.

We cannot simply copy the metamodels and use them for specifying the patterns of a rule. Firstly, the patterns that appear in rules are not necessarily well-formed models in the original modelling language. For instance, for the Production System language it may be useful to be able to specify a pattern which contains a conveyor belt that has no outgoing connections. In the original language this is not a well-formed model as each conveyor belt should have an outgoing connection to either another conveyor belt or to a machine. In order for the pattern specification language to be useful, these well-formedness rules should be relaxed. Secondly, a number of elements have to be added to the metamodel. It should be possible, for example, to identify model elements across the LHS, RHS and NACs. This is typically done by augmenting the metamodel, adding labels to entities in the LHS, RHS and NAC patterns. Thirdly, the data type of model element properties should be modified as to allow the definition of constraints on properties as well as actions to compute the new value of a property. These are the three main concepts of RAMification: Relaxation, Augmentation and Modification. The authors of the paper describe a semi-automatic process which a developer can follow to create a customized pattern language with minimal effort, starting from the original metamodel of the language.

In the next section, we explore how RAMification is used to create a transformation language specific to our example language. We do this specifically in the commercial tool MetaEdit+ and thus show the applicability of RAMification in practice, beyond its demonstration in the research tool AToM3.
4 The Rule Editor

A rule editor is an interactive (in our case visual) environment for creating model transformation rules. This is the first essential part of our implementation we will explain. The model transformation system we want to construct consist of a number of rules, which can be applied to a model. To make the rule-modelling environment as domain-expert-friendly as possible, these rules should re-use the domain-specific visual notation of the elements to be transformed. Rules consist of three parts: exactly one left hand side (LHS), exactly one right hand side (RHS) and zero or more negative application conditions (NACs). The LHS holds a pattern to indicate which part of the model is to be matched. If the rule is tried and a match is found for the LHS, the transformation engine will also try to find a match for the NACs. If a match for one of the NACs is found, the rule will not be executed. If no NAC match can be found, the rule will rewrite the model by replacing the elements found by matching the LHS by the the corresponding elements described in the RHS. In general, a transformation rule can transform elements present in the LHS of the rule to elements of arbitrary modelling languages. These are called exogenous transformations. In this paper, we restrict ourselves to the modelling of operational semantics which merely updates the state of a model. Hence, only elements of the modelling language we are creating semantics for will appear in rules. The transformations used are thus endogenous and in-place. Note that our technique also works for exogenous transformations.

The rule editor makes use of the decomposition capabilities of MetaEdit+. An object of one metamodel can, in MetaEdit+, be decomposed into a graph conforming to another metamodel. This enables us to create a layered structure for graph transformation models. Figure 1 depicts a model of a graph transformation language. We will focus on the particular case of (layered) graph transformations (or, if they are used to define languages, grammars). A transformation has a name and consists of a number of rules. The rule objects decompose into graphs conforming to the rule metamodel, which consists of the three elements mentioned above: one LHS, one RHS and zero or more NACs. A rule also has a name and a precedence, which is a positive integer. The precedence defines layers in the transformation. The transformation will, while it is executing, choose a rule at random from the currently executing layer. Once none of the rules in the current layer can be executed, the execution of rules proceeds to the next layer. As we wish to use model transformation for simulation (and hence is in principle non-terminating), our semantics loops back to the first layer once no more rules can be fired in the last layer. The LHS and NAC objects of a rule decompose into graphs conforming to the pre-condition pattern metamodel. The right hand side object of a rule decomposes into a graph conforming to the post-condition pattern metamodel. We create these metamodels starting from the original metamodel and apply RAMification on them. In particular, these are the steps we took to create the modified versions of the metamodel:

1. First, make two copies of the original metamodel: one for the pre-condition pattern language (NAC and LHS) and one for the post-condition pattern language (RHS). When we refer to “the metamodel” in the next steps, we mean one of these copies and not the original metamodel.

2. Relax the constraints on the metamodel’s well-formedness. A rule often only matches a
Adding Rule-Based Model Transformation to MetaEdit+

Figure 1: Structure of a transformation. Adapted from [5].

part of a model and this may not be a well-formed model conforming to the original metamodel. It is also possible for abstract superclasses to appear in rules, which is impossible in the original modelling language. For these abstract classes, a default visual concrete syntax is created which enables the modeller to create them in patterns.

3. Append the suffix _LHS (pre-condition pattern) or _RHS (post-condition pattern) to the class names of the objects and relationships.

4. Add a property called GG_Label of type “Number” to each object and relationship. This property is used by the graph matcher to identify nodes across the different parts of a rule.

5. Append the suffix _LHS (pre-condition pattern) or _RHS (post-condition pattern) to each property of an object or a relationship and change its datatype to “String”. The properties now define a condition (pre-condition) or an action (postcondition pattern) instead of an actual value. These strings are, in this case, Python executable code that has to evaluate to a Boolean value in case of a condition, or to the new value of the property in case of an action.

6. Add a property called “constraint” (pre-condition pattern) or “action” (post-condition pattern) to the metamodel. These represent, respectively, the condition that has to be satisfied before a rule can be executed and the action that has to be taken after the rule has executed.

In order to make simulation possible, the original metamodel has to be modified as well. As the layered architecture of the transformation gives priority to layers with a lower precedence value, mechanisms to ensure fairness have to be implemented. This is achieved by adding properties to
objects that are modified by rules. These properties can be checked in the NAC(s) or LHS of a rule: only when a particular value is found can the rule be executed. To disable the rule, the RHS sets the property to anything else than that value. In our example transformation, for instance, we have added a “moved” property to the “Operator” object which has to have the value 0 in order for the rule “MoveOperator” (which moves an operator from one machine to another one) to be applied. The RHS sets the value of this property to 1 which effectively disables the rule, preventing the rule from executing continuously. The top layer of our example transformation consists of rules that set these properties back to their initial values so all rules become enabled again. Thus, each pass through all the rules only considers each match for each rule once and fairness is achieved.

Note that while in our prototype tool AToMPM [6], we perform RAMification fully automatically, the RAMification described in this paper was done manually inside MetaEdit+.

5 Architecture

In this section, additional elements present in MetaEdit+ or implemented in Python that are important to our solution will be discussed. In Figure 2 an overview of the architecture of our solution is shown. This is a client-server architecture: the transformation engine acts as the server, MetaEdit+ as the client.
5.1 Python: Abstract Syntax Graph (ASG)

An abstract representation of MetaEdit+ models was created in Python. This component has two functions: it provides a data structure to export models to using the MetaEdit+ exporters, and it acts as an abstraction layer for the SOAP API. All methods defined on this structure make use of the SOAP API to reflect changes visually in the MetaEdit+ model. These classes are as generic as possible. It is therefore possible to export any type of MetaEdit+ model to this Python structure.

5.2 MetaEdit+: API and Generators

The SOAP API of MetaEdit+ is heavily used in our solution. It provides methods to query and update models, which are used by the ASG component in Python.

The generator editor facility of MetaEdit+ was used to create two types of generator: one for models, and one for rules. As we saw in Section 4, a rule consists of exactly one LHS, zero or more NACs and exactly one RHS. These components of a rule are, like models, mapped to the ASG structure in Python.

5.3 T-Core: Graph Rewriting

T-Core is a library of graph transformation primitives [8]. It is used in conjunction with a scheduling language, which in our case is Python. We only need a small subset of T-Core: the ARule (Atomic Rule), which chooses one match of the set of all matches (matching the LHS, considering the NACs) and transforms the LHS to the RHS. Before T-Core can be used, the following challenges have to be dealt with.

T-Core has its own data structures for graphs and rules. A compiler was built to compile the ASG representation into a T-Core graph (for models) or a T-Core rule (for rules).

To compile an ASG of a MetaEdit+ model, the compiler iterates over all nodes in the ASG twice. During the first pass, it adds all nodes to the graph. This includes both object nodes and relationship nodes. When adding a node, it copies all properties of the source node to the target T-Core node and adds an attribute to the T-Core node which will be used to identify it in the source ASG. This attribute will be used when T-Core has executed a rule on its graph representation, as the changes have to be propagated back to the ASG (see Section 6). It should be ignored by T-Core in the matching phase as it is not a property of the corresponding object in the model.

T-Core provides a mechanism to achieve this by naming the property in a particular way. During the second pass, edges are added from relationship nodes to their source and target nodes. These are not present in the source graph, but are needed by T-Core.

Compiling a rule is almost identical. First, a T-Core rule object is instantiated. Then, the LHS, RHS and NACs are compiled as outlined above (as they are represented by an ASG as well, since they are a special kind of model written in the modelling language) and added to the rule. However, an extra step has to be taken for the attributes of these nodes. In the LHS, RHS and NACs, T-Core expects the properties to be functions. Properties in the LHS and NACs have to return a Boolean value, properties in the RHS have to return a value which corresponds to the new value of that attribute. The strings that are given by the modeller for these properties are wrapped
in functions that evaluate the string as Python code and return the result of this execution. The pattern condition for the LHS as well as the pattern action for the RHS are wrapped similarly. As both the model and rules are now represented as T-Core graphs, the rules can be executed.

6 Executing Rules

We now have all elements needed to create a set of rules in MetaEdit+ and execute them on a model. This section explains how a rule is applied on a model. We start by creating a rule in MetaEdit+, then export it to Python. There, it will be compiled to T-Core together with an exported model. T-Core will rewrite its graph and report back the changes, which will be used to modify the ASG accordingly.

6.1 Creating The Rule

We will consider the moving of an operator from one machine to another as an example rule. This rule is one of the rules that define the semantics of our example production system language. It contains all of the functionality we want to demonstrate but is still basic enough to be able to explain the general principles involved.

Figure 3 depicts the rule as it appears in the generated MetaEdit+ editor. An operator in our language can be connected to either an assembler, a quality control machine or a repair machine. All three of these machines inherit from the processor abstract superclass. The LHS of the rule defines what should be matched: two processors, one of which the operator is connected to. A condition for the “moved” property, states that it should be equal to 0. As we do not want two operators connected to the same machine, we also define a NAC. In the NAC, the processor we want to move to (with \( \text{GG\textunderscore Label} = 4 \)) has an operator connected to it. By defining this NAC, we make sure that whenever the rule is executed no operator is connected to this processor. The RHS defines what the matched subgraph of the LHS should look like after executing the rule. Here, the relationship between the operator and the original processor has been removed, while a new one is created between the operator and the new processor. The “moved” property of the operator is set to 1, which ensures this rule is only executed once until it is reset back to 0.

6.2 Compiling and Executing The Rule

As explained in Section 5.3, the model and the three parts of the rule are compiled to T-Core structures. It is important to point out that subtype matching is used. Without subtype matching, the moving of an operator would have to be split into several rules, to include every possible combination of processor classes. This would lead to an explosion in the number of rules. T-Core supports subtype matching: a list of subtypes for each type can be passed to T-Core. To execute the rule, the compiled rule (which is an ARule object) is given the T-Core representation of the model. T-Core will try to match the LHS and then choose one of the matches at random in case there is more than one. This randomness is controlled through the seeding of the internal random number generator of T-Core which ensures identical results (for experiment repeatability) when the transformations are run multiple times. Then, it will perform the necessary operations as
Figure 3: Example rule: operator moving from one processor to another. From left to right: NAC, LHS and RHS.
defined by the RHS of the rule on this match. Internally, it changes its own representation of
the model, and reports back a list of changes (an “edit script”). These changes include, but are
not limited to, the changing of attributes, the creation or removal of nodes and the creation or
removal of edges.

6.3 Modifying The ASG

The list of changes made to the T-Core graph is subsequently used to modify the original ASG
of the model. In the compilation process of the ASG, we made sure the nodes in the T-Core
graph can be linked back to their original ASG nodes. This makes it possible to perform exactly
the same changes to the ASG as were made to the T-Core graph which ensures both graphs
represent the same model. In addition, the operations that change the ASG propagate these
changes through the SOAP API to the original model in MetaEdit+, which results in visual
feedback.

7 Conclusion and Future Work

In this paper, we have shown how to add operational semantics to languages created in MetaEdit+.
First, a rule editor was created in MetaEdit+ which allows us to visually create transformation
rules which are combined in a (layered graph) transformation model. The transformation model
was then exported to Python, where it can be executed on an exported MetaEdit+ model us-
ing T-Core as a backend. The execution of a graph transformation results in a series of graph
rewritings which visually propagate to the original MetaEdit+ model by using the SOAP API
of MetaEdit+.

Future work is outlined below.

- **Denotational Semantics of MetaEdit+ Languages**: In this paper, we have added op-
erational semantics to a (production system) modelling language. Further research will
investigate adding denotational semantics to languages. The difference with the work de-
scribed in this paper is that multiple metamodels have to be combined. In the rule editor,
it should be possible to use concepts of the source language as well as the target language.

- **Automatic RAMification of Meta-models in MetaEdit+**: In MetaEdit+, metamod-
els of languages can be exported to and imported from XML. It should therefore be possi-
bile to automate the RAMification process of metamodels.

- **Other Environments**: The technique outlined in this paper could be used with other front-
and backends. An example of this would be to add model-to-model transformations to the
Eclipse Graphical Modelling Project\(^2\), using for example the very efficient graph rewriting
kernel GrGen.NET as backend.

\(^2\) http://www.eclipse.org/modeling/gmp/
Bibliography


ModGraph - Generating Executable EMF Models

Sabine Winetzhammer

sabine.winetzhammer@uni-bayreuth.de
Lehrstuhl für angewandte Informatik 1 - Software Engineering
Universität Bayreuth, Germany

Abstract: Model driven software engineering aims at creating high level executable models which may be interpreted or compiled. For efficient execution of operations on model instances code generators play an important role. A well-established tool for structural modeling and code generation is the Eclipse Modeling Framework (EMF). We extended EMF by behavior modeling within ModGraph, a tool to model behavior by graph transformation rules. Each rule corresponds to an operation modeled in the Ecore class diagram. This paper focuses on ModGraph’s code generator. Therefore I describe the matching of graph transformation rules as well as the exact translation of a rule and its seamless injection into the existing EMF Java code. A running example of a simple calendar application complements the explanation.

Keywords: EMF, Code Generation, ModGraph, Model Driven Software Development, Graph Transformation

1 Introduction

Over the last decades programming languages evolved from assembler code into high level object oriented programming languages. Executable models are the next step of abstraction. These highly abstract models may be interpreted or compiled. In model driven software engineering these models are often only partially executable like it is the case within the Eclipse Modeling Framework [SBPM09] (EMF). It provides code generation from structural Ecore class diagrams. The EMF code generator creates code for classes, attributes, references and elementary operations. Elementary operations are, for example, creating links (instances of references) or setting attribute values. For user modeled operations EMF creates only empty methods, which have to be implemented by a programmer. Here ModGraph ([BWW12] and [BWW11]) steps in: ModGraph supports behavioral modeling of complex operations based upon graph transformation. An EMF model instance is considered as a graph. The graph’s nodes and edges correspond to instances of classes and references modeled in the Ecore class diagram. Each rule corresponds to one user modeled operation in the Ecore class diagram. A code generator creates executable code for this rule. Hence the modeler may use graph transformations whenever they are helpful to describe complex operations on EMF model instances in a declarative and graphical way. For everything else, in particular for the control flow, the modeler may resort to Java.

Here I present details on ModGraph’s code generation, whereas the focus in [BWW11] and [BWW12] was set on the presentation and the meta model of the graphical tool. Section 2 provides an overview on ModGraph’s modeling capabilities, Section 3 introduces the running
Figure 1: ModGraph perspective

example. Section 4 explains the code generator. It is illustrated with code snippets generated from the running example. Section 5 discusses related work, Section 6 concludes the paper.

2 Overview

Within ModGraph the body of a user-defined operation can be modeled in an Ecore class diagram. A detailed description of the meta model is given in [BWW12]. The GMF-based graphical environment in use is shown in Figure 1. Next to the Eclipse package explorer on the left the main window (marked as 1) shows the graph transformation rule. A comment can be attached to each rule using the comment view (marked as 2). A dashboard (marked as 3) provides a structured view on the process toward the executable model. All rules shown here are snapshots of the main window.

The rule itself must contain a graph pattern that specifies the subgraph to be searched inside the Ecore model instance as well as the changes to be performed. To specify these changes different kinds of nodes and edges representing instances of classes and references in the Ecore model are available.

The current object corresponds to the Java this object (see Figure 1: this). ModGraph distinguishes between unbound and bound nodes. Unbound nodes represent objects. Bound
nodes represent the non-primitive parameters of the operation. Primitive parameters are used to model constraints on objects or are employed as attribute values. If the current object is absent, at least one bound node is needed. Multi-valued parameters are shown as multi-parameter nodes, single-valued parameters as parameter nodes (see Figure 3), both with rounded corners. Any rule may contain an arbitrary number of unbound nodes represented by objects (see Figure 4) and multi-objects (see Figure 5). They need to be searched by pattern matching in the model instance. Unbound nodes must reference a class in the Ecore model. Bound nodes may reference one.

All nodes in the graph pattern – except for the current object – provide a status which is create (++) or delete (−−) or preserve (no marker). The status indicates the changes to be performed on the model instance. Nodes may be marked as return parameter (<<out>> ) or as optional (<<optional>> ).

In general nodes may be connected by edges: links and paths. The connection of two multi-valued nodes constitutes an exception, due an ambiguous assignment. If a node is optional, the related edges are assumed to be optional, too. Links represent an instance of a reference in the Ecore class diagram and provide a status. Multi-valued links may be ordered. Paths represent derived references. They are marked with a path expression written in OCL (see Figure 5) or Java.

Except for created ones, any node can be constrained by an OCL or attribute condition. On created or preserved nodes attribute values may be changed and operations may be called. These
operations are executed after the successful application of the rule simulating a sequence.

The rule’s graph pattern can be constrained using pre- and postconditions as well as a negative application condition (see Figure 3). Pre- and postconditions are formulated in OCL. A negative application condition is represented as a graph showing a pattern that must not occur, using only single-valued nodes. A special single-valued node is a NACBoundNode (see Figure 4) which references an unbound node in the graph pattern. NACBoundNodes are bound because the unbound nodes in the graph pattern are bound before the negative application condition is checked. Nodes in a negative application condition can be analogously constrained and connected by links (without status) and paths.

Each rule is validated in a threefold way. A rule must conform to the underlying ModGraph metamodel. The editor performs live validation concerning the user model during editing. The remaining errors, for example, verifying the reachability of nodes, are caught by a batch validation.

3 Running Example

A running example, an event calendar, is used to show the rules’ syntax as well as the generated code. Figure 2 shows the event calendar Ecore model. Each calendar system supports multiple users. Each user may own an arbitrary number of calendars that contain events. Each event may belong to a category that belongs to the event calendar system. There may be attendances to an event. Users are attendees. An event has a defined start and end date. It may reoccur. In the following I explain the implementations of the methods marked in Figure 2.

The implementation of addAttendee is shown in Figure 3. At the top of the rule the pre-condition is shown. The OCL expression assures the existence of the parameter newAttendee.
On the left the graph pattern is shown. It contains the non-primitive parameter `newAttendee` at the top and the current object at the bottom. In the middle an attendance `newAttendance` is to be created, and therefore shown as an unbound creation object. Its attendance status is set to pending. The new attendance is connected to the parameter and the current object by using two creation links. On the right a negative application condition shows what must not occur in the model instance in order to create a new attendance: the user, given by the parameter `newAttendee` must not be the owner of the calendar the event belongs to.

The second rule `deleteCategory` (Figure 4) consists of a graph pattern and an NAC. The pattern specifies that a category in the event calendar system is deleted if the name of the category equals the value of the method’s primitive parameter `categoryName`. This may only happen if the negative application condition does not hold. Therefore the category to be deleted, shown as a NACboundNode, may not contain any event anymore.

The last rule `getAllEvents` (Figure 5) shows just a graph pattern. The current object inside the pattern is an instance of the `User` class. A path connects it to a multi-object of type `Event`. The path specifies that all events of all calendars the user owns are collected.
4 Generating an Executable Model

This chapter provides an overview about the ModGraph code generator. The generator follows a compiler solution and is written using Xpand templates. For illustration code generated from the example rules as well as an abstract example is used.

4.1 Preliminaries for the Generation of Pattern Matching Code

Before code generation starts the rule has to be parsed. A recursive, heuristic greedy algorithm is used to transform any graph given in the rule into a forest of spanning trees following these steps:

1. Use each bound node in the pattern as the root of a tree inside the forest.
2. Consider all outgoing edges of the forest’s nodes. Choose the one instantiating the reference with minimum multiplicity. (Paths are considered like multiplicity many references. If two links instantiating references of the same multiplicity exist, one is chosen randomly.)
3. Check, if the node the link targets is mandatory and not contained in any tree yet. If true, add it as a child into the tree that contains the link’s source. (Paths are considered like multiplicity many references. If two links instantiating references of the same multiplicity exist, one is chosen randomly.)
4. Repeat steps (2) and (3) until there are no more mandatory nodes.
5. Repeat steps (2) and (3) – without the mandatory check – for all optional nodes. Please note that this prohibits searching a mandatory object from an optional one.

The forest specifies the reachability of nodes inside the graph pattern. It acts as a search plan. If any mandatory node cannot be inserted into a tree it cannot be bound and the matching fails.

To visualize these steps a schema of the search tree constructed successfully from a schematic graph pattern is shown in figure 6. The pattern has also been transformed into a schema assuming that all nodes are either to preserve or to delete. Some multiplicities are added for better understanding. The only bound node bound1 is used as a root (step(1)). unbound1 and unbound2 are added as children (step (2) & (3)). The node unbound3 in figure 6 may be searched from bound1 and unbound1. In favor of the lower multiplicity, it is added as unbound1’s child in the tree (step (4), repetition of (2) and (3)). The optional node is added as a leaf, since only optional nodes may be searched from it and the rule contains no more such nodes (step (5)).

Concretely: For the rule addAttendee the situation is shown in figure 7. For the graph pattern two roots are created, without children because the pattern only contains bound nodes and one node to be created. The NAC contains an unbound object. It may be searched either from the parameter of type user or the current object of type Event. The Ecore model specifies that an
event belongs to exactly one calendar, but a user may own several calendars. So the calendar is bound through the current object of type `Event`.

This forest is used to create the utility methods shown in the next section.

4.2 Code Generation

ModGraph injects the generated code into the EMF generated one, consequently the EMF code has to be generated independently in the first step.

The code generation itself starts by modifying the EMF generated interface. ModGraph methods may throw exceptions of type `GTFailure`, for example, if the postcondition is not matched. Through the generated interface has to be supplemented by an exception.

Next, the code generated for the implementation class is injected into the empty EMF generated method body. This is done by generating it separately and injecting it using the abstract syntax tree.

The code generated for the method `addAttendee` is shown in Listing 1 and structured in the following way: (1) The comment, written in the comment field, marked with 2 in Figure 1, is transformed into a JavaDoc-Comment as shown at Listing 1, lines 1-3. (2) As described for the interfaces the method’s header is extended by an exception (Listing 1, lines 4–5). (3) The precondition is checked using an `OCLHelper` class based upon Eclipse OCL (Listing 1, lines 6–8). (4) A utility class is instantiated and called to perform the matching and to check all conditions (Listing 1, lines 9–10). (5) All attribute values are calculated before anything is changed (listing 1, lines 11–12). (6) New objects are created, existing ones are changed or deleted. In listing 1, lines 13–15, the creation of the new attendance object is performed. The deletion of an object – not performed on the sample rule – is delegated to the utility class, see Listing 2. Its delete method is overloaded due to the possible deletion of a multi-object or multi-parameter. (7) Links are created or deleted. Listing 1, lines 16–17, shows the creation of the two links in figure 3.


```java
/** <!-- begin-user-doc --> 
 * <!-- end-user-doc --> */
public void addAttendee(User newAttendee)
```
OCLHandler.evaluatePrecondition("newAttendee<>null", this,
    new String[] { "newAttendee" },
    new Object[] { newAttendee });
GTUtil4addAttendee util = new GTUtil4addAttendee();
util.match(newAttendee, this);
final AttendanceStatus newAttendanceStatusValue =
    AttendanceStatus.PENDING;
Attendance newAttendance = CalendarPackage
    .CalendarPackageFactory.eINSTANCE.createAttendance();
newAttendance.setStatus(newAttendanceStatusValue);
this.getAttendances().add(newAttendance);
newAttendance.setAttendee(newAttendee);

Listing 2: Excerpt of the implementation and utility of Method EventCalendarSystem:
deleteCategory(categoryName:String)

    public boolean deleteCategory(String categoryName)
        throws Throwable, GTFailureImpl {
            ...
            util.delete(categoryToDelete);
            ...
        }

    public class GTMatchingUtil {
        public void delete(EObj ect eObject) {
            EcoreUtil.delete(eObject, true);
        }
        public void delete(EList<?> objectList) {
            checkResource();
            for (Object o : objectList) {
                if (o instanceof EObject)
                    EcoreUtil.delete((EObject) o, true);
            }
        }
    }

The utility class performs the matching on the instance graph. Referring to the schematic tree in Figure 6, the utility class matches this tree against an existing.ecore model instance as shown in Figure 8. The result of the matching is also shown in Figure 8 using green connectors. The mandatory unbound objects are matched as you would expect. unbound1 is matched to the object possible unbound1 (2) with an unbound3 candidate reachable which fits here. If unbound1 is matched to the other possible candidate and then it is recognized that the matching of unbound3 is impossible, the match is reverted. unbound2 can be fitted on two nodes in the Ecore model instance. The result shown here is reached assuming that possible unbound2 (1) does not fit the matching criteria. It is no problem that the optional node is not matched.
Concretely: The utility class extends the ModGraph built in class GTMatchingUtil. The ModGraph code generator generates a coordinating match method, following the logic described above. This method carries the same parameters as the operation itself and one for the method’s current object.

For the Method `addAttendee` all parameters are put into a global map in order to be accessible from anywhere (see Listing 3, lines 6–7). Next the bound nodes as well as the current object in the graph pattern are checked concerning their constraints and the links between them. Then a match of the unbound nodes in the graph pattern with the Ecore model instance is performed. Please note that both these steps need not to be performed for the sample rule. The graph in the NAC is matched analogously.

The matching methods for the running example are shown in Listing 3 for the NAC matching. The method `nacMatched` (Listing 3, lines 9–12) coordinates the matching of the NAC’s elements. In the example code the object `c` of type `Calendar` is matched starting at the current object. If the match of the NAC succeeds, an exception is thrown as shown in lines 13–22. The matching itself is put into another method shown in lines 23–33. Here the constraints given in the rule are checked.


```java
public class GTUtil4addAttendee extends GTMatchingUtil {

    public void match(User newAttendee, Event event) throws GTFailureImpl {
        map.put("newAttendee", newAttendee);
        map.put("event", event);
        nacMatched(newAttendee, event);
    }

    private void nacMatched(User newAttendee, Event event) throws GTFailureImpl {
```

Figure 8: Result of the matching on an existing example Ecore model instance (schema)
matchUnboundNodesInNAC("c", event);

private void matchUnboundNodesInNAC(String searchedNodeName, EObject boundPredecessor) throws GTFailureImpl {
    if (searchedNodeName.equals("c")) {
        Calendar c = getC4AddAttendee((Event) boundPredecessor);
        if (c != null) {
            alreadyConsideredObjects.add(c);
            throw new GTFailureImpl("c found.");
        }
    }
}

public Calendar getC4AddAttendee(Event event) throws GTFailureImpl {
    Calendar aCalendar = (Calendar) event.getCalendar();
    if (!(alreadyConsideredObjects.contains(aCalendar))
        && aCalendar instanceof Calendar
        && ((User) map.get("newAttendee")).getCalendars()
            .contains(aCalendar))
        return (Calendar) aCalendar;
    else
        return null;
}

Matching via a path is performed by the OCL-Handler, as shown in Listing 4. Here the object is found by calculating the OCL-expression and checking the constraints - not shown here - afterward.


public EList<Event> getEvents4GetAllEvents(User user) throws GTFailureImpl {
    List<EObject> allEvents = OCLHandler.evaluatePath(user, "self.calendars.events->asSet()");
    ...  
    return (EList<Event>) events;
    ...  
}

5 Related Work

During the last decades researchers presented several approaches concerning model transformation languages [CH06]. Only a few of them are EMF related, based on graph transformations [EEKR99], generate executable code, perform endogenous model transformations and provide
a graphical environment as ModGraph does. Any tool not satisfying all these constraints is not considered due to space limitations.

EMF Tiger [BEK+06] and its sequel Henshin [ABJ+10], as ModGraph, use Ecore for structural modeling. Tiger lacks some constructs, ModGraph provides, like pre- and postconditions. Concerning code generation, Henshin just provides an interpreter. This means that embedding the rules into applications depends mostly on the capabilities of the interpreter. Tiger compiles each graph transformation rule into one independent class instead of a method. If one wants to use this code in a user-defined operation, some hand-coding into the EMF-generated code is required: instantiate the class and pass the necessary parameters to execute the rule. This leads to higher run times and programming efforts. ModGraph generates code right into the EMF-generated one. It creates a standard Java method which can be used without further modification. ModGraph provides within that an efficient and seamless integration into the generated code.

Fujaba [Zün01] uses class diagrams for structured and story diagrams for behavior modeling. A story diagram represents the implementation of a method modeled in the class diagram. Story-patterns in a story diagram specify graph transformation rules. Fujaba itself was developed outside EMF. An EMF code generator is available, but the modeler has to specify a class diagram in Fujaba first and may not immediately define an Ecore model or reuse an existing Ecore model.

The idea of story diagrams was partially reimplemented in MDELab [GHS09]. MDELab uses Ecore class diagrams for static and story diagrams for behavior modeling. But story diagrams are interpreted, not compiled. This reduces the usability to the capabilities of the interpreter in working with model instances.

eMoflon [ALPS11] may use Ecore for structural modeling and may generate EMF compliant code as well as it extends professional case tools. As Fujaba and MDELab, it uses the concept of modeling behavior and control flow into one diagram, whereas ModGraph strictly separates them, in order to get clearly laid out rules. Concerning code generation eMoflon is uses the Moca Framework, which builds up a complex tree. Furthermore a separate parser is needed to generate code. ModGraph uses simply XPand templates and a simple, straightforward constructed tree for each rule.

Compared with other tools I summarize that ModGraph is the only EMF-based tool providing seamless integration into the EMF generated code and a clear separation of behavior and control flow modeling. The generated code may be used for all kinds of applications which assume generated EMF code, e.g., GMF (for building a graphical editor).

6 Conclusion and Future Work

Here I introduced ModGraph’s code generator. ModGraph is a tool to add behavior modeling to EMF at a high level of abstraction. Therefore graph transformation rules are in use. The generator is able to generate code based upon these rules. It follows a compiler solution and injects the code seamlessly into the EMF generated code. In the process each rule is treated as the implementation of a method. The result is a fully functional executable model.

Current and future work aim at extending the graph transformation rules by a control flow language, as you can see in the Dashboard (figure 1, marked with 3). This language will provide a possibility to model a control flow for the graph transformation rules.
Bibliography


Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

Integration of Triple Graph Grammars and Constraints

Stephan Hildebrandt, Leen Lambers, Basil Becker, Holger Giese

12 pages
Integration of Triple Graph Grammars and Constraints

Stephan Hildebrandt\textsuperscript{1}, Leen Lambers\textsuperscript{1}, Basil Becker\textsuperscript{1}, Holger Giese\textsuperscript{1}

\textsuperscript{1}surname.lastname@hpi.uni-potsdam.de
Hasso Plattner Institute at the University of Potsdam

Abstract: Metamodels are often augmented with additional constraints that must be satisfied by valid instances of these metamodels. Such constraints express complex conditions that cannot be expressed in the metamodel itself. Model transformations have to take such constraints of the source and target metamodels into account. Given a valid source model, which satisfies the source constraints, a model transformation is expected to return a valid target model (\textit{forward validity}). However, in current model transformation definition and tool support, such an integration with source and target constraints including validation mechanisms is often ignored or not satisfactory yet.

In this paper, we describe how the integration with source and target constraints can be achieved for the special case of model transformations defined by Triple Graph Grammars (TGGs). First, we extend the relational model transformation definition for TGGs and integrate it with source and target constraints. Moreover, we describe how forward/backward validity of TGGs with constraints can be automatically checked, either by static analysis using an invariant checker, or by generating and validating metamodel instances. Finally, we describe how to integrate constraints into our TGG-based model transformation implementation and automatic conformance testing framework.

Keywords: Model Transformation, Constraints, Test Case Generation, Invariant Checking

1 Introduction

Model-Driven Engineering puts models and model transformations into the focus of the development process. In practice, models are mostly defined using metamodels, which specify the kinds of elements that can be used in a model of that type. Metamodels are often augmented with constraints, which models must satisfy to be valid instances of a metamodel. Such constraints are complex conditions that cannot be expressed in the metamodel itself. Modeling tools have to consider these constraints in order to correctly work with models. In particular, model transformation tooling, which plays a key role in Model-Driven Engineering, needs to consider constraints as well.

A model transformation is expected to always return a valid target model if it is provided with a valid source model.\textit{Valid} means that the models not only adhere to the structure defined
by their metamodels, but also satisfy the metamodels’ constraints. We call transformations that always produce valid target models for valid source models forward valid transformations. The definition of backward valid is analogous for bidirectional transformation approaches.

However, existing implementations mostly do not consider constraints defined on metamodels at all. Instead, the problem is shifted to the designer of the transformation specification who is expected to define appropriate pre- and postconditions filtering out invalid models, which often resemble constraints already defined in the metamodels. It remains an open question how to ensure consistency between constraints in the metamodel and pre- and postconditions of the transformation specification defined by the designer.

For these reasons, we have extended our existing tools for model transformation [GHL12] and conformance testing of model transformations [HLG +12] based on Triple Graph Grammars [Sch94] to consider metamodel constraints, focusing on design-time checks as well as runtime checks revealing problems with forward or backward validity of the transformation. We describe the integration of TGGs with constraints in Section 3 and present two approaches to check the forward/backward validity of forward/backward transformations derived from a TGG in Section 4, one based on static analysis using an invariant checker, and another one based on counter example generation. Section 5 explains how we extended our conformance testing framework and our model transformation implementation to support constraints. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 Our Triple Graph Grammar Framework

Triple Graph Grammars [Sch94] are an important representative of relational model transformation specifications. To illustrate the following explanations, we will use a model transformation from simple SDL block diagrams to UML class diagrams. The metamodels of both modeling languages are shown in Figure 1. Block Diagrams are a hierarchical structure of Blocks. Blocks can be nested and the topmost blocks are SystemBlocks. In addition, Blocks can contain Pro-
(a) Axiom Rule (BlockDiagram to ClassDiagram)

(b) Rule 1 (SystemBlock to Class)

(c) Rule 2 (Block to Class)

(d) Rule 3 (Process to Class)

**Figure 2:** Example triple graph grammar

**Figure 3:** Triple Graph Grammar tool framework
cesses, which cannot contain any elements. Class Diagrams can contain Classes, Stereotypes, which are attached to a Class, and Associations, which connect two classes.

We have implemented a framework based on TGGs for model transformation and synchronization, as well as for conformance testing of TGG model transformations. It is based on the Eclipse Modeling Framework (EMF)\(^2\) and can be downloaded from our Eclipse update site\(^3\). The framework’s architecture is shown in Figure 3. We repeat the basic principles here and refer to [GHL12, GNH10, HLG\(^+\)12] for more detailed information.

The Source and Target Metamodels, \(S_{TT}\) and \(T_{TT}\), respectively, can be edited using the existing Ecore metamodel editor. TGG Rules can be created and edited using the TGG Editor. The TGG rules of the transformation between Block Diagrams and Class Diagrams are shown in Figure 2. Each rule consists of three domains: A source domain on the left, which contains elements of the Block Diagram, a target domain on the right, which contains elements of the Class Diagram, and a correspondence domain in the middle, which contains the so-called correspondence model. The correspondence model explicitly stores correspondence relationships between source and target model elements. Its metamodel \(C_{TT}\) is shown in Figure 1b. A CorrNode maps arbitrary objects in the source model to objects in the target model. These three models form a so-called triple graph, denoted by \(SCT\), where \(S\) denotes the source, \(C\) the correspondence, and \(T\) the target component. The notation used in Figure 2 combines the left-hand side (LHS) and right-hand side (RHS) of a graph transformation rule. Elements contained in the LHS and RHS are black, elements contained in the RHS only are drawn green and marked with “++”.

Like ordinary graph grammars, Triple Graph Grammars have a start graph, called axiom\(^4\), denoted by \(S_{A}C_{A}T_{A}\). The axiom creates the root nodes of all three models. The remaining TGG rules create the other elements of the models in certain contexts: Rule 1 links a SystemBlock to an existing BlockDiagram and a Class to the corresponding ClassDiagram. Likewise, rules 2 and 3 create Blocks and Processes in existing Blocks and a corresponding pattern of a Class, Association, and Stereotype in the ClassDiagram. More formally, a triple graph grammar (TGG) consists of a set of triple graph rules \(\mathcal{R}\) typed over \(S_{TT}C_{TT}T_{TT}\) (the type graph or metamodel, which results by connecting the source and target metamodels via their correspondences) and a triple start graph \(S_{A}C_{A}T_{A}\), called axiom, also typed over \(S_{TT}C_{TT}T_{TT}\). For more information about the formalization of TGGs, we refer to [GHL12].

As it is, the TGG can be used to create all three models in parallel. The basic principle is depicted in Figure 4, where the nodes represent triple graphs and the arrows represent TGG rule applications. First, the axiom rule is applied once and creates \(S_{A}C_{A}T_{A}\). A randomly selected TGG rule is then applied to create, e.g., \(S_{13}C_{13}T_{13}\). Then, another rule is selected and applied to create, e.g., \(S_{26}C_{26}T_{26}\). With \(S_{i}C_{Ti} \Rightarrow S_{i+1}C_{i+1}T_{i+1}\) we denote a rule application from \(S_{i}C_{Ti}\) to \(S_{i+1}C_{i+1}T_{i+1}\) and we write \(S_{CG}T_{G} \Rightarrow S_{CG}T_{G}\) to denote 0 to a random number of rule applications from \(S_{CG}T_{G}\) to \(S_{CG}T_{G}\). All triple graphs \(SCT\) that can be created accordingly from a TGG belonging to \(\mathcal{L}(tg)\), the TGG language. More formally, given a particular \(tg = (S_{A}C_{A}T_{A}, \mathcal{R})\), typed over \(S_{TT}C_{TT}T_{TT}\), then the triple graph language \(\mathcal{L}(tg)\) consists of all triple graphs \(S_{CG}T_{G}\) such that \(S_{A}C_{A}T_{A} \Rightarrow S_{CG}T_{G}\) via rules in \(\mathcal{R}\). The Test Model Generator

---

\(^2\) http://www.eclipse.org/modeling/emf

\(^3\) http://www.mdlab.de/update-site

\(^4\) In particular, we use a so-called axiom rule, which is applied once to the empty graph and thereby sets correct attribute values creating the concrete axiom.
Figure 4: Step-wise derivation of triple graphs using a TGG with constraints

(cf. Figure 3) uses this principle to generate a Source Model and an Expected Target Model that can be used as test input and test oracle to automatically verify conformance of a TGG with the corresponding TGG implementation. To test the forward transformation, the Source Model \((S_i, C_i, T_i)\) is input to the TGG Implementation under test. The TGG Implementation outputs a Target Model, which is compared to the Expected Target Model \((T_i, S_i, C_i, T_i)\) using EMFCompare. If both models are equal, the TGG Implementation has passed the conformance test.

To actually perform a model transformation using TGGs, operational rules have to be derived for the three different transformation directions supported by TGGs: A forward transformation takes a model of the left domain, a Block Diagram in the example, and produces the other two models. A backward transformation is the opposite, a model of the right domain is transformed to a model of the left domain and a correspondence model. A mapping transformation takes two existing models of both domains and produces only the correspondence model. For each of these directions, separate operational rules have to be derived. In our model transformation tool, Story Diagrams\(^5\) describe the operationalized form of the TGG Rules. Operational rules add the elements of the source model domain of the respective direction to their LHS, e.g., forward rule 1 (cf. Figure 2b) includes \(sb2\) in its LHS. It does not create a new SystemBlock. In this paper, we focus on the forward direction. The backward direction is always analogous.

In practice, the metamodels of source and target language are often augmented with a set of constraints \(C_S\) and \(C_T\), respectively. Our existing framework did not support such constraints. Therefore, we extended the formal definition of TGGs with constraints, added an automatic checking mechanism for forward/backward validity TGGs with constraints based on two tool components (Invariant Checker and Counter Example Generator in Figure 3), and extended our TGG model transformation implementation as well as automatic conformance testing framework with support for constraints. These conceptual extensions and the corresponding new tool components are presented in the next sections.

\(^5\) Story Diagrams are executable models consisting of a combination of UML Activity Diagrams with graph transformation.
3 Integration of TGG Specifications and Constraints

As mentioned before, metamodels are often augmented with constraints. Such constraints can be as simple as multiplicities of references but also complex conditions spanning multiple elements of the metamodel are possible. The Object Constraint Language (OCL) is the commonly used language to define metamodel constraints.

The example metamodels in Figure 1 already contain multiplicities: Trivial examples are the container ends of containment references, e.g., BlockDiagram and ClassDiagram, which always have a lower and upper bound of 1. But also several more complex constraints can be defined. In Class Diagrams, all Stereotypes of a Class must have different values in their text attributes. This is expressed in OCL as follows:

\[(C1) \text{context Class inv UniqueStereotypes:} \]
\[\text{self.stereotypes} \rightarrow \text{forAll}(e1 | \text{self.stereotypes} \rightarrow \text{forAll}(e2 | e1 \neq e2 \text{ or } e1.\text{text} \neq e2.\text{text}))\]

For Block Diagrams, we have the following constraints: First, a SystemBlock may not contain other SystemBlocks or Processes. In OCL, this is expressed like this:

\[(C2) \text{context SystemBlock inv NoSystemBlocksOrProcesses:} \]
\[\text{not self.modelElements} \rightarrow \text{exists}(e | e.oclIsKindOf(SystemBlock) \text{ or } e.oclIsKindOf(Process))\]

Second, each Block in a Block Diagram may either contain other Blocks or Processes, but not both. If a Block contains other Blocks, then there must be at least two contained Blocks. This constraint is expressed in OCL as follows:

\[(C3) \text{context Block inv BlockHierarchyConstraint:} \]
\[\text{self.modelElements} \rightarrow \text{size}(\text{isEmpty}) \text{ or} \]
\[\text{self.modelElements} \rightarrow \text{forAll}(e | e.oclIsKindOf(Process)) \text{ or} \]
\[\left(\text{self.modelElements} \rightarrow \text{forAll}(e | e.oclIsKindOf(Block)) \right) \text{ and } \text{self.modelElements} \rightarrow \text{size}(\geq 2)\]

Triple Graph Grammars as presented in Section 2 do not consider such metamodel constraints yet. Considering metamodel constraints in a TGG essentially means restricting the set of triple graphs \(SCT\) that can be created with the TGG to those that also satisfy the constraints \(C_S\) and \(C_T\) typed over the source and target metamodels \(S_{TT}\) and \(T_{TT}\), respectively.\(^6\) Given a so-called TGG constraint \(C_{tgg} = C_S \land C_T\) for a \(tgg = (S_{TT}C_{TT}T_{TT}, \mathcal{R})\), being typed over \(S_{TT}C_{TT}T_{TT}\), then the triple graph language with constraints \(\mathcal{L}(tgg, C_{tgg})\) consists of all triple graphs \(S_{G}C_{G}T_{G} \models C_{tgg}\) such that \(S_{G}C_{G}T_{G} \triangleright S_{G}C_{G}T_{G}\) via rules in \(\mathcal{R}\). We say that a model is valid w.r.t. a metamodel with constraints, if it is typed over the metamodel and, additionally, satisfies all its constraints.

Figure 4 describes a TGG language with constraints. Every node of the tree is a triple graph \(S_{i}C_{i}T_{i}\) that can be generated by \(tgg\), i.e., \(S_{i}C_{i}T_{i} \in \mathcal{L}(tgg)\). However, not all of them also satisfy the constraints of the source and target metamodels. In Figure 4, the source and target components \(S_{i}\) and \(T_{i}\) that are valid w.r.t. source and target constraints are marked with check marks to symbolize satisfied constraints. In contrast, invalid components are marked with crosses to symbolize violated constraints. Triple graphs, where the source or target component is valid and marked with a check mark, and the target or source component is not valid and marked with a cross, belong to \(\mathcal{L}(tgg, C_S \land \neg C_T)\) or \(\mathcal{L}(tgg, \neg C_T \land C_S)\), respectively. But only those triple graphs, where both components are valid w.r.t source and target constraints, belong to \(\mathcal{L}(tgg, C_{tgg})\).

---

\(^6\) For space reasons, we do not consider correspondence metamodel constraints here, but these can be handled analogously. Moreover, models and constraints typed over \(S_{TT}\) and \(T_{TT}\) are automatically also typed over \(S_{TT}C_{TT}T_{TT}\).
4 Automatic Checking of TGGs with Constraints

Recall that we say that model transformations are forward valid if they produce valid target models from valid source models. We want to automatically check for a tgg with constraints, whether it is forward valid (or backward valid, which is symmetric): Is there any $SCT \in \mathcal{L}(tgg)$, where $S \models \mathcal{C}_S$ and $T \not\models \mathcal{C}_T$? Or, equivalently, is there any $SCT \in \mathcal{L}(tgg, \mathcal{C}_S \land \neg \mathcal{C}_T)$? If not, then we have forward validity. If so, then there exists a counterexample.

We can group target constraints in $\mathcal{C}_T$ w.r.t. forward validation into the following categories:
(1) Target constraints satisfied for each $T$ in $SCT \in \mathcal{L}(tgg)$. We say that these constraints hold by TGG construction. (2) Target constraints satisfied for each $T$ in $SCT \in \mathcal{L}(tgg, \mathcal{C}_S)$. (3) Target constraints not satisfied for each $T$ in $SCT \in \mathcal{L}(tgg, \mathcal{C}_S)$. If every target constraint in $\mathcal{C}_T$ belongs to category (1) or (2), then each forward transformation derived from the tgg is forward valid.

In this section, we describe how the above question can be answered via static analysis (similar to [BLD+11], where we did this for refactorings) using our invariant checker or, dynamically, by generating explicit counterexamples.

Static Analysis The satisfaction of constraints can be invariant with respect to a set of graph transformation rules $R$. A constraint $\mathcal{C}$ is an inductive invariant of $R$ if for all graphs $G$ and for all rules $r \in R$, it holds that $G \models \mathcal{C}$ and $G \Rightarrow G'$ implies $G' \models \mathcal{C}$. We developed a static analysis technique being able to perform invariant checking [BBG+06] for constraints of the following kinds: "A specific pattern (forbidden pattern) should not occur in the model." or "A specific pattern (conditional forbidden pattern) should not occur without some other specific pattern.". Our invariant checker either reports that a constraint is indeed an invariant for a given set of rules, or it automatically computes as symbolic counterexamples all minimal situations indicating why rules might be applied to a constraint-satisfying graph leading to a violating one. Note that some of the counterexamples may represent false negatives because temporarily invalid constraints during the model transformation might be valid in the model transformation result (in case of conditional forbidden patterns) or because the invariant checker is lacking knowledge to reject counterexamples. Our invariant checker currently works with the following restrictions: it needs a flattened type graph as well as flattened rules and forbidden patterns as input [GLB+12]. Stereotypes need to be encoded by specific types. Moreover, since string attributes are not supported, we encode them by edges to data nodes as we also did in [BLD+11]. For more details to the internals of the invariant checker we refer to [BBG+06, BLD+11].

To classify target constraints into category (2) we proceed as follows: Assuming that the source and target constraints $\mathcal{C}_S$ and $\mathcal{C}_T$ have the above-described pattern form, we can apply invariant checking to verify if the target constraints $\mathcal{C}_T$ are invariant w.r.t. forward operational rules derived from a TGG, given that the source constraints $\mathcal{C}_S$ hold. Intuitively, this means that forward rules do not add any translated target structure which would violate the target constraints assuming that the source constraints hold on the source model. If the invariant checker produces at least one symbolic counterexample, which does not represent a false negative, then the constraint belongs to category (3). Constraints belong to category (1) if the invariant checker does not need the source constraints as assumed constraints to verify that the target constraints are invariants.

Consider the UniqueStereotypes constraint (C1), which can be formulated as a forbidden pat-
tern consisting of a Class holding two identical stereotypes. It holds by TGG construction w.r.t.
forward validation, since it is satisfied in all Class Diagram models created by the example TGG
(cf. Figure 2) independently of the constraints on the Block Diagram meta-model. The example
TGG creates classes only together with a single stereotype. Indeed our invariant checker is able
to identify via static analysis that this constraint belongs to category (1).

Constraint (C2) NoSystemBlocksOrProcesses is an example for category (3) w.r.t. backward
validation. It can be described by two forbidden patterns, one that consists of a SystemBlock
containing a SystemBlock and the other one of a SystemBlock containing a Process. The latter
forbidden pattern is violated by rule 3 in the TGG because it allows to add a Process to a Block,
which also includes SystemBlocks due to the generalization hierarchy defined in the metamodel
(cf. Figure 1a). Indeed, our invariant checker will output a symbolic counterexample, repre-
senting a violation of the constraint when applying rule 3 to a ClassDiagram, holding one Class
with Stereotype system and another associated Class with Stereotype process. Rule 2 violates
the first forbidden pattern analogously. The counterexamples reported by the invariant checker
may represent useful hints to the transformation developer w.r.t. repairing the transformation
specification or the different levels of expressiveness of the source and target language under
consideration. Defining a specific repair mechanism accordingly is part of future work.

The static analysis presented here has its limitations (e.g. constraint (C3) can not be analyzed),
since only (conditional) forbidden patterns can be currently analyzed successfully by our invari-
ant checker. Enhancing the expressiveness of our invariant checker is ongoing work. However,
in [Lam10], it is described more generally how the invariant checking (or constraint preserva-
tion) problem can be reduced to the implication problem for conditions. As proven in [HP09],
in the case of graphs, nested conditions are equivalently expressive to first order graph formulas.
This means that the implication problem for application conditions is undecidable, in general. In
practice, there is also the problem that OCL constraints have to be translated to graph constraints,
which is a non-trivial task [WTEK08]. Consequently, if for these reasons static analysis is not
available, we apply a dynamic analysis technique, which we present in the next section.

Figure 5: Counter Example Generator using the Test Model Generator (cf. Figure 3)

**Generation of Counter Examples** Our existing conformance testing framework can be reused
to automatically check the forward/backward validity of a TGG specification with constraints.
In particular, the Test Model Generator can be altered to search for triples SCT, where either the
source or target model S or T is valid, but the other one is not. The behavior of this Counter
Example Generator is depicted in Figure 5. First, a triple $SCT \in \mathcal{L}^2(tgg)$ is generated as before.
After that, both models S and T are validated. If one model is valid and the other is invalid, a
counter example was found. Otherwise, a new pair is generated.

The generator can be configured with a maximum number of iterations to avoid infinite loops,
occurring because a TGG generates an infinite number of language instances in general. How-
ever, this also means that it is not guaranteed that there are no counter examples if none can be found. As part of future work, we want to use specific heuristics for a systematic model generation. This should increase the success of finding counter examples.

To actually validate the models, the EMF Validation Framework\(^7\) is used, an extensible framework, which allows to integrate constraints specified in all kinds of languages, commonly OCL, but also Check\(^8\) or plain Java. The actual nature of the constraints is completely transparent to clients of the validation framework. The framework lets the appropriate interpreters evaluate the constraints, e.g., the OCL interpreter for OCL constraints. In addition to constraints, multiplicities of references are also checked.

The *BlockHierarchyConstraint* belongs to category (3) as introduced in the beginning of Section 4. Consider for example rule 3 in Figure 2d), which does not check whether *Block* bl2 already contains another *Block* before adding a new *Process*. Moreover, consider rule 2 (cf. Figure 2c), which is not guaranteed to be applied at least two times to make sure that a *Block* contains at least two subblocks. Our current invariant checker is not capable of checking such a complex constraint and, therefore, we applied our dynamic check directly here. The second randomly generated pair of models indeed already represented a counterexample.

### 5 Integration of TGG Implementation and Testing with Constraints

We also extended our conformance testing framework (cf. Figure 3), in particular the *Test Model Generator*, and our *TGG Implementation* to support constraints.

**Conformance Testing:** The goal of the testing framework is to check if each *SCT* obtained by a forward (backward) transformation of *S* (*T*) in the TGG implementation indeed belongs to the TGG language \(\mathcal{L}(tgg)\) and the other way round. Having integrated the TGG with constraints, our new testing goal is to check if each *SCT* with valid *S* and *T* obtained by a forward (backward) transformation of *S* (*T*) indeed belongs to the constrained TGG language \(\mathcal{L}(tgg, C_{tgg})\). As a consequence, we focus now on generating test models satisfying the constraints \(C_{tgg} = C_S \land C_T\) to check conformance.\(^9\) Consequently, in addition to our former framework, the *Test Model Generator* performs a validation on the generated source and target models (shown in Figure 6). If one of the models is invalid, a new random triple *SCT* is generated until a valid pair of source and target models *S* and *T* was found or a predefined number of misses has been reached to avoid infinite loops. The *Test Model Generator* and the *Counter Example Generator* share the same algorithm to generate triple graphs. Therefore, it is also possible to combine both components.

---

\(^7\) http://www.eclipse.org/modeling/emf/?project=validation

\(^8\) Check is provided as part of Xpand (http://www.eclipse.org/modeling/m2t/?project=xpand).

\(^9\) We concentrate on positive testing in our testing framework, meaning that we focus on generating valid input models.
If a pair of valid models has been generated, these can be used as test input and oracle, if one of the models is invalid, these can be used as counterexamples.

**TGG Implementation:** The TGG Implementation with support for constraints is supposed to deliver a valid target model from a valid source model. It is unnecessary to execute the model transformation if the source model is already invalid. However, because of the limitations of automatically checking validity of TGGs with constraints (cf. Section 4), the TGG implementation still needs to check at runtime if a target model is valid. For these reasons, the TGG implementation has been extended with two validation steps. Before executing the transformation, the source model is validated. During the transformation, it is possible that the target model temporarily violates target constraints. Therefore, checking these constraints while the transformation is still running is not necessary. Instead, the created target model is validated after the transformation. If it is now violating target constraints, a warning is issued along with the invalid model.

The **BlockHierarchyConstraint** (C3) is an example of a constraint that may be temporarily violated during a backward transformation from a Class Diagram containing several Classes, which are transformed to Blocks. After transforming the first Class, the constraint is violated. This constraint is also an example, where a valid source model may be transformed to an invalid target model. Considering the backward transformation (cf. Figure 2) of our running example, the following situation could occur: A Block Diagram with one SystemBlock containing a Block and a Process violates the **BlockHierarchyConstraint**. Its corresponding Class Diagram, however, would be valid. So, if we perform a backward transformation from that valid Class Diagram, the TGG implementation returns an invalid Block Diagram along with a warning.

### 6 Related Work

There is some related work concerning validation of transformation rules with consideration of constraints. A tool for the static validation of ATL transformation rules is presented in [BCG11], which considers constraints of the source and target metamodel. A transformation (meta-)model integrates the source and target (meta-)models with the execution semantics of ATL. It is possible to search for instances of the transformation metamodel that violate constraints using bounded verifiers. In contrast, we rely on a static and symbolic checking method to verify invariants given as graph constraints for a graph transformation system, which is a complete static analysis. However, constraints, usually specified in OCL, have to be translated to graph patterns first. This drawback of our approach is mitigated by the counter example generator, which generates example models and evaluates constraints directly.

To the best of our knowledge, there exist no other model transformation approaches considering the integration of metamodel constraints and allowing for automatic checking of forward/backward validity. However, most tools allow to specify additional preconditions in TGG rules using OCL, e.g., [DG09, GR12] as well as our own model transformation tool [GHL12]. In contrast, the TGG formalisms and tools presented in [GEH11, KLKS10] support TGG rule application conditions expressed as graph patterns. [DG09] and [GR12] also allow to specify postconditions in transformation rules, which are checked after the execution of a transformation rule or the complete transformation. Another approach for the verification of arbitrary model transformations based on visual transformation contracts is presented in [GLW12]. These ap-
proaches do not investigate the interplay of additional application conditions in TGG rules with metamodel constraints, except for [KLKS10], which, however, does not focus on automatic checking mechanisms w.r.t. forward/backward validity as we do in this paper.

7 Conclusion

We have shown the importance of considering constraints defined on metamodels in TGG model transformation definition, conformance testing and implementation. Automatic checking of forward/backward validity using static analysis is merely possible for a restricted kind of constraints. Therefore, we also showed how to perform this check by generating meaningful counterexamples.

The graph patterns required as input for the invariant checker have to be derived from constraints, often expressed in OCL. This complex task [WTEK08] is currently not automated, but for a restricted part of OCL, especially simple navigation expressions, we plan to implement such a translation. Moreover, the flattening of metamodels with inheritance, rules and constraints in order to serve as input for the invariant checker still needs to be automated. It is ongoing work to improve the expressiveness of our invariant checker and to investigate also weakest pre- and post-conditions [HP09] techniques. We also want to examine how the source and target model generators can be extended with more systematic generation mechanisms. This would increase the confidence in the results of the Counter Example Generator as well as our conformance testing framework. The generator also needs to be integrated with the static analysis to, e.g., check if counter examples found by the static analysis are no false negatives.

Finally, after the user has found out that his transformation rules are not forward/backward valid, he needs tool support to change his transformation accordingly. However, the causes for violation of metamodel constraints may be manifold. It may be hard for the user to deduce the cause from a counter example and a violated constraint. This remains a complex open issue for future work.

Bibliography


Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars
Claudia Ermel, Frank Hermann, Jürgen Gall and Daniel Binanzer

12 pages
Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars

Claudia Ermel¹, Frank Hermann², Jürgen Gall¹ and Daniel Binanzer¹

¹ Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany
henshin@tfs.cs.tu-berlin.de

² Interdisciplinary Center for Security, Reliability and Trust, Université du Luxembourg
frank.hermann@uni.lu

Abstract: The tool HENSHIN is an Eclipse plug-in supporting visual modeling and execution of rule-based EMF model transformations. This paper describes the recent extensions of HENSHIN by a visual editor for triple graph grammars (TGGs). The visual editor (called HENSHINTGG) supports a compact visualization of triple rules in an integrated editor panel. Internally, triple graph rules are represented as HENSHIN rules and can be simulated using the HENSHIN EMF model transformation engine. Our extension supports the automatic generation of forward translation rules for transforming source into target models. A converter from HENSHIN TGG rules to the graph transformation analysis tool AGG allows a systematic check for conflicts of forward translation rules in AGG based on critical pair analysis.

Keywords: EMF, model transformation tool, triple graph grammar, Henshin

1 Introduction

Model transformations play an important role in model driven development. In graph transformation based approaches and tools, rules express basic transformation steps. In particular, triple graph grammars (TGGs) [Sch94] are a formal technique to specify and reason about bidirectional model transformations. Using graph triples, the relations of source and target models is specified declaratively, by mapping the elements of a correspondence model to corresponding elements of the source and target model. A TGG describes how consistent graph triples are derived synchronously by applying triple rules. From such a TGG, so-called operational rules can be derived automatically to perform unidirectional forward or backward model transformations. TGGs have shown to be a suitable formal basis for model transformations and to reason about properties such as correctness, completeness, or functional behaviour [HEGO10, EEHP09].

The paper presents the new, visual TGG modelling and analysis environment HENSHINTGG that makes use of the existing formally founded EMF model transformation engine HENSHIN. In contrast to existing TGG implementations [GHL12, ALPS11, BGH+05], HENSHINTGG does not only specify and perform EMF model transformations by TGGs but generates forward translation rules (synthesized from forward and source rules) according to [HEGO10] and offers a converter to translate forward translation rules to the graph transformation analyzer AGG [AGG12] in order to benefit from AGG’s critical pair analysis for conflict detection. Fig. 1 shows an overview of the overall workflow using the main tool features of HENSHINTGG.
HENSIN is an Eclipse plug-in supporting visual modeling and execution of EMF model transformations, i.e., transformations of models conforming to a meta-model given in the EMF Ecore format. The transformation approach is based on algebraic graph transformation according to the double pushout (DPO) approach [EEPT06] which are lifted to EMF model transformation by also taking containment relations in meta-models into account [BET12, ABJ+10].

Structure of the Paper: We present our visual TGG editor in Sec. 2 and describe the generation of forward translation rules based on [HEGO10] in Sec. 3. An example for a conflict analysis of forward translation rules converted to AGG based on critical pairs is presented in Sec. 4, while Sec. 5 explains the automatic EMF model translation. In Sec. 6, we compare related approaches and tools to our tool and conclude the paper with an outlook to future work.

2 The Visual TGG Editor

Using triple graph grammars [Sch94], models are defined as pairs of source and target graphs, which are connected via a correspondence graph together with its embeddings into these graphs. In this section, we review main constructions and results of model transformations based on TGGs and introduce our visual TGG editor HENSIN TGG.

A triple graph $G = (G_S \xleftarrow{s} G_C \xrightarrow{t} G_T)$ consists of three graphs $G_S$, $G_C$, and $G_T$, called source, correspondence, and target graphs, together with two graph morphisms $s : G_C \to G_S$ and $t : G_C \to G_T$. A triple graph morphism $m = (m_S, m_C, m_T) : G \to H$ between triple graphs $G$ and $H$ consists of three graph morphisms $m_S : G_S \to H_S$, $m_C : G_C \to H_C$ and $m_T : G_T \to H_T$. A typed triple graph $G$ is typed over a triple type graph $TG$ by a triple graph morphism $\text{type}_G : G \to TG$.

Example 1 (Triple Type Graph) Fig. 2 shows the type graph $TG$ of the triple graph grammar $TGG$ for our example model transformation from class diagrams to database models. The source component $TG_S$ defines the structure of class diagrams while in the target component the structure of relational database models is specified. Classes correspond to tables, attributes to columns, and associations to foreign keys. Morphisms starting at a correspondence part are indicated by dashed arrows.

The HENSIN TGG editor uses EMF models as type graphs and EMF instance models con-
forming to the respective EMF models as typed (attributed) graphs\(^2\). The three EMF models in Fig. 2 have been edited outside the visual TGG editor using the graphical GMF editor for EMF, but any other EMF model editor or generator can be used as well. The morphisms are implemented as references between the types of the three different EMF models. EMF models are imported into the visual TGG editor which enables the use of previously produced EMF models. The names of the three imported EMF models source, correspondence and target that comprise the triple type graph, are shown in the top compartment Imports of the tree view 1 in Fig. 3.

Once a triple type graph is available (i.e., the three EMF models have been imported), triple graphs typed over this type graph may be edited, e.g. for modifying inputs and intermediate

\(^2\) For more details on the formal correspondence of typed attributed graphs and EMF models see [BET12].
states when testing model transformations\(^3\). The visual TGG editor supports editing of triple graph nodes and edges by offering the available types in the palette of the triple graph panel\(^2\). Only triple graphs conforming to the triple type graph can be created. Moreover, only source triple graph elements (colored in red) can be created and modified in the left-hand part of the editor, correspondence graph elements (blue) in the center, and target graph elements (yellow) in the right part. The separators between the different triple panels can be moved using the mouse. Morphisms from correspondence to source and target elements are drawn as edges across the separators. Fig. 3 shows a sample triple graph \textit{OrderDetails} containing a complete source part (the class diagram) but incomplete corresponding target and correspondence graphs.

\[
\begin{align*}
L &= (L_S \xrightarrow{t_L} L_C \xrightarrow{t_L} L_T) \\
R &= (R_S \xrightarrow{t_R} R_C \xrightarrow{t_R} R_T)
\end{align*}
\]

\[
\begin{align*}
& L \xrightarrow{tr} R \\
& m \xrightarrow{(PO)} n
\end{align*}
\]

Figure 4: Triple rule (left) and triple transformation step (right)

Triple graphs can be generated by applying \textit{triple rules} to the start graph. Triple rules synchronously build up their source, target and correspondence graphs, i.e., they are non-deleting. A triple rule \(tr\) (left part of Fig. 4) is an injective triple graph morphism \(tr = (tr_S, tr_C, tr_T) : L \rightarrow R\) and w.l.o.g. we assume \(tr\) to be an inclusion. Given a triple graph morphism \(m : L \rightarrow G\), a triple graph transformation (TGT) step \(G \xrightarrow{tr_m} H\) (right part of Fig. 4) from \(G\) to a triple graph \(H\) is given by a pushout of triple graphs. A grammar \(TGG = (TG, S, TR)\) consists of a triple type graph \(TG\), a triple start graph \(S = \emptyset\) and a set \(TR\) of triple rules.

\[
\begin{align*}
\text{C2T(cn: } \text{String}) & \xrightarrow{SC2T} \text{cn: } \text{String} \\
\text{A2C(n: } \text{String, t: } \text{String}) & \xrightarrow{A2C} \text{n: } \text{String, t: } \text{String}
\end{align*}
\]

Figure 5: Some rules for the model transformation \textit{CD2RDBM} (\textsc{HenshinTGG} screenshots)

\textit{Example 2 (Triple Rules)} \quad The triple rules shown in Fig. 5 are part of the rules of the grammar \(TGG\) for the model transformation \textit{CD2RDBM}. In \textsc{HenshinTGG}, triple rules are drawn in

\(^3\) Moreover, the triple graph editor can be used for resolving inconsistencies within a future extension of the tool to model synchronization.
short notation, i.e. left and right hand side of a rule are depicted in one triple graph. Elements which are created by the rule are labeled by "++". Rule CD2DB (see 1 in Fig. 3) synchronously creates a class diagram together with the corresponding database. Analogously, rule C2T creates a class with name “n” together with the corresponding table in the relational database. A subclass is connected to the table of its superclass by rule SC2T. Attributes with type “t” are created together with their corresponding columns in the database component via rule A2C.

The visual HENSHINTGG editor for triple rules consists of three panel parts like the visual triple graph editor (see 2 in Fig. 3). But in addition to the triple graph editor, the rule editor palette offers a green "++" to mark elements as created (and to unmark marked elements if necessary). Note that HENSHINTGG checks triple rules for consistency at editing time, i.e. if a node is "++"-marked, all incident edges are marked automatically, as well.

Figure 6: Triple rule C2T with NAC ClassesBeforeAssocs

HENSHINTGG supports negative application conditions for triple rules that forbid the presence of certain structures when applying a rule [EEHP09, GEH11]. A visual NAC editor can be opened via the tree view and consists of a three-panel triple graph editor again. A rule may have several NACs, the one to be shown in the visual NAC editor has to be selected in the tree view. Fig. 6 shows rule C2T with an additional NAC that forbids the synchronous creation of a class and a table if there are associations in the class diagram. The morphism from the rule to one of its NACs is indicated by equal numbers for mapped nodes (in Fig. 6, the ClassDiagram node is mapped to the NAC). Edges are mapped accordingly automatically. The rule palette entry Mapping supports the definition of a mapping from the triple rule to a NAC. Note that only unmarked elements (without "++") can be mapped to NAC elements, a consistency property which is also checked automatically by the editor.

A triple rule can be applied by clicking the button Execute Rule in the rule’s tool bar (the upper right corner in Fig. 6), and selecting the graph the rule should be applied to. The result is shown in the view of the selected graph.

3 Generation of Forward Translation Rules

From each triple rule \( tr \), so-called operational rules can be automatically derived [Sch94] for parsing a model of the source or target language (source and target rules) and for model transformations from source to target or backwards (forward and backward rules), as depicted in Fig. 7.
According to [HEGO10], the extension of forward rules to forward translation rules is based on additional Boolean attributes for all elements in the source component, called translation attributes that control the translation process by keeping track of the elements which have been translated so far. This ensures that each element in the source graph is translated at most once.

The algorithm for constructing forward translation rules from triple rules is as follows (see [HEGO10] for its formal definition): For each triple rule \( tr \), initialize the forward translation rule \( tr_{FT} = tr_{F} \) by the forward rule \( tr_{F} \). Add an additional Boolean attribute \( isTranslated \) to each source element (node, edge or attribute) of \( tr_{FT} \). In the left-hand side of \( tr_{FT} \), for each source element, the value of the \( isTranslated \) attribute is set to false if the element is generated by the source rule \( tr_{S} \) of \( tr \), otherwise it is set to true. In the right-hand side of \( tr_{FT} \), the value of all \( isTranslated \) attributes is set to true. For all source elements in NACs, the attribute \( isTranslated \) is set to true as well.

Note that in contrast to forward translation rules, pure forward rules need additional control conditions, such as the source consistency condition in [EEHP09], to ensure correct executions. In HENSHINTGG, forward translation rules are computed automatically. The translation attributes for nodes and edges are kept separately as an external pointer structure in order to keep the source model unchanged. In the source graph editor panel of a forward translation rule, all elements that are still to be translated are marked by a "<tr>" tag.

Example 3 (Forward translation rule) Fig. 8 shows the forward translation rule \( FT_{SC2T} \) generated from triple rule SC2T. The Class node and its incident edge are marked by a "<tr>" tag as to be translated, since these model elements correspond to the model elements generated by the source rule of triple rule SC2T.
Forward translation rules can be edited in a restricted visual triple rule editor which allows for a manual extension of additional NACs. All other rule editor operations are blocked because forward translation rules are generated automatically and should not be changed manually. Fig. 9 shows the abstract syntax of the forward translation rule $FT_{SC2T}$ from Fig. 8, as it is represented in HENSHIN, where left-hand and right-hand sides of a rule are kept separately, with morphisms inbetween. We can see how the translation attributes of source elements are switched from false to true.

For matching, we internally keep two tables (hashmaps) “TranslatedNodes” and “TranslatedEdges” based on the IDs of the elements of an EMF instance model. These tables are constructed and updated dynamically during transformation execution. A match is valid if for each matched element we have one of the following cases: 1) its translation attribute is true and its ID is present in the corresponding table of translated elements, or 2) its translation attribute is false and its ID is not present in the corresponding table of translated elements.

### 4 Conflict Analysis Based on AGG

According to [HEGO10], a forward translation sequence $G_0 \xrightarrow{\text{tr} \ast \ FT} G_n$ is called complete if $G_n$ is completely translated, i.e. all translation attributes of $G_n$ are set to true. A model transformation based on forward translation rules with NACs (consisting of a source graph $G_S$, a target graph $G_T$, and a complete forward translation sequence $G_0 \xrightarrow{\text{tr} \ast \ FT} G_n$) is terminating if each forward translation rule changes at least one translation attribute from false to true; it is correct if each forward translation results in a triple graph that can be generated by triple rules, and it is complete if for each source graph there is a forward translation sequence that results in a triple graph that can be generated by triple rules.

However, not all terminating forward translation sequences are complete. A counter example is a forward translation rule sequence applied to the triple graph $TwoClasses$ consisting of a parent class named $Client$ and a subclass named $PremiumClient$ connected to class $Client$ by a parent edge (see source graph in Fig. 10).

The incomplete forward translation sequence is as follows: $FT_{CD2DB}; FT_{C2T}$ (applied to class $PremiumClient$); $FT_{C2T}$ (applied to class $Client$). The sequence is terminating (no forward translation rule can be applied any more), but the result after applying this sequence is a triple graph where not all translation attributes are set to true, i.e. not all source model elements have been translated: the parent edge could not be translated.
In HENSHINTGG, elements that could not be translated are reported as error message in the triple graph panel showing the (partial) translation result (see Fig. 10). This allows the user to reason about possible conflicts between rule applications. The reason why the parent edge was not translated by the given forward translation sequence is a conflict between rule FT\_C2T (applied to class *PremiumClient*) and FT\_SC2T which could not be applied to class *PremiumClient* after the application of rule FT\_C2T.

In order to ensure completeness in the general case, the execution of model transformations may require backtracking (not implemented in HENSHINTGG). However, as shown in [HEOG10], backtracking is not necessary, if the significant critical pairs between transformation rules are strictly confluent and the system is terminating, i.e., a system satisfying this condition does not have to be confluent in the general sense. HENSHINTGG implements a converter from triple rules in HENSHIN to the graph transformation analysis tool AGG, which provides a critical pair analysis engine. A critical pair is a conflict between two rules in minimal context and it is significant, if the overlapping graph can be embedded in a possible intermediate state of a model transformation sequence. In particular, it is not significant if a fragment in the source component cannot be embedded into a valid source model due to language constraints.

Figure 11: Critical pair between rules FT\_C2T and FT\_SC2T computed by AGG
Fig. 11 shows the (only) critical pair between the rules \( FT\_C2T \) and \( FT\_SC2T \) as depicted by the AGG critical pair analyzer. In the window to the right, the critical overlapping graph of both rules’ left-hand sides is shown, and it is indicated that we have a change-use-attr conflict, since both rules want to access and change the \textit{isTranslated} attribute of the subclass.

In order to avoid the conflict shown in Fig. 11, the easiest way is to add a NAC to rule \( FT\_C2T \) that forbids its application to classes which have a parent class. According to [HEGO10], such additional conflict-avoiding NACs are called \textit{filter NACs} and can be generated automatically. Note, however, that the generation of filter NACs is not yet supported by HENSHINTGG.

5 Performing Model Transformation in HENSHINTGG

Using a set of confluent forward-translation rules, we can be sure to always get a complete forward translation sequence, i.e., all elements are translated and the result is unique. The upper part of Fig. 12 shows rule \( C2T \), now extended by a filter NAC. With this extension, the set of forward-translation rules now is confluent, since there are no critical pairs any more.

![Figure 12: Rule \( FT\_C2T \) with filter NAC (top) and input graph \textit{TwoClasses} (bottom)](image)

HENSHINTGG supports the automatic forward translation of a given source model by offering a button \textit{Execute Forward Translation} in the tool bar of the EMF source model to be translated (see the bottom part of Fig. 12). Having pressed the button, the forward translation rules are executed in arbitrary order; confluence of the transformation system guarantees a unique result. The resulting target triple graph is shown in the same window as the source model since the translation is performed \textit{in-place}. Fig. 13 shows the target triple which is the result of translating the source model in Fig. 12. In addition, the sequence of applied forward translation rules is shown to the modeller in the message window. For debugging purposes, also single forward translation rule applications can be executed, analogously as for triple rules.
6 Related Work and Conclusion

General model transformation tools such as ATL [JABK08] and MOMENT2-MT [MOM12] are usually used to perform in-place model transformations and do not restrict the structure of transformation rules. Thus, they do not ensure TGG-specific properties like preservation of source models [Sch94] and syntactical correctness and completeness [EEHP09]. Moreover, the forward and backward transformations are manually specified and not generated from a single specification. While ATL and MOMENT2-MT use textual specification techniques, graph transformation tools like HENSHIN (in-place) [ABJ10] and FUJABA [Fuj12] offer the visual specification of transformation rules, i.e., a form of visual programming interface.

In addition to HENSHIN TGG, further TGG tools based on EMF are available. The TGG interpreter [GK10] provides a feature to define OCL expressions as rule conditions, while formal application conditions cannot be specified. However, the formal results concerning correctness and completeness [EEHP09] are not available for systems with OCL conditions. The TGG tools MOTE (model transformation engine) [GW09] and eMoFlon [ALPS11] perform a compilation to the FUJABA tool suite [BGH05, Fuj12] for the execution of model transformations. While eMoFlon supports the specification of TGGs with negative application conditions (NACs), this is not the case for MOTE. MOTE offers certain optimization strategies concerning efficiency. Since correctness cannot be ensured for optimizations, the tool executes dynamic run-time checks to validate that a model transformation sequence was executed correctly [GHL12]. Moreover, MOTE uses a relaxed notion of correspondences for triple graphs, where correspondence nodes may link an arbitrary number of source and target nodes [GHL12].

In order to improve efficiency of TGG tools, suitable static and dynamic conditions have been studied that allow to completely avoid backtracking. Klar et al. use a restricted class of TGGs for which they describe explicit dynamic conditions based on pre-checking contextual edges when translating a node [KLKS10]. Lauder et al. leverage these restrictions on TGGs and introduce the notion of precedence TGGs, where rules are required to form a partial order concerning the execution [LAVS12]. However, these conditions are not checked statically. Giese et al. present efficiency conditions for a restricted class of TGGs [GHL12], where, e.g., each forward rule has to translate at least one source node and may not be in conflict with another rule via a critical pair. The first condition excludes examples where the translation of a single edge or attribute is
handled separately by one rule [HEEO12], and the second condition excludes the well-studied case study on the object relational mapping [EEHP09] used in this article. The tool was extended by a prototypical export [GHL12] of so-called bookkeeping rules to AGG for conflict analysis, but it does not provide re-import and evaluation.

HenshinTGG is based on the formal definitions for TGGs [Sch94, EEHP09, HEGO10] and supports conflict analysis via the converter to AGG. The explicit marking of edges overcomes the restriction in [GHL12] that rules are required to create at least one node. The implementation of a re-import feature for displaying and evaluating the critical pairs is work in progress. HenshinTGG allows the user to manually use the analysis and optimizations techniques presented in [HEGO10] in order to improve efficiency. The automated generation of filter NACs [HEGO10] can be implemented as a direct extension and is future work. A further line of future work is the extension of the tool to support also backward transformations, model synchronization [HEEO12], critical pair analysis directly in the Henshin GUI, and the import of EMF instances. Moreover, we plan to use HenshinTGG within an industrial case study for software translation for satellite systems and in further case studies to evaluate the results concerning correctness and efficiency.

References


Visual Modeling of EMF Model Transformations Based on Triple Graph Grammars


Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

EMorF - A tool for model transformations

Lilija Klassen and Robert Wagner

6 pages
EMorF - A tool for model transformations

Lilija Klassen¹ and Robert Wagner²

¹ klassen@solunar.de, ² wagner@solunar.de, http://www.solunar.de/
Solunar GmbH i.Gr., Germany

Abstract:
In this paper, we present EMorF - a model transformation tool for EMF. EMorF supports the specification and execution of in-place model transformations as well as model-to-model transformations. The graphical though formal specification is based on (triple-) graph grammars, which are executed by an interpreter system. In this paper, we focus on the provided tool support for the development and execution of model transformations.

Keywords: triple graph grammars, graph transformation, model transformation, EMF, EMorF

1 Introduction

Model transformations are an integral part of the model-driven software development approach. The requirements and expectations regarding model transformation techniques are very demanding. A practical solution should be formal and easy to use. It should allow specifying and executing in-place model transformations in order to support model refinement, model refactoring for software maintenance and renewal as well as model inconsistency checking with automated inconsistency resolution. In addition, a practical formalism should allow specifying model-to-model transformations which can be used for the evolution of models, for the migration of data, and for model exchange between different tools.

EMorF¹ is an open source model transformation tool for the Eclipse Modeling Framework (EMF). It supports the specification of in-place model transformations as well as model-to-model transformations. In-place model transformations are specified graphically by graph rewriting rules and model-to-model transformations are specified by triple graph grammars [Sch94]. The main advantage of triple graph grammars is that they allow executing a model-to-model transformation in both directions. Moreover, they can be used to check the correspondence between two models and to synchronize models incrementally [GW09]. Because of a strong graph and category theoretic background, graph grammars have precise formal semantics.

In order to be more expressive, EMorF integrates the Object Constraint Language (OCL) for the specification of constraints and application conditions. In addition to that, EMorF provides a simple but powerful application programming interface to the EMorF rules and the EMorF interpreter system.

There are several graph grammar based model transformation tools, e.g. AGG [AGG12], Henshin [Ecl12], eMoflon [MOF12], Fujaba [FUJ12] or TGG-Interpreter [TGG12]. But to the best

¹ http://www.emorf.org
EMorF - A tool for model transformations

of our knowledge only the TGG-Interpreter is based on EMF models and supports model-to-
model transformations specified with triple graph grammars, which are executed by an inter-
preter system. But in contrast to EMorF the TGG-Interpreter does not support in-place model
transformations.

The remainder of this paper is organized as follows. In the next section we give a brief and
informal introduction to the basic concepts of our tool. In Section 3, we present the tool support
for the specification of in-place model transformation. The tool support for model-to-model
transformations is described in Section 4. The paper closes with a conclusion and an outlook on
future work in Section 5.

2 EMF (Meta-) Model

EMorF is a tool for the transformation of models that are based on the Eclipse Modeling Frame-
work [EMF12]. The reason for using EMF is that it is a well-established technology for building
tools operating on (data) models with well-defined meta-models. For this purpose, EMF provides
tool support for meta-modeling, code generation and dynamic model instantiation. In addition,
EMF is supported by many other standards, frameworks and tools. This ensures a wide applica-
tion area for EMorF.

An EMF meta-model consists of classes with attributes, associations with multiplicity anno-
tations (including composition and reference relationships), as well as generalizations. EMorF
relies on EMF meta-models and (triple) graph grammars. For this purpose, the concepts are
mapped to each other where classes and associations correspond to nodes and edges in a typed
graph. As a consequence, models based on a meta-model are interpreted as graphs where objects
correspond to nodes and links correspond to edges.

3 In-place Model Transformation

In-place model transformations are specified using EMorF graph rewriting rules. For this pur-
pose, EMorF provides a graphical editor with a palette, a property view and an outline view.
Figure 1 shows the EMorF editor and its corresponding views.

The graphical editor in Figure 1 shows the PullUpAttribute refactoring rule. This rule removes
one attribute from a subclass and inserts it into its superclass. However, the refactoring can only
be applied, if the refactoring does not introduce any attribute naming conflicts, i.e. if none of the
subclasses of the superclass already uses the same attribute name as the attribute to be moved.

For the specification of such a graph rewriting rule, EMorF uses the short-hand FUJABA-
notation where the left- and right-hand sides of a rule are specified in a single area. To distin-
guish between patterns that have to be bound and patterns that have to be created or deleted,
the corresponding objects and links are marked with the appropriate annotations. For example,
in the PullUpAttribute rule shown in Figure 1, the objects father, son and attr as well as the
link inheritance have been marked with the $\ll$bind$\gg$ annotation, whereas the link oldAttrLink
is marked with a $\ll$delete$\gg$ and the link newAttrLink with a $\ll$create$\gg$ annotation.

The objects and links are created in the editor by simply dragging them from the palette onto
the drawing area. For this purpose, the palette offers all types that are contained in the corre-
However, in many cases it is not sufficient to specify a pattern consisting of objects and links only. Often, additional constraints have to be specified as well. Furthermore, it is necessary to provide ways and means for the specification of attribute value modification within a rule. For this purpose, we have integrated OCL support into EMorF.

The OCL support in EMorF comes in the form of expression nodes. The expression nodes are also created using the palette. The OCL expressions can be edited easily within the graphical editor due to in-place editing support with built-in content assist functionality.

The OCL was designed originally as a constraint and object query language without any side effects. Meanwhile, there are extensions that allow using OCL also for assignments. A prominent example is the QVT language. Therefore, we decided to use OCL expressions both for constraints and assignments as well. However, we did not introduce new language concepts but annotate the expression either by \(\text{constraint}\) or \(\text{assignment}\) annotations. In the case of a constraint, a live validation checks if the expression is of type boolean. In the case of an assignment expression, it is checked whether the result is of the same type as the attribute. If the validation fails, the user is informed about the malformed OCL expression in the standard problem view of the Eclipse IDE.

Figure 1 contains two OCL constraints. The first constraint expression ‘\(\text{attr}\).\text{name} = \text{x}\)’ requires that the name of the attribute that should be pulled up has to be equal to the rule parameter ‘\(\text{x}\)’. The second constraint ‘\(\text{father}\rightarrow\text{closure(}\text{subTypes}\text{)}\rightarrow\ldots\)’ ensures that the modification described by the rule is only applied if no attribute naming conflicts exist.

In EMorF, the specified rules are executed in an interpreted way. In general, there are different...
EMorF - A tool for model transformations

ways how an interpreter executes rules: (1) an interpreter can execute a rule directly (like in TGG-Interpreter [TGG12]), (2) an interpreter can translate a rule into an intermediate representation and execute this representation immediately (like in Henshin [Ecl12]), or (3) an interpreter executes a precompiled intermediate representation which has been created by a corresponding generator. In the case of EMorF, the rules are precompiled into an intermediate representation. The EMorF generator, which is part of the interpreter system, uses a minimum branching graph algorithm to produce intermediate representations. These intermediate representations serve as search plans that are executed if the user invokes the interpreter.

There are three different ways to invoke rule execution in EMorF. The first possibility is to use a context sensitive pop-up menu on the selected object of the model that has to be transformed. The pop-up menu offers potentially applicable rules, i.e. rules containing objects of the same type as the selected object. The user can choose to apply the rule only once or to apply the rule on all possible matches. In addition, if rule parameters are specified, the user is asked to enter necessary parameter values. The usage of the pop-up menu is quite simple and allows for an interactive rule execution. It is quite comfortable especially during the development of graph rewriting rules.

The second possibility to invoke a rule is to use the standard Eclipse run configuration dialog. In this dialog, all necessary information are collected and stored. Therefore, this way of rule invocation is quite suitable if the rule has to be applied over and over again.

The third and last possibility for rule invocation is made available by the EMorF API. The provided interface to the rules and the interpreter system allows to implement control structures and to inspect matching results of applied rules and their references to objects and links programmatically. In addition, the graph rewriting rules can be split up and applied in a match-only mode, i.e. modifications are not executed at all or in a later step. This allows for fine grained control which might be helpful in implementing more complex algorithms used for instance for simulation or model refactoring.

4 Model-to-Model Transformation

For model-to-model transformation the same graphical editor as for in-place model transformations is used. However, in this case we employ the visual and bi-directional transformation technique of triple graph grammars [Sch94]. A triple graph grammar is a declarative definition of a model transformation and consists of a set of transformation rules. In Figure 2, a triple graph grammar rule is shown.

The rule specifies a consistent correspondence mapping between the objects of the source and the target model and is part of a transformation between a class diagram and a relational database schema. In particular, the presented rule defines a mapping between a class and a table with a column and a primary key and uses some additional assignments and constraints.

A triple graph grammar rule is specified in the editor in the same way as an in-place model transformation. However, as shown in Figure 2, the graphical editor is separated into three different areas representing the source, the correspondence and the target domain. Since each domain can have its own meta-model, the palette is also subdivided into different areas to hold the elements of the involved meta-models.
Model-to-model transformation rules can be executed in source-to-target (forward) or in target-to-source (reverse) direction. In addition, the interpreter is able to check the correspondence mapping and to create the traceability objects of the correspondence domain (cf. Figure 2). The result of a transformation depends on the execution mode and is a file that contains either a model corresponding to the source or the target domain or a trace model holding the correspondence mappings. The correspondences can be inspected using the trace viewer shown in Figure 3.

A model-to-model transformation is executed in EMorF with the help of a launch configuration, where paths to an input, an output and a trace model are specified. After the direction mode (forward, mapping, or reverse) is configured, the model-transformation can be executed either in standard or in debug mode.
5 Conclusion and Future Work

In this paper, we presented the model transformation tool EMorF. EMorF operates on EMF models and relies on the visual and formal technique of graph grammars and triple graph grammars. EMorF supports both techniques in one tool by one and the same graphical editor, an interpreter system for rule execution, a debugger and other views. In addition, EMorF integrates the OCL standard and enables this way advanced specifications with negative application conditions, closures and flexible path expressions, whereas the included debugger enables users to detect and diagnose errors by going step by step through their model transformation.

Currently assignments in EMorF model-to-model transformation rules must be specified for each transformation direction separately. This is different to the specification in [Sch94], where the assignments are derived from declarative attribute constraints. In order to be compatible with the approach from [Sch94], we plan to introduce such declarative attribute constraints in EMorF.

Another short-term target of our future work is to improve the graphical user interface, for instance to support the specification of OCL expressions in an integrated text editor, to complete the EMorF API and to analyse the performance compared to other tools. As a long-term target, we plan to integrate our incremental model synchronization algorithms into EMorF. Then we want to investigate how EMorF can be used for inconsistency management and other use cases.

Bibliography


Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

RECONNET: A Tool for Modeling and Simulating with Reconfigurable Place/Transition Nets

Marvin Ede, Kathrin Hoffmann, Gerhard Oelker, Julia Padberg

11 pages
RECONNET: A Tool for Modeling and Simulating with Reconfigurable Place/Transition Nets

Marvin Ede, Kathrin Hoffmann, Gerhard Oelker, Julia Padberg

Hochschule für Angewandte Wissenschaften Hamburg

Abstract:
In this contribution we present a tool for modeling and simulation with reconfigurable Petri nets. Taking the idea of algebraic graph transformations to marked Petri nets we obtain Petri nets whose net structure can be changed dynamically. The rule-based change of the net structure enables the adequate modeling of complex, dynamic structures as for example of the scenarios of the Living Place Hamburg. The tool RECONNET uses decorated place/transition nets that are extended by various annotations. Especially, they have transition labels that may change when the transition fires. The transformation approach is based on the well-known algebraic transformation approach, but here we use a variant, namely the cospan approach, that inverts the relation between left- and right-hand sides and interface in the rules.

Keywords: net transformation tool, reconfigurable Petri nets, cospan DPO approach

1 Motivation
Reconfigurable Petri nets [EP03, LO04, EHP+07, PEHP08] are a powerful and intuitive formalism for describing complex systems with dynamic structures that has been available since 2003. The characteristic feature is the possibility to discriminate between different levels of change. In this paper we present a tool that directly allows modeling and simulating reconfigurable place/transition nets. The main motivation for the tool RECONNET is the formal description of scenarios from the Living Place Hamburg, that is a smart home under constant development since 2009. It is a loft apartment with dynamic mapping of functions to spaces according to the respective situation of the resident (e.g. bedroom, kitchen, living room). The Living Place Hamburg is a laboratory for applied research in different areas of ambient intelligence and covers different areas of IT-based urban living. Scenarios of the Living Place Hamburg describe the way the resident interacts with the smart home. One scenario illustrates for example an everyday procedure of the resident at the Living Place and how the system can accomplish its aim to support the residents of the Living Place in his everyday procedure. In order to achieve a better understanding, we want to abstract from sensor data using nondeterminism instead of complicated control structures. Reconfigurable place/transition nets are suitable for this purpose, as they offer the possibility to differentiate between the resident’s actions and the smart home’s reaction. The smart home’s reaction is modeled by the change of the infrastructure that is given by the underlying place/transition net structure. In a comprehensive case study [Rei12] the “morning scenarios” of the Living Place are modeled using reconfigurable place/transition nets. For illustration in Fig. 2 in Section 3 a screenshot is given with a rule similar to those in the case
study. It shows the rule `ringAlarm` that belongs to a “morning scenario”. The rule models the extension of the scenario in case an alarm clock should be used. The transition `wakeUp` is replaced dynamically by the net `R`, describing a simple alarm clock with a snooze function. Unfortunately for the case study [Rei12] the tool `RECONNET` could not be employed as both have been developed simultaneously.

In this contribution we first summarize the underlying formal technique that is based on decorated place/transition nets and net transformations in the cospan approach. Then we present the tool `RECONNET` for modeling and simulation reconfigurable place/transition nets. The tool is the result of two courses at the HAW Hamburg. Concluding remarks concern related and future work.

2 Reconfigurable Place/Transition Nets

Using the algebraic approach to Petri nets facilitates the combination of the algebraic approach to transformation. So, a marked place/transition net is given by \( N = (P, T, \text{pre}, \text{post}, M) \) with pre- and post-domain functions \( \text{pre}, \text{post} : T \to P^\oplus \) and a marking \( M \in P^\oplus \), where \( P^\oplus \) is the free commutative monoid over the set \( P \) of places. For \( M_1, M_2 \in P^\oplus \) we have \( M_1 \leq M_2 \) if \( M_1(p) \leq M_2(p) \) for all \( p \in P \). A transition \( t \in T \) is \( M \)-enabled for a marking \( M \in P^\oplus \) if we have \( \text{pre}(t) \leq M \), and in this case the follower marking \( M' \) is given by \( M' = M \ominus \text{pre}(t) \oplus \text{post}(t) \) and \( M[t]M' \) is called firing step. To gain an adequate modeling technique a few new features needed to be added. Obvious is the extension to capacities and names. More interesting are the transition labels that may change, when the transition is fired. This allows a better coordination of transition firing and rule application, for example can be ensured that a transition has fired (repeatedly) before a transformation may take place.

2.1 Decorated Place/Transition Nets

In order to provide the technical basis for the modeling of scenarios we need place/transition nets that have the following additional decorations: capacities, names for places as well as transitions and additional transition labels that can be changed by firing that transition. This last extension is conservative with respect to Petri nets as it does not change the net behavior. But it is crucial for the application of the rules and provides the possibility to control the application of rules.

A decorated place/transition net is a marked place/transition net \( N = (P, T, \text{pre}, \text{post}, M) \) together with names and labels. Based on \( A_P, A_T \) a name space with \( \text{pname} : P \to A_P \) and \( \text{tname} : T \to A_T \) we have explicit names for places and transitions. Moreover, transitions need to be equipped with labels that may change when the transition fires. For the correct application of a rule it may be important that a transition has already fired. This cannot be modeled with usual place/transition nets. Considering the tokens in the transition’s post places, does not work as these tokens may be consumed as well. So we have to change the label of the transition. For example, the label changes from `false` (indicating that the transition has not yet fired) to `true` (indicating that it has already fired). In Fig. 2 in Section 3 the transition `stopAlarm` in the net \( K \) (or \( R \)) of rule `ringAlarm` has the following node attributes (given in the upper middle panel):

\( \text{Id:28, Name: stopAlarm, Label: false, Renew: toggle, where toggle} \)
ECEASST

is a function mapping false to true and the other way round. More formally, toggle can be expressed using propositional logic \( \text{toggle}(x) := \neg x \).

Formalizing this notion we introduce a set \( W_T \) of changing labels for each transition with \( W_T := \bigcup_{t \in T} W_t \) and a function \( tlb : T \rightarrow W_T \) mapping transitions to transition labels \( W_T \). Then each transition is decorated with an endomorphism on \( W_t \), i.e. a function that maps labels to labels. Th is decoration is given by the function \( rnw : T \rightarrow \text{END} \) where \( \text{END} \) is a set containing some endomorphisms on \( W_t \subseteq W_T \), so that \( rnw(t) : W_t \rightarrow W_t \) is the function that renews the transition label. These labels are important for the control of the rules, but not for the net’s behavior.

The firing of the extended nets is the same as in place/transition net except for the changing transition labels. Moreover, this extension works for parallel firing as well. Given a transition vector \( v = \sum_{t \in T} k_t \cdot t \) then the label is renewed by firing \( tlb[v] \cdot tlb' \) and \( tlb' \) is computed by:

\[
tlb'(t) = rnw(t)^{k_t} \circ tlb(t)
\]

Analogously to the marking the transition labels also evolve during the token game.

2.2 Net Transformations in the Cospan-Approach

The cospan approach is a variant of algebraic transformations - namely the double pushout (DPO) approach - where the left-hand side and right-hand side of the rule are embedded in the interface \((L \rightarrow K \leftarrow R)\), hence cospan. In contrast to the classical DPO approach (where to morphisms starting from the interface \( L \leftarrow K \rightarrow R \)) we add new items first and then delete (some of) the old items in the second step.

In [EHP09] the cospan DPO approach has been shown to be equivalent to the classical DPO approach. In this way we are able to switch between these two approaches and, finally, omit the interface in the graphical description. In this paper we use the cospan DPO approach, because from implementation point of view, it is often more convenient to add the new items first and delete some of the old items in a second step. This idea is adopted in the cospan DPO approach where a rule is given by a cospan of morphisms, while a transformation step via a cospan rule is still defined by two pushouts. Roughly spoken, in the classical DPO approach the intermediate net obtained by rule application is often full of holes like Swiss cheese, while in the cospan DPO approach this net includes the source net and the target net.

For example in Fig. 1(a) the rule is in the DPO approach and the rule in Fig. 1(b) is in the cospan DPO approach. But both of these two rules replace the transition on the left-hand side by

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig1.png}
\caption{Comparing and cospan (DPO) approach}
\end{figure}
one on the right-hand side, that has a further place in its post-domain. In contrast to the classical approach in the cospan DPO approach we are able to relate the old and new items in an easy way. The interface of this cospan rule (see Fig. 1(b)) states for example that the environment of both transitions have to have at least the same places in the pre- and post-domain.

Thus, on the one hand several properties could be formulated in a more intuitive way and on the other hand some aspects could be investigated that have escaped our attention in the classical DPO approach. The main results in [HEH10] are not only sufficient (as in [EHP+07] using the classical approach) but also necessary conditions, so that a transformation step and a firing step can be executed in arbitrary order. Moreover in future work we will consider further property preserving net transformations in the cospan DPO approach. Of special interest are strongly connectivity and liveness and we expect that especially these properties could be formulated and proven in a more intuitive way in the cospan DPO approach.

For decorated place/transition nets as given above, we obtain with the following notion of morphisms an \( \mathcal{M} \)-adhesive HLR-category (see [Pad12]). \( \mathcal{M} \)-adhesive HLR systems can be considered as a unifying framework for graph and Petri net transformations providing enough structure that most notions and results from algebraic graph transformation systems are available (e.g. in [EEPT06, EGH+12]).

Morphisms are given as a pair of mappings for the places and the transitions, so that the structure and the decoration is preserved and the marking may be mapped strictly. Given two nets \( N_i = (P_i, T_i, \text{pre}_i, \text{post}_i, M_i, \text{pname}_i, \text{tname}_i, \text{tlb}_i, \text{rnw}_i) \) for \( i = 1, 2 \) then the morphism \( f : N_1 \to N_2 \) is given by \( f = (f_P, f_T) \) the mapping of places to places and the mapping of transitions to transitions. Additionally, the usual equations have to hold that ensure the preservation of markings and of labels [Pad12].

Given a decorated place/transition net and a rule together with an occurrence morphism \( o \), the a transformation step \( (N_1, M_1) \xrightarrow{\text{rule}, o} (N_2, M_2) \) consists of the following pushout diagrams (1) and (2).

\[
\begin{align*}
(L, M_L) & \xrightarrow{l} (K, M_K) \xrightarrow{r} (R, M_R) \\
(N_1, M_1) & \xrightarrow{o} (N_0, M_0) \xleftarrow{n} (N_2, M_2)
\end{align*}
\]

A rule in the cospan approach is given by the left-hand side, interface and right-hand side net, respectively, and a cospan of two net morphisms \( l \) and \( r \). An occurrence morphism \( o \) identifies the relevant parts of the left-hand side in the given net \( (N_1, M_1) \). Then a transformation step \( (N_1, M_1) \xrightarrow{\text{rule}, o} (N_2, M_2) \) can be constructed in two steps, provided that gluing conditions hold. The characterization of specific points is a sufficient condition for the existence and uniqueness of the so-called pushout complement which is needed for the second step in a transformation.

This construction as well as a huge amount of notion and results are available since decorated place/transition nets can be proven to be an \( \mathcal{M} \)-adhesive HLR category (see [Pad12]). Hence we can combine one net together with a set of rules leading to reconfigurable place/transition nets.

A reconfigurable place/transition net \( RN = ((N, M), \mathcal{M}) \) is given by
• a decorated place/transition net $N = (P,T, \text{pre}, \text{post}, \text{pname}, \text{tname}, \text{cap}, \text{tlb}, \text{rnw})$ and its marking $M$ and

• a set of rules $\mathcal{R}$, where rules $\text{rule}$ are given in the cospan approach

$\text{rule} = (L,M_L) \rightarrow (K,M_K) \leftarrow (R,M_R)$.

3 RECONNET: Editor and Simulator

The software tool RECONNET (REConfigurable NET) has been developed so that the modeling and simulation capabilities of reconfigurable nets are supported adequately. The most important feature of the tool is the ability to create, modify and simulate reconfigurable nets in a single tool through an intuitive graphic-based user interface. RECONNET is completely implemented in Java 6, therefore being fully portable. For the purposes of the user interface, SWING [SWI] was found to be suitable. In combination with JUNG [JUN], the JAVA universal network/graph framework, a first net rendering and editing component can be forged readily. For more customized net editing capabilities, many JUNG behaviors need to be overwritten or to be used in custom classes.

3.1 Editor and Simulator

The most important aspects of RECONNET are reflected in its graphical user interface. There are two different areas for nets and rules, underlining the importance of both those constructs. The user can basically choose among four modes that can be operated in. Note in Fig. 2 the checkbox with the options: pick (Auswählen), insert arc (Kante einfügen), insert place (Stelle einfügen), and insert transition (Transition einfügen). In the pick mode, the user can click on nodes (i.e. a place or a transition) to select them to highlight them or edit its attributes like marking. Also transition labels and the renew function can be set by typing into the attribute table in the top middle. Nodes can be easily moved by drag-and-drop. When the user drag-and-drops onto the white space, the whole net is moved. By scrolling the mouse wheel the zoom function can be used. When right-clicking a node, it can be deleted (also its incident arcs) or a color can be assigned. Colors do not interfere with the graph matching or any logical function at all. All of this applies to the rule editing, also. The renew function is restricted to one of the three modes: id, that does not change the labels, being the identity function, toggle, that swaps the Boolean values true and false, and count, that increases an integer given as the label. In the arc mode, arcs can be added by drag-and-dropping among nodes of different types. In the place and transition mode, places or transitions can be added to the net by clicking onto the white space.

Adding a node (i.e. a place or a transition) into a rule needs to differ from the procedure for nets. For each node that is created in one of the rule’s panels, one or two corresponding node are inserted to one or two other panels of the rule. This ensures that there are only injective morphisms from the left -and right side to the interface. Moreover, a unique color is auto-generated and assigned to the new places illustrating the underlying morphism.
Figure 2: Screenshot of RECONNet’s GUI
This procedure also enables the user to quickly model the basic functions of a rule: By adding a node to the left part of the rule $L$, it automatically is added into the interface $K$, this leads to the deletion of the node when applying the rule. If a node is added into the interface $K$, it is added in both left and right parts of the rule to ensure being part of the matching subgraphs. Adding a node into the right-hand side $R$ of the rule also inserts the node into $K$, meaning this node is added when the rule is applied. The same treatment is used for arcs as well except the coloring.

Another important feature of ReCONNet is the simulation of nets, which can be done in different fashions. First of all there is a button to fire one transition. The transition is chosen randomly among those that are active. A transition is represented as a dark gray rectangle if it is activated and as a light gray rectangle if not. The follower marking is computed and the tokens are moved accordingly. The application of rules leads to the transformation of the net. The rule that is applied to the displayed net is chosen randomly among those who are selected in the left hand overview. Also the morphism is found in a non-deterministic fashion. If there is an injective match morphism, then the gluing conditions will be checked for the intermediate net. Then the resulting net is computed and displayed.

Additionally there are two more advanced simulation options - k steps (k Schritte) and running simulation - which can operate in three different modes: tokens (Nur Tokenspiel), transformation (Transformation) and both (Beides). The k steps options executes a definable amount of steps on the net. In the token mode only transitions are fired, in the transformation mode only rules are applied. In the both mode it is randomly chosen whether a transition is fired or a rule is applied for each step. Those three operations can be applied by k steps instantly, but also in a simulation running mode, which can be set to run fast or slow in 10 speed levels from 1 step every 2 seconds to basically as fast as the running system can offer.

### 3.2 Architecture and Persistency

The software is designed with component based architecture to allow refinement throughout the process of development. Thus its functionalities are divided into the five broad components Petrinet, Transformation, Engine, Persistence and GUI. While Petrinet and Transformation offer the more abstract functionalities of nets and rules, the Engine combines them to a more tool-specific interface, transforming data, checking correctness of user input, performing simulations and managing session data. On top is the GUI component that defines the displays and controls of nets and rules. Aside from that, the Persistence component is adjoined by the engine to load and store Petri nets, rules and simulation results to the local file system.

Since interoperability is a desirable feature, we have used for storing place/transition nets XML schemata as given by the PNML standard (e.g. in [HKPT10]). Parsing XML files is implemented via JAXB [JAX], the JAVA architecture for XML binding. It enables to access XML files as an object tree with special classes for each type of XML element. This method requires less high level custom code for parsing the XML, rather a set of classes similar to structs that need to be written or generated. One of its downfalls is the disability to partly load an XML file into the memory that might get problematic for large nets on computers with little memory. Saving nets and rules in XML files rather than a serialized form, enables the interaction with other tools via standardized formats, later on. For rules there are no XML schemata available in the PNML standard. Therefore we have developed the following approach for saving rules.
Each part of a rule is a net element, just as a Petri net. In addition it has a net type attribute which can have one of three values L, K and R. Also each net element in K has its own id which is shared with their respective counterpart in L and R. A node that occurs in all parts of a rule will appear in each three net elements of the rule with always the same attributes and always the same id. This id is used to encode the corresponding morphism form $L \subseteq K$ and $K \supseteq R$. In Fig. 3 in the appendix an excerpt of the XML-code of the rule ringAlarm from Fig. 2 is given that represents the rule’s left-hand side $L$.

4 Conclusion

To conclude this paper we discuss related and future work. Up to now there have been two tools that also implement some kind of reconfigurable Petri nets. MCRReNet [LO05] is a tool for the specification, modeling, simulation, and verification of concurrent systems that are subject to dynamic changes by using Marked-Controlled Reconfigurable Nets. In addition to the Petri nets it is equipped with a configuration graph, a labeled directed graph whose nodes are the configurations.

The Reconfigurable Object Nets (RON) Editor [BEHM07, BM08, BEMS08] integrates transition firing and rule-based net structure transformation of place/transition nets during system simulation. In contrast to our approach they are high-level nets with two types of token: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules). Firing of high-level transitions may involve firing of object net transitions, transporting object net token through the high-level net, and applying net transformation rules to object nets. Net transformations include net modifications such as merging or splitting of object nets, and net refinement. Nevertheless the RON Editor is capable of simulating reconfigurable place/transition nets by putting all place/transitions nets as high-level tokens on high-level object-net places, putting all rules on high-level net reconfiguration rule places, and connecting a rule place with a net place by a transition that triggers the application of a net transformation rule to an object net. Moreover the RON Editor supports negative application conditions [BM08] and an independence analysis of net transitions [BEMS08].

In [GN11] the Living Place Hamburg is modeled using Algebraic High-Level Nets with Individual Token, short AH LI nets, as well as rule-based transformation of such nets following the double pushout approach. In that thesis the focus was on the detailed modeling of the data types.

Since the scenario modeling in [Rei12] makes use of negative application conditions as well as capacities these are the obvious extensions that need to be implemented next. In [Pad12] the construction of a reachability graph for reconfigurable place/transition nets is given. Moreover, the modeled scenarios - due to negative application conditions and capacities – are bounded leading to bounded reachability graph and hence to the corresponding possibilities of model checking.

Acknowledgements: This tool is the result the student’s efforts in two courses given at the HAW Hamburg. We want to thank Nora Berg, Mathias Blumreiter, Andreas Jäckel, Tobias Markmann, Maximilian Meyer, Dmytro Oleynyk, Alexander Schulz, Oliver Willhöft, Pierre-Henri Nodin (fall semester 2011) and Steffen Brauer, Roman Genez, Till Gerken, Christian Götz,
Philipp Kühn, Thorsten Paech, Florian Reiter, Zabihullah Safai, Niklas Schreiber, Moritz Uhlig (fall semester 2010).

References


http://www.oracle.com/technetwork/articles/javase/index-140168.html

[JUN] Java Universal Network/Graph Framework.  
http://jung.sourceforge.net/ 


doi:http://doi.ieeecomputersociety.org/10.1109/QEST.2005.18


http://tfs.cs.tu-berlin.de/publikationen/Papers08/PEHP08.pdf

http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/reiter.pdf

http://docs.oracle.com/javase/6/docs/technotes/guides/swing/6.0/index.html
Appendix

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pnml
    <node size="50.0" type="rule" xmlns="http://www.pnml.org/version-2009/grammar/pnml">
        <net id="8" nettype="L">
            <page>
                <arc id="21" source="17" target="13">
                    <graphics/>
                    <dimension/>
                    <position/>
                </graphics>
                <inscription>
                    <text>undefined</text>
                </inscription>
            </arc>
            ...
            <place id="10">
                <graphics>
                    <color b="0" g="0" r="255"/>
                    <position x="68.0" y="23.0"/>
                </graphics>
                <initialMarking>
                    <text>1</text>
                </initialMarking>
                <placeName>
                    <text>with alarm</text>
                </placeName>
            </place>
            <place id="13">
                <graphics>
                    <color b="255" g="0" r="0"/>
                    <position x="281.0" y="215.0"/>
                </graphics>
                <initialMarking>
                    <text>0</text>
                </initialMarking>
                <placeName>
                    <text>awoken</text>
                </placeName>
            </place>
        </page>
    </net>
</pnml>
```

Figure 3: Left-hand side of rule ringAlarm as XML document
Proceedings of the
7th International Workshop on Graph Based Tools
(GraBaTs 2012)

Rooted Graph Programs

Christopher Bak and Detlef Plump

12 pages
Rooted Graph Programs

Christopher Bak and Detlef Plump

The University of York, UK

Abstract: We present an approach for programming with graph transformation rules in which programs can be as efficient as programs in imperative languages. The basic idea is to equip rules and host graphs with distinguished nodes, so-called roots, and to match roots in rules with roots in host graphs. This enables graph transformation rules to be matched in constant time, provided that host graphs have a bounded node degree (which in practice is often the case). Hence, for example, programs with a linear bound on the number of rule applications run in truly linear time. We demonstrate the feasibility of this approach with a case study in graph colouring.

Keywords: Graph programs, rooted graphs, time complexity, constant-time graph matching, graph colouring

1 Introduction

The bottleneck for using graph transformation rules in programming is the inefficiency of graph matching. In general, to match the left-hand graph $L$ of a rule within a host graph $G$ requires time $\text{size}(G)^{\text{size}(L)}$. As a consequence, linear graph algorithms are slowed down to polynomial complexity when they are recast as programmed graph transformation systems.

One way to speed up graph matching, going back to Dörr [5], is to equip rules and host graphs with distinguished nodes, so-called roots, and to match roots in rules with roots in host graphs. The same idea underlies Fujaba’s requirement that each method must have a “this” node at which graph matching starts [8, 12]. A related concept in GrGen are rules that return graph elements to restrict the location of subsequent rule applications [6].

Dodds and Plump [4, 2] have considered rooted graph transformation by using uniquely labelled nodes as roots. They show that graph matching can be achieved in constant time if rules have a connected left-hand graph and host graphs have bounded node degrees. In addition, they use rooted rules in a rule-based extension of C that allows to check the shape safety of pointer manipulations [3]. In this paper, we generalise the approach of [4, 2] from plain rules to programs in the graph programming language GP 2 [10]. We extend GP with rooted rule schemata and present a matching algorithm which deals with the label expressions in these schemata.

Our main contribution is to identify fast rule schemata, a large class of rooted conditional rule schemata, and to prove that they can be applied in constant time if host graphs have a bounded node degree. In practice, the latter assumption is often satisfied. For example, traffic networks, digital circuits and social networks usually have an upper bound on the number of edges attached to nodes. In Section 6, we apply fast rule schemata in a case study on graph colouring. We give a GP program which checks whether the input graph is 2-colourable and, if this is the case, colours the graph. We prove that this program runs in time linear in the size
of input graphs, demonstrating that rooted GP programs can achieve the time complexity of programs in imperative languages.

2 Graph Transformation

We first recall the graph transformation approach underlying GP, namely the double-pushout approach with relabelling [7], and then accommodate this framework to rooted graphs.

2.1 Non-rooted graph transformation

A (partially labelled) graph $G$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ where $V_G$ is a finite set of nodes, $E_G$ is a finite set of edges, $s_G$ and $t_G$ are functions that assign to each edge a source and a target node respectively, $l_G$ is the partial node-labelling function and $m_G$ is the total edge-labelling function. We write $l_G = \bot$ if $l_G(v)$ is undefined. Both node and edge labels are taken from a fixed label set $\mathcal{L}$. Unlabelled nodes are used in rules to relabel nodes (see below). There is no need to relabel edges because they can be deleted and reinserted with a new label.

A node $w$ is reachable from a node $v$ if $v = w$ or there are an edge $e$ and a node $v'$ such that $v$ and $v'$ are incident to $e$ and $w$ is reachable from $v'$. (Note that this defines undirected reachability.) An edge $e$ is reachable from $v$ if the source and target of $e$ are reachable from $v$. A graph is connected if every node is reachable from every other node.

Given graphs $G$ and $H$, a premorphism $g: G \to H$ is a pair of functions $g_V: V_G \to V_H$ and $g_E: E_G \to E_H$ that preserve sources and targets. That is, for all edges $e$ in $G$, $s_H(g_E(e)) = g_V(s_G(e))$ and $t_H(g_E(e)) = g_V(t_G(e))$. If $g$ also preserves labels, that is $m_H(g_E(e)) = m_G(e)$ for all edges $e$ and $l_H(g_V(v)) = l_G(v)$ for all nodes $v$ with $l_G(v) \neq \bot$, then $g$ is a morphism. A morphism whose node and edge functions are both injective and surjective is an isomorphism. If $g$ satisfies $g(x) = x$ for all nodes and edges $x$, then $g$ is an inclusion.

A rule $r = (L \leftarrow K \rightarrow R)$ is a pair of inclusions $K \to L$ and $K \to R$ where $L$ and $R$ are totally labelled graphs. We refer to $L$, $R$ and $K$ as the left-hand side, the right-hand side and the interface, respectively.

Given a graph $G$ and a rule $r = (L \leftarrow K \rightarrow R)$, an injective morphism $g: L \to G$ satisfies the dangling condition if no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$. In this case $G$ directly derives graph $H$, denoted by $G \Rightarrow_{r,g} H$ or just $G \Rightarrow_r H$, if $H$ can be constructed from $G$ as follows:

1. Obtain a subgraph $D$ by removing all nodes and edges in $g(L) - g(K)$.
2. Add (disjointly) the nodes and edges of $R - K$ to $D$, keeping all labels. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.
3. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is $H$.

Note that $H$ is specified only up to isomorphism, that is, every graph isomorphic to $H$ qualifies as a result of the rule application. Abstractly, a direct derivation can be defined by a pair of natural pushouts in the category of partially labelled graphs; we refer to [7] for this characterisation.
2.2 Rooted graph transformation

We extend the above definitions to include distinguished root nodes in both rules and host graphs. Our approach is to treat rooted graphs and root-preserving morphisms as “first-class citizens” instead of encoding roots with labels. Unlike [4, 2], we allow multiple roots in rule schemata and host graphs; this may be useful in applications with disconnected host graphs.

A rooted graph is a pair \( \langle G, P_G \rangle \) where \( G \) is a graph and \( P_G \subseteq V_G \) is a set of roots. A morphism \( g : G \to H \) is root-preserving if \( g(P_G) \subseteq P_H \). Note that rooted graphs (over some label set) and root-preserving morphisms form a category.

A rooted rule \( r = \langle \langle L, P_L \rangle \leftarrow \langle K, P_K \rangle \to \langle R, P_R \rangle \rangle \) is a pair of root-preserving inclusions \( \langle K, P_K \rangle \to \langle L, P_L \rangle \) and \( \langle K, P_K \rangle \to \langle R, P_R \rangle \) where \( L \) and \( R \) are totally labelled. Given a rooted graph \( G \) and a root-preserving injective morphism \( g : L \to G \) satisfying the dangling condition, a direct derivation \( G \Rightarrow r, g H \) is constructed as above and by defining \( P_H = (P_G - gV(P_L - P_K)) \cup (P_R - P_K) \). This construction can be characterised by a pair of natural pushouts in the category of rooted graphs and root-preserving morphisms (omitted for lack of space).

3 Rooted Graph Programs in GP

We extend the graph programming language GP with rooted programs. A complete definition of GP and its revised version GP 2 is given in [9, 10]. In this section, we describe GP’s most important features informally.

3.1 Conditional Rule Schemata

GP’s principal programming constructs are conditional rule schemata. These extend the rules of Subsection 2.1 with expressions in labels and with application conditions. For example, Figure 1 shows the declaration of a conditional rule schema bridge, where roots are depicted as nodes with bold borders. Only the left- and right-hand side of the rule schema are declared. By convention, the interface is the unlabelled and rootless graph consisting of the numbered nodes.

![Figure 1: Declaration of a conditional rule schema](image-url)

The top line of the declaration states the name of the rule schema and declares the variables that are used in the labels and in the condition. All variables occurring in the right-hand side and
in the condition must also occur in the left-hand side because their values at execution time are determined by matching the left-hand side with a subgraph of the host graph.

Each variable is declared with a type which is either `int`, `string`, `atom` or `list`. Types form a subtype hierarchy in which integers and character strings are basic types, both of which are atoms, which in turn are considered as lists of length one. In general, a label in GP is a list of atoms each of which is either an integer or a character string. Labels in host graphs do not contain expressions; they are fixed values in \((\mathbb{Z} \cup \text{Char}^*)^*\), where Char is the set of available characters.

Lists are constructed by the colon operator which represents list concatenation. For example, the label of node 1 on the left of Figure 1 stands for a list whose first element `a` is an integer or a character string, followed by a (possibly empty) rest list `x`. String concatenation is signified by the dot operator, as in the edge label `s.t` on the right of Figure 1. Labels in the right-hand side of a rule schema may contain arithmetic expressions such as `n*n` in node 3 on the right of Figure 1.

Besides having labels, both nodes and edges can be marked. Graphically, a marked node is shaded, and a marked edge is dashed. Marked items in rule schemata can only match marked items in host graphs. Vice versa, marked items in host graphs can only be matched by marked items in rule schemata. Marks are used as boolean flags and should not be confused with roots.

Rule schemata have an optional condition, declared with the keyword `where`. The condition is a boolean expression containing built-in predicates and functions, label expressions, and node identifiers. For example, the subexpression `not edge(1,3,s.t)` in Figure 1 demands that there must not be an edge in the host graph from node 1 to node 3 that is labelled with the string `s.t` (where `s` and `t` stand for the host graph labels of the edges in the left-hand side). To apply the rule schema according to a match of the left-hand side, the condition must evaluate to true for that match and its induced assignment of values to variables.

### 3.2 Programs

GP programs consist of a finite number of rule schemata and a command sequence which controls their application to a host graph (see Figure 3 for an example program). The main control constructs are: application of a set of conditional rule schemata \(\{r_1, \ldots, r_n\}\), where one of the applicable schemata in the set is nondeterministically chosen or otherwise the command fails; sequential composition \(P;Q\) of programs \(P\) and \(Q\); as-long-as-possible iteration \(P!\) of a program \(P\); and conditional branching statements `if C then P else Q` and `try C then P else Q`, where \(C\), \(P\) and \(Q\) are arbitrary command sequences.

To execute `if C then P else Q` on a state \(G\) (the current graph), first the condition \(C\) is executed on \(G\). If this produces a graph, then this result is disposed and \(P\) is executed on state \(G\). Alternatively, if \(C\) fails on \(G\), then \(Q\) is executed on \(G\). This behaviour makes it possible to encode complex tests in the condition \(C\) which do not alter the current state.

The command `try C then P else Q` also first executes \(C\) on \(G\) and, if this fails, executes \(Q\) on \(G\). However, if \(C\) produces a graph \(H\), then \(P\) is executed on \(H\) rather than on \(G\).

GP also allows to define (non-recursive) macros, which are command sequences represented by identifiers. For example, the program in Figure 3 contains the macro `colouring`. Macros are used for better readability but have no semantic significance.
4 A Matching Algorithm for Rooted Rule Schemata

We present a matching algorithm for rooted rule schemata which extends the corresponding algorithms in [4, 2]. The main difference is that instead of matching plain graph transformation rules, we have to deal with the syntax of GP 2-rule schemata. In particular, the algorithm must compare label expressions of the left-hand side with values in host graph labels and, besides finding matches of the graph structure, compute assignments of values to variables. These assignments are used both in evaluating the application condition of the rule schema and in calculating the labels of new and relabelled items when the rule schema is applied. Another extension to the previous algorithms is that we allow multiple roots in rule schemata and host graphs, while the cited papers assume a single root. Moreover, it is now possible to designate arbitrary nodes as roots whereas before a root had to be identified by a uniquely occurring label.

First we introduce some notation used in the algorithm. A \textit{partial premorphism} \( g: G \rightarrow H \) is a pair of partial functions \( g_V: V_G \rightarrow V_H \) and \( g_E: E_G \rightarrow E_H \) such that for each edge \( e \) in \( G \), if \( g_E(e) \) is defined then \( g_V(s_G(e)) \) and \( g_V(t_G(e)) \) are also defined and satisfy \( s_H(g_E(e)) = g_V(s_G(e)) \) and \( t_H(g_E(e)) = g_V(t_G(e)) \). We write \( \text{Dom}(g_V) \) and \( \text{Dom}(g_E) \) for the sets of nodes and edges on which \( g \) is defined. Given partial premorphisms \( f, g: G \rightarrow H \), \( f \) extends \( g \) by a node \( v \) if \( \text{Dom}(f_V) = \text{Dom}(g_V) \cup \{v\} \) and \( \text{Dom}(f_E) = \text{Dom}(g_E) \). Also, \( f \) extends \( g \) by an edge \( e \) if \( \text{Dom}(f_V) = \text{Dom}(g_V) \cup \{e\} \) and \( \text{Dom}(f_E) = \text{Dom}(g_E) \cup \{s_G(e), t_G(e)\} \). Given a rooted graph \( \langle L, P_L \rangle \) and \( p \in P_L \), an edge enumeration \( p \) for \( p \) is a list of edges \( e_1, \ldots, e_n \) such that \( \{e_1, \ldots, e_n\} \) is the set of all edges reachable from \( p \), \( e_1 \) is incident to \( p \), and for \( i = 2, \ldots, n \), \( e_i \) is incident to the source or target of some edge in \( \{e_1, \ldots, e_{i-1}\} \).

\begin{algorithm}
\begin{algorithmic}
\State \( A \leftarrow \{\langle h: L \xrightarrow{\text{par}} G, \emptyset \rangle \mid \text{Dom}(h) = \emptyset\}\)
\While {there exists an untagged root \( p \in P_L \)}
\State \( A_0 \leftarrow \{\langle h: L \xrightarrow{\text{par}} G, \alpha_L \rangle \mid h \text{ is injective and root-preserving, and} \)
\State \hspace{1cm} \text{there exists} \( \langle h', \alpha_L' \rangle \) in \( A \) such that \( h \text{ extends} h' \text{ by} p \}\)
\State \hspace{1cm} \text{tag} \( p \)
\State \hspace{1cm} \text{AssignmentUpdate}(A_0)
\For {\( i = 1 \) to \( p_n \)}
\State \( A_i \leftarrow \{\langle h: L \xrightarrow{\text{par}} G, \alpha_L' \rangle \mid h \text{ is injective and root-preserving, and} \)
\State \hspace{1cm} \text{there exists} \( \langle h', \alpha_L' \rangle \) in \( A_{i-1} \) such that \( h \text{ extends} h' \text{ by} e_p \}\)
\State \hspace{1cm} \If {\( s(e_p) \in P_L \)} \text{tag} \( s(e_p) \)
\State \hspace{1cm} \If {\( t(e_p) \in P_L \)} \text{tag} \( t(e_p) \)
\State \hspace{1cm} \text{AssignmentUpdate}(A_i)
\EndIf
\EndFor
\State \( A \leftarrow A_{p_n} \)
\EndWhile
\Return \( A \)
\end{algorithmic}
\end{algorithm}

Figure 2: Algorithm Rooted Graph Matching

Given a rooted host graph \( \langle G, P_G \rangle \), the left-hand side \( \langle L, P_L \rangle \) of a fixed rooted rule schema, and an edge enumeration \( e_{p_1}, \ldots, e_{p_n} \) for each \( p \in P_L \), the algorithm in Figure 2 computes all
matches of \( \langle L, P_L \rangle \) in \( \langle G, P_G \rangle \). The algorithm assumes that each node in \( L \) is reachable from some root. It incrementally constructs a set \( A \) of pairs of partial premorphisms \( h: L \xrightarrow{\text{par}} G \) and partial assignments \( \alpha_h \). By a partial assignment we mean a partial function \( \text{Var}(L) \rightarrow (\mathbb{Z} \cup \text{Char}^*)^* \), where \( \text{Var}(L) \) is the set of variables occurring in \( L \). The roots in \( L \) are tagged whenever they are matched; initially they are all untagged.

The algorithm calls the procedure AssignmentUpdate, which exploits that expressions in the left-hand side of a GP rule schema are constrained to prevent ambiguous variable assignments. Expressions must not contain arithmetic operators, more than one occurrence of a list variable, or more than one occurrence of a string variable in a single string expression.

Lists and strings are stored internally as doubly-linked lists with pointers \text{first} and \text{last} pointing to the first and last element. Hence the first and last element of a list or string, as well as the predecessor and successor of the current element, can be accessed in unit time. With this data structure, only the pointers \text{first} and \text{last} are needed when assigning a value to a list, atom or string variable, as the rest of the list or string can be accessed through the \text{next} and \text{prev} operators.

AssignmentUpdate is omitted for lack of space; we give a brief outline of its operation. The procedure iterates over its input, a set of pairs of partial premorphisms and partial assignments. For each pair \( \langle h, \alpha \rangle \), it iterates over all untested labels \( l \) in the domain of \( h \). Each \( l \) and corresponding host graph value \( h(l) \) are evaluated by a local procedure which will also update the partial assignment if \( l \) contains a variable and the two expressions can be matched. In particular, expressions containing a list variable or a string variable are tested by comparing the individual components (atoms or characters) that occur before and after the single variable. If all components match, then the variable has a unique mapping. This mapping is specified by assigning locations to the \text{first} and \text{last} pointers of the string or list variable. This is sufficient; the rest of the string or list can be accessed through the list operators as the value is stored as a doubly linked list in the graph data structure.

**Proposition 1** (Correctness of Rooted Graph Matching) The algorithm Rooted Graph Matching returns the set of all pairs \( \langle g, \alpha \rangle \) where \( g: L \rightarrow G \) is an injective root-preserving premorphism and \( \alpha: \text{Var}(L) \rightarrow (\mathbb{Z} \cup \text{Char}^*)^* \) is a total assignment such that \( g^\alpha: L^\alpha \rightarrow G \) is label-preserving.

Here \( L^\alpha \) is the graph obtained from \( L \) by replacing each variable \( x \) with the value \( \alpha(x) \). According to the semantics of GP 2 [10], \( g \) must be label-preserving after this replacement, that is, it must be a graph morphism \( L^\alpha \rightarrow G \). We omit the proof of Proposition 1 for lack of space.

## 5 Complexity of Rooted Rule Schemata

In this section, we analyse the complexity of the rooted graph matching algorithm and of applying a conditional rule schema with a given match. We make the following general assumption.

**Assumption 1** (Complexity model)

When analysing the complexity of rule schemata and programs, we assume that

1. rule schemata and programs are fixed, and
2. integer operations and character comparisons are computed in unit time.
The first assumption is customary in algorithm analysis where programs are fixed and running time is measured in terms of input size. In our setting, programs consist of fixed rule schemata and the input size is the size of a host graph and its labels. The second assumption is consistent with the uniform cost criterion for random access machines, the standard complexity model in algorithm analysis [1, 11].

Our matching algorithm assumes that each node in a left-hand graph is reachable from some root. This alone does not guarantee that rule schemata can be applied in time independent of the host graph. To achieve this, we need to impose some more restrictions on the form of rooted rule schemata. We will show that, under mild assumptions on host graphs, rule schemata of the following form can be applied in constant time.

**Definition 1** (Fast rule schema)
A rule schema \( L \Rightarrow R \) with application condition \( c \) is fast if

1. each node in \( L \) is reachable from some root,
2. \( L \) and \( R \) do not contain repeated list, string or atom variables, and
3. \( c \) does neither contain the edge predicate nor a test \( e_1 = e_2 \) or \( e_1 \neq e_2 \) where both \( e_1 \) and \( e_2 \) contain a list, string or atom variable.

The first condition ensures that matches can only occur in the neighbourhood of roots. The second condition makes it unnecessary to check the equality of lists or strings, or to copy lists or strings. The third condition rules out tests that require more than constant time.

Applying a conditional rule schema \( L \Rightarrow R \) to a host graph \( G \) requires several phases: finding a root-preserving match of \( L \) in \( G \) and constructing the induced variable assignment; checking the dangling condition and the application condition; removing items from \( L - K \); adding items from \( R - K \); and relabelling nodes. In the following we focus on the complexity of the matching phase because, in the worst case, it is far slower than the other phases.

We give the following lemma without proof due to lack of space.

**Lemma 1** Given a fast rule schema \( L \Rightarrow R \) and a host graph \( G \), the procedure AssignmentUpdate compares each label in \( L \) in constant time with the corresponding label in \( G \).

We can now show that fast rule schemata can be matched in constant time, provided that both node degrees and the number of roots in host graphs are bounded. The degree of a node \( v \) is the sum of the number of edges with source \( v \) and the number of edges with target \( v \).

**Theorem 1** The algorithm Rooted Graph Matching runs in constant time for fast rule schemata if there are upper bounds on the maximal node degree and the number of roots in host graphs.

**Proof.** Consider a fast rule schema \( L \Rightarrow R \) and a host graph \( G \). Let \( l \) be the number of roots in \( L \). Let \( b \) and \( r \) be upper bounds on the maximal node degree and the number of roots in host graphs respectively.

First we count the number of times the set of partial premorphisms \( L \xrightarrow{par} G \) is updated. We assume a data structure where adding either a node, an edge, or a node and an edge to an existing

---

1 We need to consider the size of labels because they can contain arbitrarily large lists and strings.
Rooted Graph Programs

morphism takes unit time. There are at most \( l \) iterations of the while loop and, within each iteration, at most \( m = |E_L| \) iterations of the for loop. Note that, by Assumption 1, both \( l \) and \( m \) are constants.

Consider the execution of the first iteration of the while loop. First, a single root from \( L \) is matched with all unmatched roots in \( G \). Since no roots have been matched yet, \( r \) partial morphisms are created. Then, in each iteration, either a single edge or an edge and a node is added to the domain of one of more morphisms in the current set. As node degrees in \( G \) are bounded by \( b \), no more than \( b \) additions can take place. This gives a worst-case running time of \( r + b|A_0| + b|A_1| + \ldots + b|A_{m-1}| \). The set \( A_0 \) contains at most \( r \) morphisms, \( A_1 \) contains at most \( br \) morphisms, etc. It follows that the running time is \( r + br + b^2r + \ldots + b^mr = r\sum_{i=0}^{m} b^i \).

Next, the second root of \( L \) is matched. Since one root in \( G \) has already been matched, the maximum size of the new morphism set is \( b^m r(r - 1) \). Hence, by the same argument as before, the maximal running time after the second iteration of the while loop is

\[
r \sum_{i=0}^{m} b^i + r(r - 1) \sum_{i=m}^{2m} b^i.
\]

After the \( l \)-th and final iteration of the while loop, the total running time is bounded by

\[
r \sum_{i=0}^{m} b^i + r(r - 1) \sum_{i=m}^{2m} b^i + \ldots + r(r - l + 1) \sum_{i=(l-1)m}^{lm} b^i.
\]

The procedure AssignmentUpdate is called after each update of the set of premorphisms. Each execution checks at most two labels for every premorphism in the set since on each update, at most two new items are added to the domain of the premorphism. Thus, by Lemma 1, it follows that the total execution time of Rooted Graph Matching is bounded by a constant factor of the above expression.

Given a match of the left-hand side of a fast rule schema, checking the application condition and the dangling condition, and deleting, adding and relabelling items can be done in constant time. Hence we obtain the following corollary of Theorem 1.

**Corollary 1** Fast rule schemata can be applied in constant time if there are upper bounds on the maximal node degree and the number of roots in host graphs.

**Proof sketch.** Consider again a fast rule schema \( L \Rightarrow R \) with condition \( c \) and a host graph \( G \). By Theorem 1, constructing a premorphism \( g: L \rightarrow G \) and induced variable assignment \( \alpha \) (or determining there is no such pair) requires only constant time. We need to prove that the remaining phases of rule schema application can be executed in constant time, too.

By Definition 1, the condition \( c \) is a boolean combination of subexpressions each of which is either (1) a relational operator applied to integer expressions, (2) a test \( e_1 = e_2 \) or \( e_1 \neq e_2 \) where \( e_1 \) and \( e_2 \) do not both contain list, string or atom variables, or (3) a type check \( \text{int}(e) \), \( \text{string}(e) \) or \( \text{atom}(e) \). Subexpressions of the first kind can be evaluated in constant time by (note that all expressions in \( c \) are of constant size). By the same assumption, tests according to (2) take only constant time because no comparisons are made between atom, string or list variables. Type
checks according to (3) can be done in unit time if the data structure for labels records type information suitably.

The dangling condition for an injective premorphism \( g : L \to G \) can be checked by comparing the degree of each node \( v \) in \( L - K \) with the degree of its image \( g(v) \). We assume a graph representation where nodes are stored together with their indegree and outdegree. This operation then takes time of order \(|V_L|\), a constant.

Given a match satisfying the dangling condition, removing the items in \( g(L - K) \) can be executed in time proportional to \(|L - |K||\). Similarly, the addition of nodes and edges takes time proportional to \(|R - |K||\).

Finally, relabelling a string or list only requires redirecting the pointers \( \text{first} \) and \( \text{last} \) to a particular label in \( G \). For string concatenation, two more pointer redirections are required to combine the two strings. There are at most \(|V_K|\) relabellings, so the time needed is proportional to \(|V_K|\).

The overall time complexity of a fast rule schema is largely determined by the number of roots in both the rule schema and the host graph. This is to be expected since the number of roots available for matching will increase the number of matches. Indeed, if all nodes were roots, then rooted matching would be identical to conventional graph matching. In practice, we aim to limit the number of roots. For example, in our case study in the next section, we use only one root in both rule schemata and host graphs.

## 6 Case Study: 2-Colouring

Vertex colouring has many applications [11] and is among the most frequently considered graph problems. We focus on 2-colourability: a graph is 2-colourable, or bipartite, if we can assign one of two colours to each node such that the source and target of each edge have different colours.

The GP program 2colouring in Figure 3 expects a connected and unmarked input graph \( G \) with atomic node labels and a single root. The program will either produce a 2-colouring for \( G \) by appending the integer 0 or 1 to each node label, or return \( G \) unmodified if no 2-colouring exists. For the rest of this section, by a rooted graph we mean a connected graph with a single root.

In Figure 3, the roots in rule schemata are depicted with a thick border. For notational convenience, the rule schemata \( \text{colour} \), \( \text{illegal} \) and \( \text{back} \) contain bidirectional edges. Each of these rule schemata actually represents a set of two distinct rule schemata with normal edges such that the edge direction is the same in the left- and right-hand side. For example, \( \text{colour} \) stands for the set \{\( \text{colour1} \), \( \text{colour2} \)\} where \( \text{colour1} \) and \( \text{colour2} \) differ only by the edge direction. When \( \text{colour} \) is called by the main program, \( \text{colour1} \) or \( \text{colour2} \) is selected non-deterministically and applied. If it is not applicable, then the other rule schema is attempted.

The program traverses a host graph in depth-first order, starting at the unique root which is coloured with 0. Whenever an edge is encountered whose source or target has a colour \( i \) and whose other node is uncoloured, then the other node is coloured with \( 1 - i \). If \( \text{colour} \) is no longer applicable, then the rule schema \( \text{back} \) moves the root one position back on the path of coloured nodes and the colouring process starts anew.
Upon termination of the macro `colouring`, the rule schema `unmarked` checks whether the root is unmarked or not. If not, then the rule schema `illegal` has detected an edge whose ends have the same colour and `stop` has marked the root. In this case the input graph is not bipartite and by the semantics of the `try` command, the input graph is returned as the application of `unmarked` failed. On the other hand, if the root is unmarked, then the whole graph has been correctly coloured.

**Proposition 2** (Correctness of `2colouring`) Given a rooted input graph $G$ with atomic node labels, the program `2colouring` returns a 2-coloured version of $G$ if $G$ is bipartite, otherwise it returns $G$ unchanged.

We omit the proof for lack of space. Note that each of the nine rule schemata can only be applied at the unique root of the current graph. Therefore the root needs to be moved around, which happens with both `colour` and `back`. However, care must be taken to prevent the root being moved back and forth between the same nodes forever. The program avoids this kind of looping by marking an edge only when an incident node gets a colour and unmarking this edge when `back` is applied to it (without altering the colours).

We now analyse the time complexity of `2colouring`. First note that all rule schemata are fast in the sense of Definition 1. Hence, by Corollary 1, we know that each rule schema takes only constant time on rooted graphs of bounded degree. Moreover, none of the rule schemata increases any node degree or the number of roots. Hence repeated rule schema applications in program runs preserve the assumptions of Corollary 1.

Then, to show that the running time of `2colouring` is linear in the size of the input graph, it suffices to show that the maximal number of rule schema applications is linear. This argument takes into account the linear overhead of the `try` command, which can be implemented by
copying the input graph and returning the copy in case the command sequence fails.

As to the number of rule schema applications, observe first that the rule schemata `init`, `unmarked`, `illegal` and `stop` can be ignored because each of them is applied at most once in a program run. Next, we notice that `colour` reduces the number of uncoloured nodes and `back` does not increase this number. Hence `colour` is applied at most \( n \) times, where \( n \) is the number of nodes in the input graph. Moreover, `back` cannot be applied more often than `colour` because the input graph is unmarked, only `colour` creates an edge mark, and `back` removes one edge mark. Thus, there are at most \( 2n \) applications of `colour` and `back`. Altogether, we have shown the following.

**Proposition 3** (Time complexity of `2colouring`) *On rooted input graphs with atomic node labels and bounded node degree, the running time of `2colouring` is linear in the size of graphs.*

This is significantly better than what can be achieved with unrooted programs. For, in the worst case of rule schema matching, even a clever algorithm requires at least linear time as it needs to search the complete host graph. Since each node of a bipartite input graph gets coloured, it follows that such a program has at least quadratic running time.

### 7 Conclusion

We have presented an approach for programming with graph transformation rules in which the bottleneck of graph transformation—the inefficiency of graph matching—is circumvented by using rooted rules which only match in the neighbourhood of host graph roots. Rooted graph transformation has been cleanly embedded in the framework of the double-pushout approach and has been extended to rule schemata in the graph programming language GP.

We have shown an algorithm which matches a large class of rooted conditional rule schemata in constant time, provided that host graphs have bounded node degrees. Our case study demonstrates that algorithms of practical importance, such as graph colouring, can be implemented with rooted GP programs whose time complexity is as good as that of programs in imperative languages. Moreover, we have demonstrated that due to the simplicity of GP and its semantics, the correctness and complexity of rooted graph programs is amenable to formal analysis. Essentially, because fast rule schemata can be applied in constant-time, to show that a program with fast rule schemata has a certain time complexity \( T \), it suffices to prove that the maximal number of rule schema applications is of order \( T \).

In future work, we will consider alternative sufficient conditions that make rooted programs fast. For example, in [4] it is shown that graph transformation rules can be applied in constant time if the outdegree of nodes in host graphs is bounded and the left-hand sides of rules contain a directed path from the root to each node. A corresponding result should hold for fast rule schemata if each node in a left-hand side is reachable from some root by a directed path.

We will also aim at confirming our theoretical results by implementing a rooted version of GP and comparing the performance of graph programs with that of programs in traditional programming languages.
Bibliography


