Efficient and Usable Model Transformations

Māris Jukšs
Supervisors: Clark Verbrugge, Hans Vangheluwe

Doctor of Philosophy

School of Computer Science

McGill University

Montreal,Quebec

2017-12-15

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

DEDICATION

To my parents and Mehrnoosh.

ACKNOWLEDGMENTS

I would like to acknowledge both of my supervisors Prof. Clark Verbrugge and Prof. Hans Vangheluwe for believing in me, providing support, guidance and advice.

I would like to express my gratitude for fruitful collaborations and discussions to MSDL members and in particular to Dr. Levi Lucio, Dr. Joachim Denil, Dr. Bruno Barroca, Dr. Sadaf Mustafiz and Simon Van Mierlo. I would also like to express my gratitude to the NECSIS project that was the main source of my funding.

I would like to thank Prof. Daniel Varró and Dr. Maged Elaasar for their advice and collaboration resulting in publications relevant to this thesis.

For all the help and learning I am grateful to McGill School of Computer Science and specifically to Prof. Jörg Kienzle and Prof. Bettina Kemme for being on my graduate progress committee. I would also like to acknowledge my undergraduate advisor Prof. Oksana Nikiforova from Riga Technical University for her continued and valued support.

Finally, I am very grateful to my family, my parents Janis and Tatiana Jukss for all their sacrifices and to my wife Mehrnoosh Azodi who was alsways there for me through difficult times.

ABRÉGÉ

L'adoption généralisée de l'ingénierie dirigée par les modèles dépend de la disponibilité d'outils et de techniques efficaces et utilisables prenant en charge les transformations de modèle (TM). Dans cette thèse, nous abordons l'efficacité de TM à travers des techniques qui exploitent mieux la localité dans les transformations, et la facilité d'utilisation en adressant le manque important et actuel de support d'outils pour le débogage de tels systèmes.

L'efficacité dans de nombreux systèmes de programmation est basée sur la propriété fondamentale de la localité de calcul, souvent exprimée en utilisant une certaine notion de portée. Dans le contexte du modèle, et en particulier de la transformation graphique, l'utilisation de la portée peut présenter deux avantages : premièrement, une expression plus naturelle de la localité d'application de transformation, et deuxièmement, une réduction du nombre de candidats correspondants. Dans cette thèse, nous présentons deux approches de portée. La première approche fait de la portée un citoyen de première classe dans les transformations de modèles basées sur des règles en définissant un formalisme de portée avec des modifications associées à la spécification de la règle de transformation. Par conséquent, un ingénieur TM peut manipuler et raisonner sur les étendues dans les règles de transformation. La deuxième approche vise à tirer parti de la localité dans TM sans la contribution de l'ingénieur. Nous proposons ici une conception probabiliste, de boîte noire à l'exécution, qui observe et apprend des transformations au fur et à mesure de leur exécution, en effectuant des prédictions sur les étendues d'application TM possibles

dans le modèle d'entrée.

Le support d'outils est également essentiel pour utiliser efficacement les systèmes de transformation de modèles de haut niveau. Les travaux dans ce domaine se sont principalement concentrés sur les éditeurs frontaux, visuels ou d'autres domaines spécifiques, laissant la tâche tout aussi importante de débogage aux débogueurs traditionnels basés sur le code et fonctionnant sur le code généré ou le traçage via l'implémentation de bas niveau d'un système de transformation. Nous abordons ici ce problème en décrivant une approche en couches du débogage, en mappant les opérations de débogage familières à différents formalismes, ainsi que les transitions entre eux. Notre conception permet un débogage transparent à travers les différentes abstractions entrelacées communes aux transformations de modèles, permettant à un ingénieur TM de déboguer dans un formalisme approprié à chaque niveau de débogage. Notre approche présente l'avantage supplémentaire de prendre en charge à la fois le débogage traditionnel et impératif ainsi que les approches déclaratives fondées sur des requêtes.

Nos deux approches générales sont démontrées par des implémentations non triviales, validant les conceptions, et nous permettant d'explorer le comportement et d'assurer la faisabilité de nos techniques. La création d'outils pratiques et bien fondés est un élément important de la poursuite de l'amélioration des problèmes d'utilisabilité dans les transformations de modèles.

ABSTRACT

Wide adoption of Model-Driven Engineering (MDE) depends on the availability of efficient and usable tools and techniques supporting Model Transformations (MTs). In this thesis, we address MT efficiency through techniques that better exploit locality in transformations, and usability by addressing the important, current lack in tool support for debugging such systems.

Efficiency in many programming systems is based on the fundamental property of locality of computation, often expressed using some notion of scope. In the context of model, and in particular graph transformation, the use of scope can present two advantages: first, more natural expression of transformation application locality, and second, reduction of the number of match candidates, promising performance improvements. In this thesis, we present two scope approaches. The first approach makes scope a first-class citizen in rule-based model transformations by defining a scope formalism with associated modifications to the transformation rule specification. As a result, a MT engineer can manipulate and reason about scopes within the transformation rules. The second approach aims at leveraging locality in MT without the engineer's input. Here we propose a runtime, probabilistic, black-box design observing and learning from the transformations as they are executed, making predictions to the possible MT application scopes within the input model.

Tool support is also essential to effectively using high-level model transformation systems. Work in this area, however, has mainly focused on front-end, visual or other domain-specific editors, leaving the equally important task of debugging to

traditional code-based debuggers operating on generated code or tracing through the low-level implementation of a transformation system. Here we address this issue by describing a layered approach to debugging, mapping familiar debugging operations to different formalisms, as well as the transitions between them. Our design allows for seamless debugging through the different, interleaving abstractions common to model transformations, allowing a MT engineer to debug in a formalism appropriate to each level of debugging. Our approach has the additional advantage of supporting both traditional, imperative debugging as well as declarative, query-based approaches.

Both of our general approaches are demonstrated through non-trivial implementations, validating the designs, and allowing us to explore behaviour and ensure feasibility of our techniques. Building practical, well-grounded tools is an important part of continuing to improve usability concerns in model transformations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure page

Chapter 1
Introduction

Model Transformations (MTs) are at the heart of Model Driven Engineering (MDE) [89]. In MDE and multi-formalism modeling contexts, models are primary artifacts used to describe complex systems at different levels of abstraction, using the most appropriate formalisms [71]. The benefit of using models at various stages of system development is a reduced conceptual gap between the problem domain and the resulting code-based implementations. The model transformations are then used to specify how the models are manipulated to produce other models or the final code-based solution.

Therefore, efficiency, scalability, and usability are of significant concern in the application of model transformation systems to industrial MDE problems. Scalability concerns in terms of graph matching magnify greatly with the input graph size. On the other hand pattern matching is the most expensive part of the graph-based model transformation system, thus efficiency is highly dependent on the capability of the subgraph matching algorithms, where the underlying subgraph isomorphism problem is known to be NP-complete [18]. To alleviate this, in this thesis, we propose to improve the pattern matching efficiency by using domain-specific heuristics based on locality, grouping or *scopes* in model transformations.

From the usability point of view, debugging of MTs is as important as debugging of general code-based programs. Because the MTs can be complex mixing declarative

and imperative constructs, engineers may require comprehensive debugging facilities helping them to find hidden bugs responsible for incorrect model manipulations. Hence, in this thesis, we aim to address the debugging of MTs with respect to the MT schedule, rule applications and the input model being transformed that would benefit tool builders and transformation engineers. We now proceed with a more detailed introduction to scope and MT debugging.

Scope. Grouping, modularity, or locality is a common approach to dealing with complexity and performance used throughout computer science and software engineering. Optimizing compilers for general purpose programming languages (GPL) may use locality or scoping to confine the optimizations to a part of the program. Optimizations, for example, can have local, intra-procedural, or inter-procedural scopes. Modular systems design and modular programming, in particular, aim to produce systems by grouping independent components resulting from systems subdivision into modules. In the context of MTs, scoping or locality is typically a transient effect of computation. For example, a *pivot graph* [1], a subgraph resulting from a rule application and passed to a subsequent rule execution possibly through rule parameters [84]. Locality can be an integral feature of the model, such as the subtree of an abstract syntax tree (AST) in term rewriting systems [15]. In this thesis, we propose to make scope a first-class citizen in MTs and harness the possible performance benefits this entails. Our approach aims to reason about scope (locality) in MTs at the level of models being transformed. In situations where scoping (locality) is not an integral part of the model we can create persistent scope avoiding repeated computations and thus possibly improving performance.

We approach scope in two ways. First, we introduce *static scope* concept that is applied to the input model resulting in the *scoped model* (graph). Formally, the scoped graph represents a persistent relationship between scope and the model elements. This relationship can be stored and reused. In addition, static scope can be used to encode additional domain information on top of the existing domain specific model because of its hierarchical nature. Static scope is created by a transformation engineer within the model transformation rules and as a performance enhancing measure is used to constrain the pattern matching routine to parts of the graph. Static scope added to the input model, in certain situations, creates favorable conditions for pattern matching in *local search-based* approaches such as *search plans* [43, 107].

Although static scope is an effective way of dealing with locality, it does require the conscious effort of an engineer who modifies MT rules to define scope. Our second scope approach, *dynamic scope*, aims at discovering scopes in model transformations automatically. This also facilitates contexts in which the model transformation may not be available for static scope application. This can happen when the MT is compiled or hides proprietary, sensitive information. Therefore, instead of manually creating scope as a transformation performance optimization, we attempt at discovering scopes automatically in the input model during transformation. We observe the transformation activity at runtime and use that to predict the areas (scopes) where transformation will be active, constraining, and thus reducing the match effort to those designated areas. We employ a temperature-based approach to tracking transformation activity in conjunction with machine learning to restrict scoping areas even further. This technique is also intended to be primarily used with the local

search-based approaches and simulation-like transformations where the first match discovered is sufficient as our approach is optimistic and requires a fall back step to guarantee a match discovery.

MT debugging. Debugging is an essential feature in most programming languages. The user can typically step through program execution at the code or machine instruction level. In the context of MTs, we are dealing with a complex system consisting of different abstractions. It is not always possible or desirable to transform the MT model into a textual representation for debugging. Therefore, we need comprehensive debugging support in MTs without additional conversion into a debuggable representation.

The complexity of MT debugging is at least partly due to the existence of multiple layers of MT execution stack. The stack consists of several layers utilizing different formalisms at the different levels of abstraction. For example, we can distinguish the modeled part, where the MT and the input/output models are concerned. The MT specification language deals with scheduling the rule execution and can be graphical or textual. Inside the schedule, we discover MT rules containing patterns. These declarative patterns, in turn, can contain imperative, textual action code.

The MT operates on the input model at another, lower level of the MT execution stack that contains pattern matching and application routines. This level is a complete departure from the formalized specification of the modeled levels of the stack. Here we find the implementation specific routines often hidden from the user but necessary for the understanding of how a MT works and to find potential problems: bugs may occur at any level of the MT stack, or between levels. We want to

debug this stack in a systematic way with an aim to advance the debugging support for MT tools, contributing to the wider adoption of MTs in the MDE context.

In this thesis, we adopt a structural, hierarchical, and item-based view of navigating a general debugging target using debugging commands. We apply this view to MT debugging. We make use of the MTs themselves to model the interaction with a debugger in order to realize automated debugging scenarios. These scenarios are intended to automate the debugging process in situations where the human interaction is not necessary/desirable. The benefit of explicitly modeling such interaction is the reuse and analysis of the solution and its exchange between engineers. In addition, the MT engineer stays in the context of MTs he or she intends to debug. In this debugging language, we also describe declarative queries that utilize the formalisms that are taken from the MT being debugged. This reduces the cognitive gap between the MT and MT debugging. Finally, our scope concept can be beneficial in the debugging language to indicate the areas of application for the declarative queries specified in the debugging rules.

We evaluate the feasibility of our approach in three prototype implementations and present our experience. We use a modeled approach to specifying the main part of our debugger. The modeled solution demonstrates a good degree of reuse in porting the initial implementation to another MT tool. Although these are prototype implementations, our experience demonstrates that design is feasible, and applicable to different MT system designs.

1.1 Contributions

In this section we outline the contributions pertaining to this thesis. The first part of the contributions is dedicated to the static and dynamic scopes and addresses MT efficiency. The second part deals with MT debugging aiming to improve MT usability. We leave more detailed contribution description in each topic's respective chapter.

Contributions addressing MT efficiency:

- We propose to treat the common notions of scoping, locality, and grouping as first class citizens in MTs. Our static scope concept, designed to capture these notions allows the transformation engineer to create, modify, and store these abstractions in a general way.
- To enable this we describe and formalize a modified MT rule structure, permitting scope creation and utilization in MTs. Based on a prototype evaluation, we demonstrate that scoping in the MT context has a potential to improve MT efficiency by constraining pattern matching effort to the particular parts of the input model. In addition, scope fits well into the search plan-based pattern matching [6, 107].
- Finally, we demonstrate an automatic scope discovery technique that does not require the input of a MT engineer. We propose to learn from the MT by observing its effects on the models and make automatic decisions whether to include the model elements into a scope of interest. The scope can then be used for constraining the matching efforts to improve performance for example.

Contributions addressing MT usability:

- We propose a structured view of a debugging process. This view deals with navigable items located at the hierarchical levels within a debugging target. The structured view specification does not impose a strict requirement on what these items are, except that they expose and represent something pertinent to the debuggee execution. We define debugging operations that allow us to move between the levels and sequentially step through debugging target items.
- We apply this view to a debugging of MTs. The view abstracts the MT process as a hierarchical stack of heterogeneous items, allowing for a seamless transition between the stack levels.
- We tailor a MT specification language to create a debugging scenario that essentially models a user interaction with a live and interactive debugger. This keeps the user in the MT context providing the benefits of a declarative specification reusing the DSL originally present in MT being debugged.

1.2 Publications

Below is the list of publications relevant to this thesis. All works were published in conjunction with my supervisors who provided supervisory input.

- "Scope in Model Transformations" - Māris Jukšs, Clark Verbrugge, Maged Elaasar, Hans Vangheluwe. Accepted for publication in Software and Systems Modeling (SoSyM) Journal in 2016. This paper presents the concept of scope and in particular addresses the topic of static scope. In this paper, I developed the concept of scoped model transformations including its syntax and semantics and demonstrated the prototype implementation and evaluation of the concept. Maged Elasaar contributed a section on scope concept mapping to QVT.

- "Dynamic Scope Discovery for Model Transformations" - Māris Jukšs, Clark Verbrugge, Dániel Varró, Hans Vangheluwe. International Conference on Software Language Engineering (SLE) 2014. This paper develops the scope concept further with runtime scope discovery. I developed the method, prototype and performed the evaluations. Dániel Varró contributed to the background and related work sections.
- "Transformations Debugging Transformations" - Māris Jukšs, Clark Verbrugge, Hans Vangheluwe. The 1st International Workshop on Debugging in Model-Driven Engineering@MODELS 2017. This paper develops structured view of the debugging and its application to the MT stack. The paper demonstrates the initial evaluation and introduces the debugging language and scenarios.

The following are the publications that were realized during the course of my PhD and were not directly related to the topics in this thesis or were omitted from the thesis.

- "Towards a Unifying Model Transformation Bus." Māris Jukšs, Bruno Barroca, Clark Verbrugge, Hans Vangheluwe. MPM workshop at MoDELS conference 2015. In this paper, in essence, we present a modeled solution to data exchange between programs joined by a bus architecture. Bruno Barroca contributed introduction section, collaborated with me on the bus specification language and the concept.
- "Search-Based Model Optimization Using Model Transformations." Joachim Denil, Māris Jukšs, Clark Verbrugge, Hans Vangheluwe. SAM conference 2014.

This paper discusses MT use to perform search-based optimizations in design space exploration problem. My contributions were: a search plan pattern matcher implementation used for performance evaluation of the approach and the performance data collection, analysis and plotting.

- "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains." Levi Lucio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, Māris Jukšs. SDL Forum 2013. This paper introduces the formalism transformation graph and process model describing the models and possible transformations between them. This eliminates the uncertainty in MDE processes and permits generation of transformation chains. I contributed to the AToMPM prototype permitting the automated execution of the transformation chains.

1.3 Thesis Roadmap

In this section, we present the outline of this thesis. We begin, in Chapter 2, with an introduction to modeling and model transformations. We discuss the optimization of model transformations and introduce *search plans*, a local search-based pattern matching approach. The chapter is concluded with an overview of relevant MT tools from the point of view of efficiency and usability in terms of debugging.

In Chapter 3, we introduce our scope contribution. We begin with the *static scope* and its formal definition. We then introduce scope syntax for use in MT rules and the semantics of scoped rule application. In order to evaluate scope, we use several examples also described in this chapter. Our scope concept can be implemented in other tools which we discuss in the context of our examples. The chapter is finalized with performance evaluation and discussion of the results.

Another part of scope contribution, a *dynamic scope*, is presented in Chapter 4. We introduce the approach that discovers scope in the input model by observing MT activity. We conclude the chapter with an evaluation of the approach and discussion of the results.

The usability aspect in terms of MT debugging is discussed in Chapter 5. We propose a structured view of a general debugging process and apply it to the debugging of model transformations. We show how we can use the MT rules to model interaction with a debugger and present debugging scenarios. We conclude the chapter with prototype evaluations of our approach. This thesis is finalized with Chapters 6 and 7 addressing related work and final conclusions respectively.

Chapter 2
Background

Our work depends on a number of specific technologies and tools. In this section, we give the essential background information that is necessary for understanding our work. We begin by introducing models and the model transformations (MT). We then discuss local search-based pattern matching and finalize this chapter by discussing a selection of MT tools in terms of efficiency and usability, such that we are able to draw parallels to our own work addressing these two issues.

2.1 What are models?

The abundance and complexity of real world or conceptual systems creates the need for a concise and pragmatic way of representing or modeling them. According to Kühne: "*A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.*" [51]. For a further explanation, we continue with the features of the model given by Stachowiak [91]. The model needs to be based on an original system, typically ensured with a *mapping feature*. In practice, the original system may not be present. This is usually the case in a software modeling context where the model represents an idea or a concept going to be implemented. A model has a *reduction feature* that abstracts away unnecessary details of the system. A human nervous system, when modeled, may not need to consider digestive organs, for example. Finally, a model has a *pragmatic feature*. A model created to reflect a particular system could be used in place of that system, provided that the model

is adequate and correct [51]. This is true in the case of a model-based simulation. Instead of using the actual, physical object in crash tests, for example, a computer-based simulation can be used resulting in the cost-saving benefits.

Typically, the main concepts of a system or a problem domain are captured by an associated metamodel (MM), together with their attributes and relationships thus defining the *abstract syntax* of a corresponding *domain-specific language* (DSL) [1] . The DSL is a language tailored to a specific problem domain. In contrast, a general purpose language (GPL) is suitable for a wide range of applications. According to Mellor et al., a metamodel is a model of a modeling language defining structure, semantics and constraints [64]. The metamodel then describes all possible models that can be generated according to the said MM. To represent our DSL we can define a *concrete syntax* based on the abstract syntax of the DSL. The concrete syntax defines how the DSL is presented, whether in textual or graphical forms. The elements of the concrete syntax are created such that they represent abstract syntax elements. This typically means a many-to-one relationship allowing for several concrete syntaxes for a single abstract syntax. In Figure 2–1 we demonstrate an example DSL tailored for Petri Net (PN) modeling. The language MM and the abstract syntax is defined using the UML class diagram language. Two possible, graphical concrete syntaxes are shown above and below the MM. Each concrete syntax element is mapped onto the abstract syntax element using dashed lines. In this case, the difference is in the presentation of the transition. A PN model conforming

[1] In this thesis we use the terms DSL and modeling language interchangeably.

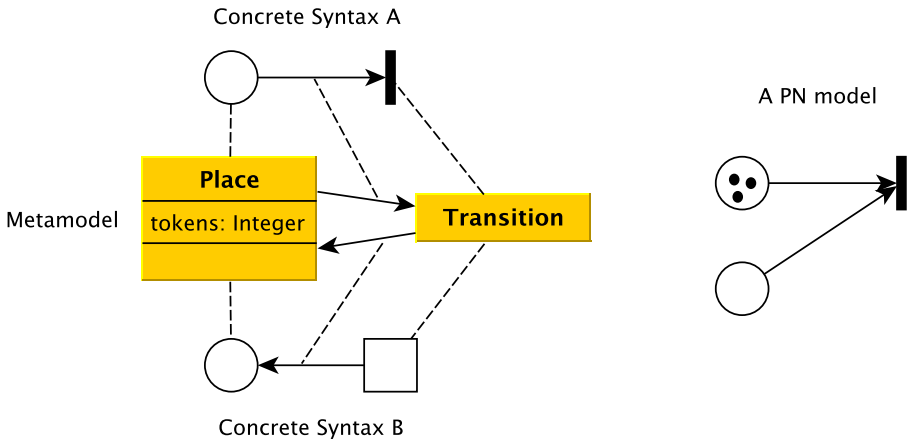

Figure 2–1: The Petri Net DSL with its abstract syntax metamodel and two graphical concrete syntaxes. Dashed lines represent a mapping between concrete and abstract syntax elements. A PN model example is shown on the right.

to the defined MM is also shown on the right. Another example of a graphical notation is the well known unified modeling language (UML). Program source code, such as Java, an example of a textual notation, can be considered a model as well. The code is modeled by its abstract syntax tree (AST) with its language grammar being the metamodel. Finally, to discover more about models and metamodeling we invite the reader to an article by Kühne [51].

In model-driven engineering (MDE) process, models are primary artifacts used to describe systems at a high level of abstraction. Typically, model transformations are then used to manipulate models with an aim of reducing the level of abstraction of models until an executable code is derived (semi)automatically. There exists a variety of MT approaches and a MT can be implemented using a general-purpose

programming language. In this thesis, however, we are concentrating on the declarative, *rule-based* MTs and the models in the MT systems will be represented as graphs. The following section is an introduction to MTs.

2.2 What are Model Transformations?

The definition from the Object Management Group (OMG) states that the MT is "*the process of converting one model to another model of the same system*" [70]. The definition of a model transformation given by Syriani [93] explains it further, "*the model transformation is an automatic manipulation of models with a specific intention*". The second definition clarifies the first in that the model transformation is an automated process and is associated with an intention. An example of an intention is generation of the GPL source code from a DSL model.

Model transformations can be used for many purposes. Here we continue with some generic examples originally compiled in [93] as a primer to MTs.

- **Query** - originates from data management in databases. A query transformation, provides a view over the model or the repository of models. There is no modification of the model and the model is accessed in a read-only fashion. A *restrictive query* for example reveals all, none, or some elements of the model. This is useful, for example, in the context when two observers of the same model M will see the different views of M based on the access rights of the observers.
- **Synthesis** - a transformation from a higher level specification to a lower level specification. A model-to-code is a specialized case of the model-to-model

transformation and a code generation from the UML diagrams is one such example.

- **Reverse engineering** - is the opposite to synthesis. Here, the higher level specification is extracted from a lower level one. For example, a transformation from source code to UML class diagrams.
- **Translational semantics** - the translational semantics is a model-to-model transformation defining the semantics (the meaning) of the source DSL in terms of the target DSL. This is necessary when the original semantics are not formal enough to perform mathematically rigorous reasoning about the model. For example, one can define by transformation, the semantics of a modeling language describing a concurrent system. This can be achieved by transforming the concurrent system model to an equivalent Petri Net (PN) model. The PN semantics are mathematically well defined and may be used to explore a variety of reachability problems, such as detection of a deadlock, a property useful in a concurrent system domain.
- **Simulation** - a model transformation in this case updates the state of the modeled system. For example, the PN modeling a checkout queue in a store is simulated by a transformation discovering enabled transitions, firing them and propagating the PN tokens. In software language engineering terms, the transformation defining the simulation can also serve as operational semantics.
- **Optimization** - a transformation improving operational qualities of the model while preserving semantics of the model. For example, transforming a model

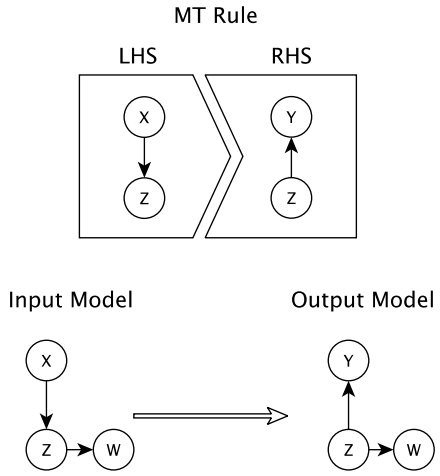

Figure 2–2: A typical rule from rule-based model transformations and its effect on the input model resulting in the output model.

incorporating a single linked list structure to a double linked list in order to facilitate deletion operations on the list.

We now proceed to describe MT rules, an important element in the rule-based MTs. A **MT rule** is a declarative construct, operating on an input (source) and an output (target) model. The rule consists of *pre-condition* and *post-condition* parts that are also called the left hand side (*LHS*) and the right hand side (*RHS*) respectively. In Figure 2–2 we present a typical MT rule, the input model to be transformed and the output model after executing the rule. The input model is changed with the removal of a node $\textcircled{X}$ and its incident edge, the addition of a node $\textcircled{Y}$ and creation of an edge according to the specification in the rule.

The *LHS* and the *RHS* of the rule contain patterns. The patterns can be presented in a textual or graphical form. As shown in Figure 2–2, the patterns are represented graphically using concrete syntax of a *transformation language* that is

based on the concrete syntax of a DSL being transformed (labeled circles connected with directed edges). In this case, the patterns are also represented as graphs. In other transformation systems the patterns can be defined as terms for tree representations of models as in Stratego [15] or strings for the template based transformations as in Xpand[2] .

Rule application. The pre-condition of the rule must be satisfied in the input model before a modification to the model according to the post-condition is made, and which would result in the output model. Therefore first, an occurrence of the *LHS* pattern in the input model is found by means of subgraph pattern matching. This establishes the *binding* between the pattern elements and the input model. Then, the occurrence (a subgraph within the input model) of a *LHS* pattern is modified according to the *RHS* of the rule.

The rule application steps above have a theoretical foundation in graph grammars and graph transformation theory. One of the them is an *algebraic graph transformation* approach based on a category theory and pushouts on the category of graphs [27]. Algebraic graph transformations can be defined using Single-Pushout (SPO) or Double-Pushout (DPO) approaches. In Figure 2–3 we show, with a kind of pushout diagram, how the DPO approach can be used to achieve the input model manipulation shown in Figure 2–2. The arrows represent graph morphisms.

We continue with a simplified explanation of the DPO approach. In DPO, a transformation rule or a production p consists of L, K, and R graphs. The L and R

[2] `http://wiki.eclipse.org/Xpand`

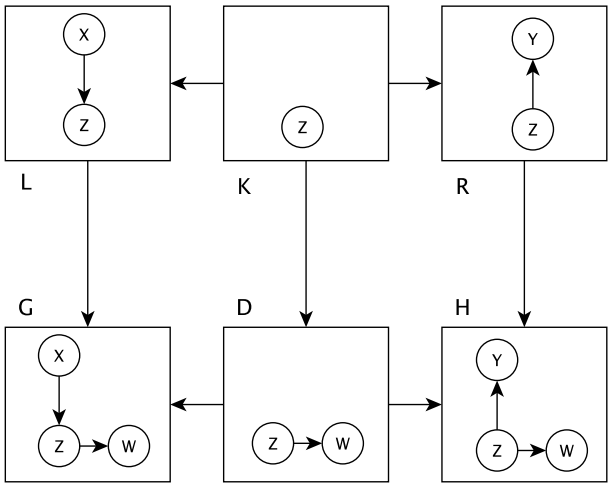

Figure 2–3: DPO approach for the rule application in Figure 2–2

graphs are the *LHS* and *RHS* parts and K is the *gluing graph* sharing the common elements with *LHS* and *RHS* (an intersection between two graphs). In essence, the K represents the graph that is not modified and that has to exist in order to apply the rule. Once the match of L in G (a graph morphism represented with an arrow in Figure 2–3) is found, the graph elements in L missing from the K will be deleted from G resulting in the context graph D. The context graph must satisfy a *gluing condition* in that it must be a valid graph without any dangling edges. The resulting graph H is achieved by gluing the graph D with the elements in R not found in K. The SPO approach is different from the DPO in that a production does not have a gluing graph K. This results in the absence of a context graph in the process of rule application. In addition, the handling of the dangling edges is more relaxed. The dangling edges are removed and do not inhibit rule applications.

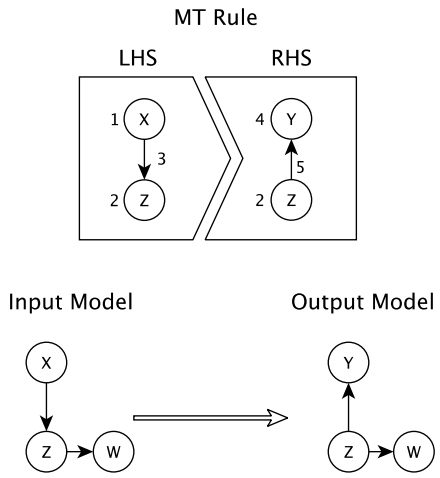

Figure 2–4: An example rule from Figure 2–2 with unique labels added.

The patterns in graphical syntax can be represented in a traditional (as in Figure 2–2) and compact notations. In a traditional notation, the *LHS* and *RHS* parts are clearly distinct. The pattern elements have unique labels associated with them which allow transformation engine to decide which elements need to be added or removed from the input model. For example, if a label present in the *LHS* pattern is missing in the *RHS* pattern then the pattern element (with a label in *LHS*) binding in the input model will be deleted. We demonstrate this with our example rule augmented with unique labels in Figure 2–4, the match of the pattern elements 1 and 3 in the input model are deleted and the elements with labels 4 and 5 are created.

In a compact notation, as used in FUJABA [30], elements to be created or destroyed are explicitly marked. We show a mockup of such a rule in Figure 2–5 and mark the elements to delete in red and the new elements to create in green.

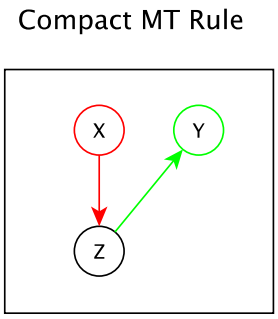

Figure 2–5: A rule in compact notation, equivalent to the rule in Figure 2–2.

Finally, the MT rule can have Negative Application Conditions (NAC). Similar to the LHS and the RHS, a NAC contains a pattern. The NAC pattern is matched in the input model, and if it is found the rule cannot be applied. We demonstrate the syntax of the NAC part of the rule in Figure 2–6. In this case, the rule is not applied because the NAC pattern in matched in the input model. The model therefore remains unchanged.

We continue with some notable MT features, such as rule application control (RAC) and tracing. RAC relates to two issues, one is where to apply the rule in the input model and another one is how to execute multiple rules. Tracing relates to keeping track of changes the MT is causing.

2.3 Model Transformation Features

MTs form a complex domain with an array of possible features pertaining to MT execution and specification. A good classification of those features was necessary and was created by Czarnecki et al. [23]. Here we present the necessary basics.

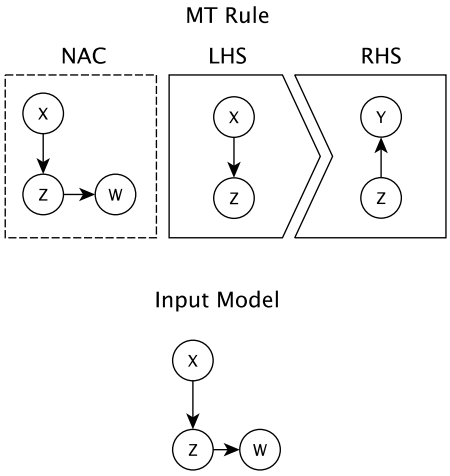

Figure 2–6: A typical rule from rule-based model transformations with added *NAC* pattern. The presence of the *NAC* pattern prohibits rule application and the input model is unchanged.

The rule presented in the previous sections is a minimal transformation. It designated the "smallest transformation unit" (TU) in Czarnecki et al.'s terminology. The TUs are defined over one or several domains or meta-models. The source and target meta-models can be different (*exogenous* transformation) or the same (*endogenous* transformation). The TU can also take the form of a function or a relation. A function is an imperative construct specifying how the modification of the source model produces the target model. A relation acts on the domains and states the relation between them. Relations may be used for creating bi-directional transformations, such as in QVT-R [76]. The transformation is called *in-place* when it operates on the same model, otherwise it is called *out-place*.

Often in MTs, the number of rules is larger than one. Different approaches exist as to how to execute a set of rules in order to achieve the desired model modification. Below we discuss the topic of rule application control.

Rule application control specifies where the rules are applied in the model and the order in which they are executed (scheduling of the rules). A deterministic transformation ensures the same output is produced after repeated execution. In Stratego [15], a deterministic traversal of the model is defined to allow deterministic transformation. A non-deterministic transformation can be achieved in several ways. A rule can be applied to a single, non-deterministically selected location or to all of the applicable locations in the model concurrently. In AToM^3 [25] a user can specify the application location interactively during transformation execution. A non-determinism situation is demonstrated in Figure 2–7. There we show a rule from Figure 2–2 with a different input model. The *LHS* of the rule can have three matches shown in different colors. We, therefore, need to decide where to apply the rule.

The transformation rule scheduling can be positioned in two groups: *implicit* and *explicit*. With implicit scheduling, a transformation engineer has no direct control over the order of rule executions. For example, the transformation language in the tool Groove [82] can be unordered: the execution order depends on the patterns and is determined at runtime, and rules are executed until there are no more rules to be executed. Another example is found in graph grammars [28], which can be used to generate graphs and perform graph rewriting (transformations). A grammar

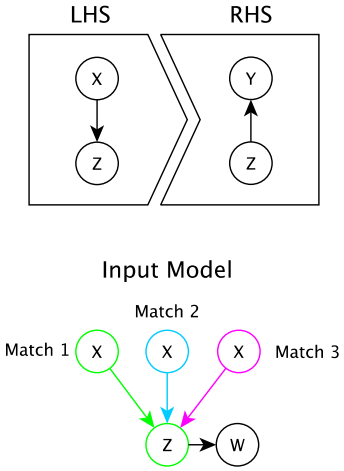

Figure 2–7: An example rule from Figure 2–2 applicable to a different model where several matches exist (marked with different colors). We can thus apply the rule to a single non-deterministically selected location until the rule can no longer apply.

can also be viewed as unordered transformation. Similarly, ATL transformations language does not provide explicit scheduling [45].

An explicit scheduling gives control over the rule execution to the engineer. External explicit scheduling defines a scheduling separate from the transformation rules. This results in an ordered transformation. The schedule can contain a control structure using a branch and loop syntax or a priority-based ordering, etc. An example of such a schedule in a graphical representation is demonstrated in Figure 2–8. The first rule to execute is Rule A. The choice of the following rule is based on the result from the previous rule. An internal explicit scheduling is usually present in the imperative transformation units. For example, in QVT-R when/where clauses may invoke a next rule, thus defining a schedule. Another option is to use a general-purpose programming language to specify MT schedules. Finally, an event-driven

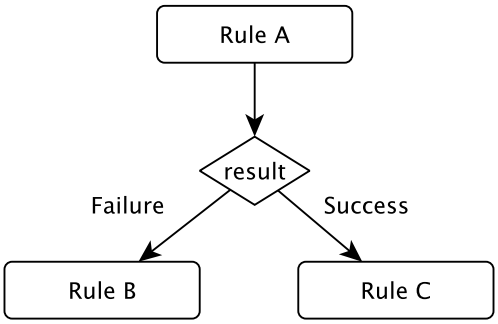

Figure 2–8: An explicit MT schedule using branch and loop constructs.

scheduling can also be used. There, the external events trigger rule executions. An example of an event-driven MT system is VIATRA [7].

Another important MT feature is **tracing**. In transformations, it is important to know which elements in the target model relate to the elements in the source model. The traceability links provide that information. They are also used for debugging and correctness verification of MTs. Some languages, such as QVT support traceability automatically, others leave the creation of links to the transformation engineer. Traceability links are more valuable in unidirectional transformations. In multi-directional transformation, it is easier to go back to original source model by reversing the transformation.

We conclude our general discussion on MTs, rules and MT features. In the following sections, we address the existing challenge of using MTs in the industrial context, namely the efficiency of execution.

2.4 Optimizing Model Transformations

An efficient execution of a MT is important in industrial contexts. The input models can be large, resulting in long execution times for complex MTs. The declarative nature of MT patterns (found in the *LHS* of the rule) and their subsequent matching generally lead to a subgraph isomorphism problem, where an exact match of the pattern is established. Due to the fact that the subgraph isomorphism problem is solved in an exponential time in the worst case, an effort in MT community is made towards heuristically improving pattern matching performance. In this section, we present the efficient pattern matching techniques. These include *search plans*, incremental techniques, general search space reduction and pattern matching strategies. Search plans are of particular importance to this thesis and we explain them in more detail.

2.4.1 Search Plans

In local search-based (LS) techniques, a pattern matching starts from an initial binding of a pattern in the input model (graph). Typically a single node of a pattern is matched. The match is then expanded from that node by following the edges according to the pattern definition. We present a brief overview of the search plan-based graph pattern matching (an LS technique) in order to explain the matching of scoped patterns presented later in this thesis. In addition, we explain *primitive* match operations composing the *search plan* (SP) and discuss the cost of an SP. For in-depth explanation of SPs consult the original works of Batz [6] and Varró et. al [107].

A search plan is an ordered list of primitive match operations. The execution of these operations results in a binding of pattern nodes and edges to the input graph nodes and edges. The operations are executed in order. The ordering can be based on the cost of these operations in terms of their *branching factor*. The branching factor corresponds to the number of bindings each primitive match operation returns. Only a single binding is considered for expanding the match further while others are kept for a *backtracking* step which happens if the next operations fail to produce a binding. In a bad case, the search plan execution can result in a lot of backtracking, causing all of the bindings to be explored while constructing a match. Therefore, it is desirable to first execute operations with a small branching factor and thus minimizing the backtracking. Typically, the following primitive match operations are distinguished:

- A *lookup* operation *lkp(x)* establishes a binding from the pattern node or edge x to the matching host graph's node or edge. Valid search plans must start with a lookup operation to create the initial binding.
- The *incoming* and *outgoing* edge operations: *in(v,e)* and *out(v,e)* require an already bound node v as a parameter to establish the binding for the incoming or outgoing edge e of the node v.
- The *source* and *target* operations: *src(e)* and *trg(e)* require an already bound edge e as a parameter to establish the binding with its respective source and target nodes.

For a binding to be valid, the pattern element type must match the input graph element type. In addition, for operations that concern edges, corresponding incidence

relationships must exist. Pattern attribute conformance, such as for node labels, may be treated within the primitive match operations or as a separate operation. In this thesis, we assume the treatment of node label conformance is within the primitive operations and disregard attribute verification for brevity.

In Figure 2–9 an input graph is shown in column *a* and a pattern to match in column *b*. Column *c* presents the *search graph* corresponding to the pattern. The nodes (as circles) and edges in the pattern are labeled with their respective types.

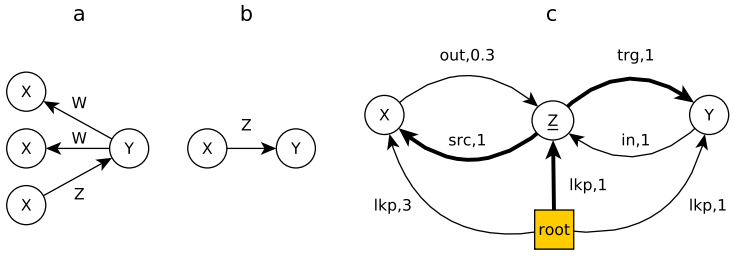

Figure 2–9: Input graph in column *a* with the pattern to match in column *b*. A corresponding (input sensitive) search graph is shown in column *c*, with its minimum spanning tree indicated by bold edges.

The edges in the search graph correspond to primitive match operations, and each node in the search graph corresponds to a pattern element, with the addition of a special *root* node (we underline the node in the search graph corresponding to an edge in the pattern). Nodes representing pattern elements are connected according to pattern connectivity and in a way to allow for bidirectional navigability. The root node is connected to each search graph node with a single outgoing edge representing a lookup operation. To produce a match each node in the search graph must be visited once by following the edges representing primitive match operations and

executing them, in essence traversing a *spanning tree* of the search graph, as shown by the bold edges in column c of Figure 2–9.

Statistical information about the input graph can be used to estimate operation costs (e.g., their branching factors). The host graph information typically includes the number of nodes and edges of a particular type. This information results in a production of *input model sensitive* search plans [33, 107]. Model sensitive search plans are constructed from a weighted search graph representing the match pattern. In column c of Figure 2–9, for example, each edge is weighted with the cost of its operation.

There are several search plans possible for the pattern in Figure 2–9. Let us consider a search plan P_1=*lkp(X),out(X,Z),trg(Z)* represented by the spanning tree on the left of Figure 2–10. This is not the best possible SP. The lookup operation, executed first, returns the bindings for all three nodes of the type X. The following outgoing edge operation, in the worst case, fails for two of the three $\textcircled{X}$ nodes because of the missing outgoing edge. When the match operation fails to produce a binding backtracking occurs and other unexplored candidates are considered. Thus, a single binding for X is used to bind the outgoing edge successfully. As backtracking is expensive, the search plan that causes the fewest backtracking steps is preferable. With this in mind, a better search plan in our case is P_2=*lkp(Z),src(Z),trg(Z)*. P_2 is based on the search graph's minimum spanning tree, as shown on the right in Figure 2–10 (and also in Figure 2–9), which will produce no backtracking. In this case, the first lookup operation binds the lone edge of type Z, and execution of the source operation binds pattern element $\textcircled{X}$ to the node of type X at the endpoint of

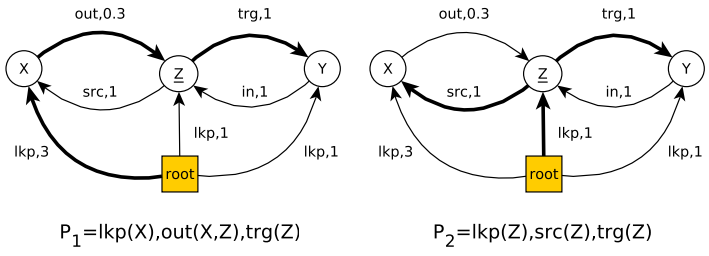

Figure 2–10: Spanning trees corresponding to the search plans P_1 and P_2.

the edge Z. The target operation is executed last, resulting in a pattern element $\textcircled{Y}$ binding to the node of type Y in the input graph.

The minimum spanning tree of a directed graph, weighted with operation costs can be constructed by using Edmonds' polynomial time algorithm [26]. The minimum spanning tree is then used to produce the ordered search plan with the smallest cost.

The cost of a search plan corresponds to the size of its search space tree representing the number of host graph elements visited during matching. The i^{th} level of the tree corresponds to the execution of the i^{th} match operation. The number of nodes at the i^{th} level of the tree is equal to the product of the costs of match operations (branching factors) up to the i^{th} level.

Let us first consider the cost of each individual operation. The cost of a lookup operation is equal to the number of candidate bindings. Therefore $c(lkp(X)) = 3$ and $c(lkp(Z)) = 1$. In case of the incoming and outgoing edge operations let us consider $out(X, Z)$ for the same graph and pattern in Figure 2–9. Depending on the $\textcircled{X}$ node, we may or may not have an outgoing edge Z resulting in a number of candidate bindings equal to 0 for the two nodes and 1 for the third. We then

consider $c(out(X, Z)) = 0.3$ as the average between the three nodes. The source and target operations are simple and both cost 1, because once the edge is bound, the number of candidates at each endpoint is equal to 1 (we do not consider hyperedges in this thesis).

Therefore, the cost of the search plan P=$\langle o_1,...,o_k \rangle$ is calculated by $c(P)$=$c_1 + c_1c_2 + \cdots + c_1c_2 \cdots c_k$ or $\sum_{j=1}^{k} \prod_{i=1}^{j} c_i$. Here c_i is the cost of the i^{th} primitive match operation and k is the number of pattern elements. For example, the cost of P_1 is $c(P_1) = 3 + 3 * 0.3 + 3 * 0.3 * 1 = 4.8$, which is larger than $c(P_2) = 3$, as all three operations in the latter case have branching factor of 1. Note that the search plan cost equation is dominated by the early terms, and therefore it is important to reduce the cost of early match operations.

One of the benefits of our static scope technique presented in Chapter 3 is to reduce the cost of search plans by introducing early match operations with a small branching factor. These operations are then prioritized in an SP and positively affect its overall cost. Another efficient technique presented in the following section is incremental pattern matching.

2.4.2 Incremental techniques

The incremental pattern matching techniques are based on caching intermediate, partial matches. This accumulation of partial matches results in a cache of all matches for a given pattern. The matches are then available for use at the constant time. The performance benefits are evident, for example in a situation when it is necessary to apply a rule to all occurrences of a pattern in the input model. In essence, this approach is well suitable for match intensive transformations, but less

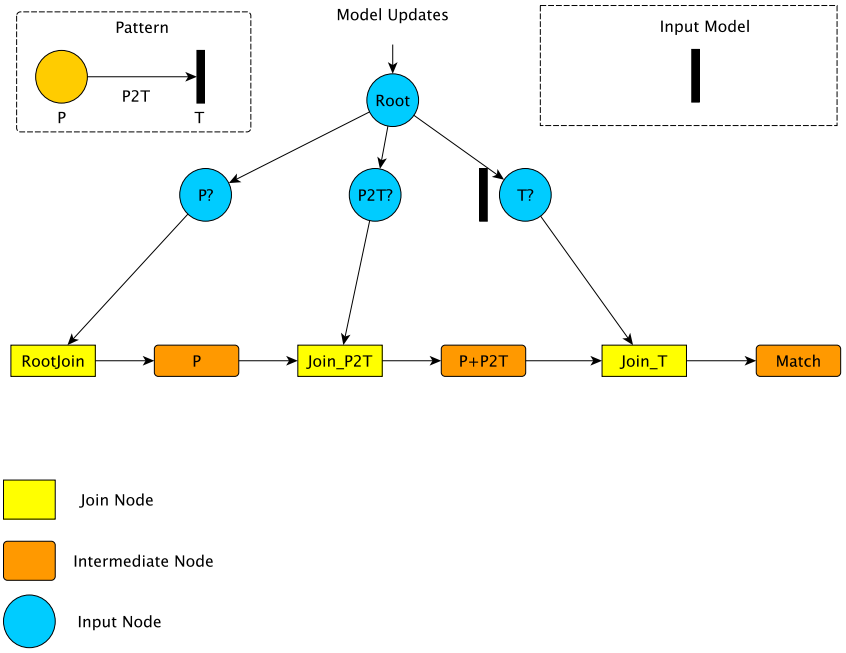

Figure 2–11: A Rete network made for the pattern on the left with the input model on the right. The input model contains a single transition and the pattern is not matched.

appropriate for the transformations where the model is often updated as the match caches need to be updated. The authors of VIATRA tool [10] propose to use the *Rete algorithm* [31] and its network to achieve caching of matches and match retrieval in constant time. The Rete was originally used in rule production systems. In Figure 2–11, we show an example Rete network along with the corresponding initial PN input model on the right and the pattern to match on the left (both contained in dashed rectangles for separation from Rete network). Note that for illustration purposes, the input model contains a single transition as to demonstrate a manual PN drawing in an editor for example. The network for the pattern consists of the

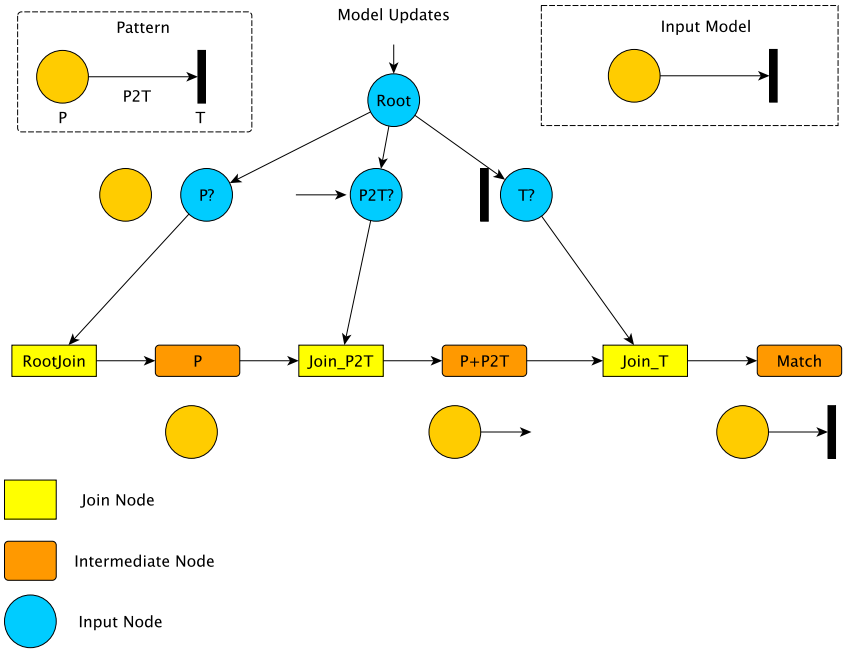

Figure 2–12: A Rete network demonstrating an update phase when a place and an edge was added to the input model.

input, join and intermediate nodes. The input nodes receive a single model element and perform conformance test, they also maintain or keep track of the individual elements. Here we have a single transition cached inside the input node. In Figure 2–12 we see how an update of the input model changes the contents of the Rete network. We add the place and connect it to the transition. The new elements are propagated through the network. The input nodes now contain a place and the edge (connecting place with transition).

In Figure 2–12 the outgoing edge from the input node is connected to the appropriate join nodes to produce the combinations of the individual pattern elements into pattern constructs. After update, we see that the element P can be joined with

P2T element followed by the final join with T to produce the desired match (here we store all occurrences of a pattern in the current input model). The intermediate nodes maintain the partial matches. Anytime during transformation execution the input model changes, the update operation is executed and changes are propagated down the network to modify partial and complete matches accordingly. Efficient construction and implementation of the network is important as the memory cost to caching the matches can be high. Memory cost is reduced by constructing a network where similar nodes are shared between patterns.

It is possible to combine several pattern matching strategies to achieve better results depending on the MT at hand. We discuss this in the next section.

2.4.3 Pattern matching strategies

We already presented two of the state-of-the-art graph pattern matching approaches: search plans (an LS technique) and incremental pattern matching. In [9] the authors evaluate a hybrid approach that combines LS techniques with incremental techniques (INC). The INC pattern matching can provide the match in constant time due to caching of matches, so this technique usually outperforms LS. However, there are several cases where the LS technique is preferable over the INC.

- **INC Cache of matches exceeds available memory**. This is more apparent on a desktop machine rather than in the cloud. Regardless, the authors provide an example where the Java Virtual Machine (JVM) memory limit was reached when the input model size was increased from 10 model elements to just 575. In order to avoid memory starvation, some of the pattern matching was delegated to the LS matcher.

- **INC Cache construction time is prohibitive**. There is an initialization step in the INC techniques that creates the match cache. The time required is not less than finding all occurrences of a pattern in the input model. In the case when only the first match is necessary, LS can stop matching and therefore be faster than INC, provided that the patterns can be efficiently matched by LS.
- **Expensive model updates**. Frequent model manipulations with infrequent pattern matching introduce additional time overhead on INC. The cost of updating the match caches when the input model changes may exceed the time to find a match using the LS. Therefore, the authors suggest using the LS for pattern matching after identifying the MT rules that perform significant updates.

Finally, the authors discuss static factors that help transformation designers choose between LS and INC at the design time. Structural properties of a pattern hint at the complexity of matching: number of nodes, edges, node degrees (statistical information about the input model is also indication of the LS performance). The frequency of repeated matching of a single pattern (taken from MT schedule) suggests that use of INC is more appropriate. On the other hand, the great impact on the model structure described in the 3rd item of the list above suggests that using LS is more desirable. An automatic approach was also proposed that is based on observing dynamic factors at runtime. Observing the match cache size with respect to the available system memory will cause the system to switch to LS matching. The

average matching times of hybrid matching strategies were comparable to INC yet easier on the system memory.

In our work on dynamic scope described in Chapter 4 we present a dynamic technique for learning from the MT by observing its interaction with the input/output model. This allows us to influence the matching process by reducing the input model the pattern matcher is using. It is possible to use our technique with some modifications to make decisions on the matching strategies as we observe the MT engine and its effects on the input/output models.

2.4.4 Rule optimizations

It is possible to optimize model transformations by optimizing transformation rules. In particular, MT rules can be parameterized to allow parameter passing. For example, an application of one rule can produce a *pivot*. In a graph-based transformation, the pivot is a graph that provides an initial match binding. This means that subsequent rules can use the pivot to start matching a *LHS* pattern from and around the pivot graph. In GReAT [110, 1] for example, this is called *pivoted matching*. The pivots are typically useful in the rule execution sequences sharing and passing previously discovered matches. This leads to reduced computation effort. In addition, the pivots are one of the ways to reduce the logical complexity of the rules and complex MT use scenarios. Scope concept presented in this thesis can be seen as a compliment to pivots. Both can be used to discover pattern matches sooner as they represent the areas of interest in the input model. However, depending on how scope areas are created and used, they may not necessarily contain parts of input

model where a valid match can be discovered. This is particularly true in case of dynamically discovered scope areas presented in Chapter 4.

Sometimes transformation rules are executed repeatedly in loops. The authors in [72] propose to automatically identify which model elements from one rule execution can be reused in the subsequent rule. In a way, this is an automatic pivot detection and propagation. In addition, the technique is aimed at producing new and improved rules from combining several old rules. The new rules contain combined matching patterns by exploring the similarities between the patterns in different rules.

We conclude the discussion on the MT optimizations and continue with the discussion on MT tools.

2.5 Overview of Relevant MT Tools

The previous section deals with the efficiency challenge MTs face. In addition, this thesis discusses the usability aspect of MTs through their debugging support in Chapter 5. Therefore, in this section, we discuss a selection of MT tools relevant to this thesis, from the point of view of their efficiency and/or debuggability aspects.

2.5.1 AToMPM

AToMPM [59], a tool for Multi-Paradigm Modeling developed in our lab, is a multi-formalism, multi-abstraction meta-modeling and model transformations tool, conceived as a successor to AToM^3. One of the distinguishing features of AToMPM is the multi-view, multi-user browser-based user interface that eliminates the need for complex installations and setups for DSL engineers and users. In AToMPM everything is modeled explicitly, from the tool bars in the browser to the user interface

behavior. The tool is under active development and aspires to be the answer to the shortage of accessible and usable model-driven engineering (MDE) tools.

Efficiency. The graph rewriting capability of AToMPM is powered by T-Core [93], a collection of transformation primitives which abstracts the graph matching and rewriting aspects of graph transformation rules and provides a universal way of dealing with graph rewriting problems and in particular, the rapid design and implementation of transformation languages.

Graph representation and transformations are actually carried out within the C-based `igraph` library [22], contained within a Python implementation called Himesis [81]. Graph structures in `igraph` have a simple and well-optimized representation. Nodes, for instance, are implicitly represented by an integer index, allowing the system to allocate nodes simply by incrementing a maximum node counter. Once a node is allocated, edges can be constructed as (directed) pairs of node indexes, and are themselves referenced by an index. This design allows for straightforward and efficient access to graph structures, using array-like access semantics.

We base many of our experiments on AToMPM. The tool in its original form provides a good use case in testing our techniques. Before the thesis-related prototypes were implemented, the pattern matching routine present in AToMPM was a variant of a VF2 algorithm [19]. This defined a baseline case for our evaluations of scoped model transformation efficiency.

Debugging. The tool supports certain debugging functionality. The user is able to step through a single rule application using a debugging toolbar. The input model and the transformation effects can be observed in the browser window.

The visualization of a control flow through an explicit MT schedule is possible with highlighting if an active rule is in a separate window.

2.5.2 GrGen

GrGen [33] is a highly efficient graph rewriting and model transformation system. GrGen compiles everything that is necessary to run the transformation into executable binaries. This, along with the use of search plan-based matching results in very fast transformation executions [8].

Other than the use of a textual language, the process of specifying the transformation and DSL in GrGen is similar to AToMPM. A meta-model (MM) is defined and models conforming to the MM are instantiated. The rules and their execution sequences are specified. Then, the pattern matching backend generates model sensitive search plans.

Efficiency. As described in the GrGen manual, several optimizations aimed at speeding up the graph rewriting are available for a transformation engineer with deep knowledge of the problem domain. Below we list some of the available optimizations we used in our evaluation of GrGen to implement our scope concept. The way each of the following optimizations was used to implement our scope concept and the experimentation results are presented in the Chapter 3.

- An annotation *prio* is used in rules to indicate to the transformation backend which pattern element should be bound first in the search plan-based matching. Assume, for example, the engineer anticipates that there are ten nodes of type A versus thousands of nodes of type B in the input model. Marking the A type related pattern element with *prio* annotation will then force the first lookup

operation to bind a node of type A, significantly reducing the cost of the search plan. This annotation overrides model sensitive search plan generation or is used in the case when the input model statistics is not available.

- A transformation engineer can define custom model attribute indices in the meta-model. These indices are then used in the rules to reduce the search space. During matching, model elements can be queried based on the exact value or the range of attribute values.
- It is also possible to use containers in the transformation rules. Sets and maps promise node lookup performance gains in addition to the convenience of passing them between the rules.
- The GrGen manual also suggests to introduce extra edges into the DSL. Provided the number of such edges is smaller than the number of DSL type elements, the use of these edges in the model will cause model sensitive search plans to bind with these edges first.

It is not, of course, our goal to present an exhaustive list of possible GrGen optimizations. We concentrated on including those optimizations that can be used to implement our scope concept or those that have some overlap in the functionality of reducing the search space.

Debugging. Below we present GrGen's debugging support. The complete description can be found in the user manual [3] of GrGen. The tool suite provides

[3] `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`

GrShell enabling interactive execution of GrGen rewrite rules. For visualization purposes, GrGen uses its own *yComp* editor displaying the input model as a graph. GrShell (a text-based user interface) commands control the execution step by step. Below are some of the most interesting commands. It is possible to step through a rule application in three steps: match, rewrite and apply. The match step highlights the match in the graph, rewrite highlights the elements to be changed and the application step performs the actual rewrite operation and displays the resulting graph. The step-over command, results in a one shot rule application, producing the modified graph. There exist a step-out operation, it continues execution of the MT until a single iteration of the loop (in MT schedule) or to the end of the rewrite sequence. A resume functionality is available too, the remaining rules will be executed to the end of the rewrite sequence or a breakpoint. It is also possible to save the visualized graph in a file for later inspection and print the local and the global variables of the rewrite sequence. In addition to breakpoints, GrGen provides *choicepoints* that halt the execution expecting some user input. The choicepoints can be used to deal with MT non-determinism by providing, in the event of uncertainty, all available choices to the user for selection.

2.5.3 ATL

ATL is a de facto standard in the Eclipse model-to-model transformations. A combination of declarative and imperative constructs in rules are specified in a textual format. The ATL rules are then compiled into specialized byte code that is executed on a stack-based virtual machine. Importantly, the whole transformation is compiled into two notable operations called *match* and *exec*. The first runs rule

specific match parts and creates automated trace links (with empty target elements at the end). The second initializes those empty target elements according to the specification in the rule.

Efficiency. In [14] the author compares the ATL MT execution to QVT on several transformations. The author explores ways of improving ATL performance through augmenting the transformation specification. One such approach is to move the input model navigation to the attribute helper functions so that the matched or queried model elements are reused (cached). This approach proved to be beneficial for the rules that query the elements more often thus justifying the caching. Otherwise, the standard declarative ATL rules outperform the rules with model navigation in the attribute helper functions.

Debugging. Debugging in ATL is supported through the Eclipse launch configurations. Transformations can be executed step by step or continuously. Breakpoints can be set in the rules and variable inspection is possible as a proxy mechanism for model inspections. The debugger is essentially based on the underlying virtual machine (VM) operations. This is supported by the presence of the callback hooks in the VM. Those hooks are for VM stack frame entering and leaving as well as stepping. We use those callback mechanisms in our debugger prototype evaluation.

2.5.4 VIATRA

VIATRA stands for Visual Automated model Transformations [7]. In VIATRA, rule-based model transformations are specified using Viatra Textual Command Language (VTCL). The rules have pre-conditions and post-conditions. VIATRA also supports negative patterns or NAC. The NAC pattern is specified within the LHS

directly thus explicitly marking the elements that must not be matched in the model. The rules are parameterizable, similar to pivoted matching in GreAT [1] and their scheduling is explicit with a possible non-determinism of rule application location using random constructs. In addition, the branching construct if-then-else in combination with *forall* and *iterate* constructs allows for advanced control over the transformation schedule. Additionally, rule hierarchies exists in VIATRA that facilitate rule packaging and reuse.

Efficiency. VIATRA tool is a pioneer in utilizing the incremental pattern matching in a MT system [10]. The Rete network, described in Section 2.4.2, is used to provide pattern matches in constant time. In addition, VIATRA supports search plan pattern matching. These two efficient approaches can be used in the tool depending on the MT problem at hand by combining pattern matching strategies [9].

Debugging. The tool supports debugging through the Eclipse Debugging framework. It supports breakpoints, including rule activation breakpoints, conditional breakpoints (specified using VIATRA query language), and rule halting breakpoints. Various views over the debugging state (including the models) are also supported with a possibility for a user to control the transformation interactively.

We conclude this section describing the selection of MT tools. We now move to the chapter describing scoped MTs and start with the static scope concept.

Chapter 3
Scope in Model Transformations

In this chapter we describe our *static scope* concept applicable to model transformations (MT). The purpose of static scope is to provide the means to specify and reason about scopes in MT. Our scope becomes part of the model and is definable by the transformation engineer inside the MT rules.

Scope is primarily aimed at improving pattern matching performance. For this, to demonstrate possible and practical performance benefits, we build on the search plan-based pattern matching described in the previous chapter. We begin with the introduction into scopes in MTs and specific contributions described in this chapter.

3.1 Introduction

Scoping, i.e., grouping of related elements within a model is a common approach to dealing with complexity. Use of scope in models in some form or another has advantages in terms of improving scalability, especially for visualization of large graphs[1] [11], as well as in representing the natural hierarchy or scoping that exists within the underlying problem domain. In the context of model transformation, the integration of scope allows for a more natural expression of locality of transformation rule execution, and has the potential to provide performance benefits as well.

[1] Throughout this thesis we refer to model representations as graphs, and use both terms interchangeably.

The latter is of particular importance, since the declarative nature of rules in many model transformation environments leads to expensive matching procedures based on subgraph isomorphism [18]. A matching process that is constrained by scope to a subgraph may be faster (depending on the implementation and the problem), addressing one of the main concerns in the industrial-scale implementation of graph-based model transformation systems. A difficulty exists, however, in that the scope best used for a given model transformation may not trivially conform to the notion of scope used in the base modeling formalism—the method by which a hierarchical system is transformed does not always need to respect its original hierarchy.

In this chapter, we address this concern by developing a graph transformation language that incorporates scope directly into the input (host) graph, while also allowing for easy and natural manipulation of scope within rule syntax. This is in contrast to previous efforts at using scope that either concentrated primarily on developing hierarchical rule structures [84], or focused on domain-specific implementations of transformations [34]. Our effort is aimed at integrating a general and flexible form of hierarchical scope directly into model transformation, while still maintaining practicality of implementation, and indeed heading towards useful efficiency improvements. The system we design has the further advantage of being a natural extension of existing graph transformation approaches, and we present an initial non-trivial prototype implementation that demonstrates real speedup in a state-of-the-art research modeling environment. Specific contributions of this chapter include:

- We present a unified way to model and utilize scope in MT. Instead of being a runtime artifact, the scope becomes a first class citizen in MT. We develop a formalism for representing multiple scope hierarchies in a host graph, orthogonal to any internal hierarchy of the underlying model. Our approach is designed as a natural extension of basic graph transformation environments, allowing for implementation within an existing framework and supporting tools. This enables straightforward and incremental migration to an optimized, scope-aware transformation system.
- We define a modified rule syntax that tightly integrates scope into rule matching and rewriting. Rule structure is selected to elegantly reflect an intuitive understanding of how scope is used, without overly compromising the ability to ensure efficiency in a realistic implementation.
- Practical utility of our scope model is demonstrated by an initial prototype implementation within the AToMPM [59] meta-modeling tool. Performance evaluation is performed on a forest-fire spreading simulation and a distributed mutex benchmark (from a model transformation benchmark suite [109]). In addition, we map our scope concept to an efficient and highly optimized transformation tool *GrGen* [33]. Our experience indicates that our scope design is very usable and also capable of significant speedup.
- The actual scoped pattern matching is presented in the context of *search plans* (SP) [107] (introduced in Section 2.4.1). We demonstrate that adding scope to an existing pattern may result in the reduction of SP costs. This leads to accelerated pattern matching and performance gains in MTs in general.

The rest of the chapter is organized as follows. Section 3.2 formally introduces our interpretation of scope and offers a running example. Section 3.3 investigates the use of scope in rule-based model transformations. Section 3.4 concentrates on implementation, experimental evaluation and interpretation of results. Section 3.5 concludes the chapter and discusses future work.

3.2 Static Scope

There are many possible ways to express and use scopes in a model transformation system. Our approach here is necessarily a compromise intended to allow for reasonable expressiveness, while still ensuring a realization that is as efficient as possible. Below we describe the core ideas underlying our approach and give an initial formal definition, followed by an introduction of our running example, and efficiency motivation behind scope-based matching.

The basic idea of scopes we use is built on the notion of a secondary *scope forest*, connected with, but logically distinct from the underlying host graph. The scope forest is formed as a set of hierarchies, represented by a scope hierarchy forest (SF) consisting of one or more scope hierarchy trees (STs), and such that each node in the SF has a unique label. The use of multiple STs allows a node to simultaneously exist in multiple hierarchies, while unique names and the single-parent and acyclic properties of STs make certain scope patterns unambiguous and improve the efficiency of determining whether a scope pattern applies to a given node.

Figure 3–1 shows a simple example of a *scoped graph* as used in this thesis. Labeled, dashed rectangles identify the two main components of a scoped graph, the host graph, and the SF. To avoid confusion we represent the host graph in

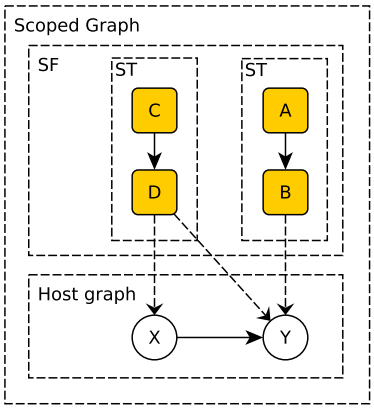

Figure 3–1: The scoped graph with the scope hierarchy forest (SF) containing two scope hierarchy trees (ST).

terms of connected, labeled circles, while the SF is represented using labeled and rounded-rectangles. In Figure 3–1, we have two distinct STs in the SF. Dashed lines from ST nodes to host graph nodes represent innermost scope labeling or mapping, while edges within ST nodes represent the scope hierarchy relation. Here, node $\textcircled{X}$ is understood to be in scope C and D with the innermost scope D, and node $\textcircled{Y}$ is in all 4 scopes, with the innermost scopes D and B. Note that the expression of scope in this fashion allows for a straightforward implementation, even in scope-unaware systems, either by including scope nodes directly or by expressing scope as an additional attribute of host graph nodes. In the case of encoding scope as attributes, the tool may gain performance during matching by performing attribute indexing. A scope-aware implementation will obviously exploit the extra information to increase performance.

We now proceed with a formal definition of our scope concept.

3.2.1 Formal Definitions

The above description can be formalized by defining a scoped graph as a 7-tuple $G = (V_G, E_G, L_G, V_S, E_S, L_S, R)$, where:

- V_G is a finite set of host graph *vertices*,
- $E_G \subseteq V_G \times V_G$ is a set of *directed edges* in the host graph,
- $L_G : V_G \to String$ is a node labeling function; duplicate names are allowed.
- V_S is a finite set of scope *vertices*, disjoint from V_G ($V_S \cap V_G = \emptyset$),
- $E_S \subseteq V_S \times V_S$ is a set of of *directed edges* such that (V_S, E_S) forms a forest,
- $L_S : V_S \to String$ is a scope-node labeling function, assigning unique labels to each scope node: $\forall v_1, v_2 \in V_S, L(v_1) = L(v_2) \Rightarrow v_1 = v_2$.
- $R \subseteq V_S \times V_G$ defines the innermost scoping relation; which must fulfill the conditions enumerated below.

With this definition, we have a formal way of evaluating whether a node is in a scope or not. A node $n \in V_G$ is considered in a scope $s \in V_S$ if there exists a path from s to some s' such that $(s', n) \in R$. We will also be concerned with an innermost scope: $s \in V_S$ is an innermost scope of $n \in V_G$ if $(s, n) \in R$. Note that we allow a node to belong to multiple scopes, and thus innermost scope is not a unique property. However, to better reflect the conceptual hierarchy each ST represents, we will also impose a requirement that each node has at most one innermost scope in each ST; that is, $((s, n), (s', n) \in R \wedge s \neq s') \Rightarrow \nexists s''$ s.t. $\text{path}(s'', s) \wedge \text{path}(s'', s')$.

In this thesis, we are dealing with directed, typed, and attributed graphs. We leave the use of hierarchical graphs, in the context of scoped MTs, for future work. However, the theoretical foundations of algebraic hierarchical graph transformations

exist [77] and can be used as a foundation for our approach. In addition, we demonstrate our scope use in the context of the tool GrGen in Section 3.4. This tool does support hierarchical graphs and we believe that their scoped transformation would be straightforward. We conclude the formal definition of static scope and move on to the running example.

3.2.2 Running Example

To motivate the use of hierarchical scopes in model transformations we present our running example. For this, we used a simulation of forest-fire spreading, where fire spreads across a 2D grid of neighboring cells. Each cell in a grid represents a forested area which may catch fire if any neighboring cells are on fire. Once fully burned, a cell represents a barrier to further fire spreading. The simulation terminates when no burning cells remain. There are of course many fine-grain details that may be added to the base simulation model, such as the duration of forest burning, wind effects, and so forth [50]. This overall geometric approach, however, is recognized as a standard way of modeling the dynamics of fire spreading scenarios [29]. Additionally, in [109] the grid approach is used to benchmark transformations that mostly perform matches without changing the structure of the source graph. Due to strong localization in where rule transformations occur and which rules can apply, the forest-fire constitutes an interesting problem to evaluate efficiency in model transformations. This localization is highly dynamic, changing as the simulation progresses. Thus, it provides an excellent test-case for evaluating the impact and suitability of scoped versus non-scoped transformations.

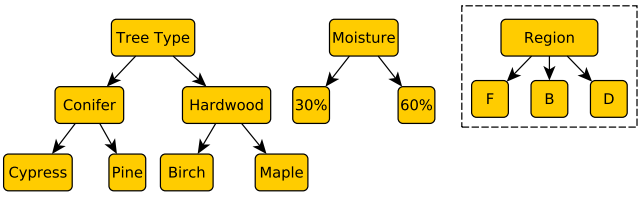

Figure 3–2: Extended forest-fire scope hierarchy.

We use a *SF* to capture both the domain-oriented structure and the dynamic knowledge of forest-fire spreading dynamics. Tree burning time can be affected by several factors. For example, the type of trees in the forest and their moisture content may be static aspects of our model that form a natural hierarchy of the domain. Thus, the forest cells (assuming sufficiently small cells to guarantee homogeneity) can be grouped into scopes by their type and moisture content. In Figure 3–2 we thus introduce two *ST*s to represent this structure, dividing trees into classes of hardwoods and conifers in the *Tree Type* scope hierarchy and introducing two moisture levels in the *Moisture* scope hierarchy.

The dynamic property of fire spreading is also captured in the scope hierarchy. This represents a scoping orthogonal to the natural, static domain hierarchy, and introduces a *Region ST* with three active scopes F, B, and D (highlighted with a dashed rectangle in Figure 3–2). Scope F contains burning forest cells, scope D represents cells with dead trees, and scope B represents cells with healthy trees that border cells in scope F (think of scope B containing smoldering trees to ignite soon). We illustrate the entire scoped graph in Figure 3–3, where filled round nodes representing cells in the 3 by 3 grid of host nodes are connected by dashed edges from *SF* nodes, indicating the innermost scope relationships.

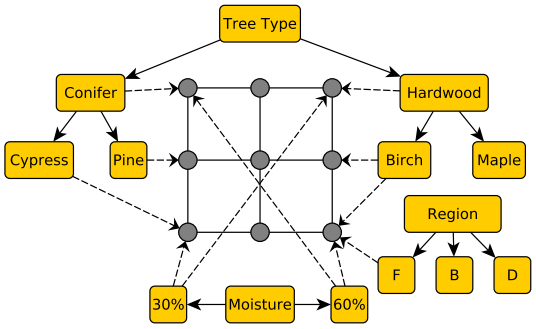

Figure 3–3: Forest-fire scope hierarchy applied to the forest grid.

In this example, we can see that some of the cells are in the scope of conifers while others are in the scope of hardwoods. There is one cell in the fire scope and the moisture content scope encompasses several cells. Note that there are unmarked cells in the grid. These indicate nodes without any scope relationship.

The dynamics and evolution of the forest-fire spreading, encoded in scopes, can be related to the concept of *activity tracking* in a cellular or a Discrete Event System Specification (DEVS) based modeling and simulation, such as presented by Muzy *et al.* [75, 73]. In fact, the active regions of the transformation were some of the first candidates to be represented as scopes. This is one of the examples of how hints from the problem domain can be used for defining scopes. Similarly, in previous work on automated and runtime scope discovery [47], the active parts of the transformed model are considered as dynamic scope candidates. The next section presents the efficiency motivation of using scopes in MTs.

3.2.3 Efficiency Motivation

We use *scope areas* to demonstrate possible matching efficiency gains. Consider an $N \times N$ grid of forest cells, as shown on the left in Figure 3–4. This is the actual

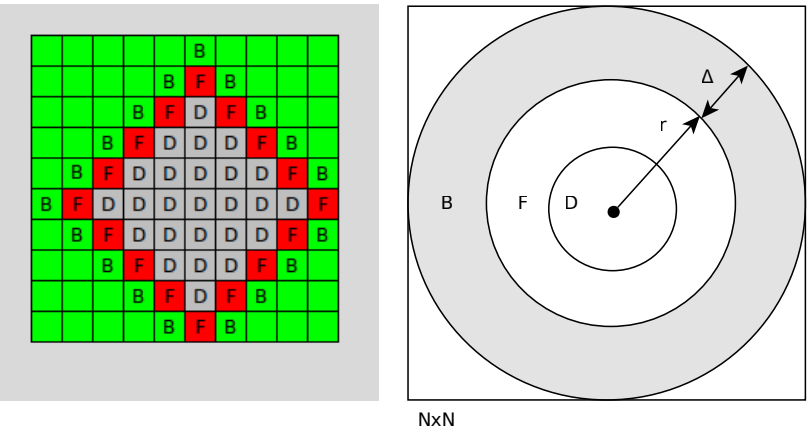

Figure 3–4: A screenshot of the forest-fire simulation and the model of the forest-fire spreading over the grid.

simulation of forest-fire spreading using our scope concept. On the right is a model of the forest-fire spreading over the grid on the left. Concentric annuli marked B, F and D represent the *Region* scope hierarchy from our running example and correspond to the scope regions marked on the screenshot. The areas bound by these regions approximate the number of grid nodes that need to be iterated over to find all matches of a pattern containing a single forest cell. We are interested in a symmetrical spreading of the forest-fire (conceptually circular, but appearing diamond-shaped due to the taxicab geometry in the forest-fire spreading screenshot), and so are primarily interested in the dynamically expanding fire-front area B containing candidate cells to be moved into F scope (as well the F cells which eventually finish burning and change to D cells).

The number of match searches performed without using scope is equal to N^2, the entire area of the grid. Considering only the scope relevant to a rule application can dramatically decrease this cost. The area of the B annulus, for instance, is

$B = 2\pi r\Delta + \pi\Delta^2$. Dividing both sides of the equation by r^2 we get $\frac{B}{r^2} = 2\pi\frac{\Delta}{r} + \pi(\frac{\Delta}{r})^2$. We can eliminate $(\frac{\Delta}{r})^2$ as negligible when r is significantly larger than Δ. Now we can approximate the area to $B \approx 2\pi r\Delta$. When $r \approx \frac{N}{2}$ the annulus area is $B \approx \pi N\Delta$, this yields a linear complexity of $O(N)$ to enumerate the nodes inside the scope defined by annulus B.

Dynamically changing scopes such as the one represented by annulus B in the previous example requires runtime modification to scope membership—we gain nothing in efficiency if the entire grid must be traversed to change scopes during the transformation. To solve this problem, we can maintain scope B from within its neighbor scope F. For this, we need to incorporate scope modification directly into the rule syntax as we demonstrate in the next section.

3.3 Scope in Rule Based Model Transformations

In this section, we describe the syntax and semantics of scoped model transformations. First, we look at possible scope patterns, their use in transformation rules, and provide some justification based on usage scenarios and constraints. We then introduce an extension to our own transformation rules that allows for the manipulation of scope hierarchies. We describe the semantics of scoped transformations and scoped matching using search plans. Finally, we address scoped graph rewriting with its well-formedness concerns.

3.3.1 Scope Syntax

Our full design applies a syntax to the conceptualization of scope presented earlier. This simplifies and constrains rule specification in the presence of scopes, reducing the potential for specifying malformed rules, and overall providing a more

intuitive format for describing scope matching and construction. Our goal is to be sufficiently expressive while ensuring that an efficient implementation is possible.

In general, we will need to know whether nodes are in different scopes, and may be concerned with combinations of scopes, or inheritance within a scope. For this, we define scope *patterns* as core constructs of our syntax. Figure 3–5 visually summarizes this approach. For each of the 6 panels, a scope pattern syntax is presented on the left, and a corresponding scoped graph (host graph and SF) that would match the pattern is shown on the right. In patterns, the shaded, rounded-rectangles represent scopes, and labeled circles represent nodes of a source graph. The items placed inside a scope imply a direct relationship. Thus the semantics of a circle node drawn inside a scope rectangle is a node inside that scope (this is similar to the notion of containment in hierarchical graphs described in [77]), and that scope also constitutes the node's innermost scope. Similarly, a scope immediately inside another scope represents an edge in the scope hierarchy relationship, with the outer scope being the direct parent of the inner scope.

The graph on the right of each panel shows source nodes slightly larger than on the left in order to emphasize the distinction between template host nodes in our pattern syntax and host nodes in the actual scoped graph.

We now discuss each of the 6 panels. As an example, and to help motivating the syntax of our scope patterns, we use our running example.

- Panel 1. The node $\textcircled{X}$ is not placed inside a scope pattern. The intention is to ignore the scope during matching. Using this construct we can match any

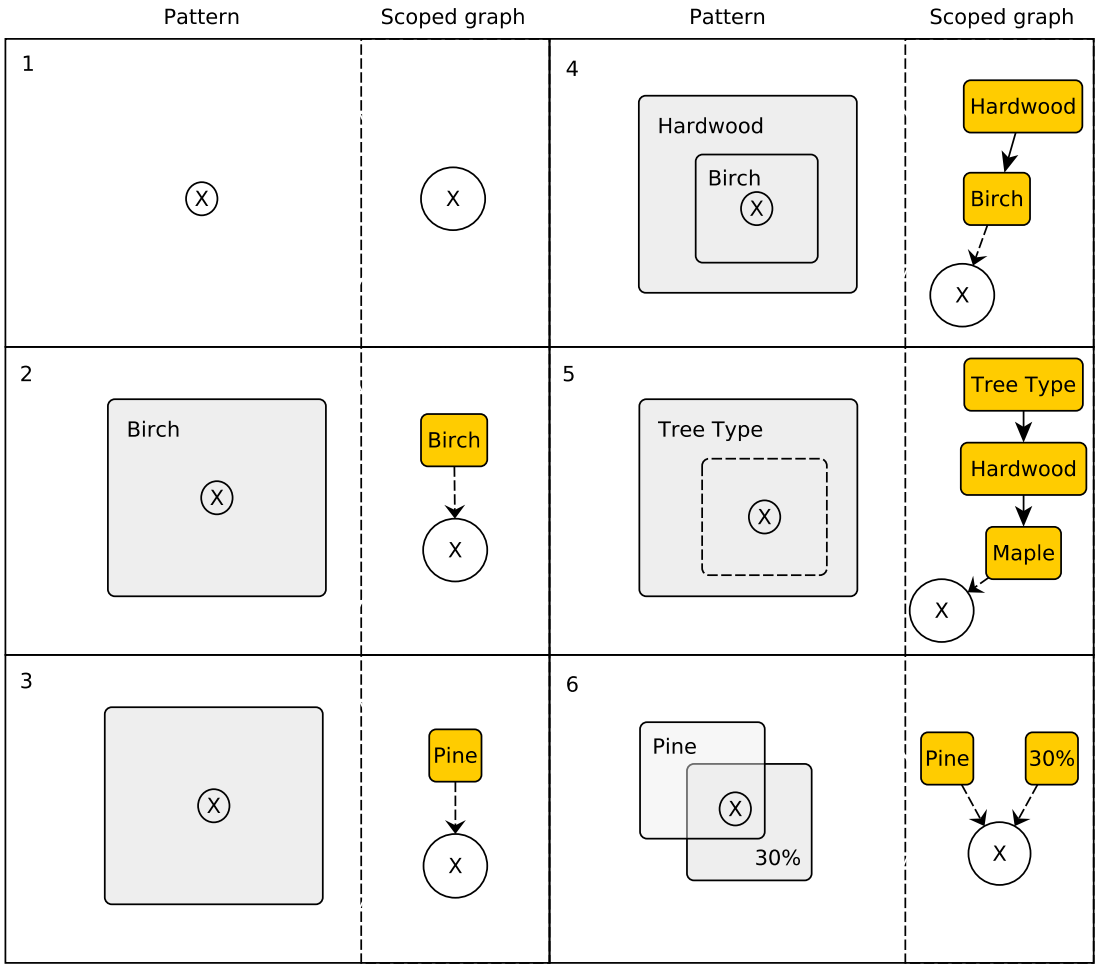

Figure 3–5: Six core scope patterns, each with the pattern on the left, and a possible matching scoped graph on the right. Panel 2 shows a *labeled scope* pattern, panel 3 contains an *anonymous scope* pattern, and nested and overlapping scope patterns are shown in panels 4 and 6 respectively. A *dashed scope* pattern is shown in panel 5.

forest cell in the grid, regardless of its type, moisture content, or fire region. We refer to this construct as *no scope.*

- Panel 2. The node (X) is placed inside a single, *labeled scope.* This indicates the innermost scope relationship for node (X). Here we match all birch cells in the forest.
- Panel 3. The node (X) is placed inside an unlabeled scope. Such a pattern represents an arbitrary scope in the SF, and we refer to it as an *anonymous scope.* This pattern will produce a match if the corresponding source node has any (innermost) scope relationship. In a negated rule application condition, the anonymous scope can also be used to determine the absence of scope for the node it contains.
- Panel 4. The node (X) is placed inside a hierarchical (nested) scope construct. Such patterns can be used to designate a specific portion of the scope hierarchy while matching. In hierarchical scope constructs, labeled or anonymous scope represents one level of scope hierarchy inside a single ST below the scope that directly contains it. Here we are trying to match all hardwood cells that are birches.
- Panel 5. Here we introduce an unlabeled, *dashed scope.* It represents 0 or more levels of scope hierarchy inside a single ST below the labeled or anonymous scope that directly contains it. As opposed to panel 2, this allows us to identify nodes within an inherited scope; here, any scope under Tree Type. A dashed scope is not allowed to be used on its own or as the outermost scope in nesting constructs, as that results in the source of the inheritance being

undefined. Dashed scope patterns are in a way similar to the use of rules with inheritance [24], where a pattern is specified using an abstract type and is applicable to the subtype model elements.

- Panel 6. Pattern 6 demonstrates a node having multiple innermost scopes. Here we are matching nodes that are both pine cells and have 30% moisture content. Multiple scopes enclosing a node must be in separate STs, as mentioned in the formal scope definition. In our design, an attempt to match a node in multiple scopes from the same ST will result in an error. We base this on the assumption that a node in two scopes of the same ST properly belongs to its single least-ancestor.

Core scope constructs, labeled, anonymous, and dashed, can be combined using nesting (such as in panel 4 in Figure 3–5) and intersection (panel 6 in Figure 3–5). Not all scope nesting combinations make sense. In particular, nesting of dashed scopes is redundant if expressed as a single parent-child nesting, and so is disallowed.

The intersection of scopes is meant to be simple, representing a conjunction of distinct scope patterns that must apply to the same host graph node. Thus nests of scope pattern specifiers may not intersect except at the leaf-level. As well, to ensure we can easily distinguish which nest of patterns to apply to a given scope tree, we also require the outermost scope of each intersecting scope nest to be a labeled scope.

It is also worth pointing out that the knowledge of the exact scope hierarchy may render some scope combinations such as nested labeled scopes unnecessary. In panel 4 of Figure 3–5, for example, it is actually sufficient to use a single labeled scope construct (such as in panel 2) to indicate just the innermost scope relationship, since

Table 3–1: Operators

Scope Operator	RE Operator
(L) *labeled scope*	α
(A) *anonymous scope*	.
(D) *dashed scope*	.*
(n) host graph node	n

we already know that birches are a direct subscope of hardwood. Specifying parts of the hierarchy is mainly useful when the scope hierarchy is dynamically modified, or when the same rule set may be used in distinctly different scope contexts. We now move to a discussion on the expressiveness of scope.

3.3.2 Expressiveness

Our scope pattern constructs are defined to accommodate an intuitive understanding of how scope may be used and required in practice, while trying to guarantee that an efficient runtime test will still be possible. We would still like, however, to guarantee some degree of expressiveness, ensuring reasonably general applicability, and also better formalizing our allowable scope pattern constructs. For this we can relate our constructs to other ways of expressing path properties in graphs, and consider our patterns as a form of *path expression* [111] or *regular path query* [65] over the *SF*. To do this we show that the paths within the *SF* encoded in the scope constructs can be mapped to simplified regular expressions (RE) over the labels of the scoped graph. Table 3–1 gives the basic translation, relating each of our scope pattern operators to corresponding RE syntax. In this mapping α represents a label of a *SF* node and n indicates a label of a host graph.

In order to perform this mapping, we first define the language of well-formed scope path expressions (SPEs) by converting the nesting hierarchy to an RE-language,

$$\text{SPE} = (L|A)(D?\ (L|A))^*\ D?\ n$$

with a restriction of no nested dashed scopes to respect the constraints given earlier. A scope hierarchy path starts with L or A, followed by a combination of scope operators and terminates at the host graph node n. The sequence of operators read from left to right defines scope nesting order. The leftmost operator indicates the outermost scope, and subsequent operators represent subscopes of the parent scope denoted by the preceding operator. The innermost scope relationship for the terminating node n is defined by its immediate predecessor. Scope intersection is interpreted through a finite set of paths, each terminating at the same host node, but otherwise evaluated separately. Recall that we also require intersecting patterns to have a labeled node at the top level.

Mapping a particular pattern $p \in \text{SPE}$ to an RE itself is then defined by translating p according to the mapping given in Table 3–1: $.*(p[A \rightarrow .][D \rightarrow .*])n$ (an arrow here means replacing construct on the arrow's left with the construct on its right side). Here we also prepend $.*$ to represent an arbitrary starting point in the *SF* for RE matching. Matching over the *SF* is then conceptually performed in a depth-first manner, beginning at the first operator and terminating at n. We also perform the shortest match, that is useful in the context of search plan-based matching described in Section 3.3.4. There we navigate up the scope hierarchy from the graph node and if the anonymous scope contains the dashed scope we terminate

our matching at the very first scope node encountered (innermost scope) instead of searching such a construct all the way up the hierarchy until the root scope node.

We now map as an example some of the patterns in Figure 3–5 to REs.

- Panel 5 is: $. * \textit{Tree Type}. * X$
- Panel 6 is a combination of two REs: $. * \textit{Pine } X$ and $. * 30\% \ X$

All our patterns are required to terminate at a single host graph node. In many cases, however, a designer may wish to specify that the same pattern applies to multiple host nodes, and patterns similar to the one on the left in Figure 3–6 may thus be desirable. Since the host graph does not, in general, conform to the same restrictions we impose on our *SF* that make RE expression straightforward, describing such patterns introduces significant complexity into our RE translation process. To reason about such patterns, we thus instead rely on a *flattening* operation, conceptually decomposing a non-conforming pattern to duplicate the scope pattern such that there is a single well-formed path expression for each host node. The result of flattening is shown on the right in Figure 3–6. Note that this reduces the ability for a pattern to guarantee it refers to the same scope node containing different host nodes: even if we draw X and Y within the same anonymous scope A, once flattened the respective anonymous scopes could be bound to different STs. Avoiding the need to find a scope binding that is the same for all nodes, however, simplifies the matching process, and so we only permit non-conforming patterns such as these when there is an unambiguous use of scope (no rewriting of anonymous scopes). The use of anonymous scope (or dashed scope) in conforming patterns similar to the pattern on the right in Figure 3–6 is allowed. The user should be aware of the resulting matches

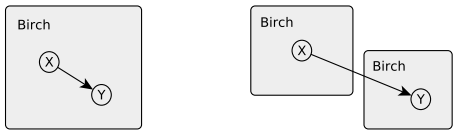

Figure 3–6: Flattening of a scope pattern on the left, with a result on the right.

of different scopes or different parts of the scope hierarchies (in the case of dashed scope use).

Finally, we note that extensive use of scope combinations such as intersection can stress the visual syntax of our scope formalism. In more complex cases we can use a textual format to describe SPEs, or a layered, hierarchical visual representations to deal with the complexity. In the next section we introduce *scoped rules*.

3.3.3 Transformation Rule Structure

Graph transformation rules in our design follow a traditional composition of a left-hand side (LHS) graph pattern, a possible negative application condition pattern(s) (NAC), and a right-hand side (RHS) transformed pattern. Note that the rule systems traditionally also include unique labels associated with pattern elements, to allow LHS, NAC, and RHS elements to refer to specific matched instances, and thus identify elements which are specifically deleted or modified. For clarity and simplicity in depicting the patterns in transformation rules, we do not show these unique labels in our examples.

A straightforward rule design would be to simply allow our scope syntax to be employed in LHS, RHS, and NAC specifications. An interesting complexity in defining rules for a scoped transformation system, however, arises from the need to express scope manipulations independently of source graph manipulations. This is

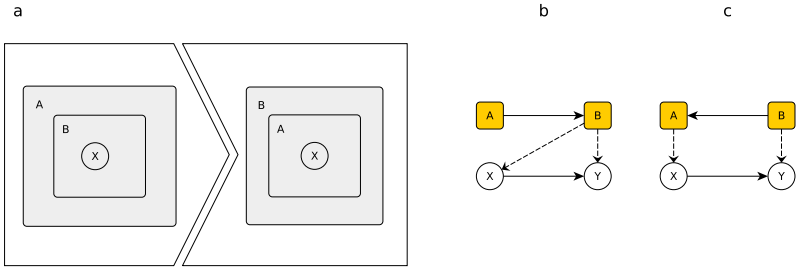

Figure 3–7: Ambiguity in modifying scope hierarchy in RHS.

due to the fact that a naive change in scope hierarchy has the potential for non-trivial side-effects on the graph structure, not easily visible in the rule structure.

An example motivating this concern is given in Figure 3–7. The rule in column a is attempting to express an inversion of the scope hierarchy relation between A and B. Consider, however, a source graph consisting of two connected nodes $\textcircled{X}$ and $\textcircled{Y}$ and a single ST, as depicted in column b of Figure 3–7. The LHS will match a node labeled $\textcircled{X}$ which is marked with innermost scope B, such that scope A is a parent scope of B. After execution of the RHS, node $\textcircled{X}$ is moved into the scope A. And the scope hierarchy manipulation occurs: scope B now becomes the parent of scope A, as shown in column c. The new scope hierarchy, however, indirectly affects node $\textcircled{Y}$; its immediate scope is still B, but scope B is now a parent of scope A, and thus $\textcircled{Y}$ is no longer (by transitivity) in scope A. This may be the desired, correct behavior, but is also potentially confusing in that it is not clear whether a scope pattern in the RHS is intended to represent a global transformation of scope relations or just specification of a single, bound host node's new scoping relation.

To more cleanly separate these issues we decided to perform SF and ST manipulations orthogonal to the main transformation rule rather than implicitly in

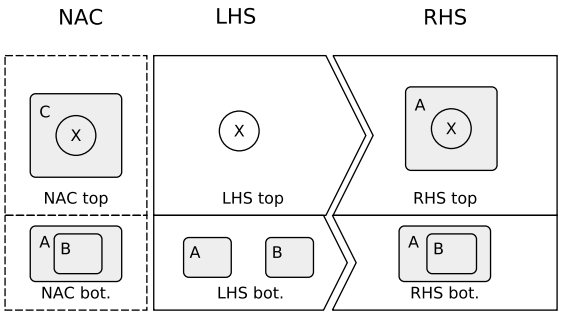

Figure 3–8: Extended model transformation rule.

the normal *RHS*. This ensures that the intent of any scope manipulations is overt. Attempts to modify the scope hierarchy itself in the *RHS* are thus disallowed.

Figure 3–8 presents an extension to our existing transformation rule that illustrates our final rule design. This rule creates a scope hierarchy between unconnected ST nodes A and B and places node $\textcircled{X}$ into scope A. Node $\textcircled{X}$ must not be in C scope as dictated in the NAC part of the rule. The top part contains DSL and scope patterns, while the bottom part of the rule is reserved for scope hierarchy manipulation only, with matching and rewriting performed in typical model transformation fashion on the SF. Thus, only labeled scopes with nesting to represent hierarchy are allowed in the bottom part. We will now refer to the top part as LHS_T, RHS_T, NAC_T and to the bottom part as LHS_B, RHS_B, NAC_B.

There are constraints on placing certain scope patterns in the top parts of the rule. Anonymous and dashed scope constructs are ambiguous in labeling source graph nodes in the RHS_T and should be disallowed when creating new innermost scope relationships. Nested scope constructs such as in pattern 4 from Figure 3–5

are not allowed in RHS_T if they attempt to modify the scope hierarchy because of the hierarchy manipulation problem described in Figure 3–7.

In RHS_T, labeling of source graph nodes with the corresponding innermost scope occurs and the bottom part with its RHS_B is reserved for scope hierarchy manipulation. A runtime check will ensure that the RHS_T is using (assigning) a scope that exists in the scope hierarchy. This is necessary when the scope is not matched prior in LHS_T or present in RHS_B, which guarantees scope existence through either matching or creation of the scope. If the scope does not exist the rule application will result in a failure. Scope well-formedness is discussed in Section 3.3.6.

Returning to our example of the ambiguous intent in the transformation shown in Figure 3–7. We can now use the extended rule structure to more clearly express the transformation engineer's intent. Figure 3–9 shows the resulting rule (a), and behavior in terms of the input scoped graph (b) and rewritten output graph (c). Here, in the top left part of the rule it is clear that we wish to match $\textcircled{X}$ with an innermost scope of B and a parent to that scope A. The top right shows us changing $\textcircled{X}$'s innermost scope relation from B to A. The actual SF manipulation itself, however, is now separately specified in the bottom part of the rule, showing that we require B to be nested immediately inside A, and wish to invert that relation. The end effect is the same, but the change to the SF by the rule designer's choice is intentional, explicit and is more clearly affecting the entire scoped graph.

Final argument in favor of our extended rule structure is to allow for the flexibility to execute the top part of the rule when the bottom part is applicable (LHS_B match is found in SF) and vice versa. It is also possible to omit either the top or

bottom parts of the rule. In this way a rule designer can easily separate *SF* manipulation from any transformation of the host graph and the scope relationships it includes.

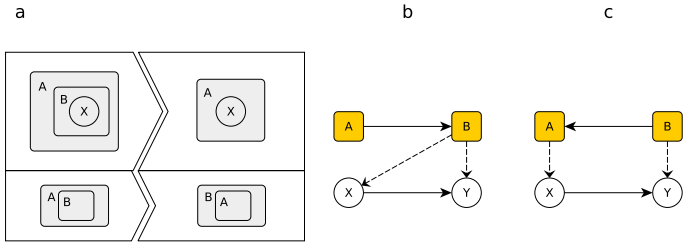

Figure 3–9: The extended rule applied to the ambiguity problem.

In the following section, we describe the semantics of scoped rule application.

3.3.4 Semantics

In this section, we give the semantics of our scoped rule application a visual description. In Figure 3–10 an extended rule application is shown using notation similar to that used in algebraic single-pushout (SPO) graph rewriting [58] (with the *NAC* portion of the rule omitted for brevity). Top and bottom parts of *LHS* and *RHS* are shown separately as they represent different parts of the extended rule. For simplicity the morphisms are indicated by the bold outlines of the nodes and edges. A morphism of the bottom parts pertaining to the *SF* are indicated by dashed bold rectangles. The application of this rule creates a node *Birch* in the scope hierarchy and a node Ⓨ connected to node Ⓧ. Both top and bottom patterns must have a match in the scoped graph for the rule to apply. That is $m\ top : LHS_T \rightarrow G$ and $m\ bot. : LHS_B \rightarrow G$ morphisms must exist.

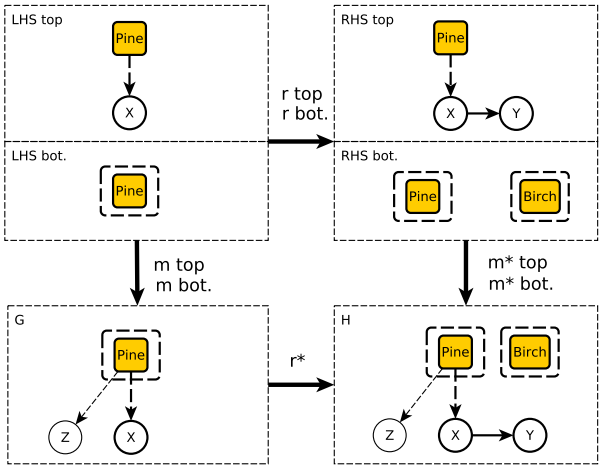

Figure 3–10: Scoped matching visualized (scoped graph syntax for patterns) using SPO notation.

Note that matching the scope labeled *Pine* in the top and bottom parts of the rule may seem redundant, as both ensure the existence of a node labeled *Pine* in SF. This need not be true in general, however, and there is no requirement that LHS_T be related to LHS_B, giving us the flexibility to modify the SF as a consequence of rule application, without needing that rule to directly refer to host nodes in the portion of the SF being modified.

Finally, due to a restriction on the use of unique labels in the SF, we ensure that the *Pine* scope patterns in both LHS_T and LHS_B match the same *Pine* node in G. The use of anonymous and dashed scope constructs is forbidden in the bottom part of the rule for the reason of ambiguity in modifying the scope hierarchy (see Section 3.3.6). This eliminates the situations with unclear matching semantics, such

as the presence of the anonymous scope constructs in both LHS_T and LHS_B parts of the rule.

3.3.5 Scope Matching Using Search Plans

We now present scoped matching in the context of search plans (SP). The SPs were presented in the background chapter in Section 2.4.1. We believe that this is the most natural way of dealing with scoped matching, because our scope constructs translate into search plans in a straightforward way. Below we discuss the generation of SPs for the scoped patterns and address possible matching scenarios and strategies.

Matching is a process of finding an occurrence or a binding of a pattern (including scoped pattern) in the input graph. We consider the scoped graph as a whole for the purpose of SP-based matching. Note that matching routines typically produce a set of occurrences of a pattern in a host graph, called the *matchset* [94]. A match is then selected from a matchset for a rewrite. To understand our process, however, it is sufficient to provide a description of locating a single match, and this can be trivially extended to multiple matches.

To demonstrate how scope augments the SP-based matching we start with a non-scoped pattern and add scope. In Figure 3–11 an input graph is shown in column *a*. Assume that the host graph is analyzed to collect statistical information for the purpose of calculating the costs of primitive match operations. In this demonstration, we count the number of types in the input model. We get 4 $\textcircled{X}$ node types and 4 *e* edge types between the nodes. The resulting search graph for the pattern given in column *b* is shown in column *c*. Edges are labeled with corresponding primitive match operations, their cost is shown after the comma, and a minimum spanning tree

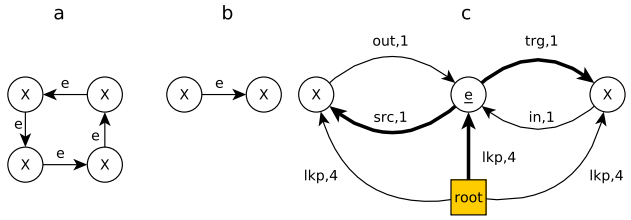

Figure 3–11: A minimum spanning tree over the search graph in bold edges on the right. The input graph is on the left, and the pattern in the middle.

is marked over the search graph with bold edges. Note that this minimum spanning tree is not unique, and thus other lowest-cost search plans are possible. In this case a SP is P_1=*lkp(e),src(e),trg(e)* with a cost of 12 ($c(P_1) = 4 + 4 * 1 + 4 * 1 * 1 = 12$).

Consider Figure 3–12 with the same layout as Figure 3–11. We add scope to the input graph and augment the pattern with the scope (shown using input graph syntax, a dashed edge representing an innermost scope relationship). The input graph statistics remains unchanged by the addition of a single scope node and its immediate scope edge (labeled s). The resulting SP from a minimum spanning tree P_2=*lkp(Pine),out(Pine,s),trg(s),out(X,e),trg(e)* is longer in terms of the number of operations. The scoped search plan however, is much cheaper at a cost of 5 ($c(P_2) = 1 + 1 * 1 + 1 * 1 * 1 + 1 * 1 * 1 * 1 + 1 * 1 * 1 * 1 * 1 = 5$).

The generation of scoped SP can be applied in the same way to labeled and anonymous scopes with nesting, as well as the scope intersections after flattening operation. Simple scoped patterns are treated just like any other pattern. If the anonymous scope construct is present in the search graph, the primitive match operations should accommodate returning a binding for any scope node in the *SF*.

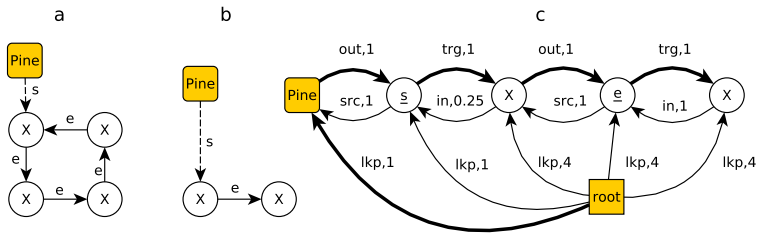

Figure 3–12: After addition of scope to the input graph and the pattern, the search plan generated from the search graph on the right has a cost of 5 as opposed to 12 without scope.

This could mean considering all the nodes in the *SF* which is expensive. Therefore, depending on the input graph statistics, it could be beneficial to force the SP to start from the input graph node and then match up the scope hierarchy until the outermost scope construct. This can be demonstrated in an alternative (and not necessarily the best) search plan, searching right to left through Figure 3–12. In this case, we start at an input graph node and match up the scope hierarchy: P_3=*lkp(X),in(X,e),src(e),in(X,s),src(s)*.

Dashed Scope Matching is addressed next. The dashed scope constructs require slight modification to the search plan-based matching. We introduce two extra match operations *down* and *up*. These two new operations are functions that contain calls to the primitive match operations with additional branch and loop control structures. In Figure 3–13 in column *a* is the scoped pattern and the search graph corresponding to that pattern in column *b*. Now, during search graph generation the dashed construct within a labeled scope is treated with these new operations. The match operation *down* is used when the matching process starts binding the pattern from the scope node (*Pine* node in this example) down the hierarchy towards the

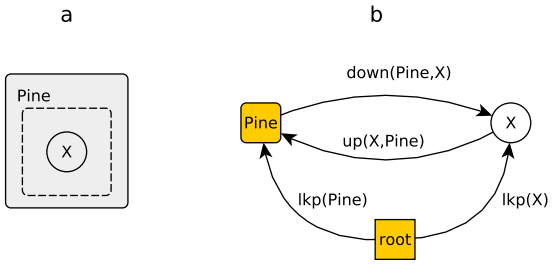

Figure 3–13: The search graph in column *b* displaying two new match operations targeting dashed scope constructs

input graph node (*X*). The *up* operation, on the contrary, is used when we match up the scope hierarchy from the input graph node. In Figure 3–13 lookup operation edges are present. Based on the input graph statistics, lookup of the node (including the scope hierarchy nodes) whose type is least represented in the model may be chosen to be at the start of a search plan list. This results in a matching up or down the hierarchy.

The *down* operation requires two parameters, an already bound scope node and the information about the non-scope node where the pattern terminates. The information about the non-scoped node is used to decide whether matching reached the terminating host graph node. In reverse, the *up* operation requires an already bound host graph node and the information about the terminating scope pattern node. In a case when the scope pattern node is an anonymous scope, the generic type of the scope node is provided to indicate that any node of the *SF* will be satisfactory for a binding. Then, the *up* routine should terminate as soon as the innermost scope relationship from the bound host graph node is found by traveling up the scope relationship edge satisfying shortest match semantics. In Figure 3–13

operation costs are not shown. At this point we do not attempt to reason precisely about the cost of the *up* and *down* operations, except to estimate the maximum or minimum values based on the information about SF. The cost depends on the number of levels and branching of the STs. Let us first consider dashed scope patterns inside labeled scope as in Figure 3–13. The best case for an *up* operation is when *Pine* is the immediate scope to $\textcircled{X}$. The worst case is when the scope of interest is not present or is the root of the ST. Similarly, for a *down* operation the intermediate scope relationship presents the best case. However, the worst case is when all of the ST is traversed from the root scope down.

When the anonymous scope contains dashed scope, a *down* operation can be very expensive. Bindings to all scope nodes may need to be evaluated in an effort to discover the host graph node down the ST. In addition to that, the match can contain portions of the ST of different length (depending on the starting point). The *up* operation starting from the host graph node will match the very first immediate scope node discovered. Depending on the cost of a binding to the host graph node, this can be an expensive operation. Calculating the cost of the nested operations presents an interesting case for future work in the field of search plans.

Below are algorithmic descriptions of *down* and *up* operations. For brevity, we assume that the primitive operations inside these functions return a single binding out of all possible; anonymous scope treatment is also omitted because extending these functions to support such scopes is trivial. To concisely convey the semantics of dashed scope matching in a simple way, in these functions, we do not deal with failures to produce a binding and the resulting backtracking. The S parameter passed

into *in* and *out* match operations represents a generic scope hierarchy edge type that needs to be bound, including the innermost scope edge.

```
function DOWN(Scope, Node)
    result = Scope
    while result != Node do
        ScopeEdge = out(result, S)
        result = trg(ScopeEdge)
    end while
    return result
end function
```

```
function UP(Node, Scope)
    result = Node
    while result != Scope do
        ScopeEdge = in(result, S)
        result = src(ScopeEdge)
    end while
    return result
end function
```

Inside the *down* function, the outgoing scope hierarchy edge is bound and stored inside the variable *ScopeEdge*. The edge is then used to bind the target node at its endpoint. The binding stored in *result* is tested for the conformance to the terminating non-scope node. If it conforms, the binding is returned, if not, the routine continues looking for the non-scope node in a depth-first fashion. Note that intermediate scope hierarchy bindings are not returned from this function. If necessary, these bindings can be stored within the function and returned. In the case of the *up* function, the same principles describing *down* operation apply, only in reverse. From the starting binding of the non-scope node, the matching travels up the scope hierarchy edges until the labeled scope (outermost scope in the pattern) is reached. We continue with a description of the rewriting stage of the scoped rule application.

3.3.6 Rewriting

After the discovery of the match, the rule application process enters the rewriting phase. This phase is described in this section.

As mentioned in Section 3.3.3, we rely on unique labels within the rule patterns to determine which nodes and edges are being added as opposed to being removed or updated in the match bindings. If we have a match for both LHS_T and LHS_B, we perform the corresponding graph modifications to rewrite the host graph and the immediate scope relationships according to RHS_T, and the SF, according to RHS_B. Recall that in RHS_T, we only deal with innermost scope relationships. Therefore, all scope relationships and nodes that are not intermediate to the host graph nodes in the LHS_T are ignored for the rewriting. Rewriting within the scoped graph occurs using standard graph transformation operations that add, update, or remove nodes and relations. Note that treatment of NAC scoped patterns is similar to the treatment of LHS patterns described above. The efficient treatment of NAC patterns when common parts of LHS and NAC are matched first is outside of the scope of this thesis.

Scope Well-Formedness is addressed next. The additional complication shows up in terms of ensuring that the innermost relation between host and SF nodes is properly updated and still well-formed. Our constraints on RHS_T ensure that we only need to consider innermost bindings of RHS_T, but even there we still need to ensure that the innermost bindings are not created to the new or matched

scope nodes that would violate our property of each host node having only one innermost scope in a given ST. Failures in this represent runtime errors, as do rewrites that would violate the forest nature of the SF.

A constructive technique presented in [40], which derives application conditions from global constraints and adds them to the transformation rules to ensure valid models (w.r.t. constraints) by design, could possibly be used in this context as well. In [12] the authors ensure EMF model consistency, such as avoiding cyclic containment, by introducing restricted rules. That work could be applied to ensuring certain consistency requirements of our scope hierarchy forest, such as an absence of cycles. Another way to ensure consistency is by using a tool such as IncQuery [100]. This tool is used successfully in industry to facilitate verification of models by efficiently matching (IncQuery uses incremental pattern matching) anti-patterns, the patterns that break consistency. It would be possible to implement scope hierarchy consistency verification as the rules are being constructed. This would be facilitated by the fact that the bottom part of our scoped rule is reserved for scope hierarchy manipulations. The MT engineer could then be alerted to any problems in advance of executing the transformation.

As mentioned in Section 3.3.3, the top part of the rule is intended for innermost scope manipulations. Thus, RHS_T will only have labeled scope constructs without any nesting when new innermost scope relationships are created. The scoped pattern found in the LHS_T may need to be replicated in the RHS_T if the intent is to preserve the relationship discovered. The bottom part of the rule can only contain labeled scopes (with possible nesting) to allow for SF manipulations. Anonymous and

dashed constructs are not permitted in the bottom part. We may also need to delete innermost bindings from arbitrary other nodes if a scope node is destroyed. This requires simply removing dangling edges between deleted scope nodes and affected source graph nodes. Runtime well-formedness verification implies the use of some form of the transactional rewriting system with checkpointing and backtracking. In case of a failure, the previous well-formed state of the scoped graph is restored. Note that T-Core within AToMPM supports backtracking.

We can also envision automated debugging scenarios, presented in Chapter 5, used to evaluate scoped transformations. Debugging scenarios can be defined (using declarative patterns) to catch illegal *SF* modifications, such as loop creation, etc. The MT is then executed under the supervision of the debugger running the scenario. In case an illegal pattern is discovered, the execution can be halted and appropriate alert issued to the engineer debugging scoped MT.

In general, and although pathologically expensive scope-based manipulations can be easily defined, we envision the scope hierarchy to be much smaller than the related host graph, and scope modifications to be much less frequent. We thus expect the overall *SF* modification time to be negligible, at least in comparison to the cost of host graph manipulations. In the next section, we continue with the possible implementations of scope applicable to several benchmark transformations.

3.4 Implementation

In this section, we first describe the mutual exclusion and the forest-fire simulation transformations. We then discuss a basic design in the industry-standard context of QVT and the state-of-the-art graph rewriting tool GrGen demonstrating

that our approach does not require any fundamental changes to existing transformation systems in order to realize an implementation. The implementation of our scope concept in GrGen in the context of the forest-fire and mutual exclusion benchmarks are evaluated to establish the performance benefits. In addition, a prototype implementation in our tool AToMPM, allows us to show a preliminary performance comparison between a scoped and a non-scoped (baseline)[2] implementations of the forest-fire and mutual exclusion benchmarks. These examples are used throughout the thesis. We proceed with a description of a mutual exclusion MT benchmark.

3.4.1 Mutual Exclusion

We evaluate the use of scope on a benchmark from the model transformation community. For this, we chose the distributed mutual exclusion benchmark in its as long as possible (ALAP) form, as introduced in [109] with a complete specification presented in [39]. In this benchmark, we model the processes that are attempting to access shared resources. In order to ensure exclusive access to resources, the processes are interconnected to model a token ring. The access to resources is achieved through the passing of the tokens in the ring.

A metamodel of the mutual exclusion problem is shown in Figure 3–14. There are two classes called Process and Resource. These classes are connected by associations of type next, request, held_by, release, token, and blocked. The final reference blocked is not used in the ALAP transformation.

[2] Non-scoped and baseline implementations are the same and are used interchangeably.

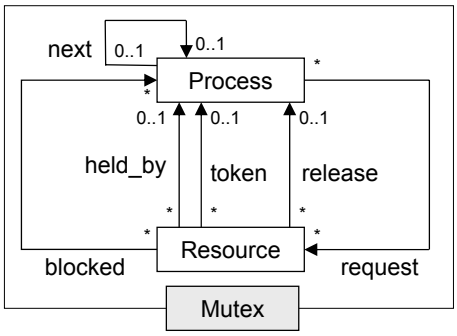

Figure 3–14: Metamodel for the mutual exclusion problem [108]

In Figure 3–15 the four rules of the ALAP mutual exclusion benchmark transformation are presented (the original, complete problem contains the total 13 rules in [39]). These rules constitute the baseline, non-scoped MT. Note that we will revisit this baseline transformation in Chapter 4 for another evaluation. We give an explanation of the releaseRule as an example. The rule in its *LHS* requires Resource 1 to be held_by a process Process, while in its *RHS* the same nodes Process and Resource 1 become connected by an edge of type release. The *releaseRule* also has a *NAC* (shown in dashed rectangle) which ensures that the process Process does not have any requests issued for any resources. A single iteration of the mutex benchmark transformation is scheduled such that the rules are executed in the following order: *releaseRule*, *giveRule*, *requestRule*, and *takeRule*. Each rule is executed N times, this number is equal to the number of both resources and processes in the input model. Another version of the benchmark exists where there is only one resource for the process ring, called short transformation sequence (STS) [108].

To explain the essence of the transformation, on the left in Figure 3–16 is an example of an initial mutex model. The edge labels are omitted, but the processes are interconnected using next associations, forming a ring of processes. The resources

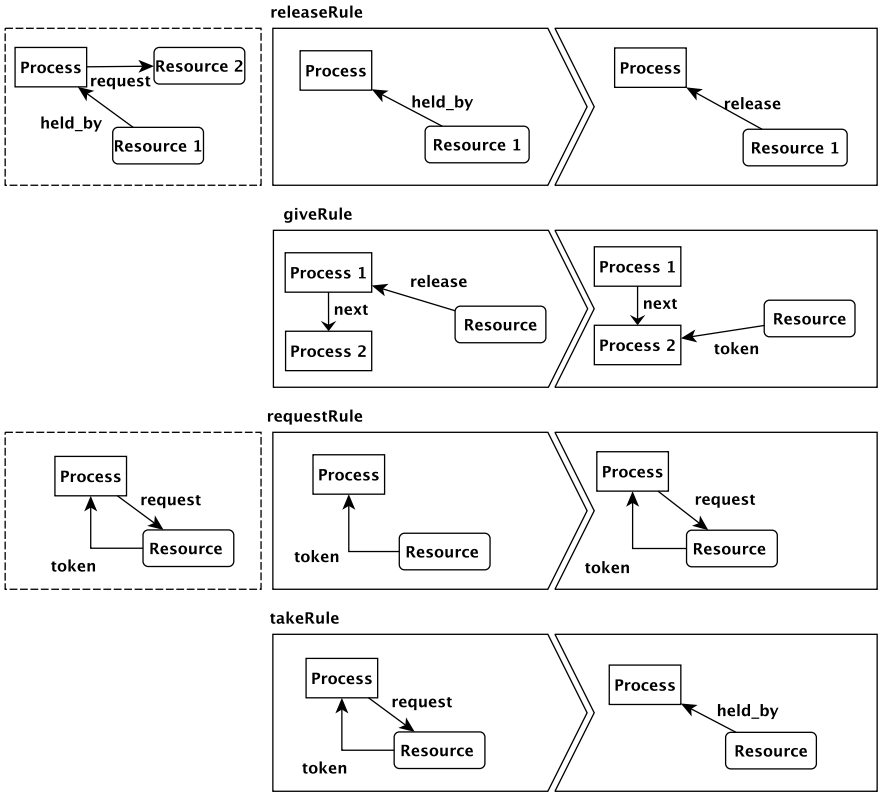

Figure 3–15: A subset of rules describing the ALAP mutual exclusion transformation

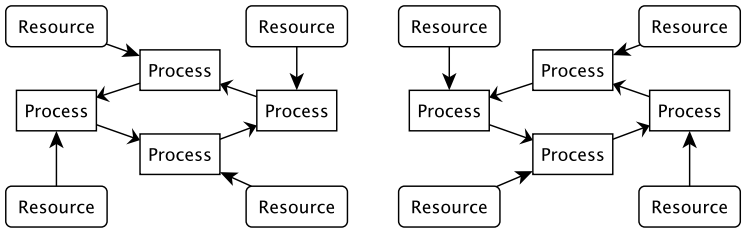

Figure 3–16: The initial mutual exclusion model on the left and the resulting model after transformation sequence execution on the right. Each resource is moved to the next process in the process ring.

corresponding to a process are linked with held_by association. The resulting output model is shown on the right. As a result, each resource is moved to the next process in the process ring. Parallel execution of this benchmark is possible, however, we are concerned with the sequential execution of this benchmark because AToMPM does not support parallel rule execution.

To achieve scoped MT, we place the resource model elements into a scope, and the other model elements such as processes and associations are not placed into scope. Thus, each rule of the transformation uses scope for matching resources except for the *releaseRule* transformation rule that is used to place each resource into the only scope in the *SF* called *sResource*. In Figure 3–17 is the example of the *giveRule* transformation rule with the added scope labeled *sResource* (other scoped rules are omitted for brevity).

Note that it is, of course, possible to implement scoped transformation in different ways (this is also true for the forest-fire simulation described in the following section). For example by placing processed model elements into the scope to eliminate them from consideration in the following rules. In this thesis, we added the

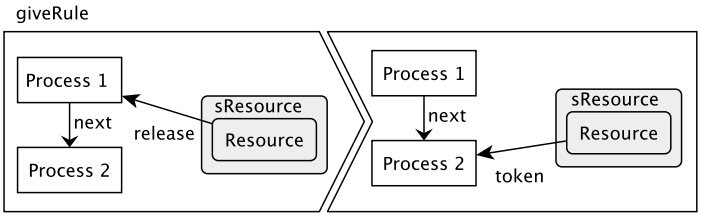

Figure 3–17: Scoped *giveRule*; resources in scope *sResource* are used for matching the pattern in *LHS*.

scope to the benchmarks in a way to keep the likeness of scoped and non-scoped rules as much as possible (no introduced *NAC* for example) and capture the resulting performance effects. A short transformation sequence (STS) mutex benchmark was briefly evaluated in GrGen (using the implementation provided in the GrGen source tree). Next, we describe the forest-fire simulation MT and its implementations in several MT tools.

3.4.2 Forest-Fire Simulation

In experimental work we wanted to validate that our scope model was feasible in implementation, allowed for reasonably intuitive rule constructions, and was able to demonstrate some improvement in efficiency. For this we used our running example with its *Region* scope hierarchy, examining both scoped and non-scoped designs in AToMPM and GrGen. The forest-fire example was partially motivated by the "comb structure" MT benchmark from the MT benchmark suite [109]. This benchmark is used especially for evaluating the pattern matching efficiency. We show the benchmark input model and the comb pattern itself in Figure 3–18. The essence of the comb structure benchmark is the discovery of the comb pattern in the grid input

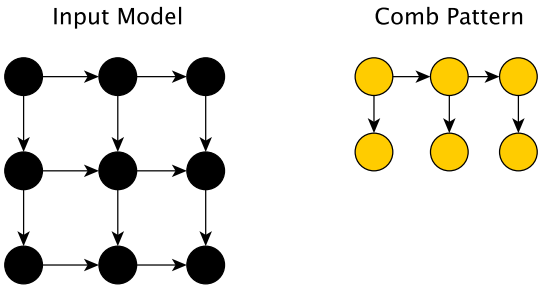

Figure 3–18: The grid input model and the comb pattern from the comb structure benchmark

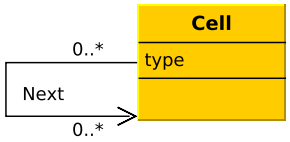

Figure 3–19: Forest-fire abstract syntax

model that itself is not modified. The grid of this benchmark is also similar to our forest-fire grid. Note that in this thesis we demonstrate the application of scope to in-place transformations; however, there is no restriction on the use of scope in model-to-model transformations.

Figure 3–19 shows the abstract syntax model of the forest-fire spreading formalism. The *Cell* class instances can form the forest by connecting to each other using the *Next* association. For simplicity we do not create any specialization of the connection between the cells such as North, East, or West. The state of the forest cell, healthy, on fire, smoldering or dead, is denoted by an integer attribute *type*, as inspired by Muzy *et al.* [73]. The concrete (visual) syntax of the cell icon is a rectangle (although the underlying representation is fully graph-based), colored

according to the state of the cell: healthy–green, on fire–red, and dead–gray. The smoldering cells have no color of their own for simplicity and to allow scope labels B displayed over them as shown in Figure 3–23 on the right.

Implementation in AToMPM

The operational semantics of the forest-fire simulation are defined using transformations. To be able to compare the scoped transformations to non-scoped ones, we design transformations to produce similar simulation results. In our case, the fire-spreading pattern is uniform for both transformations.

The baseline transformation is presented in Figure 3–20. The first rule, "Cells catch fire" marks (by changing the *type* attribute to smoldering) the cells neighboring the cells with burning trees (they catch fire). Note that in Figure 3–20 and Figure 3–21 we show undirected edges in "Cells catch fire" rule. In the actual implementation, the rule has two versions for the incoming and the outgoing edges from and to the neighboring cell to catch fire. The smoldering cells are then set on fire in the rule "Cells ignite". Note that we do not regulate burning time for both transformations, since this is an as fast as possible simulation. The final rule "Cells burn out" finds the cells that are on fire and marks them as dead. Thus the fire front spreads in the same fashion as in the scoped transformation described below.

Our scoped transformation uses the *Region* scope hierarchy from our running example. For simplicity in this simulation, the scoped rules do not utilize the hierarchical nature of the SF. Instead we just use the leaves of *Region* ST, such as with F as shown in Figure 3–21. There are 4 rules in the core of fire spread simulation, as displayed in Figure 3–21. The first rule "Init F scope" is intended for

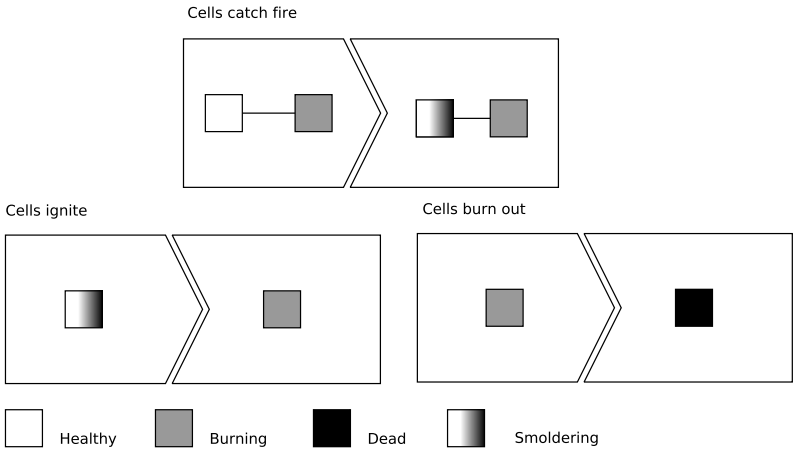

Figure 3–20: Core rules of the baseline forest-fire simulation shown using AToMPM syntax. Here and in the scoped transformation the cells are colored according to the *type* attribute value.

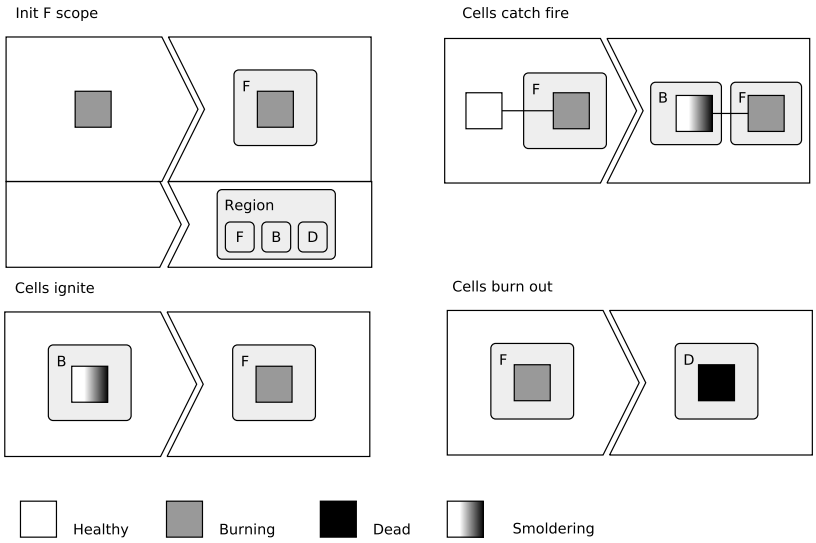

Figure 3–21: Core rules of scoped forest-fire simulation shown using AToMPM syntax.

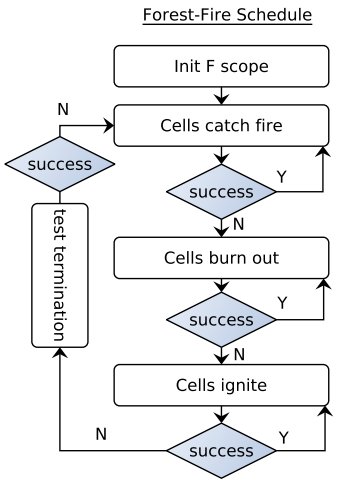

Figure 3–22: The sequential scheduling of the scoped and non-scoped rules.

one-time initialization; it places a cell on fire into F scope and also creates *Region* ST in SF. The initialization rule also uses our extended rule structure to change the SF. Note that the subsequent rules omit the bottom part of the scoped rule, because it is not used. Rule "Cells catch fire" puts any neighbors of cells in the F scope into the B scope (they catch fire) and marks them as smoldering. Rule "Cells ignite" sets the cells in B scope (smoldering) on fire and puts them into F scope. Finally, the rule 'Cells burn out" seeks out the cells in F scope. The cells are then marked dead and put into scope D. The transformation rules for both the scoped and non-scoped implementations are then scheduled as per the sequential scheduling approach is given in Figure 3–22 (the first rule "Init F scope" is scope-specific). The sequential approach aims to produce uniform fire spreading (in our case circular, as in taxicab geometry, since we do not model the effect of the wind).

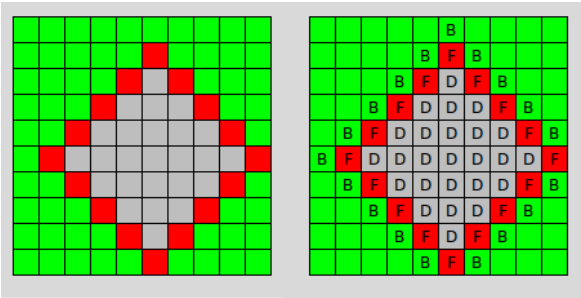

Figure 3–23: The Result of execution in both baseline (left) and scoped (right) cases in AToMPM simulation: the fire spreads uniformly.

AToMPM simulation screen shots of the results for both baseline and scoped transformations are shown in Figure 3–23. We chose to use Tkinter [44] as our canvas for fast prototyping and running transformations in a stand-alone rendering outside AToMPM. The use of Tkinter allows us to display extra information that is not part of the formalism, e.g. scope of the source node.

Implementation in QVT

We include excerpts of QVT-Relational (QVT-R) [76] and QVT-O transformations implementing our scoped forest-fire simulation. These examples were provided by Maged Elaasar a co-author of the "Scope in Model Transformations" journal paper [46]. In the examples below, the class *Cell* models a single cell in the forest grid. The cell neighborhood relationship is denoted by the *next* property (similar to the *next* association in Figure 3–19). Property *type* represents the enumeration corresponding to the state of the forest cell described in Section 3.4.4. The QVT transformation's intermediate property called *Scope* is used to model our scope. This property is defined as an enumeration of F, B, and D scopes from our running example. Note that the intermediate property can be of arbitrary complexity to model

our scope formalism (in Figure 3–24), enabling hierarchical scope constructs during transformation. The intermediate property is a good facility to implement scope as it does not exist outside the context of the transformation that defines it. This ensures a non-invasive scope application as the MM of the language transformed will remain unchanged. This can be a good strategy for scope use in *scope-unaware* MT systems.

Below is the QVT-R transformation with the implementation of our scoped "Cells catch fire" rule (in Figure 3–21). The rule "Cells catch fire" puts any healthy forest cells adjacent to cells in F scope into the B scope (they catch fire). Definitions of the QVT classes *Cell* and *ForestFire* are not listed.

```
metamodel ScopeMM {
    enum Scope { F, B, D };
}
transformation FireSimulation (ff : ForestFire) {
    intermediate property Cell::scope : Scope;
    toplevel relation CellsCatchFire {
        checkonly domain ff healthy:Cell {
            type = healthy,
            next = burning:Cell {
                scope = Scope::F
            }
        }
        enforce domain ff healthy:Cell {
```

```
            scope = Scope::B
            type = smoldering
        }
    }
}
```

The same rule in QVT-O is presented below.

```
metamodel ScopeMM {
    enum Scope { F, B, D };
}
transformation FireSimulation (inout ff : ForestFire) {
    intermediate property Cell::scope : Scope;
    mapping inout Cell::CellsCatchFire()
    when {
        self.type = healthy
        self.next->exists(scope = Scope::F)
    }
    {
        scope := Scope::B
        type := smoldering
    }
}
```

Implementation in GrGen

We also consider the forest-fire simulation for scope evaluation using GrGen. We ported the non-scoped transformation from Figure 3–20 and its schedule to GrGen and refer to it as a baseline transformation. Below is the baseline "Cells catch fire" rule.

```
rule CellsCatchFire {
    t1:Cell <-:Next- t2:Cell;
    if {t1.type == 1 && t2.type == 0;}
      modify {eval {t2.type = 2;}}}
```

We then use some of the available optimizations GrGen provides (listed in Section 2) to implement/simulate a scope-aware system. Note, that in the actual implementation of the simulation we use a node replacing an edge of the type *Next* between the forest cells to align the benchmark with the implementation in AToMPM.

Container. First, to model the scope, sets are introduced directly into the rules, a transformation we refer to as "Container". The forest cells, such as the cells on fire are maintained inside a single set container and so there exists a container for each region of the forest-fire. GrGen then performs the search plan pattern bindings from these containers. It is up to transformation engineer to maintain model elements inside the containers, whereas the AToMPM scope implementation aims at a transparent and automatic scope implementation. Below is the "Cells catch fire" rule. Notice the familiar F and B scope mirroring variables.

```
rule CellsCatchFire (ref F:set<Cell>, ref B:set<Cell>){
    t1:Cell{F} <-:Next- t2:Cell;
```

```
    if {t2.type == 0;}
      modify {
        eval {
          B.add(t2);
          t2.type = 2;}}}
```

The *t1:Cell*{*F*} construct signals to the search plan backend to perform a binding to the pattern variable *t1* during a lookup from a container set named *F*.

Index. The forest cell attribute *type* goes hand in hand with the forest-fire regions. The cells with *type* values equal to 1 represent the fire front and so forth. We thus implement the attribute indexing inside GrGen, referring to this transformation as "Index." First, the index for cell types is specified in the MM. Then, the index is used in rules for binding pattern elements. GrGen takes care of index maintenance. Below is the same "Cells catch fire" rule using the *TYPE* attribute indexing that embodies scoping or grouping based on attributes values.

```
rule CellsCatchFire {
    t1:Cell{TYPE==1} <-:Next- t2:Cell;
    if {t2.type == 0;}
      modify {eval {t2.type = 2;}}}
```

The *t1:Cell*{*TYPE=1*} construct signals to the search plan backend to perform a binding to the pattern variable *t1* during a lookup from the dictionary where keys are *type* attribute values.

Scoped graph. Finally, we implement the scoped graph by adding scope nodes and scope relationships to the forest-fire MM. We refer to this transformation as

"Scoped graph" in this section, and our scope implementation is a direct mapping of our scope concept onto GrGen. Placing of the forest cells into scopes is performed by drawing a scope relationship edge between the cell nodes and the scope nodes. In the rule below, the immediate scope relationship is matched in the pattern using the edge of type S. Below we show the "Cells catch fire" rule using the scope directly in the input graph.

```
rule CellsCatchFire {
    F[prio = 10000]:Scope{NAME=="F"} -:S-> t1:Cell <-:Next- t2:Cell;
    B:Scope{NAME=="B"};
    if {t2.type == 0;}
      modify {
        B -:S-> t2;
        eval {t2.type = 2;}}}
```

The scope is now part of the pattern. In this rule, we use *prio* to ensure that the search plan starts matching from the scope node F. We also use scope name indexing to locate a scope node based on its name. In the rewrite part we place the *t2* forest cell into scope B with *B -:S-> t2;*.

Amalgamated Forest-Fire. It is possible to use amalgamated rules to implement forest-fire transformation, and we also modified the forest-fire example to investigate the effects of scope on the amalgamated rules. For this, we use undirected edges between the forest cells. Before, to align the GrGen example with the implementation in AToMPM, the forest-fire transformation was implemented using directed edges. The undirected edge implementation is simpler and cleaner, as it

allows us to specify successor and predecessor neighbors with one rule (in GrGen, it is still possible to make a pattern that matches directed edges in both directions). Below is the meta-model of the modified forest-fire example including the scoping information.

```
node class Cell { type: int; }
node class Scope { name: string; }
edge class S
    connect Scope [1] --> Cell [1];
undirected edge class C2C;
```

We amalgamate the "Cells catch fire" rule. Its non-scope, baseline version is shown below. In this rule, after a single execution all cells neighboring burning cells catch fire. The attribute indexing in the baseline transformation is also performed.

```
rule CellsCatchFire {
  multiple{
    t1[prio=10000]:Cell{TYPE==1};
    multiple {
      t1 -:C2C- t2:Cell;
      if {t2.type == 0;}
        modify {eval {t2.type = 2;}}
    } modify {eval{}}}}
```

We now add scope in a similar fashion to how we showed earlier. In the rule below, a single match of the scope node F, is used to iterate over all scope edges to find the neighboring cells of all burning cells.

```
rule CellsCatchFire {
  B:Scope{NAME=="B"};
  F[prio = 10000]:Scope{NAME=="F"};
  multiple{
    F -:S-> t1:Cell;
    multiple {
      t1 -:C2C- t2:Cell;
      if {t2.type == 0;}
        modify {
          B -:S-> t2;
          eval {t2.type = 2;}}
        } modify {eval{}}}}
```

Note that we now use an iterator *multiple* in the amalgamated rules of the forest-fire example. Previously, in AToMPM and GrGen the rewrites happened on the first match found. As demonstrated with an amalgamation example in Section 3.4.5, the iteration may not always be beneficial. In the next section, we briefly discuss an implementation of scope-aware MT system in AToMPM.

3.4.3 Implementation of Scope in AToMPM

A prototype of scoped transformations is implemented in the AToMPM meta-modeling and graph rewriting tool [59]. Our design is intended mainly to establish additional proof of feasibility, and to serve as a baseline for further research into optimizing performance. At the same time, we use this naive approach as another

example, demonstrating that our scoped model can be easily and incrementally integrated into an existing framework.

For scope implementation in AToMPM we use our model sensitive search plan matcher. There, the *SF* is implemented trivially, by using sets containing input graph node identifiers. Each set represents a single scope. This implementation is not well suited for dealing with hierarchical scopes as opposed to what was shown in GrGen where we directly write scope hierarchy into the input graph. Nevertheless, this approach allowed us to easily use the existing subgraph matching algorithm. Within this process, we do make one important modification to the part of the algorithm where candidate nodes are evaluated for compatibility in type/name and degree during a primitive lookup operation. Here we prioritize scope in the search plan generation process causing search plan to start from scope node. The candidate bindings are then taken from the scope sets. Only the nodes that are *in scope* are considered as candidates. As we will show, even this simple change led to a performance improvement.

Other changes were made to accommodate the new rule and the scope formalism in the tool. For this, we need syntactic changes to introduce a universal scope formalism that can be used with any AToMPM DSL model in transformation rules. For abstract syntax, AToMPM uses a variant of the UML diagram formalism, and a reconstruction of the abstract syntax model is shown in Figure 3–24. The class *Scope_* represents a scope. The class *$** is an implementation-specific way of defining a wildcard class (i.e., *any* class). The type of *Has_* association is containment, such that it allows *Scope_* instances to contain any class instance, including *Scope_* instances

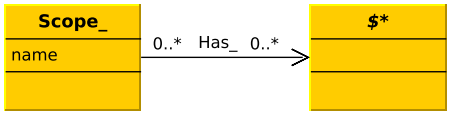

Figure 3–24: Scope abstract syntax used in AToMPM.

themselves. This allows for hierarchical construction of scope. In addition, the resulting implementation makes our tool scope-aware. Scope is now a part of MT language formalism, applicable to any DSL without modification of its MM. Such scope integration becomes transparent to the user, and can now be used without the far-reaching effects of a language specification modification. This also addresses an aspect of a model and model transformation evolution problem [69]. Finally, as a side effect, in this fashion the scope hierarchy can also be used to encode certain domain-specific information omitted at the language design time. We now continue with a description of our experimental setup to evaluate scoped MTs.

3.4.4 Experimental Evaluation

In the experimental evaluation, we investigate if our scope concept can indeed bring performance improvements. This implies the following research questions:

- Does the application of scope to the baseline transformation reduce the total transformation execution time?
- What is the penalty for scope maintenance?

To answer these questions we perform the following experiments. For the forest-fire benchmark, the forest grids of $N \times N$ cells similar to one on the left in Figure 3–4, are generated in conformance with the meta-model. In AToMPM the grid is generated programmatically for N values of 100 and 200. In GrGen transformation rules

are used to create the grid for N values of 100, 1000, and 2000. Smaller grid sizes in our tool compared to a highly optimized GrGen were necessary because of the performance advantage of GrGen. A single cell in the middle of the grid is placed on fire before executing the transformation. In AToMPM scoped and baseline transformations were executed, while in GrGen baseline (non-scope), index, container, and scope graph experiments were executed. Each experiment was executed 3 times and the average was taken for total, match, and rewrite times of the simulation. Rewrite time allows for estimating scope maintenance penalty. The amalgamated forest-fire example was evaluated on a grid of 1000 by 1000 cells, on which baseline and scoped transformations were executed. Because the baseline amalgamated transformation is already using attribute indexing, we exclude indexing/container type experiments for the amalgamated forest-fire transformation in GrGen.

The mutual exclusion ALAP benchmark was executed in AToMPM. Initial mutex models for N values of 100, 1000, and 10000 were generated programmatically (N represents the number of both resources and processes in the input model). Each experiment for scoped and non-scoped transformations was executed 3 times. The average total, match, and rewrite times were taken. The short transformation sequence (single resource) mutex benchmark implementation found in GrGen source tree was augmented with scope and evaluated on a million process model. Parallelization was purposefully omitted from the evaluation as the benchmarks implemented in our AToMPM tool do not execute transformations asynchronously. The evaluation was performed on an x64 i7 mobile quad-core processor with 16 GB RAM running

Ubuntu 12.10. The results of the static scope experimental evaluation are presented next.

3.4.5 Results

Note that throughout this section standard deviation is not reported, as it was within ten percent of the mean.

Forest-fire results. In Figure 3–25 total, rewrite and match times are presented for the forest-fire benchmark in GrGen. The total time is displayed to contrast for the unaccounted time in the case of container and index experiments. GrGen match and rewrite times were taken as reported by the system. It appears that container and index maintenance time is not fully accounted for. There is also an anomaly for the index simulation. For $N = 1000$ match and rewrite times add up to match the total execution time. In $N = 2000$ however, this is not the case. A possible explanation is the large graph size and that at such size index maintenance becomes an issue and it is not tracked.

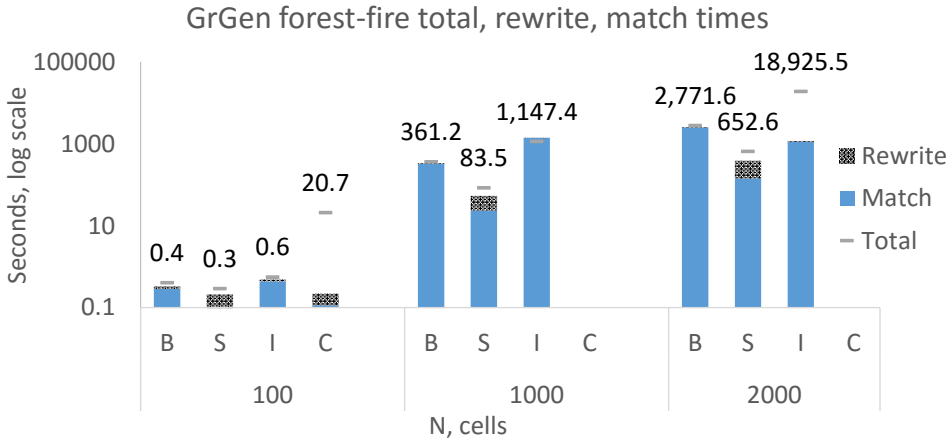

Figure 3–25: GrGen forest-fire total, rewrite and match times for Baseline (B), Scope Graph (S), Index (I), and Container (C) variations. Note the log-scale in time.

The container transformation results are only reported for $N = 100$. Due to large forest grids, the runtime exceeded the practical limit of 10 hours and further experiments were not executed. The container scope implementation is not feasible due what likely is a prohibitive penalty for container maintenance. The scope implementation using indexing does not show a speed up compared to a baseline transformation, although index maintenance is evidently cheaper than container maintenance judging by total time. Finally, from the results, we conclude that applying the scope directly into the host graph is a better optimization to an already fast baseline transformation in GrGen. This is attributed to the search plan cost reduction described in Section 3.3.5.

The only downside to the scoped graph use is shown by the larger portion of the transformation dedicated to the rewriting comparing to other transformations (this is also true for the container transformation). However, the rewriting penalty associated with the scope maintenance within the input graph is diminished by the improvement in matching and total time compared to the baseline transformation.

In Figure 3–26 we show total, rewrite and match times for the forest-fire benchmark executed in AToMPM. Is it clearly evident that the scoped transformation outperforms baseline by close to two orders of magnitude, again the only side effect being the increase in the rewriting time due to scope maintenance.

The results of the amalgamated forest-fire transformation demonstrate the effects of scope inclusion in GrGen. On average, the baseline amalgamated transformation on 1000 by 1000 cell grid took 1448 seconds. This result is similar to the indexed transformation for the same grid size model in a non-amalgamated version.

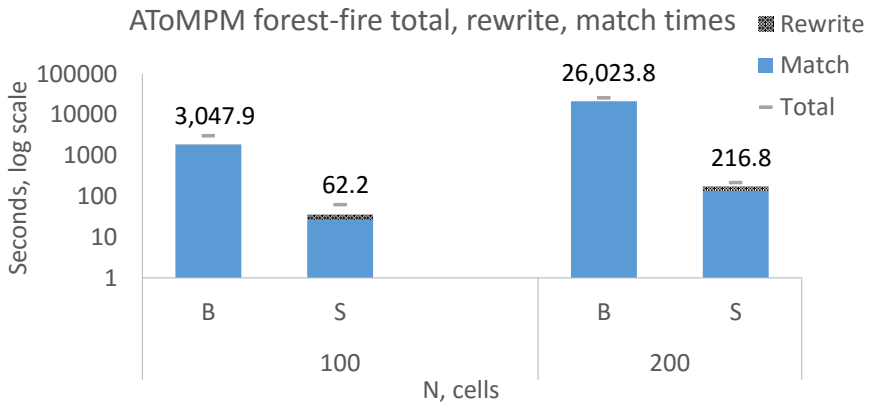

Figure 3–26: AToMPM forest-fire total, rewrite and match times for Baseline (B) and Scope Graph (S)

This result does not demonstrate an improvement. We believe that this is due to the use of undirected edges and possibly the amalgamation itself. The amalgamation may help to reduce matching costs in theory, because a series of steps is merged into one. However, the actual implementation of the amalgamation within a tool does not necessarily guarantee improved efficiency. It remains unclear whether amalgamation would improve efficiency in another tool.

The addition of scope shortened the total simulation time to about 43 seconds. This demonstrates an excellent use case for scope. Even though the non-amalgamated forest-fire model was larger (due to the encoding of directed edges), its baseline runtime is faster than the amalgamated baseline. The indexing in the amalgamated example, as opposed to the non-amalgamated one, was beneficial. Removal of indexing in the amalgamated example was detrimental to performance resulting in a runtime of about 5500 seconds on average. This indicates that the indexing was a good performance hint to the pattern matcher in this situation.

Mutex results. In Figure 3–27 total times for the ALAP mutual exclusion benchmark executed in AToMPM are presented. In this case, adding scope to the transformation did not result in an improvement. This was due to the fact that the number of host graph nodes in scope was large and constantly equal to the number of resources in the input model. In the context of search plans, this means that match operations related to scope had large branching factor (as if scope was not applied) and the cost of search plans with the introduction of scope did not improve. There is a minuscule improvement for the scoped transformation and that most likely is attributed to faster lookup inside the sets used for implementing the scope in AToMPM.

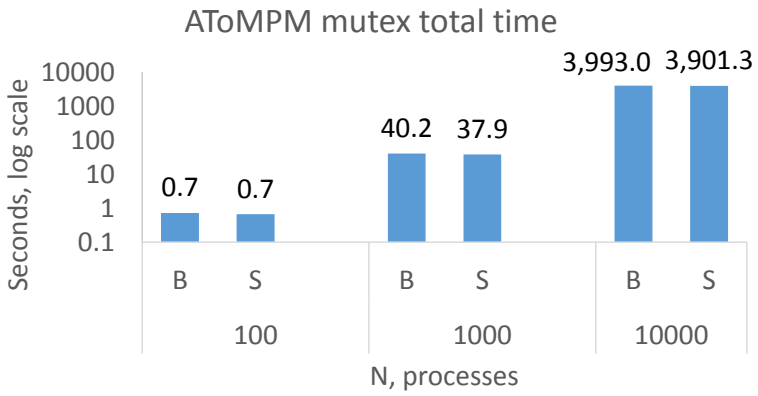

Figure 3–27: AToMPM ALAP mutual exclusion total time for Baseline (B) and Scope Graph (S)

The inclusion of scope in a single resource version of the mutex benchmark (STS) was evaluated. We applied the scope to the STS benchmark found in the GrGen source tree. The test was executed on a one million process model. Inclusion of scope resulted in a close to doubling of the runtime of the STS benchmark on

average. This is due to the fact that the baseline transformation is already exploiting the single resource node for search plan matching. The inclusion of scope in this situation creates additional, unnecessary overhead. This example represents the case where scope use may result in performance degradation. We complete this chapter with the conclusions and future work discussion presented in the following section.

3.5 Conclusions and Future Work

The use of hierarchical scope represents an interesting and potentially fruitful research topic in the context of model transformations. Our experience demonstrates by reducing the matching time by close to one and two orders of magnitude in GrGen and AToMPM respectively, that application of scope to the host graph is a path towards more efficient model transformations. Scope has the additional benefit of proving an intuitive and natural mechanism for expressing hierarchical concepts that transcend individual DSL boundaries.

Our scope concept, as demonstrated in the case of GrGen may be implemented in various ways, such as container, indexing, and finally scoped graph. The use of scope syntax at the rule level and automatically translating scope into the most efficient implementation is the best way to deal with our scope concept and to apply optimizations to transformations without the engineer knowing the inside of the tool engines. In addition, we demonstrate that scope can be beneficial for the amalgamated rules as well.

We do observe situations where scoping may not be as useful (the mutex example). This is usually the case when the matcher already exploits matching hints to

the fullest. It is interesting, therefore, to investigate what types of transformations benefit from the application of scope.

Future work for our design is currently concentrated on implementing changes to AToMPM to more efficiently support the scoped formalism. Even though the naive approach taken in our prototype implementing scope used sets, as was demonstrated in the case of GrGen the use of scope directly in the input graph is feasible and desirable. We are also interested in variations and extensions of how scope is represented in transformation rules. Our syntax and semantics here are aimed at a restricted use of scope, as a reasonable balance between implementation complexity and ensuring sufficient expressiveness. More varied parametrization of scope specification in rules, allowing for example complex scope specifiers that do not always resolve to simple path searches, is also potentially interesting, and while this can pose challenges in terms of maintaining an efficient match process it may also offer even more flexibility in scoped graph specification and manipulation. A thorough investigation would also need to look at how scope hierarchies and various scope management strategies impact transformation efficiency.

In addition, the scoped graph scalability would be an interesting future work topic. The input graph size is increased with the inclusion of an additional scope hierarchy and the edges representing scoping relationships. From the examples in this thesis, we observe, that the classic scope has edge-heavy effect on the input model (as the number of scope nodes is small wrt. the host graph). We can estimate that the number of edges representing scoping relationships between the host graph nodes and the SF nodes to be linear in size and at most $V_{SF} * V$ (considering the

scoped graph requirements) where V_{sf} is the number of nodes in SF and V is the number of nodes in the host graph. Therefore, it is important for any tool, dealing with scope, to have efficient edge representation. In case the scoped graph grows out of memory (e.g. due to extremely large benchmarks or problems), distributed (cloud) storage and MT approaches can be used. This is demonstrated, for example, in [96] where incremental model querying, possibly adoptable to scoped pattern matching, is performed in the cloud to alleviate memory limitations.

Finally, we would also like to further investigate scope workflow in solving various MT problems. An appropriate workflow intended for a transformation engineer should include well-defined guidelines on scope creation and scope use. A user study could be a source of valuable information in designing such a workflow.

We continue with a new chapter presenting a different scope concept. A dynamic scope that is meant to be used without the input of a transformation engineer.

Chapter 4
Dynamic Scope

Although the static scope presented in the previous chapter is an effective way of specifying locality in MTs, it does require the attention of a MT engineer. Therefore, in this chapter, we present a different scope concept that does not require modification of MT specification. The *dynamic scope*, contrary to the static scope, aims at discovering the regions of interest in the input model automatically, at runtime. These regions, in the context of the local search-based techniques such as search plans, are then used to reduce the initial search space, promising performance benefits. We begin with the introduction into dynamic scope.

4.1 Introduction

Local search-based techniques (an example search plan technique was presented in Chapter 2.4.1) are often used in model (or graph) transformation systems [115, 30, 43, 107, 105, 33]. These techniques start pattern matching from some initial node (or a graph) and then expand the match candidate along edges in the vicinity of previously discovered match nodes. This happens in accordance with some search plan. The search plans provide an order of matching pattern elements based on metrics calculated mostly ahead of execution [115, 30, 43] or adaptively at runtime [105, 33, 36]. This strategy is effective but requires ahead-of-time inspection of the model transformation system, and may be less applicable when MT rules are dynamic or difficult to inspect, such as when the rules are compiled (for example in the GrGen

tool). In addition, MTs can be proprietary, similar to software programs, and not available for an inspection.

Therefore, in this chapter, we present a dynamic, black-box approach intended to reduce the search space of graph pattern matching in model transformations. This approach is complementary to existing search plan-based approaches in that it does not change the algorithms of SP matching but rather acts on the matcher input. We apply our vision of scope as the region of interest in the input model. The aim of the approach is to discover these regions or *dynamic scopes* where the MT is active and feed them to the pattern matcher. The dynamic scope area (similar to static scope) is necessarily smaller than the originally intended area of pattern search. The SP matcher then takes advantage of this situation in creating the initial binding. In essence, our design observes the transformation process to group the nodes which will likely form a match of the following transformation rule, thus forming dynamic scopes.

For dynamic scope discovery, we use two techniques. First, we employ a "temperature-based" coloring of the input model elements inside the transformation engine. We observe model elements as they are interacting with the transformation, and color the nodes according to a basic temperature specification. Thus we obtain a heat-map that can be used to construct a scope intended as an initial, reduced search space that is passed to the pattern matcher.

Further refinement of this search scope is then achieved by applying a variant of a supervised machine learning technique, based on a Naive Bayes (NB) classifier [63]. At runtime, successful matches are used incrementally as positive training examples

to help with further prediction. The training examples reference the heat-map in addition to domain-specific properties of the model elements that contribute to matches and may include structural graph information as well. The classifier is then used to decide whether the node should be included in the refined search scope. Our approach is primarily intended for long-running, simulation-oriented transformations, where transformation evolves in the neighborhood, and where the first match discovery is usually sufficient. In addition, our approach involves a fall back step to matching in the whole input model in order to guarantee a match or lack thereof. The benefit and applicability of our approach to other types of transformations needs to be investigated in the future work.

To validate and assess our process for model simulation transformations, we experimentally examine two non-trivial graph transformations. We show the effect of different parametrizations of scoping on both transformations. Through this, we demonstrate that the temperature-based approach in itself and also combined with NB are effective at reducing the search scope. We achieve a high overall success rate of 90 percent in case of single resource mutual exclusion benchmark and reduce the size of the search scope at least 10 times in case of forest-fire simulation. Specific contributions of this thesis include:

- We describe a temperature-based system for tracking and predicting the scope of rule matches in model transformation. This runtime technique provides heuristic information that helps identify possible matches without explicit reference to rule content or scheduling information.

- Improvement to the scope discovery is further facilitated by incorporating machine learning into the search process. A Naive Bayes classifier is trained at runtime, filtering "warm" nodes to more accurately identify model elements that have a high probability of being part of a successful match.
- Feasibility and performance of our design is evaluated by experimenting with both a mutual exclusion benchmark (see Section 3.4.1) and a forest-fire simulation (see Section 3.4.2) using the research oriented tool AToMPM [59]. This work demonstrates effectiveness in both graph-modifying and pure simulation contexts, and illustrates the impact of different parametrization of our technique.

In Section 4.2 we present our approach, explaining both the temperature and NB scope refinement mechanisms. Experiments and their results are discussed in Section 4.3 and finally, Section 4.4 gives conclusions and future work.

4.2 Dynamic Scope Discovery

The cost of graph pattern matching in the case of SPs (as described in Section 2.4.1) can be expressed as the size of a search tree. The nodes of this tree represent the model elements visited during the pattern matching steps. The size of the tree depends on the size of the graph pattern and the branching factor at each decision point or SP primitive operation execution. A smaller search tree therefore represents a more efficient pattern discovery.

Existing approaches (see also Section 6.1), in essence, focus on a good ordering of SP operations, either statically before execution of pattern matching or dynamically at runtime. The main idea of these approaches is to start executing cheap SP

operations first, *i.e.* those with a small branching factor. For example, extending a match along an edge with an at most one multiplicity (according to the metamodel specification of the input model) is guaranteed to either succeed or fail. The branching factor of this operation is therefore 1.

In SP-based pattern matching, where to start matching is often very important, as the first pattern node can typically be matched to many input model nodes. This situation was demonstrated with an application of static scope (in Chapter 3) in the context of search plan-based pattern matching. An addition of scope created a favorable place for an initial pattern binding by introducing the first SP operation with a reduced branching factor. In the case of our dynamic scope discovery, the SP is not augmented, but rather a reduced input is provided to the pattern matcher. Still, this results in reduced branching factor as the SP operations are applicable to smaller search space.

In this chapter, we propose a complementary optimization technique to existing local search-based approaches in order to *reduce the branching factor at each decision point* dynamically at execution time independently from the search plan of a graph pattern. We aim at filtering match candidates by giving priority to (1) recently touched nodes (calculated using a heat map) and (2) nodes which constitute a match with a higher probability (estimated by Naive Bayes classifier). This filtering effectively creates the dynamic scope that can be used to reduce the search space.

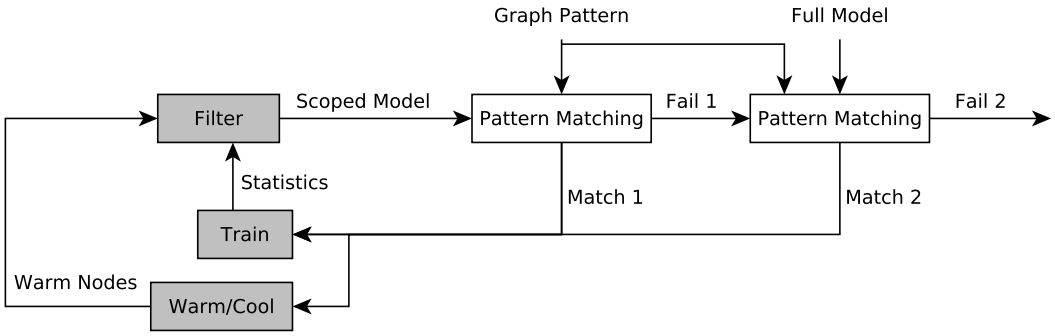

Figure 4–1: Dynamic scope discovery and matching (new components are shaded).

Our overall design for dynamic scope discovery builds on two main components. Below we first give an overview of how the process is integrated into a transformation system, followed by details of the temperature and the Naive Bayes classifier components.

4.2.1 Overview

Our approach to scope discovery is embedded within the graph pattern matching process of a typical model transformation system. Figure 4–1 presents a general overview of this integration with rectangles representing the major steps carried out in the pattern matching phase model of a transformation system.

- **Operation phase:** During the main operation, our basic design carries out graph pattern matching in two phases using any existing matcher component. First, a reduced (scope) graph is computed by filtering, and pattern matching is initiated on this reduced graph (middle pattern matching box in Figure 4–1). A valid match on this reduced graph is guaranteed to be a valid match of the full graph. But since this approach is optimistic, we retain the pattern

matching of the full graph as a fallback (right box) when pattern matching on the reduced graph fails.

- **Filtering phase:** The filtering of nodes for obtaining the reduced scope graph is carried out by a combination of two techniques. (1) First, *a heat map of model nodes is calculated*: if a transformation rule touches (matches, modifies) a model node then it becomes a warm node. Several subsequent matches heat up a node, which gradually cools down if it is no longer part of a match. The number of warm nodes in the system, and subsequently the size of the scoped model, is directly dependent on parameters of the warming and cooling process, which will be described in Section 4.2.2. (2) The exact population of the warm set is also reduced by a Naive Bayes classifier. This classifier is trained using the matches produced during the pattern matching phase and is then used to further filter the warm set. Note that this is a simplified view, and the filtering step can more generally contain an arbitrary chain of filters that refine the warm node input and thus the dynamic scope.
- **Initialization and Training phase:** Initial pattern matching is performed on the full model when all nodes are cold in the system. The scoped model is thus initially empty, and the scoped pattern matching fails. As we observe the matches of transformation rules, warm nodes are discovered, populating the Scoped Model subset, and scoped pattern matching may succeed. The training of the Naive Bayes classifier can be carried out either in a preprocessing phase (i.e. prior to a transformation run) or during transformation execution.

As a result, matching a pattern on a scope graph will probabilistically reduce the complexity of matching by reducing the branching factor. We expect that this can be a significant reduction for simulation kinds of model transformations, which may exploit the strong locality of subsequent execution steps. We continue with a description of how we use the temperature and heatmaps in our system.

4.2.2 Warming the Nodes

Coloring input graph nodes with temperature values is a straightforward way of representing the frequency of access to the graph nodes and thus the temporal locality of transformations: high-temperature nodes are frequently accessed (or near to ones that are), and so likely to be part of a future, successful match and/or rewrite, while low-temperature nodes are outside the current locus of activity, and so less likely to be part of a match.

Node temperature is maintained by augmenting the transformation engine with the ability to color/heat the nodes belonging to a match. At rewrite time, every node in a match chosen by the engine for a rewrite will be tagged with a temperature value. In our system, this means updating temperature attribute of a node. This temperature attribute is created at runtime, transparent to the language engineer and is not part of the attributes specified in the metamodel for the language being transformed.

Node temperature is expected to increase on frequent access, and decrease if not accessed over time. We track the temperature changes of a node using a global timer that counts the number of rule executions during transformation. References to warm nodes and the time of the last temperature change are kept in the *temperature list*,

which defines the temperature scope. Nodes that are not participating (not matched and/or colored) in the transformation for a number of rule executions will be cooled down. We call the number of rule executions that must occur before a node begins cooling as the node's *warm time*. The decision to cool down nodes is made at every transformation step. Reference to a node is removed from the temperature list once the temperature of the node cools down to zero.

Temperature values in our system range from 0 to 100 (temperatures exceeding maximum value are scaled back to 100), with increments occurring in discrete steps. For simplicity, we chose to decrease the node temperature to zero after its warm time expires. However, temperature decrease step can be equal to a discrete value similar to temperature increase step. For each node in the match, the temperature is increased by 40 degrees. Nodes in the neighborhood of a match are also colored with a temperature, although with a smaller increase to indicate less confidence; we used a step of 20. In our case we consider only immediate neighborhood, using single hop distance, and exploring the effects of variable-size neighborhoods on our approach is future work. All temperature related values were specifically chosen for the purpose of this thesis and their variation needs to be explored in the future work as well.

Once the temperature is updated, we compose the warm set as a subgraph of the instance model where all nodes have the temperature higher than 0, without making any distinction between the temperature values. Diversity in temperature steps is intended primarily for the NB classifier described in the next section. Without NB, it would be sufficient to use two values for the temperature: 0 or 100, cold or hot.

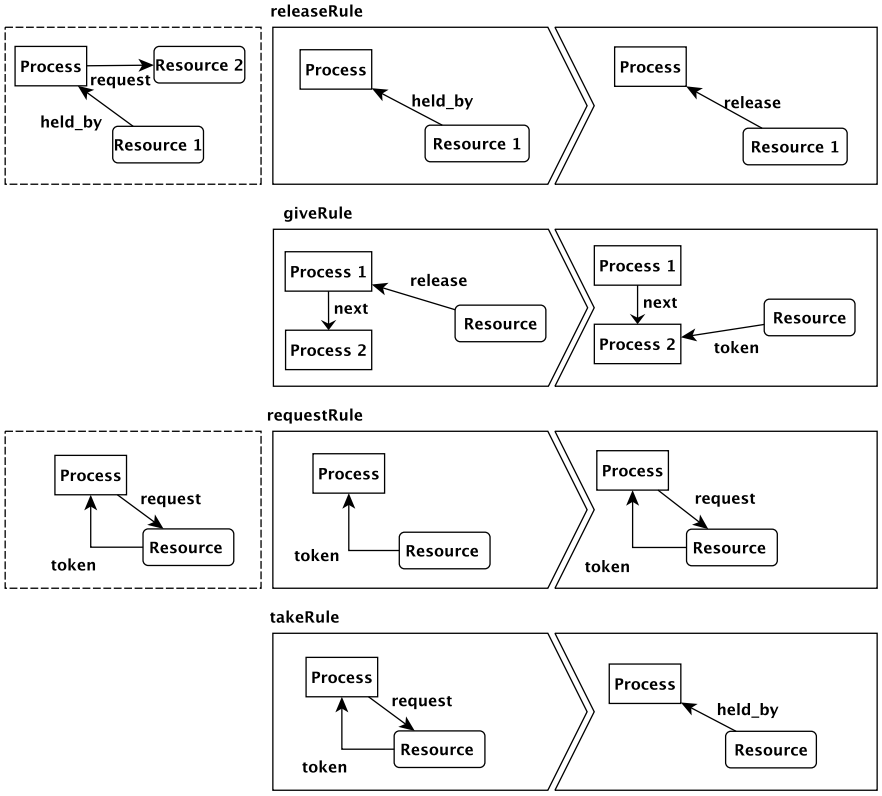

Figure 4–2: A subset of rules describing the mutual exclusion algorithm.

Heat map example: Mutex. Figure 4–3 illustrates the basic heating process, showing the application of *releaseRule* (from the mutex benchmark introduced in Section 3.4.1 and shown again in Figure 4–2 for convenience) on a model of a ring of processes with a single resource. Note that we omit labels on the connections between processes; they are of type *next*. Here nodes participating in the current rule application are shaded, and all nodes in the model initially have temperatures equal to zero (we only show temperatures of the elements participating in the rule

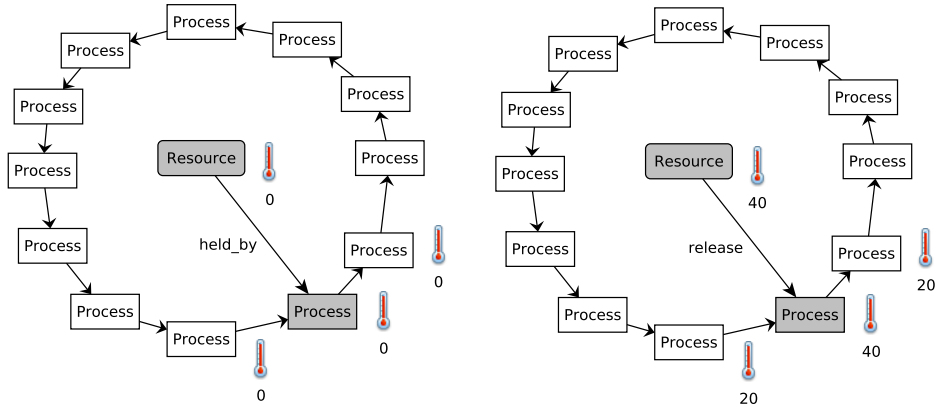

Figure 4–3: Application of *releaseRule* on a model (left) results in warmed up nodes (right).

application and their neighbors). The right side of the figure shows the result after the matching; as described above, matched nodes are warmed up to 40 degrees, and immediate neighbors are warmed up to 20.

Heat map example: Forest-fire. A finer-grain example of temperature is shown in Figure 4–4, showing a stage in the forest-fire simulation transformation we will evaluate with our approach in Section 4.3. Note that we omit the metamodel and the transformation rules of this example for brevity; full details are available in Section 3.4.2. In this simulation, a fire spreads across a 2D grid of neighboring cells starting in the center. Each cell in a grid represents a forested area which may catch fire if any neighboring cells are on fire. Once fully burned, a cell represents a barrier to further fire spreading. The simulation terminates when no burning cells remain. Assuming all cells are exactly the same and in the absence of wind effects, fire will spread in a circular fashion (discretely represented).

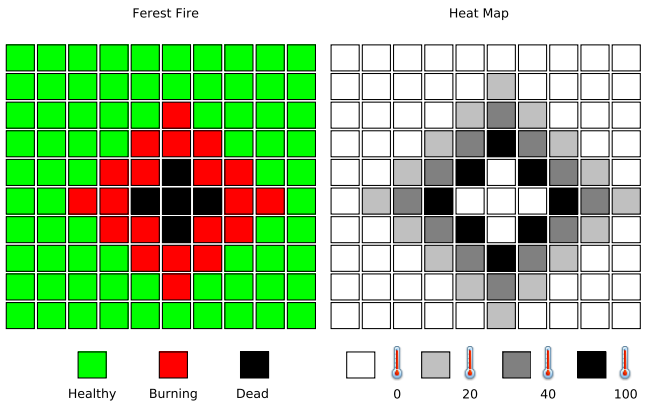

Figure 4–4: Forest-fire simulation rendering (left) with burned out, black cells in the middle and the model heat map over the cells (right).

In Figure 4–4 on the left, there are dead, burned out cells in the middle of a fire ring of a width of 2. On the right is the heat map over the input model of the cells on the left. We can see that white nodes in the middle of the heat map are cold (0 degrees). These nodes correspond to dead trees on the left, they are not touched by transformation anymore and therefore cool down. Black nodes are the hottest, they were matched several times recently. Going outwards the temperatures of the nodes decrease, as some rules are yet to match these nodes in the process of spreading the fire. The least warm, outer nodes have the temperature of 20 degrees, they are immediate neighbors of match nodes and are likely to be touched in the next iteration transformation.

A subgraph created from the temperature list is then passed to a matcher for the initial match attempt. Note that temperature scope approach does not guarantee a match especially for the transformations with random behavior, and correlation of the heat-map with match success strongly depends on the degree of locality in matching.

Our design also depends on rule application mainly being interested in finding *any* match, rather than *all possible* matches. The contexts, where all possible matches are necessary, are not suitable for the heuristic filtering enabled by temperature based matching. We now continue with a discussion on the complexity of heatmap maintenance.

Complexity of dynamic heatmaps. Maintaining match scope through temperature allows for a reduced search space, but requires non-trivial bookkeeping to track the warm set. Depending on the warm time of the nodes and the transformation process, warm set can grow at most to the size of the input model. Therefore it is important to handle the warm set efficiently. For this we use an ordered set data-structure, with node ordering based on the time the temperature of a node was last changed.

The first heating of a given node implies inserting it into the tree while reheating a node requires removing and reinserting. Node cooling requires searching the tree, with possible removals from the tree if nodes were cooled down to 0. A simple minimum-temperature value can be used to avoid processing cool-downs until necessary, but identifying the nodes needed for cooling requires searching for the n nodes at minimum temperature. By using an augmented red-black tree that allows interval search, we can perform inserts and deletions in $O(log\ n)$ worst-case time, and find the now-cold nodes in time $O(log\ n)$. The efficiency of this approach thus depends on the trade-off between performing these additional data-structure operations and the corresponding reductions in search cost. Investigation of this trade-off

is planned in future work. We can refine the dynamic scope further with machine learning techniques. We discuss this in the next section.

4.2.3 Scope refinement by Naive Bayes classifiers

The temperature scope described in the previous subsection may be larger than necessary to find a match. This is especially apparent when nodes cool down slowly and the temperature scope grows correspondingly large. More aggressive cooling would mitigate this, but requires a cool-down threshold well tuned to the transformation. Our design thus instead makes use of a Naive Bayes classifier to learn from features describing the nodes involved in the transformation at runtime, including temperature, and so further refine the subset of relevant nodes fed to the matcher.

A Naive Bayes classifier is a simple machine learning technique that makes an independence assumption on the training data [63]. With a training vector of features for a given class, we assume that each feature is independent and does not affect the conditional probabilities of these features given a class. This approach is simplistic of course, but NB is known to perform well in many classification applications, and speed in our design is important—an independence assumption greatly simplifies the calculations necessary. The incremental training phase for NB is also an advantage.

Training is performed after each rewrite, with each node that is part of a successful match participating in training before being changed in the rewrite step. We consider each node in the pattern as an independent entity disregarding relationships between nodes. This simplifies training and the classification. The class or the label of the training example is the identifier of the rule, and the training features of the

nodes are the domain specific attributes, node type, and the temperature. For example in the case of the mutual exclusion transformation described in Section 3.4.1, nodes may have domain attributes such as the name of the process and resource, process priority, *etc.* Each of the domain-specific attributes including the temperature constitutes an independent training feature. Graph structure also carries a lot of information that can be harnessed, and we plan in future work to also consider structural graph attributes such as node degree, a number of incoming or outgoing edges, *etc.*

Training consists of keeping track of the number of distinct feature values encountered for each rule identifier. At this stage, features with continuous values such as temperature require discretization for efficient training [112]. Here we use a simple binning approach, based on 10 bins of equal size. Other approaches are possible, such as to assume the numerical value to be part of normal (Gaussian) distribution [112].

In NB, the probability $P(Y|X)$ of a class Y given set of features $X = \langle X_1, X_2, .., X_n \rangle$ is calculated as:

$$P(Y|X) = P(Y) \prod_{i=1}^{n} P(X_i|Y) \tag{4.1}$$

In Figure 4–5, we show the application of *giveRule* from mutual exclusion benchmark (in Figure 4–2) on a portion of the model shown in Figure 4–3. The match is shaded on the left, and the effect of the rule is shown on the right with updated temperatures according to rules described earlier. Before rewriting, the match is used for training NB. We have three feature vectors corresponding to the three

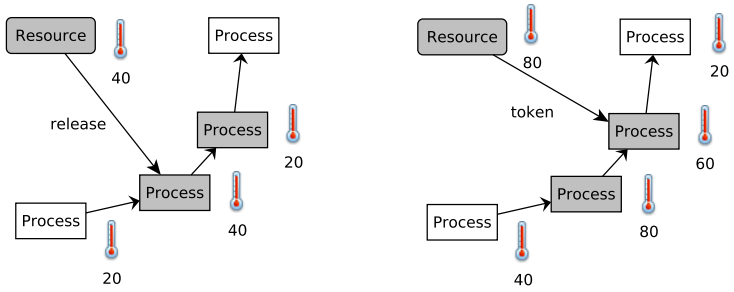

Figure 4–5: Application of *giveRule* on a portion of model (left) and the result (right).

nodes in the match on the left. We only use type of the node and its temperature to demonstrate the concept. $X_1 = \langle \text{process}, 40° \rangle$, $X_2 = \langle \text{process}, 20° \rangle$, $X_3 = \langle \text{resource}, 40° \rangle$. The label for training is the *giveRule* identifier. Thus statistics accumulated for *giveRule* is two processes, one resource, two 40-degree and one 20-degree temperatures. At the filtering step given a node described by the feature vector $X = \langle \text{process}, 20° \rangle$ and the upcoming execution of *requestRule* from the mutual exclusion running example (in Figure 4–2), the probability that the node will be part of a match (or should be included in the scoped model) is $P(\mathsf{requestRule}|X) = P(\mathsf{requestRule})P(\text{process}|\mathsf{requestRule})P(20°|\mathsf{requestRule})$.

Knowing the probability, a decision is then made whether to include the node in the refined scope or not. Essentially, there are two ways we can approach the classification of nodes using NB. One is by calculating the probability of nodes belonging to all possible rules (classes), using a competition approach in which the class with highest probability "wins". Another is through judging the probability of one class, choosing a threshold for accepting the probability as a description of the class. The higher computational cost of the former (competition) approach makes it less

appealing in our performance-oriented context, and so we use the latter approach with a threshold of zero. We now move to the experimental part of this chapter to evaluate our approach.

4.3 Experiments

In our experimental evaluation we investigate if our concept is feasible in the context of long-running, simulation-oriented transformations. Feasibility in our context implies two research questions:

- **RQ1:** Does our dynamic scoping technique effectively reduce the search space?
- **RQ2:** Does dynamic scoped matching provide satisfactory success rate?

In order to answer RQ1, we compare the size of the scoped graph wrt. the size of the entire instance graph. In order to address RQ2, the failure of our scoped matching technique is when the original rule has a match on the entire graph, but a scoped matching fails to detect it. We consider scoped matching a success when scoped matching produces a match or when the fallback matching on the full input model fails. We evaluate our approach using two non-trivial transformations: a mutual exclusion problem from the transformation benchmark suite [108], and a forest-fire simulation both introduced in Section 3.4 and decribed above. The latter constitutes a pure simulation benchmark, mainly modifying node attributes, while the former requires some amount of node creation and destruction in representing changing edge relations in the model (model edges are represented in AToMPM using graph nodes). First, we present the benchmarks in more detail, explain the experimental setup, followed by results presentation.

4.3.1 Benchmarks and Measurements

Both mutual exclusion and forest-fire transformations were executed with three different node warm times of 10, 50, and 300. These represent short, medium and long node warm times. We used a simple model for cooling, immediately reducing node temperature to zero and removing it from the temperature list. Each benchmark was executed using temperature scope matching (Temp), followed by additional filtering using NB (Temp+NB). For evaluation purposes, we maintain several metrics at each transformation step. Metrics have global and individual rule resolutions. We track: the success rate of scoped matching, size of temperature scope, size of scope resulted after additionally using NB filtering. We report these in the following section (individual rule results are omitted). The evaluation was performed on x64 i7 mobile quad-core processor with 16Gb RAM running Ubuntu 12.10.

Experimental setup for mutual exclusion simulation. The mutual exclusion experiments were executed on two types of the input models each containing 1000 processes in accordance with the benchmark setup published in [108]. This is the same baseline, non-scoped as long as possible (ALAP) transformation applicable to both input models and presented in Chapter 3 with the MT rules shown again in Figure 4–2. The first input model contains a single resource (similar to model in Figure 4–3) corresponding to the input model to the STS benchmark and the second input model contains multiple resources (one resource for each process) corresponding to ALAP benchmark. We show examples of both types of input models in Figure 4–6. In the single resource case, the size of the graph underlying the input model was 2002 nodes and the multiple resource model contained 4000 nodes (counting

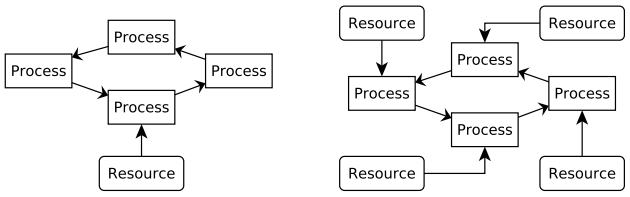

Figure 4–6: Single STS (left) and multiple ALAP (right) resource model types. Association types omitted.

model associations represented as nodes). The transformation applicable to both input models was executed in an ALAP fashion using the following rule sequence: *releaseRule*, *giveRule*, *requestRule*, and *takeRule*. Each rule was scheduled to execute exhaustively as long as the matches were found, followed by the next rule scheduled in the same fashion. Each model was simulated for several cycles after which transformation terminates. One transformation cycle is defined by sequential execution starting from *releaseRule* and terminating at *takeRule*. The multiple resource model was transformed for 4 cycles, resulting in approximately 20000 rule executions. The single resource model was simulated for 100 cycles with close to 1000 rule executions. We observed that this is quite sufficient to demonstrate the stability of the success rate of scoped matching in the system *i.e.* transient effects (such as initialization and training) are no longer visible.

Experimental setup for forest-fire simulation. The forest-fire simulation was executed on a grid of 100 by 100 cells with one cell burning to start the simulation. The number of nodes in the underlying graph is 29800 (including the nodes representing association edges). The simulation ran until all cells burned out. We

observed that the success rate in the forest-fire simulation stabilizes after 15000 iterations. We now present the results of running the evaluation.

4.3.2 Results

In this section we present results, demonstrating the overall success rate and the size of scopes with respect to the iterations of transformation, where iteration is equal to a single rule execution. All figures in this section contain legends that are following the order of the graphs in the plot: the top line in each graph corresponds to the top entry in the legend list.

Mutual exclusion results. Overall success rate of our single resource mutex benchmark with node warm time (WT) of 10 is presented in Figure 4–7 on the left. On the right in Figure 4–7 sizes of scopes are shown for node warm times of 300 and 10 (log scale on the y axis).

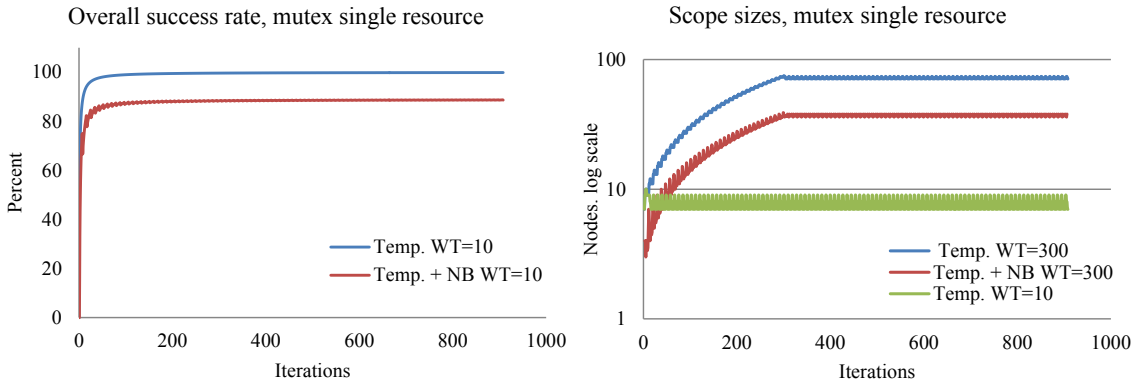

Figure 4–7: Single resource model. Overall success rate and scope sizes.

We observed that success rate does not improve after increasing WT, and a short WT of 10 is sufficient to demonstrate a good success rate. It is evident that

NB filtering reduces success rate by about ten percent. On the right in Figure 4–7, warm scope at WT equal to 10 is presented to contrast the long WT. With a long WT, the reduction of warm scope by NB is more evident. After the system stabilizes, a warm scope is reduced by approximately 30 nodes with NB, equivalent to 30 percent of the warm scope. NB filtering does reduce the scope size, however, it reduced the success rate due to exclusion of some of the match candidates.

In Figure 4–8 we present the success rates of Temp (left) and Temp+NB (right) scope matching in the multiple resource mutex model. Highest success rate in both

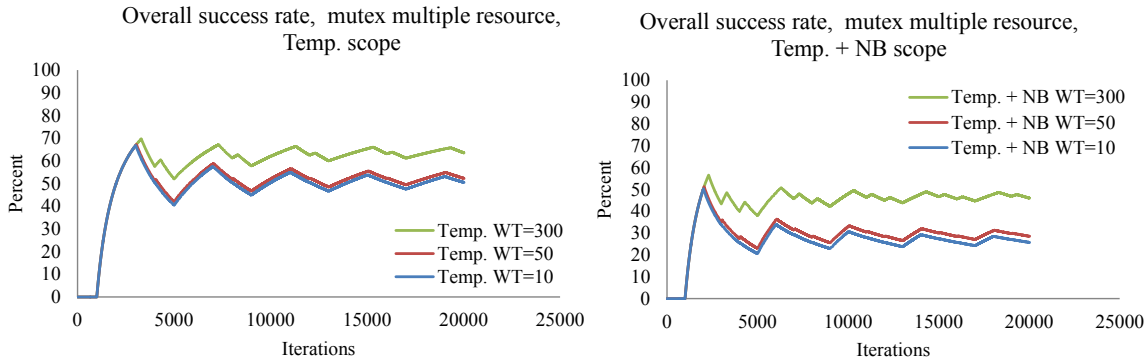

Figure 4–8: Overall success rate, multiple resource model.

filtering situations is achieved at WT equal to 300. The success rate with temperature filtering is at 50 percent in the worst case with WT of 10. We observe a similar reduction in success rate to the one seen in the single mutex benchmark, by applying NB filtering. NB filtering reduces the overall success rate by approximately 10 percent compared to warm scope matching.

Scope sizes for multiple resource model are shown in Figure 4–9. Temperature filtering is on the left and NB filtering on the right. The peaks on the left plot are due

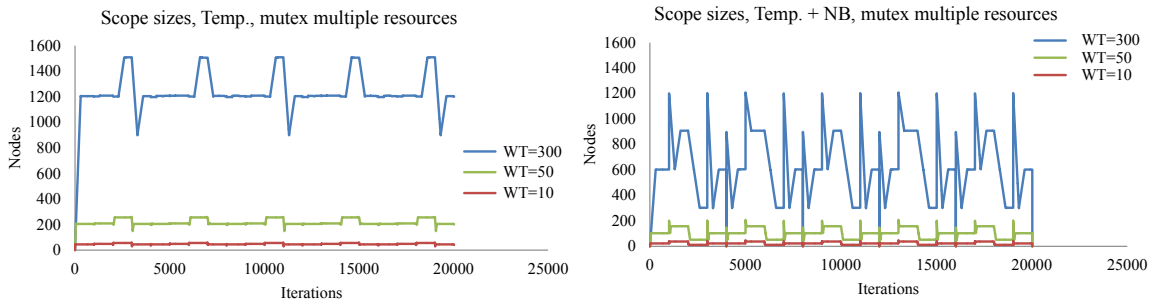

Figure 4–9: Scope sizes, multiple resource model.

to the increased number of nodes in the graph after repeatedly executing *requestRule*, which adds an extra node corresponding to the *request* association. Temperature based filtering reduced the scope to sizes ranging from 100 nodes (WT=10) to 1400 nodes (WT=300) (full model 4000 nodes). Consider the area under the graph for WT of 300 in the left and right plots. NB filtering does reduce the temperature scope even though there are peaks to 1200 nodes. When the number of nodes in the temperature scope peaks on the left and remains flat, NB scope size does not follow the trend closely.

Forest-fire results. Figure 4–10 presents overall success rates for temperature scope matching (left) and additional NB filtering (right). We observe satisfactory and equivalent success rates with WT of 300 for both filtering cases. This is likely because we use more data for NB training compared to the mutex example, such as the "burning" state of the forest cell. On the right in Figure 4–10 we can clearly see a gradual rise in success rate for WT 300. This is due to the initial NB training as well as the increase of the warm scope. The success rate in both plots is high at the

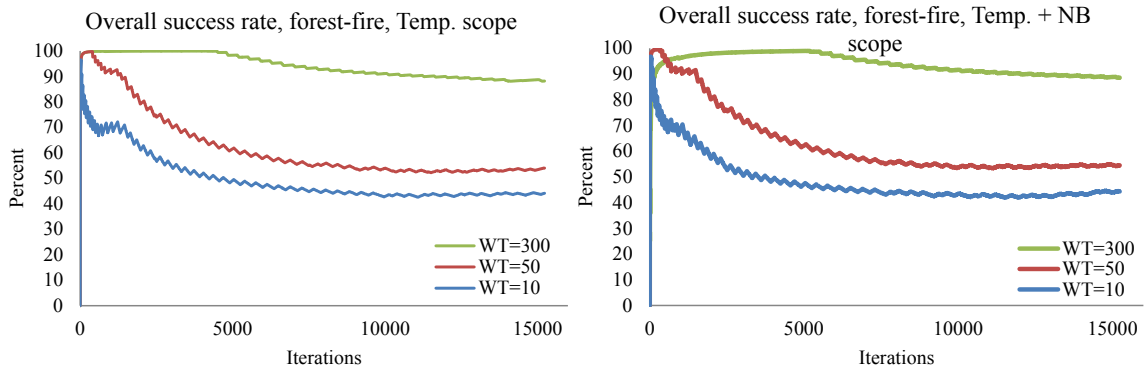

Figure 4–10: Overall success rate, forest-fire simulation.

beginning because a small portion of the graph is active. As the active region grows, the rate reduces and stabilizes sometime after 10000 rule executions.

Table 4–1: Forest-fire scope sizes (nodes), full input graph 29800 nodes

	Temp		Temp+NB	
WT	Ave.	Std. Dev.	Ave.	Std. Dev.
300	1110	319	357	415
50	357	415	157	155
10	77	17	37	36

The table 4–1 shows the average scope sizes in nodes and standard deviation for different warm times. The temp+NB scope is three times smaller on average than Temp scope. Average Temp scope is close to 20 times smaller than a number of nodes in the forest-fire graph.

Result summary. After running both benchmarks we learned that scoped matching in our approach is promising based on success rates of 30 to 90 percent. This depends on the warm time of the nodes and the additional filtering, such as NB in our example. In certain cases such as for the single resource mutex model,

the success rate was over 90 percent with a significant reduction to the search space, reducing it to just 10 nodes compared to 2000 nodes for the full input graph. NB filtering does reduce the search scope further at the slight expense of success rate. NB filtering performed best in the case of the forest-fire simulation both in terms of success rate and in reducing search scope. It is important to note that these results were achieved using the initial prototype. Even though we observe interesting results, a deeper investigation of performance, parameter values and applications is necessary for future work. We wrap up this chapter in the following section with conclusions and future work.

4.4 Conclusions and Future Work

In this chapter, we investigated the dynamic scope approach to reducing search scope of model transformations by tracking the transformation process within the input model. For this, we used a temperature inspired underlying graph node coloring. Temperature regions constitute the likely rule application areas that we explored in pattern matching during runtime, with a fall back to matching in the whole input model in case the match was not discovered.

We investigated additional filtering based on Naive Bayes classifier. In the context of simulation oriented transformations, where first match is usually sufficient, we demonstrated that our approach works well in certain situations: the success rate of matching within the scope defined by our filtering was over 90 percent in single resource mutex benchmark. We also observed the reduction of search scope by using our filtering approaches. NB application further refined the search area, however in some instances at the expense of the matching success rate.

In future work, we would like to explore the cost of the warm scope maintenance and the MT runtime effects of our concept. This involves answering the question whether the dynamic scopes help to improve MT runtime in general. We believe that deeper investigation of temperature scope related parameters, with an addition of structural graph information for NB training, will be beneficial to the performance and accuracy of the approach. Temperature scope in the context of search plans could in the future provide dynamic information to search plan generation at runtime. In addition, the fall back step does not necessarily need to search in the whole input model, it can do so in some special scope or an enlarged dynamic scope (where matching has just failed). The incremental matching technique is another area of model transformation that could possibly incorporate our approach. Another area to investigate is the pre-trained NB classifier used as a static filter in future transformation applications offloading the training expense.

Chapter 5
Debugging Transformations

Model transformations are complex specifications possibly combining declarative and imperative constructs. Similar to regular text-based programs, MTs can contain errors resulting in undesired behavior. Therefore, it is important to provide comprehensive debugging support for MTs.

Debugging MTs in a consistent, systematic way is a complex task due to a presence of a heterogeneous, hierarchical execution stack. Multiple stack execution layers are used, consisting of different languages/formalisms at different levels of abstraction. A schedule language, for example, contains the rules, which themselves hold patterns using action code. Further down the stack, we find the pattern matching and application routines. As problems can manifest at each layer, or between layer interactions, thorough debugging requires a tool permitting inspection and modification throughout the execution stack. In this chapter, we present our contributions to MT debugging and explain how scopes, from previous chapters, can be used in the debugging context. We begin with the introduction.

5.1 Introduction

Model Transformations are an important part of Model Driven Engineering (MDE) [89]. In MDE, models are primary artifacts and manipulation of models is achieved through transformations. The usability of tools supporting MTs and

modeling in general affects wide adoption of MDE, with usability enhanced by comprehensive debugging facilities akin to the ones found in software tools. These should enable exploration of what a transformation does, and also how the transformation does it, both of which are important to understanding a transformation. In addition, to be useful, the debugging of MTs should be automated. This point is supported by Zeller [114] who believes that interactive debuggers are not the most appropriate tools. Another observation in favour of automated tools was made by Parnin et al. [78] stating that automated debugging tools may help users fixing faults rather than patching failures. In the context of MTs, debugging could also benefit from declarative nature of models. This is of great importance to improving usability of MTs by narrowing the knowledge gap between the process of debugging and the problem domain specified by means of various domain-specific languages (DSL). The aim of the automated debugging is an unsupervised program execution that can be useful in the context of long-running and complex MTs. In addition, specifying debugging scripts or scenarios that can be reused and exchanged between users can supplement the interactive debugging tools. Such debugging specifications are presented in this chapter.

The purpose of a debugger is also to aid a developer in locating and eliminating software failures [113]. The same can be said about debugging of model transformations. Although this execution context can be quite different from the execution and debugging of a procedural, textual language, a generic vocabulary exists for understanding the basic approach, and main components of a debugger. We thus briefly

discuss the criteria and facilities related to a debugger described in this chapter, following the classification described by others [86, 60, 88].

Debugging can be generally divided into *live* and *forensic* contexts. As the names suggest the former is used during execution and the latter is effective after execution, where traces are available and can help identify the problem. The MT debugger presented in this chapter is intended to be a live debugger, and thus requires a fundamental set of facilities, suitably adapted to the MT context. These facilities include:

- *Selection* of a specific part of MT for debugging. This also implies that the matching process is exposed and can be influenced, with the pattern bindings in the input model during the matching process available to the user. When present, non-determinism of multiple choices during MT execution needs to be resolved either automatically (communicating the choices made to the user) or by giving control to the user for a decision. Breakpoints in MTs should be available as in program debugging, to halt MT execution automatically based on some state condition.
- *Investigation* of the state of MT execution. *State inspection* is common in program debuggers. For MTs a developer needs to inspect the state of source/target models, pattern bindings, the current match, *etc.* This also includes access to traces of the transformations. *Visualization of control flow* allows one to see the flow of the MT. In textual languages, this is done by displaying lines or line numbers, and in graphical languages by highlighting relevant parts of the MT such as the rules being executed.

- *Dynamic behavior* investigation by means of stepwise execution. In textual languages, a step corresponds to advancing execution to the next line number, or syntactic element. The notion of a "step" in declarative languages is more ambiguous, as rule application can involve a large number of underlying smaller steps that are normally invisible to the MT developer. This leads to the necessity of presenting information at the appropriate level of abstraction.
- *Adapting* the parts of MT during execution. This includes modification of the input model, match bindings and finally the transformation itself.

In order to draw parallels with the MT debugging discussed in this chapter, we continue with relevant debugging concepts and operations from the area of interactive debugging. A variety of general purpose language debugging tools exist, here, we adopt a view of a *gdb* debugger [1] as it represents a ubiquitous application in the open source universe.

The inspection of a debugging target control flow can be achieved through commands executed inside a debugger. The target can be instructed to execute until the end through a *continue* operation and a stepwise execution of the target can be achieved through *stepping* commands. In case of stepping, a step is usually a single line of code (LOC) or a machine instruction. A *step-over* command executes a single LOC or a machine instruction (without exploring inside of the function or procedure calls) and returns control to the user. The *step-into* command operates similarly to *step-over* except that it will inspect inside of a function or a procedure call. A

[1] `https://www.gnu.org/software/gdb/`

stepping command can also execute several lines of code or machine instructions. An example of this is the *step-out* command that executes a target just until after the return out of a function in the selected stack frame. Certainly, the *step-over* command can be included in this category as well because it implicitly causes the function or procedure call execution within a span of a single step debugging operation.

Regardless, whether the target program is in the process of continuing or stepping, the execution can be interrupted by means of, most notably, *breakpoints*, *watchpoints* and *catchpoints*. A *breakpoint* is associated with a certain place within a debugging target. Once that place is reached during execution, the target stops and returns control to the user. A *watchpoint* (sometimes also called a *data breakpoint*) is intended to stop execution when a value of an expression of interest changes. A *catchpoint* is a special breakpoint designed to stop the execution on special events such as exceptions. This brief introduction is not intended to exhaustively explore the well-established area of program debugging. We direct the user to a *gdb*[2] manual for more information.

MT debugging brings in additional complexity in terms of a layered execution stack. The stack layers consist of different languages/formalisms at different levels of abstraction. We can envision the top of the stack to be the MT schedule specifying rule application order. The declarative rules are representing hierarchical structures with their *LHS* and *RHS* parts, internally holding the patterns in turn containing imperative action code. At the bottom of the stack, we can discover the pattern

[2] `https://sourceware.org/gdb/current/onlinedocs/gdb/index.html`

matching and application routines with their own implementation-specific complexity. This heterogeneity of MT stack layer requires thorough debugging throughout the execution stack in a systematic and unified way.

Modern transformation tools already provide some debugging functionality, although typically at the schedule/rule level, with possible areas of relevant models highlighted, as in *AGG* [97], GrGen [33], or Fujaba [32]. MT tools may also delegate debugging tasks to the low-level debugging facilities provided by the hosting general programming language (GPL), such as Java in *ATL* [45]. To our knowledge, there are no MT tools that fully address debugging of the whole MT stack, from the schedule level down to the pattern matching and application routines details.

Automated debugging support is also limited. The declarative nature of models and MTs is a natural setting for many debugging tasks, particularly event or watch-based goals, such as pausing execution when a pattern matcher accumulates a certain portion of the match, or when an undesirable pattern appears in the output model. In this context, query-based debugging techniques [56] yield further inspiration for our debugger design, allowing us to inspect all the relevant MT stack levels with declarative queries during the debugging session. In this way we facilitate deep introspection of a MT execution, taking advantage of the high level abstraction provided by a MT environment itself. Ideally, every aspect of a MT is modeled at the most appropriate level(s) of abstraction, using the most appropriate formalism(s) [71]. An important property of our debugger is thus that we try to avoid resorting to code-level debugging and instead remain at the level of abstraction similar to the Domain-Specific Language (DSL) of the models being transformed.

In this thesis, we aim to provide a practical and flexible solution for MT debugging for modern tools, validating our design by describing our experience with realizing such a debugger in a research tool AToMPM. In addition, we demonstrate the applicability of our approach by applying our debugger implementation approach to other tools such as AToM^3 and ATL. Specific contributions of this chapter include the following.

- We describe a structured view of a debugging process that lends a unified way of navigating the *debugging target*. We consider debugging as a movement over items in horizontal and vertical planes, an abstraction applicable to many stacked execution contexts.
- We describe a simple language for automated debugging of MTs spanning the levels of the MT stack. In addition, we introduce declarative breakpoints/watchpoints applicable throughout the MT stack.
- Our design builds on the use of familiar transformation rule and schedule syntaxes. This enables us to reuse the core declarative nature of the MTs, namely the pattern matching. The user is able to stay in the MT mindset while defining debugging scenarios. Similar to higher order transformations (HOT), where the transformations themselves are being transformed, we strive to apply the MT syntax and semantics to the debugging of MTs.
- Our declarative approach subsumes a separate control scheme for direct user interaction. We show how the action of a step issued during interactive debugging can be modeled with our debugging rules in terms of calling a MT. In this

way our design provides a unified debugging model, suitable for automation or interaction.

The rest of this chapter is structured in the following way. The structured view of a general debugging process is presented in Section 5.2. In Section 5.3 we apply the structured view to the MT stack. We present our debugging language with the debugging rules and transformations in Section 5.4, and describe our prototype implementation in Section 5.6. The conclusions and future work finalize the chapter.

5.2 Structured View of Debugging

In this section, before moving on to the MTs, we approach the general process of debugging in a structured way. We discuss navigation of a general debugging target with an aim to use this in defining debuggers for situations where a general programming language paradigm may be less applicable, such as in debugging DSLs and in particular MTs. This view allows us to clarify the notion of a step that may be ambiguous in declarative debugging contexts.

Consider *event-based* view of debugging, a powerful and flexible approach to realizing debuggers [4, 17]. Debuggers reason about the program execution from the stream of events to perform debugging, whether it is live or forensic. In general, a process of live debugging follows the hierarchical structure of the execution control flow. This implies descending and ascending the composite structures and thus exploring the existing hierarchies. Instead of reasoning about the program execution from the sea of events, it may be advantageous to employ a *structured view*, that can be used across a wide range of applications. This view is intended to complement an event-based paradigm as events can still be used to realize the structured view of

debugging. Here, we are simply imposing a conceptual structure on the view. Let us now discuss this generic view in more detail.

5.2.1 Navigating the debugging target

To take a generic view of debugging control flow (process) we can imagine it to be somewhat similar to navigating the multi-story building representing the debugging target. On the vertical dimension (here we use the term without adhering to its strict definition) we have a multitude of floors representing the many levels of nested structures. These hierarchical levels can be found in nested function calls, hierarchical models, etc. We call these nested levels **vertical levels** (*VL*) and the movement between them as **vertical movement**.

Each vertical level also has a horizontal dimension, this can be imagined as the apartments on a floor. If we use the programming language analogy, these levels represent a single kind of scope. We call this horizontal dimension a **horizontal level** (*HL*) and the movement within that level as **horizontal movement**. Each horizontal level contains items of interest for the debugging. These can be statements, expressions, nodes and edges in the model. The unifying characteristic of these items is the fact that when the debugging target has processed, visited or executed one item, it will move to the next item—items on a horizontal level are dynamically enumerated as a result of executing general step-over operations on the debugging[3] target. In

[3] For the rest of this chapter, we will use the term debugging target and target interchangeably.

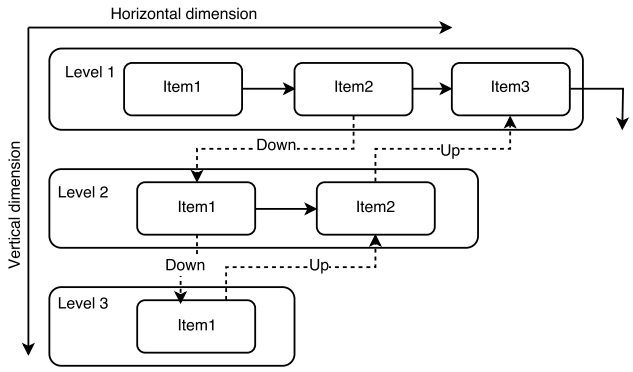

Figure 5–1: Horizontal and vertical dimensions or levels. Arrows between items on a horizontal level represent horizontal movement operation. Vertical operations are dashed arrows and labeled.

Figure 5–1, we show the horizontal and vertical dimensions[4] of a debugging control flow. Some of the items on the horizontal level can contain other items (nested) and therefore lead to lower vertical levels. For example, these can be hierarchical model elements or function calls to descend into. Therefore, the vertical dimension in a debug process is explored by moving up and down between the hierarchical items (as illustrated by the dashed arrows in Figure 5–1).

Figure 5–1 gives us a conceptual graph structure which we can apply general forms of stack-based program execution. Each vertical level embodies a particular behavior, execution scope, or level of abstraction. The horizontal elements then correspond to the smallest units of processing, computation, execution, or specification within that level of abstraction. Program execution, whether procedural or MT-based, can then be understood in terms of navigating this 2D hierarchy. This view

[4] In this chapter, we use the terms level and dimension interchangeably.

can be further expanded to include additional dimensions. Consider concurrent execution, an additional dimension can be added representing the threads of execution (although in this thesis we concentrate on sequential MT execution).

Navigation pointers. For our purposes, we need to be aware of the control flow position during debugging with respect to the horizontal and vertical dimensions. Horizontally we are concerned with the *item pointer* (*IP*) and vertically with the *level pointer* (*LP*). We can then define a Navigation Pointer (NP) as a tuple (*IP*,*LP*), describing the control flow position in the target, similar to the position of a point on a plane, described by its two coordinates. Both *IP* and *LP* are non-negative integer values, representing the ordered movement of control flow within, or between levels. The use of integer values is convenient in the specification of debugging operations shown below. However, as demonstrated later in this chapter, pointer values can also be symbolic if they are properly ordered and allow us to move within and between them. One of the benefits of symbolic values is readability.

We also define certain constant values for the *NP* tuple elements. These allow us to refer to the typical positions and the debugging situations within the target.

The following are the constant values the *IP* can take.

- *NULL* - this value indicates that the pointer is not initialized or more specifically it is not pointing to any item on the *HL*. This value is useful in describing the situation when the *IP* moves past the last item on the *HL*.
- *FIRST* - the first item of interest on the *HL*. Note, that in some of the following examples we use the first item's value directly. As in tuple $(1, TOP)$ for example.

The following is the constant value the LP can take.

- TOP - the very top VL in a debugging target.

Taking the above values into account, a program begins at NP=$(FIRST, TOP)$, and the end of the debugging (termination) can be specified with a tuple $(NULL, TOP)$. Note, that our intention is also to use the NP to specify breakpoints (described later in this chapter). Therefore, we assume that within the same program, two equal NP values represent the same location inside a program execution control flow. For example, NP_1=$(FIRST, TOP{+}2)$ is equal to NP_2=$(FIRST, TOP{+}2)$.

Operations. We now propose the operations that allow us to navigate the vertical and horizontal dimensions in a debugging target. Operational semantics essentially follows a controlled stack-based traversal, giving us operations to move between (ordered) siblings, into child nodes, or back to parent nodes. The effect of these operations includes the modification of LP and IP pointers, and presumes an execution stack of horizontal item pointers, σ. We begin execution at $(FIRST, TOP)$, with σ empty.

- ***Next***. This operation processes the current item on a HL and moves on to the next item at the same level. Given non-$NULL$ values for (IP,LP) and stack σ,

$$Next((IP, LP), \sigma) = \begin{cases} ((IP + 1, LP), \sigma) & \text{if } IP + 1 \text{ exists} \\ ((NULL, LP), \sigma) & \text{if } IP + 1 \text{ does not exist} \end{cases}$$

  Given ($NULL$,LP),σ, a $Next$ operation delegates to an Up.

- ***Down***. This operation moves one vertical level down if possible, pushing the current state and setting the IP to the first item on the next level. If no deeper level exists from this item, this delegates to a $Next$ operation.

$$Down((IP, LP), \sigma) = \begin{cases} ((FIRST, LP+1), IP{:}\sigma) & \text{if } LP+1 \text{ exists} \\ Next((IP, LP), \sigma) & \text{if } LP+1 \text{ does not exist} \end{cases}$$

- ***Up***. This operation completes processing of all remaining items on the current horizontal level and moves one level up vertically. This is idempotent, and implies terminating the program if it attempts to ascend past *TOP*.

$$Up((IP, LP), \sigma) = \begin{cases} ((IP', LP-1), \sigma') & \text{if } LP \text{ is not } TOP \text{ and } \sigma = IP'{:}\sigma' \\ ((NULL, TOP), \emptyset) & \text{if } LP \text{ is } TOP \end{cases}$$

We can now describe the program execution in terms of the navigation pointer evolution. Using a simple textual example, Figure 5–2 demonstrates a graph of possible pointer values in the nodes and the operations that result in the changes as edges. In this example, the items were the statements of the program and were identified by the line number. At each point a debugger may move to the next statement at a given level, either as a typical *step-into* (black *Next* arrow) or *step-over* (red arrow) any lower levels (the latter being *Down* operations that delegate to *Next*). An actual *Down* operation can be performed on the method call to enter the method body, at which point an *Up* operation can be requested to complete execution, skip debugging the method body and return to the caller, or *Next* can be used to flow through the execution of the increment statement, and *Up* executed when no more horizontal execution is possible.

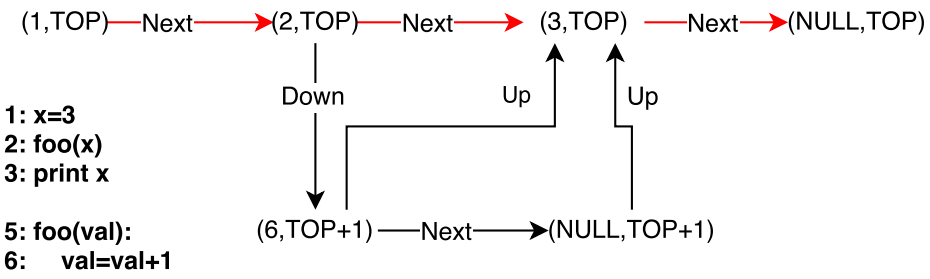

Figure 5–2: Navigation pointer evolution. Red arrows represent a step-over debugging scenario.

We now demonstrate, in Figure 5–3, a possible *NP* evolution while debugging a graphical formalism representing sequential execution of two MT rules. The simple schedule contains two rules enclosed in rounded rectangles connected with an arrow to indicate sequential execution control flow. The MT schedules as demonstrated in Section 3.4 hide rule details in plain rectangles, here the rule contents are shown. In addition, here, we demonstrate the use of symbolic pointer values. The ordering is provided by the MT schedule structure. For example, *Next* operation for *IP*=*Rule 1* results in *IP* change to *Rule 2* as it is the following item on that horizontal level.

In this example, the items are the elements of the MT schedule. At the top level, the *step-over* (red arrow) operation is performed using the *Next* command by enumerating the items at the *TOP* level. A typical *step-into* (black *Down* arrow) operation can be performed on the nested MT schedule element. In such case, the hierarchy of *Rule 1* is explored resulting in the adjusted *LP*. At this point the *step-over* results in sequential exploration of the *LHS* and the *RHS* parts of the rule *Rule 1*. Finally, the debugger is taken back to the *TOP* level from the *LHS* processing

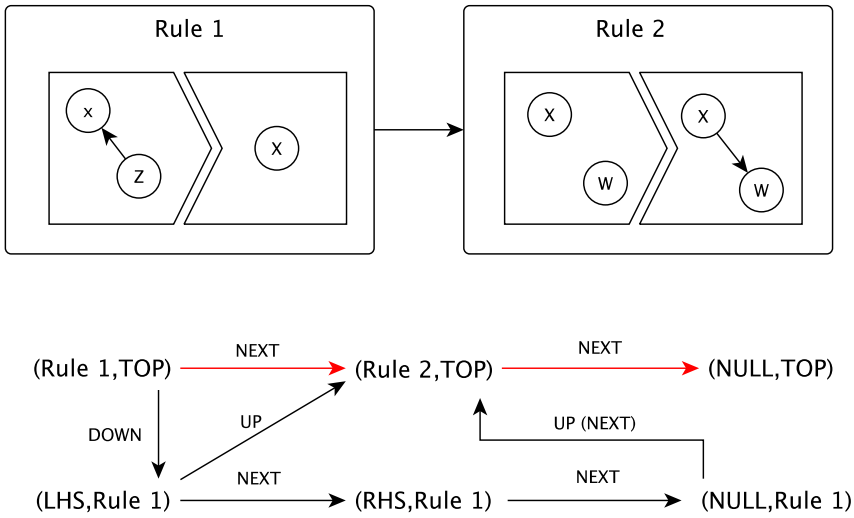

Figure 5–3: Navigation pointer evolution for a graphical formalism. Red arrows represent a step-over debugging scenario.

(skipping the debugging of the *RHS*) by *Up* command or by the delegated *Next* command after exploring the items of the *Rule 1* level.

In order to handle breakpoints, described in Section 5.5, we need to distinguish between two kinds of *NP* evolutions. The first evolution is *explicit*. This evolution produces *NP* traces when the debugging target execution is controlled by means of navigation commands. This trace does not contain the fine-grained details of the debugging target execution, and the details of *NP* evolution from the lower vertical levels is omitted. This is similar to stepping over the nested items of interest without exploring them deeper. For a comprehensive information about the target execution, we consider an *implicit NP* evolution. This trace, visible to debugger at all times, contains all *NP* value changes. The implicit values are necessary for us

to enable breakpoints based on *NP* values, pinpointing various locations throughout the debugging target execution flow.

Debugger behavior is of course not entirely addressed by this control flow model. We also need to consider how and when a debugger accesses data. Global, static data is universally available, but access to other, local data can depend on the language semantics given by the position in the control flow (such as with local, stack variables in a procedural language). As this depends on the language being debugged, we will require the target-language context to provide a means to expose (and represent) data, given a (current) navigation state, allowing the debugger read and write access in accordance with the expected semantics. We now move to apply the structured view of debugging to MTs.

5.3 Structured View of MT Stack

In this section, we apply our structured view of debugging to a MT stack typically found in rule-based MT systems. We describe the MT stack in terms of levels, items, and data. We need this in order to enable seamless debugging across the MT stack layers.

Typically, in MT debugging we are mainly concerned with MT specification, its use of the source model and the final effect the transformation has on the target model. Inside the transformation specification, we can discover the hierarchical structure of the schedule encompassing MT rules. Further down, we find individual patterns contained within the pre/post-condition parts of the rule. These individual patterns are used for matching in the source model and modifying the target model. Pattern matching and application steps are also important and need to be debugged.

In Figure 5–4 we outline a conceptual, level-based view of the MT stack. Rectangles represent the components such as static models and dynamic routines. The nesting relationship represents hierarchy or containment. There is a clear separation of data where the model artifacts are concerned. The data found inside the operational semantics of related components, however, such as the matcher of the pattern for example, is not clearly distinguishable on the diagram. We will clarify this concern below. In the following paragraphs, we discuss the MT stack in more detail and investigate how it fits within our debugger.

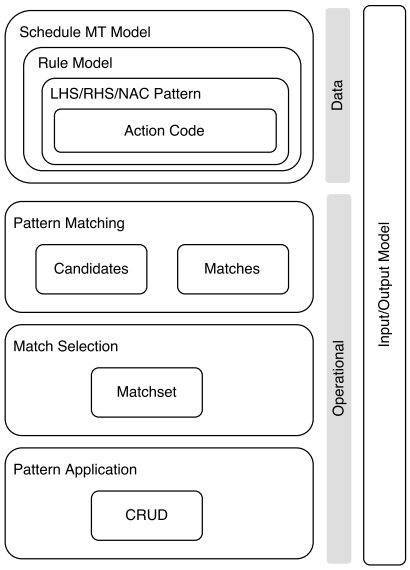

Figure 5–4: A MT stack view. Nesting of boxes represents hierarchy.

Input/Output Model Level. Shown on the right of Figure 5–4, this is the main data part of a model transformation. In our prototype example we perform in-place transformations, and therefore we expose that single model to the debugger. The model is global access data and we decide whether it should be accessible to

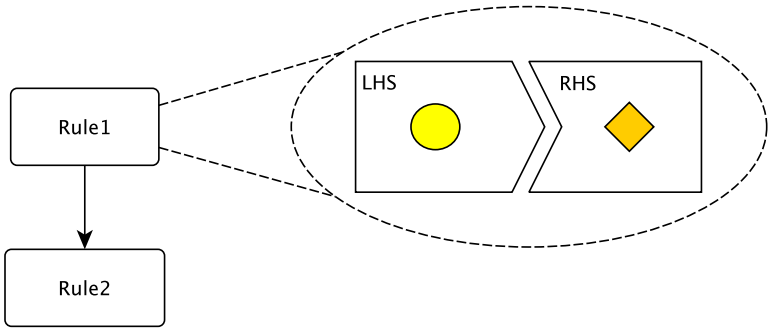

Figure 5–5: Example of MT schedule.

the debugger for inspection at any point in the target control flow space (in terms of navigation pointers that is). Of course, the model may contain sensitive, proprietary data, to which the target may want to limit or obfuscate access, depending on the requirements. Our choice is motivated by our desire to provide automated, declarative debugging of MTs and for that purpose, we assume full model access for querying from any point in the execution.

Schedule/Rule Model Level. The MT specification describes the control flow of the MT execution. This level is the heart of the debugging target. The nested structure of MT specification is giving us hints to the vertical and horizontal dimensions for the debugging. Note that although we think of this as specifying control flow of the target, at the same time it also represents data for our debugger to query, and we can use that perspective to guide a more MT-driven debugging process. In Figure 5–5 we demonstrate a mock-up of a MT schedule. Syntactically, rectangles represent rules (and possibly transformations as shown in Figure 5–3). To describe the hierarchical nature of MT schedule, we expand the *Rule1* rectangle

revealing the actual rule model the rectangle contains. The MT rule typically consists of *LHS* and *RHS* parts (with an optional *NAC*), each in turn containing patterns. We need to decide on the *VL* and *HL* items. We do this by analyzing the possible movement in horizontal and vertical dimensions. In this case, in terms of debugging we would navigate horizontally from *Rule1* to *Rule2*, as rule are processed at the same level of abstraction (the vertical arrow shown in the example is indicating that application of *Rule2* follows *Rule1*). Note that the *Rule1* is shown with its *LHS* and *RHS* parts and their individual patterns. The *LHS* tells us to find a circle in the input model, and *RHS* shows it converted to a diamond. The actual rule transformation is at a different level of abstraction from rule scheduling and represents a descent into a deeper, vertical level, wherein there is horizontal movement between the process of applying first the *LHS*, and then the *RHS* parts of the rule.

Action Code. The execution of the *LHS* and *RHS* parts contains further, nested execution complexity, most notably in terms of the presence of some action code (AC) used to specify imperative constructs otherwise too complex to express declaratively. Model elements attribute evaluation is one example of AC use. Treatment of AC requires a context switch in interpreting the MT specification. Execution semantics depend on the action language, and so requires a formal view of the language in terms of the debugger navigation pointer values, or would need to relay to the underlying general purpose language debugging facilities. We utilize those facilities in our prototype evaluation to deal with AC.

In summary, for this part of MT stack, the *VL* can be obtained by exploring the containment relationship in the MT schedule presented. If the schedule is presented in the textual format the hierarchical relationship could also be explored by descending into the function/procedure calls. In turn, a horizontal dimension of each *VL* is exposed by enumerating the items without exploring the hierarchy, as in the case of the *LHS* and the *RHS* parts of one rule. We continue with the low-level rule application details.

5.3.1 Pattern Matching/Application Level

We now discuss the parts of MT stack which are arguably overlooked the most in the MT debugging or are exposed to a limited degree of in the MT tools.

Pattern matching. Going a level down into the pattern matching process we are presented with the tool specific implementation. We need to agree on the unifying concepts of pattern matching applicable to various solutions that can be used uniformly during debugging. Local search-based techniques will operate differently from the constraint satisfaction pattern matchers (such as VF2 [19]). However, with some degree of certainty, we may assume that both will return the bindings representing the match and both will iterate over the candidate bindings in an attempt to grow the match further.

We propose below unifying concepts related to pattern matching. This gives us the ability to use them regardless of the pattern matching algorithm employed. The items below can also be considered as data that we can query during the debugging.

- *Candidates* - a list of bindings between the pattern elements and the input model the matcher considers to produce the match. Observing candidates gives us an insight into how the algorithm goes about finding a match.
- *Bindings* - list of bindings between pattern elements and the input model that positively form a part of an incomplete match.
- *Match* - a complete match as a set of bindings.
- *Matchset* - a set of valid matches for one pattern. Typically, from this set, a match is selected for the rule application when non-determinism is present.

Movement in the horizontal dimension within these sets happens by iterating over set items by means of the *Next* command. In addition, we decided to treat each of these categories as being on the same horizontal level, and one vertical level below (inside) the pattern matcher. Certainly, it would be possible to assume a different structure in treating these groups of items inside the pattern matcher. For example, a set of candidates inside the match item.

Match selection. Another part of the match/apply process is the match selection from the matchset containing all matches found for the pattern. This addresses the question of non-determinism (described earlier in Section 2.3). It is important to announce to the user which of the matches were selected by the tool for a rewrite (application) phase. This can be approached simply by exposing the selected match. Alternatively, depending on the implementation, the candidates for the selection can be exposed and offered to the user for inspection and modification. In this paper, we treat this simply and define the *selected match* concept. The non-determinism exposed here may possibly translate into an additional dimension of our structured

view of the MT debugging, as each match selected for the application stage may result in a different control flow. We leave the treatment of additional dimensions in our structured view to future work.

Pattern application. For the pattern application process related to the *RHS* part of the MT rule we also need execution concepts to consider in our debugger. This design can also be quite implementation specific. As a general solution, however, we rely on the Create, Read, Update, Delete (CRUD) operations affecting the input model and define the sets related to each one (except for the Read, since in this context the Read operation belongs in the pattern matching domain). We also consider these items as data available for querying. Our interpretation of the CRUD operations is based on element-labeling typical of MT rule design, where unique labels on pattern elements are used to assist in identifying elements meant to be the same or different in the *LHS* and *RHS* patterns; the labels absent in the *LHS* but present in the *RHS* indicate the creation of an element and element deletion in the opposite situation. We present below the sets related to the pattern application.

- *Create* - a set describing the elements that will be created in the output model. This set is populated by *RHS* elements that do not have corresponding labels in the *LHS*.
- *Update* - a set describing the elements that will only be updated. This set is formed from pattern elements that have mirroring labels in both parts of the rule.

- *Delete* - a set of items that will be deleted from the input model. It is created from items that are only present in the *LHS* and have no correspondence in the *RHS*.

Just as in the case of pattern matching, navigation in the horizontal dimension on this level should happen by enumeration of the set items presented in the sets above. Note, that while we aim to explicitly expose pattern matching and application, we leave user-modification of these processes to future work.

In the next section, we describe a debugging language that is utilizing the structured view and the navigation commands to model interaction with a debugging target.

5.4 Debugging Language

Our debugger design builds on a custom debugging language expressed through *debugging rules*, which follow the familiar MT rule structure, including *LHS* and *RHS* parts. This approach allows us to incorporate domain-specific syntax for different layers. Debugging rules can also be chained to form *debugging scenarios*, which can be executed separately from a target MT, facilitating automated debugging.

We begin with the discussion about the *LHS* part of the debugging rule, followed by a description of the possible action the debugging rule can perform. Also in this section, we describe the use of *scopes* to indicate where in the MT stack the *LHS* query will perform the search.

Querying. The use of the *LHS* of a MT rule is that of a pre-condition for the application of the *RHS*. The *LHS* patterns are matched in the input model and for our purposes, this can be considered as a query over data or state of a debugging

target. Therefore, it seems natural to use the *LHS* for all queries related to the debugging problem. These queries include:

- *Input/output model* querying. The problem domain formalisms used in the MT are then reused in the debugging rule specification without the need for any extra effort on the part of an engineer. The input/output models represent the main data part of the MT system. An engineer may have an idea about an undesirable pattern appearing in the model resulting from the application of a MT. It may become quite tedious to interactively step through the transformation and monitor the transformation results until the problematic pattern appears. Therefore, it is important, for MT tools to facilitate declarative and automatic querying of the models (or a debugging state of MT in general). This is the intended application of queries this thesis. In a way, this is similar to anti-pattern querying in the tool IncQuery [16]. These patterns represent undesired changes in the model that the tool monitors through incremental pattern matching.
- *MT specification* querying. The MT specification is a model itself and is already treated as an input in the area of higher order transformations (HOT) [99]. This permits us to issue queries over the MT specification. Further if necessary, the HOTs can also be applicable in the context of debugging. The debugging rule may modify the image of a query found in the MT schedule thus providing the *adaptation* facility of a debugger mentioned in Section 5.1.

- *Pattern matching and application* querying. The data involved in the pattern matching and application process can also be the target of queries. For example, a match containing the bindings between the pattern and the input model or any other data can be exposed to the debugger. We can envision the use of an appropriate graphical formalism to query in this context, as the complete match, for example, should conform to the MM of the input model and is a graph (even though it may not be represented as such in the pattern matcher implementation). If this low-level data does not conform to the meta-model of the languages involved in the transformation we are debugging, the use of normal *LHS* mechanisms for pattern matching may therefore not be available, and thus we currently make use of action code to read and write that data. The complete treatment of such cases we leave for future work. Finally, in this context, we also reserve certain keywords for the match discovery described in Section 5.3.1 for more precise querying. The keywords include: *candidates*, *bindings* and others.
- *Navigation pointer* querying. In order to reason about the location within the debugging target, we need to form a query based on the navigation pointer pair, or some part of it. This is useful in order to perform an action when the MT control flow enters a desired location in the MT specification. For example, we may want to break the execution of the MT when a specific rule is executed. We intend to use action code in order to match the current *NP* value to the one we may be interested in.

Note that the *LHS* condition can be further augmented through the use of *NAC*s to specify negative application conditions, and so inhibit application of the debugging rule.

Action. After successful query discovery (or simply a pre-condition satisfaction), we want to perform an action. For this, we use the *RHS* of the rule, specifying traditional debugger actions, as well as modification of the various parts of the query domains. We focus mainly on the former in this chapter, but many other effects of the *RHS* action are possible, including modifications to the input/output and the MT specification models. The latter, for example, allows us to perform the *adaptation* of MT specification, described in Section 5.1, for the exploration of new execution scenarios. We may also want to influence the pattern matching and application process by modifying such data as match candidates, matches, and matchsets for example. More detailed investigation of execution adaptation and pattern matching/application influence represents an advanced debugging session, which we leave for future work. Finally, we may also want to perform simple miscellaneous operations such as printing the values to the console or file. For this, we rely on the action code facilities typically available in the MT implementation.

Navigation Commands. One of the goals of our debugger is to control the execution of the debugging target by means of issuing navigation commands. To issue navigation commands we embed them within the *RHS*. Application of the rule and successful matching of the *LHS* pattern then results in the command being performed. Such debugger commands can be simply issued through the use of action code. A visual representation, however, better fits the MT paradigm and allows for

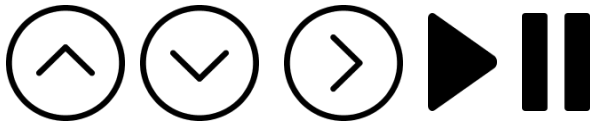

Figure 5–6: Concrete syntax of the navigation command language. The last two icons are for the resume and pause operations.

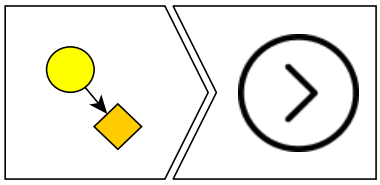

Figure 5–7: Rule example that advances the execution of MT based on a *LHS* pattern match.

a more natural integration with other *RHS* elements. We have thus chosen a simple graphical concrete syntax, shown in Figure 5–6. These symbols naturally map to the *Up*, *Down*, and *Next* operations comprising our navigation strategy, as well as a resume (repeated *Next*), and asynchronous pause behaviors found in debuggers. In terms of common debugging parlance, the *Next* operation represents *step-over*, *step-into* maps to *Down*, and the *Up* operation maps to the *step-out* operation.

Debugging rules. With this in mind, we can now create the first debugging rule as shown in Figure 5–7. This rule advances the execution of the MT by issuing a *Next* command if the pattern in the *LHS* (a query) of the rule is found. At this moment we ignore the location of the input (model) applicable to this query.

The rule in Figure 5–7 represents a slight semantic departure from typical MT rule design. In traditional MT rule design, the absence of the *LHS* pattern in the *RHS* of Figure 5–7 indicates that the occurrence of the *LHS* pattern should be

deleted (as well as *Next* invoked). As this is not typically the combined intent of a debugger action, we assume that when a navigation command is present in the *RHS* the rule becomes read-only, and the occurrence of the query present in the *LHS* will not be modified.

This approach limits the expressiveness of *RHS* actions, but avoids the need to repeat the *LHS* pattern in the *RHS* when performing common navigation commands. When modifications to the occurrence of a query in combination with navigation are desired, the user will need to perform this action with two rules in sequence, one to perform the modification, and another to issue the navigation command. Overall, the execution order of debugging rules depends on the debugging scenario scheduling. In this case, however, it makes sense to execute the write rule while the target MT is paused and before the second rule advances the target execution by issuing a navigation command.

In order to understand how the debugging scenarios work and their possible limitations, we need to agree on the semantics of interaction between the debugging target and the debugger. When a debugging scenario is first executed, the target should be paused. The debugger is then free to control the target through the application of debugging rules. The debugging scenario can issue the continue/resume command after which the target can be paused again and another debugging scenario executed. The debugging rule application when the target is in the running state should be avoided because the state of the target will evolve independently resulting in an undefined scenario behavior.

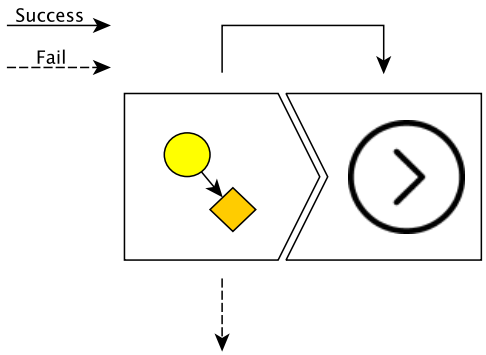

Figure 5–8: Simple debugging scenario that repeatedly advances the execution of MT based on a *LHS* pattern match.

With this in mind, Figure 5–8 demonstrates a simple debugging scenario, that uses the rule from Figure 5–7 executed repeatedly in loop. The successful rule application results in a repeated, *step-over* debugging target execution on the current *VL*. The failure, terminates this debugging scenario. We continue with a discussion on how the scope concept from Chapter 3 can be applied to the debugging rules.

5.4.1 Scope use to indicate location

Given the many sources of data described in the previous section, the rule in Figure 5–7 is ambiguous in terms of which input model is applicable to the query. A straightforward approach to enumerating query pattern occurrences is to search for them everywhere, from the input model and the MT schedule, down to the match bindings in the matcher routine. Rule efficiency, clarity, and readability can be improved, however, by indicating the location of the input for a query, allowing the query to be applied with more specificity.

To represent the search area in a meaningful way, within the MT rule paradigm, we make use of our scope formalism proposed in Chapter 3. Scope provides a means to specify the locality of pattern matching and application in the input model using graphical and textual formalisms. Our debugger thus defines several, global scopes to represent the fundamental parts of the model transformation we are debugging. Note that while in principle scopes can be modified by transformations, the value of doing that in a debugger context is less clear, and so we assume scopes are read-only, with appropriate scoping preserved in any *RHS* transformation. In addition, for simplicity, these scopes are not hierarchical. However, it would be possible to use scope hierarchies if necessary. Below we propose several scopes suitable for MT debugging context.

- *Host* - this scope gives our debugger access to the host graph encoding the input model.
- *MT specification* - this scope is used for MT querying and HOTs. For simplicity, in this paper, we define a global access to the MT specification—regardless of the navigation pointer values, all the rules of MT are available to debugging scenario for query and modification.
- *Navigation* - this scope allows us to query the navigation pointer. This actually represents a set of scopes, parametrized by navigation pointer values, so as to allow a rule to focus on a specific navigation instance.

The rule in Figure 5–9 demonstrates the use of the scope formalism in the *LHS* part of the rule. This rule issues the *Down* command to the target MT if the pattern is found in the input model. The rectangle "Host" represents the scope of the input

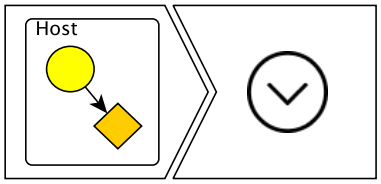

Figure 5–9: Rule example that issues *Down* command to MT based on a *LHS* pattern match in the input model. A scope formalism is used to indicate location of the query.

model—any item placed in the scope rectangle, through the containment relationship is said to be in that scope. Now the rule is clearly indicating the location for query matching.

Use of scope helps with disambiguation but is meant mainly as a performance optimization. To better take advantage of the possible performance benefits, however, the creator of a debugging scenario may want to create their own scope, dynamically modifying it as necessary. For example, suppose the user relies on a complex query based on part of the input model, but which may need to be checked frequently, such as in the case of data-based debugging scenario. Using a write rule in the debugging scenario, the user marks part of the input model with a unique scope. This part of the input model, now in scope, represents an incomplete query, but will eventually grow to produce a pattern completely satisfying the query. The pattern matcher can start checking for that query from and around the scope, potentially greatly reducing the matching cost as we demonstrated in Chapter 3. Search plan-based matchers can be particularly helpful in this context; tools supporting search plan matching include GrGen and Viatra [33, 42].

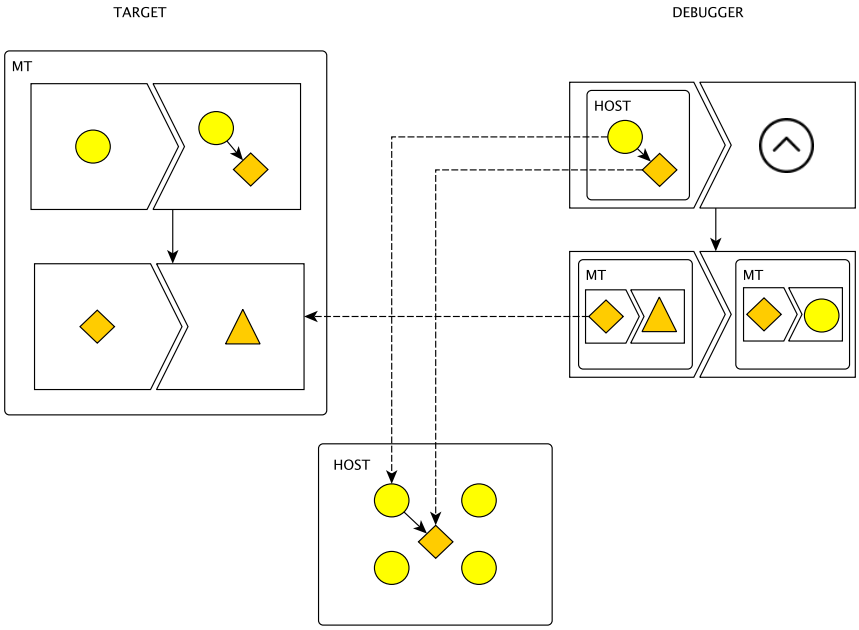

Figure 5–10: On the left is the target MT consisting of two rules, on the right is the debugging scenario. The target specification is contained within the MT scope rectangle and the input model within the Host scope. Dashed lines represent matches for the *LHS* parts of the debugging rules.

In Figure 5–10 we show an overview picture of a debugging situation demonstrating the use of scopes. On the left, is a MT rule schedule, on the bottom is the input host graph. Both the schedule and the host graph are displayed enclosed within their respective scopes. On the right is the debugging scenario, consisting of two rules. The first rule expects a pattern in the host scope. A successful query of this pattern issues the *Up* command and schedules the next debugger rule. The next rule uses the MT scope and has the purpose of a HOT on the MT specification,

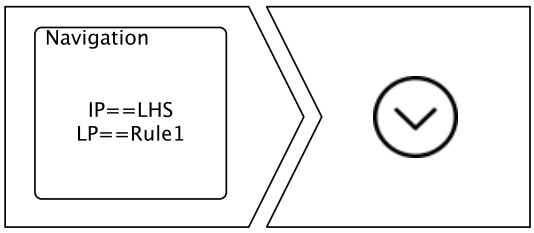

Figure 5–11: Rule example that uses scope to query navigation pointer and results in a debugger action.

modifying a rule that transformed diamonds to triangles into one that transforms diamonds to circles. The dashed arrows are indicating where the *LHS* of the debugger rules find their matches.

Finally, in Figure 5–11, we demonstrate the rule implementing debugger action based on the navigation pointer values. We utilize the scope labeled *Navigation* to encapsulate the *NP* values. In the event the execution enters the *LHS* of the rule, the action will result in *step-into* operation. Note that in this example, the navigation pointer values are encoded as strings. This facilitates the exchange of these values between the debugger and the debugging target, and increases readability. The ordering, allowing us to move between the *NP* values, is provided by the debugging target. This includes, for example, the schedule of the MT being debugged encoding the ordering. The evaluation of the *NP* happens using the action code within the scope pattern element in the *LHS* of the rule. We can also envision the use of regular expression style matching over the *NP* values to cover a range of *NP* values in one query. For example, we can query when the debugging target processes the *RHS* of every rule (see Figure 5–12) or the input model is modified (enumeration of *Update*

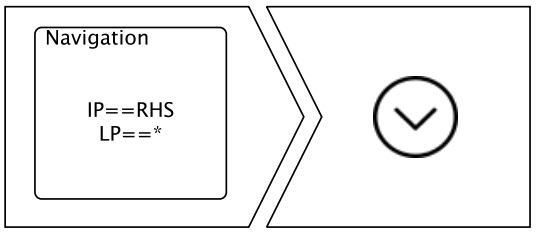

Figure 5–12: Debugging rule example that results in a debugger action upon the processing of the *RHS* of every rule in MT. The star represents any value.

set during pattern application). We now continue with a discussion on breakpoints and debugging scenarios modeling the watchpoint-like functionality in our debugger.

5.5 Breakpoints and Watchpoints

In order to support a well featured MT debugger, we need a comprehensive breakpointing support. In this section, we describe breakpoints in our debugger concept. We begin with the manual breakpoints set within the MT stack. This is similar to setting breakpoints over the lines of code. We then describe breakpointing/watchpoining realized through the debugging scenarios. Note that, for our purposes, we redefine the common notion of a watchpoint in debugging. The classic watchpoint interrupts execution of a program and returns control to the user when a change to the observable state of a debugging target is detected. A watchpoint in this chapter is a debugging mechanism intended to query the state of a debugging target. Upon the satisfaction of a query, some watchpoint action is performed. The action is not limited to returning control to the user but can include tracing, debugging state modification, or even the navigation commands.

Manual breakpoints. Let us first consider breakpointing in the context of a typical interactive debugger, without the use of the debugging scenarios modeling debugger interaction. *Manual breakpoints* are the breakpoints that can be placed within the MT stack before MT execution. Intuitively, such breakpoints are more applicable to the MT specification as it is typically a model available for inspection and modification. Recall, that our debugger observes the debugging target through the *NP* evolution. Each *NP* then represents a possible breakpoint location at which the execution can be halted and the state of the debugging target inspected. Clearly, for this, the debugger needs to be aware of the implicit evolution of the *NP*s. Manual breakpoint location then needs to be associated with the *NP* value. This can be achieved in two ways. First, the debugging target can label the *NP* value as a breakpoint. The debugger then halts the target execution upon receiving such *NP*. In case the debugging target has the facilities to indicate breakpoints in the MT stack (such as within the rules or even in the action code), the implementation responsible for producing the *NP*s needs to handle labeling of the *NP*s within the target. Second, the breakpoint as the *NP* value can be specified in the debugger itself. The debugger then compares incoming *NP* values to the internal list of breakpoints it has. Naturally, the latter kind of breakpoint specification needs to conform to the *NP* values produced by the debugging target.

Watchpointing using debugging rules. Watchpoints in our design are modeled and driven by debugger rules, depending on the variety of query specifications. Introducing a break or other action in an explicit debugging scenario is shown in

Figure 5–13. The purpose of this debugging scenario is to step through the execution, by issuing *Next* command (with an unconditional query that always evaluates to the true value) until the creation of a circle in the input model can be detected by a second rule. Rule scheduling applies the first rule (which always succeeds), then the second, returning to the first rule when the second fails to apply. Upon success of the second rule a debugging action is performed, in this case, a trace is done using action code.

A pause or other navigation action could be performed instead or as well. Recall, however, that the debugging target is in the paused condition after a debugging rule application, unless the resume/continue command was issued. Therefore, defining a pause action in the rule (as to create a breakpoint) is unnecessary and can be used simply for readability purposes and to make the target pause intent explicit. This is the reason why we chose to refer to such debugging scenarios as watchpoints rather than breakpoints.

The pace of debugging in Figure 5–13 is given by the use a *Next* operation, which depends on the navigation pointer position at the time the command is issued; it may also be desirable to step execution at a finer granularity, such as by using a *Down* operation in the first rule to model a fine-grain (step-into) execution. Modeling watchpoints in this fashion mimics the user repeatedly invoking navigation command until a condition is reached.

Writing explicit watchpoints can become tedious. Therefore, we extend the MT language with an addition of a syntactic sugar construct to represent the repeated attempt of a debugging rule application. Once the rule is successfully applied (the

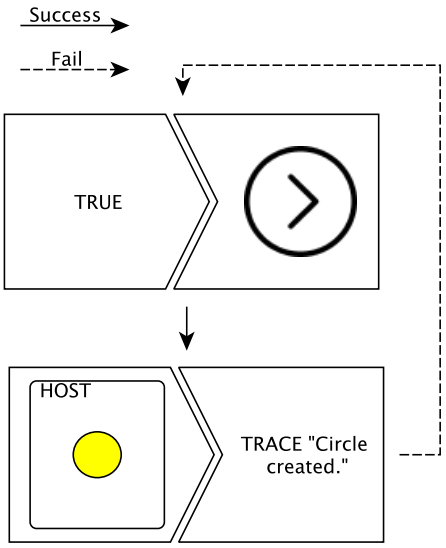

Figure 5–13: Explicit watchpoint debugging scenario resulting in a trace after the query in the input model is found.

query is satisfied) the execution of a watchpoint construct is terminated and the control can be passed to a next MT language construct. In Figure 5–14 we show this construct as a grayed out rectangle housing the debugging rule. While this construct is executed in a debugging scenario, the target will be stepping through its execution, with rule failure implying repetition. The debugger controls the granularity of the target in the explicit debugging scenarios (as shown in Figure 5–13), in the proposed syntactic construct however it is unclear what granularity is implied. Parametrization is certainly possible, however by default, we propose that the finest step granularity should be used in order to catch the minuscule and incremental debugging target state changes. Therefore, the construct in Figure 5–14 is equivalent to the construct shown on the right in Figure 5–15. Here the finest granularity is ensured with the *Down*

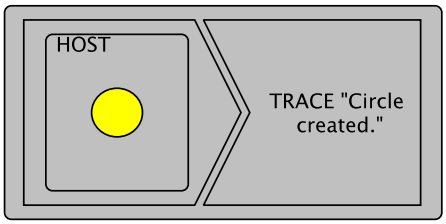

Figure 5–14: A syntactic sugar debugging scenario resulting in a trace after the query in the input model is found.

command execution. Certainly, some watchpoints do not require fine granularity and the performance of a target execution will suffer if the syntactic sugar constructs are used unnecessarily frequently.

The new construct can also house several watchpoint rules as shown on the left in Figure 5–16. There, we see two watchpoints, one is for the occurrence of a pattern in the input model and another for navigation pointer values to be pointing to the *LHS* of a particular rule. The latter is having the pause action for readability purposes described earlier. The semantics of this construct is that each of the watchpoint rule queries contained within will be checked once while the target transformation is suspended. The process repeats after a single step in the target transformation. We show the equivalent, explicit debugging scenario on the right in Figure 5–16. After advancing the target execution the debugging rules are checked in sequence. The first rule to succeed breaks the debugging scenario's execution for further processing.The execution of the target remains paused and the control is handed over to the next rule in the debugging scenario.

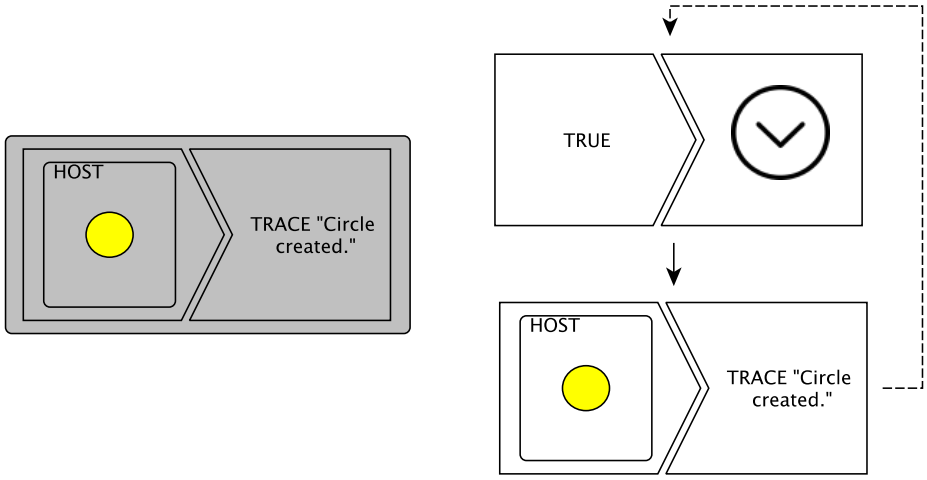

Figure 5–15: A syntactic sugar debugging scenario on the left and its equivalent explicit scenario on the right.

Constructs for coarser granularity may also be interesting to explore of course, and the explicit debugging scenario form remains as a means to give the user full control. We now proceed to discuss possible implementation.

5.6 Prototype Implementation

Evidence of the utility and practicality of our design is given by a sample implementation. In this section, we discuss the realization of our debugger within an existing research-oriented MT tool AToMPM used in our lab. We show how our initial implementation can be reused to a certain degree in other MT tools such as AToM^3 [25] and ATL. We conclude this section with a discussion on performance implications of our solution.

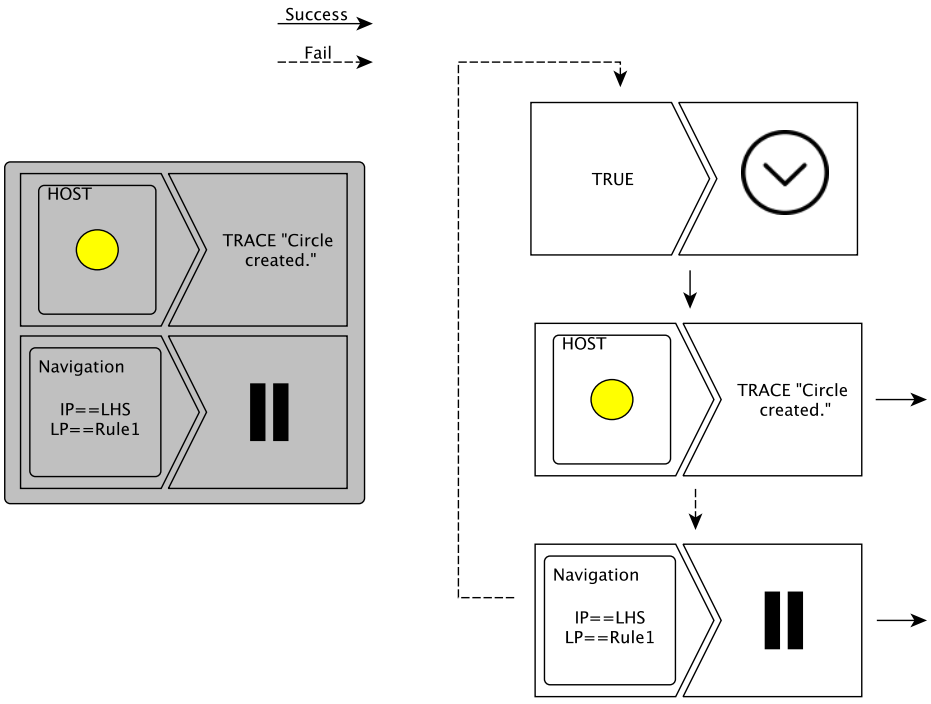

Figure 5–16: A *watchpoint construct*, watchpoint rules inside are tried until one is applicable, resulting in a debugging action.

5.6.1 Implementation in AToMPM

AToMPM [59], a Tool for Multi-Paradigm Modeling, is a multi-formalism/multi-abstraction meta-modeling and model transformation tool. The front-end of AToMPM is a browser-based user interface. In AToMPM everything is modeled, from user interaction to the tool bars in the browser. The tool provides the class diagram-like meta-model used to create custom DSLs and the MTs. Finally, the graph rewriting functionality of AToMPM is implemented in a Python-based back-end.

We begin by describing the general architecture and the high-level view of the interaction process between the debugger and debugging target. We then proceed by describing modeling activities we perform to produce the debugging formalism artifacts described up to this point (for brevity, we do not include the meta-models we modify), followed by the development activities performed.

General Architecture. In Figure 5–17 we show a general architecture of our system in AToMPM. The squares in the figure represent components of the system. This is a single process system where a debugging target, debugger and a *Statecharts* controller have separate threads of execution. The components, however, have shared memory access. Shared memory allows for an easy access to the debugging target state. In this case, it was trivial to enable data exchange between two threads of execution, one being the target model transformation, and another being the debugging scenario. In a different implementation scenario, shared memory access may not be possible. In such cases, extra effort may be needed to implement data exchange through a network or other means.

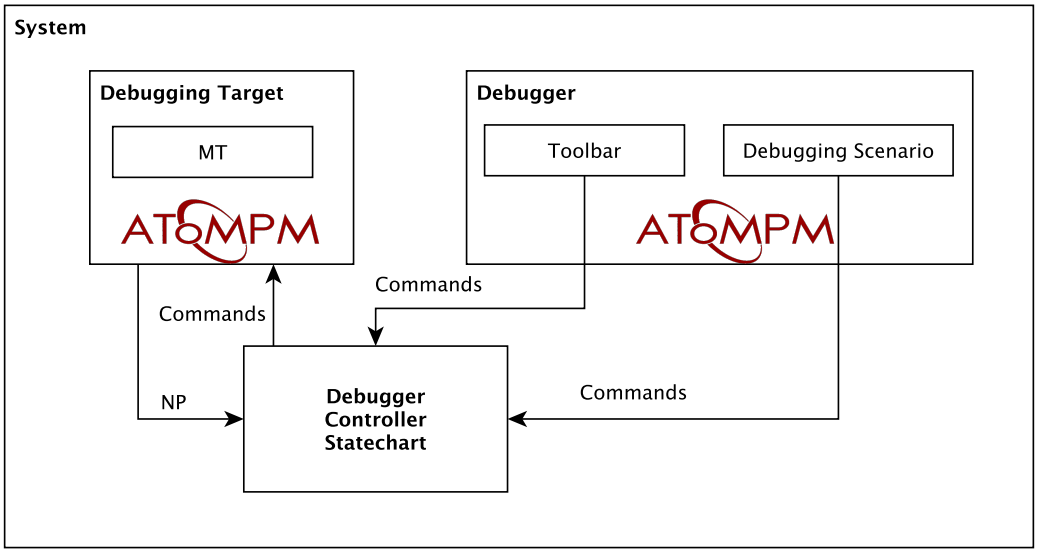

Figure 5–17: A general architecture of our debugger in AToMPM. Debugging target and debugger are two AToMPM instances (threads of execution) communicating through a *Statecharts* and have a shared memory access.

The debugger and debugging target are two separate AToMPM instances. These are typically two browser windows that have access to a Python back-end process (each AToMPM instance has a separate thread of execution in the Python-based back-end). The back-end is the heart of AToMPM MT system where the graph rewriting and rule handling takes place. Further, in this section, we describe the modifications to the back-end necessary to implement our debugger.

The heart of our debugger is realized in the debugger controller using the well-known *Statecharts* formalism [38]. *Statecharts* were chosen because of their convenience in describing autonomous, concurrent, and reactive systems. The inspiration for our solution comes from work by others on reimplementing existing model execution engines in *Statecharts* with the addition of a debugger related functionality [101]. We chose a different path and implement the main logic in a central *Statecharts* model (Python-based). The *Statecharts* controller resides in the AToMPM back-end. Its purpose is to receive debugger commands either from the debugger toolbar, for interactive debugging, or from the debugging scenarios. In addition, the controller receives the *NP*s from the debugging target and instructs the target whether to halt or proceed with the execution. We will return to the actual *Statecharts* model later in this section with more details. Finally, the shared data access model permits the debugging scenario running in the debugger to access the state of the debugging target. The scenario can read the input model, the MT specification, and the pattern matching/application related data.

Modeling activities. To implement our debugger, certain modeling activities were necessary. We model navigation commands as visually shown in Figure 5–6.

This allowed us to place the commands in the existing MT rules without additional modification to MT specification. The treatment of such rules had to be however augmented in the internal implementation (recall the read-only debugging rules).

The scope formalism, as described in Chapter 3, is able to contain any formalism (DSL). Recall, that we use the scope to indicate the location of a debugging query. We reuse the scope formalism unchanged in this prototype. Some back-end modification was necessary in order to associate the scopes with the debugging target data.

Toolbars in AToMPM are explicitly modeled, and therefore, for the debugger interface, we modify the MT control toolbar to include the buttons corresponding to the navigation operations. Pressing the button then sends a command to the *Statecharts* controller resulting in the reaction of the debugging target. In essence, the toolbar can be used for interactive debugging of the target without the need for constructing a debugging scenario. In addition, it is possible to model the button press as an execution of a debugging rule responsible for a particular navigation command.

Debugging target development activities. The first major development activity was to modify the back-end of AToMPM to generate a *NP* at every point of interest. Here, the engineer should make choices to what *NP* values the target will expose to the debugger. Our choices here are based on the debugging items and *NP* values from the MT stack we describe in Section 5.3. We identify all locations in the Python back-end code that are processing debugging items. These locations are relevant to the changes in navigation pointers and processing of the items of interest. We are mainly concerned with the schedule, individual rules, the *LHS* followed by the

RHS, down to the relevant action code evaluation and pattern matching/application routines. We need to communicate debugging items, as *NP*s to the *Statecharts* controller model. The items can also be hierarchical leading to other internal items. Therefore, we need to indicate the beginning and the end of item processing in order to have an image of the hierarchies. For example, the beginning of an item such as the processing of an individual rule constitutes a beginning of a rule item. This, followed by the processing of the *LHS* (before the end of rule item is announced) indicates that the rule item is nested. The end of a rule item alerts the controller to the completion of the composite item and that the particular *VL* is explored to the end. We add extra information to the *NP* sent to the controller. This information can contain action code line numbers, rule names and can be used for such purposes as code line highlighting and highlighting of control flow in the debugging target, as necessary.

The point where the debugging item begins also marks a place where the debugging target may be paused before processing the item. The pausing functionality is achieved by waiting for a message from the *Statecharts* controller to advance the execution. This is achieved with a blocking queue provided by the *Statecharts* implementation. With these modifications on the debugging target side, we can already have an interactive debugger. The debugger will have the pause/resume and other debugging command support (ignoring, for now, the *Statecharts* controller implementation details).

More modifications are necessary in order to support the debugging scenarios. We modify handling of the MT rules to support read-only rules issuing commands to

the *Statecharts* controller. We also ensure that the debugger executing a debugging scenario waits for a notification from the controller about reaching paused states after execution of the last debugging rule.

In summary, in AToMPM, the implementation of a simple interactive debugger is achieved without complex, intrusive modifications to the debugging target. Depending on the complexity of the debugging target, however, these modifications may not be as straightforward, and for instance, involved additional complexity porting our solution to ATL (described in the following sections). We now address the *Statecharts* controller implementation.

The *Statecharts* controller. The *Statecharts* controller is implemented in *SCCDXML* format [102]. This format is an extension to *SCXML* format[5] specifying *Statecharts* models. The extension adds support for runtime instantiation and destruction of objects that internally represent *Statecharts* models. This additional functionality was not explored in this thesis. Instead, we utilize the SCCDXML compiler for basic *Statecharts* model compilation, producing executable Python code from an XML-based specification. The SCCDXML format is textual and we convert it to the SCXML format with some modifications. For display purposes, we then import and edit the resulting file in QTCreator's[6] *Statecharts* graphic editor. Due to a particularity of the tool, not all of the information related to the *Statecharts* transitions is displayed. For example, the tool hides transition guards. Where it is

[5] `https://www.w3.org/TR/scxml/`

[6] `https://www.qt.io/ide/`

important, we will describe these hidden transition attributes and add guards. If the transition in the diagram is missing an event that triggers it, then that transition is unconditional.

In Figure 5–18 we demonstrate the debugger *Statecharts* controller model. Note that this is just one of the possible ways to implement the *Statecharts* controller model. Our version of the *Statecharts* model deals with the navigation commands, receives navigation pointer values from the debugging target, and permits the target to advance the execution. The navigation pointer events from the target are processed in the *ProcessingItems* state. Here, the parametrized *newitem* event signals to the controller the beginning of the debugging item and arrival of a new *NP*. The new item is processed in the nested *PushingStack* state. In this state, the custom Python code, embedded in the *Statecharts* model reacts to the creation of a new item and possibly newly nested *VL*. Once the target signals the end of the item with an *enditem* event, the controller performs necessary computations to indicate the end of a particular *VL* in the *PoppingStack* state.

Upon receiving an item, a new event called *newitemtrig* is raised. It is processed in the *MTSpecificActionsVisualization* state. Each *newitemtrig* transition has additional guards to distinguish between various new items. Only one is shown for brevity in the model. New items may carry information such as rule names, action code line numbers, etc. In this state, we use the additional information to perform such actions as visualization or highlighting in the target user interface. For example, action code line number highlighting. This state is the place where we would add processing of new types of items, if necessary. For example, an engineer wants

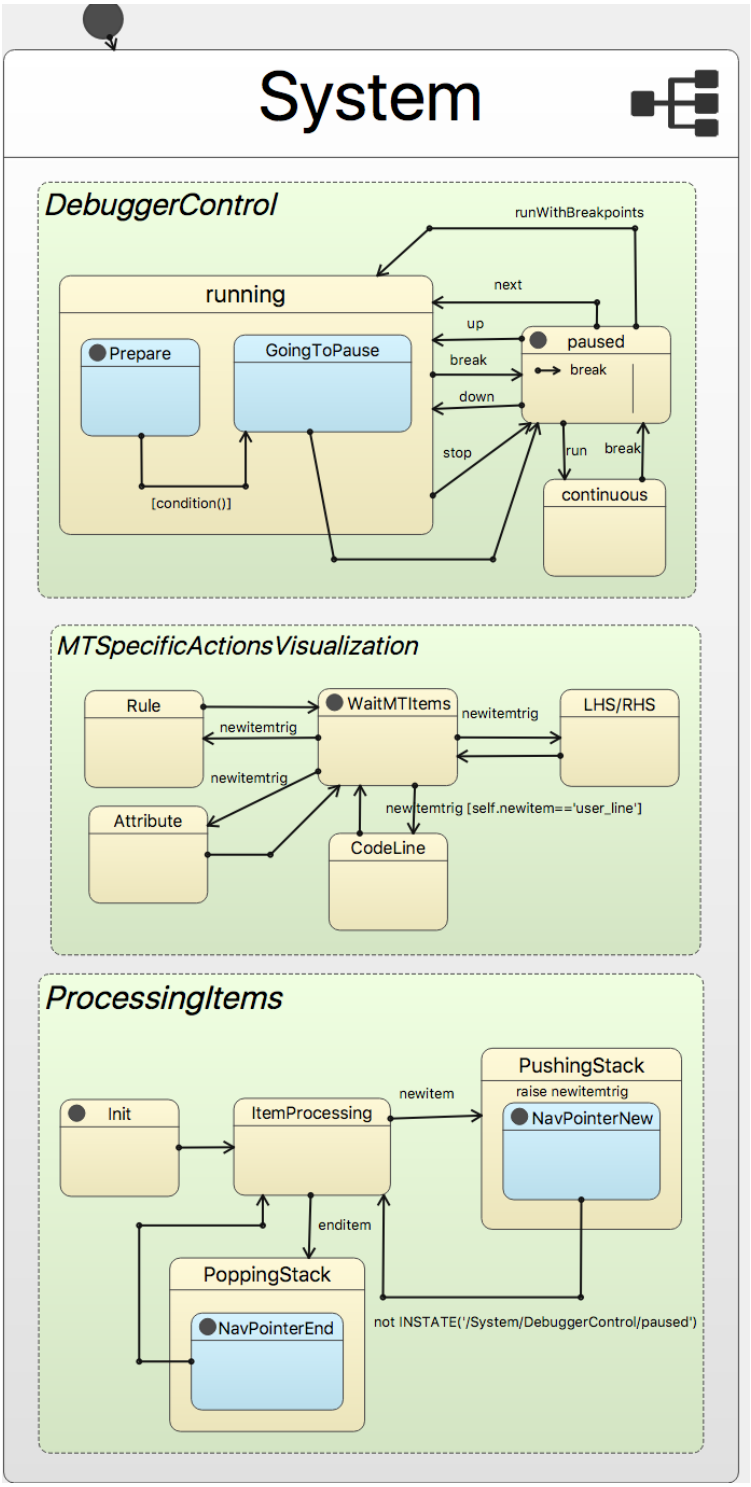

Figure 5–18: A *Statecharts* model of the debugger controller. Orthogonal components are responsible for processing navigation commands, navigation pointers, and implementation specific items corresponding to navigation pointers. Empty transitions are unconditional.

to introduce highlighting of a function body in the action code, in addition to already present line-by-line highlighting. The *MTSpecificActionsVisualization* state is intended to be implementation specific, as opposed to other states aiming to be as generic as possible, because it mostly deals with the user interface of a specific tool.

The pausing functionality of the target is achieved through the transition from states *NavPointerNew* to *ItemProcessing*. This transition is taken if the *paused* state of the *DebuggerControl* state is inactive as indicated with a guard. On taking the transition an event is raised, that is sent to a halted debugging target, signaling the target to proceed with the execution.

Runtime context is maintained in the *DebuggerControl* state, where we process the navigation commands. The controller can be in three main states. The state *paused* indicates that the debugging target is paused. From there, the target can run continuously as indicated by *continuous* state. Finally, when the target is paused, navigation commands can take place, placing the runtime context into a *running* state. The duration of the *running* state depends on the navigation command. The execution will be automatically halted by a transition from *GoingToPause* state to the *paused* state. For example, issuing a *Next* command will enter the *running* state, after which the single item in the given *VL* is processed. Upon receiving an end item event, the controller will ensure that the *paused* state is entered before the next item on the same *VL* arrives. Finally, the *running* state is also equipped to run the target until an *NP* with a manual breakpoint is encountered.

User interface. In Figure 5–19 we show a screenshot of the tool. The symbol outlined in red with a number one indicates the button that opens the second window

to load a debugging scenario for the transformation within the current window. The set of symbols outlined with number two shows the debugger toolbar. Left to right are symbols for loading a debugging scenario (transformation), continue, *Next*, pause, stop, *Down*, and *Up*. (As we reused an existing symbol set here these symbols slightly differ from the ones we showed earlier.) It is possible to use the debugging toolbar

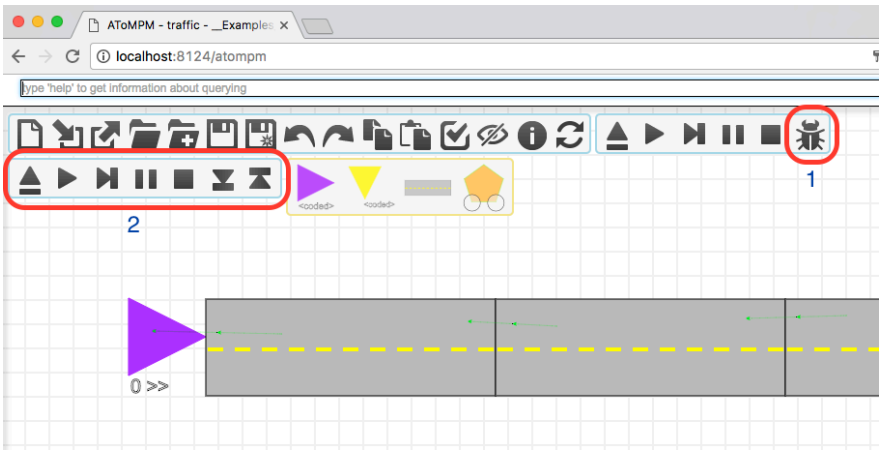

Figure 5–19: A screenshot of our tool and the debugging toolbar.

without the use of a debugging scenario for an interactive target control.

The workflow of the debugging scenario is the following. We load the input model and the transformation. We press the bug icon to open a second window where we load a debugging scenario (a transformation) using the toolbar outlined in Figure 5–19. We play the debugging scenario which in turn controls the target MT.

Action code. Treatment of action code (AC) used in MT rules deserves additional explanation. The implementation is approached by utilizing the AC's language facilities. In the case of AToMPM, the AC is Python, which provides an interface to develop custom Python debuggers called *BdB*. By extending this class, we can then

process Python code specific events and initiate communication with the *Statecharts* model to announce changes in navigation pointers and debugging item processing. For example, here, the navigation pointers change on events such as when the control flow descends into the function or the next code line/statement is processed. The navigation pointers can carry line numbers to the *Statecharts* controller that we can use for highlighting in the action code as demonstrated in Figure 5–20. In addition, the same Python language facilities allow us to inspect variables if necessary. The

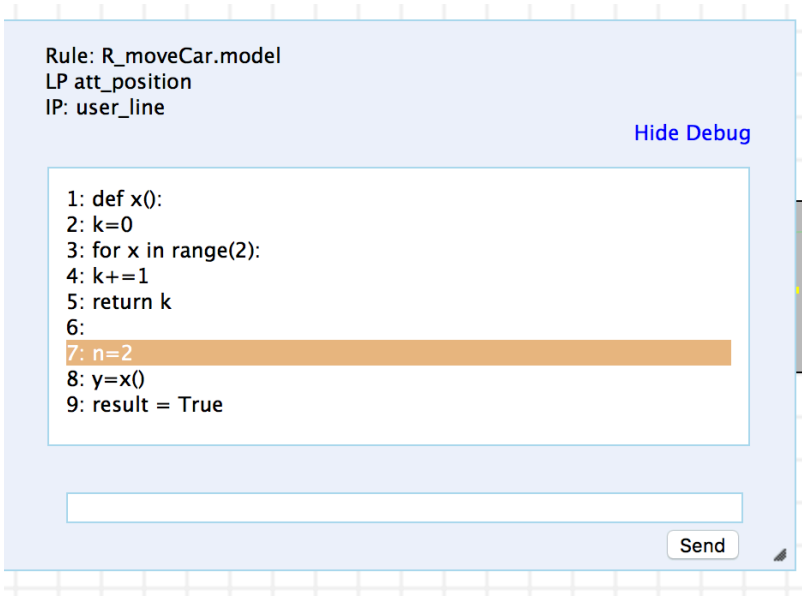

Figure 5–20: A screenshot of our tool and processing of action code.

newly added debugging window displayed in Figure 5–20 is also used to display the *IP* and *LP* comprising the *NP*. In a similar fashion to dealing with AC, we envision dealing with MTs specified entirely in AC (a loop calling a function representing the MT rule for example).

In reusing an AC's built-in debugging API our solution to debugging AC is expedient, but also not entirely seamless with our overall approach. With a simpler, standardized AC design more structured views may be possible, such as by, which would let us expose deeper levels in terms of *NP* values. We now move on to a discussion on our experience in porting our debugger implementation to other MT tools.

5.6.2 Porting Implementation

In order to give validity to our debugger solution beginning from the unified, structured view to the *Statecharts* controller, we port the implementation to two additional MT tools. One of the two is the industry standard ATL tool described in Section 2.5. Another tool $AToM^3$, is a precursor tool to AToMPM. These ports realize interactive debuggers without the debugging scenario support, which would require language engineering efforts beyond the scope of this thesis.

AToM^3 is a MT tool. Compared to AToMPM, it has a different MT language, implementation, and user interface. Both tools, however, share the Python-based implementation language. Therefore, it was easy to reuse the *Statecharts* controller in $AToM^3$. At the beginning, we modify the *Statecharts* state responsible for domain specific actions, such as the action code highlighting, to adopt the new tool's user interface specifics. Other states, for runtime context and *NP* processing, remain largely unchanged. Similar to AToMPM, we identify the locations in the tool's implementation that treat the items of interest, such as rules, rule parts, and action code. In those spots, we communicate with the *Statecharts* controller, signaling the debugging items and pausing the debugging target execution if necessary.

We modify the MT controller window adding new buttons issuing debugger commands. We also add a text field area, in order to display some state of the debugging target. In Figure 5–21 we show a screenshot of $ATOM^3$ debugging a MT simulating Petri Net execution. In this figure, we demonstrate line by line debugging

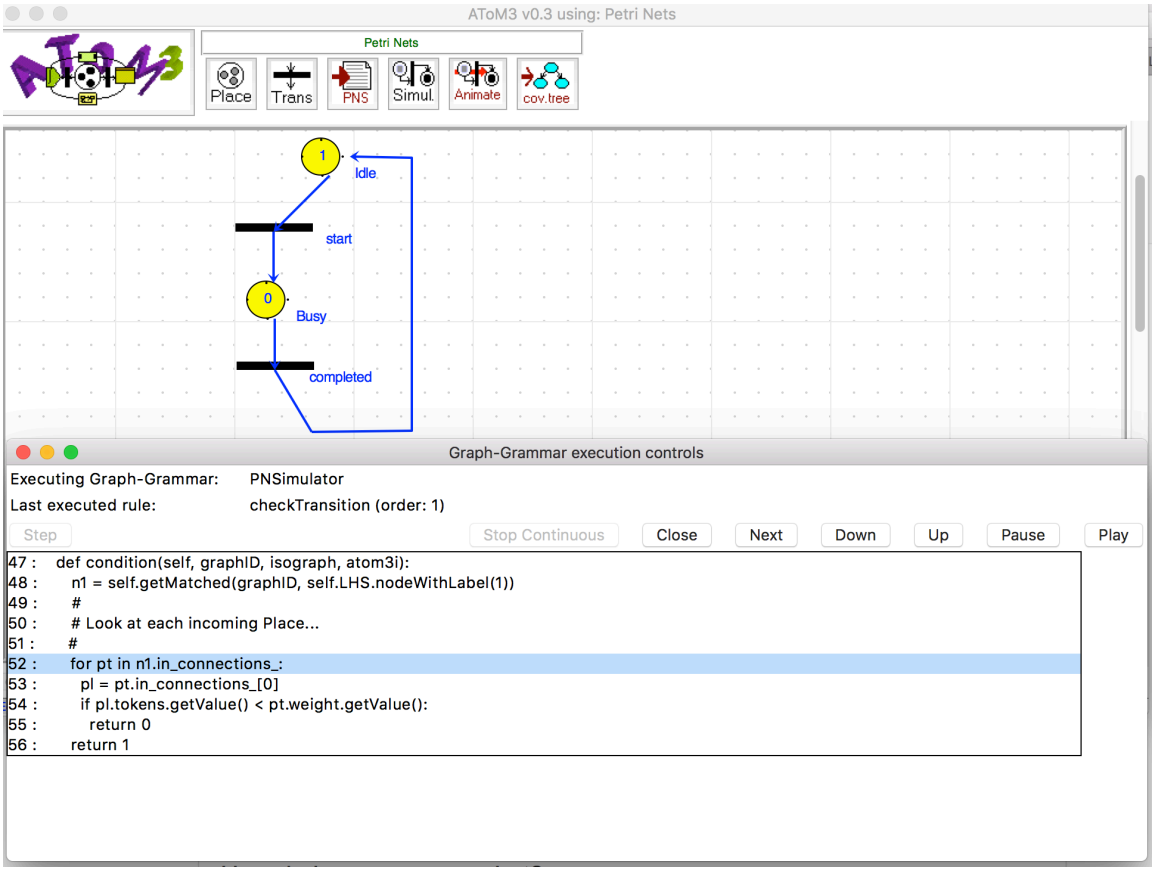

Figure 5–21: A screenshot of $ATOM^3$ tool debugging action code.

of action code similar to the one we showed in AToMPM. Porting the action code treatment was relatively easy because both tools use Python as their AC and we use the same approach.

ATL tool was chosen with an aim to demonstrate the applicability of our concept to a widely used MT tool. The situation is different with ATL and several complications hindered a seamless porting of our solution.

ATL is an Eclipse[7] and Java-based tool. Because the *Statecharts* compiler used initially does not produce Java code, we reimplemented the controller model in another tool, *Yakindu*[8] *Statecharts* tool. Yakindu does not support embedding of custom executable code within the *Statecharts* model. Instead, external functions need to be implemented. This results in an additional implementation effort because of a widened cognitive gap when modeling the controller.

Instead of interpreting the transformation specification, ATL compiles them into bytecode, suitable for execution in its own virtual machine (VM). In porting our solution to ATL, we thus need to work at the level of VM. This certainly increases the complexity of identifying items of interest and the *NP* values coming out of the debugging target as these are based on different, custom bytecode sequences and API entry points. The MT language of ATL is also textual, with a mix of declarative and imperative constructs. In debugging such transformations a line-by-line approach, based on stepping through the VM operations, is used by ATL itself. We aim for a similar way to interactive debugging of ATL. Note that is is not our goal to implement a fully featured ATL debugger or build a graphical user interface environment, but rather to sufficiently demonstrate the applicability of our concept and approach.

---

[7] `http://www.eclipse.org/`

[8] `https://www.itemis.com/en/yakindu/state-machine/`

We know (from the ATL documentation and VM investigation) that there is a certain structure to the ATL rule application inside the VM. It consists of several nested functions that are applicable to each rule execution. The main function is the top level routine containing three-stage rule application: helper function initialization, rule matching, and rewriting. These main functions themselves consist of many other functions and VM operations (such as stack load operation), perhaps too low-level to be considered of interest in a typical debugging use case. We approach this situation explicitly, by providing the debugger and the controller with as fine-grain items as possible. Certainly, with more implementation effort and investigation, we can filter our the low-level VM operations and concentrate on the high-level ones only or more appropriately integrate low-level operations into our multi-level navigation approach. We now move to the *Statecharts* controller implementation.

In Figure 5–22 we demonstrate the *Statecharts* controller model. The model

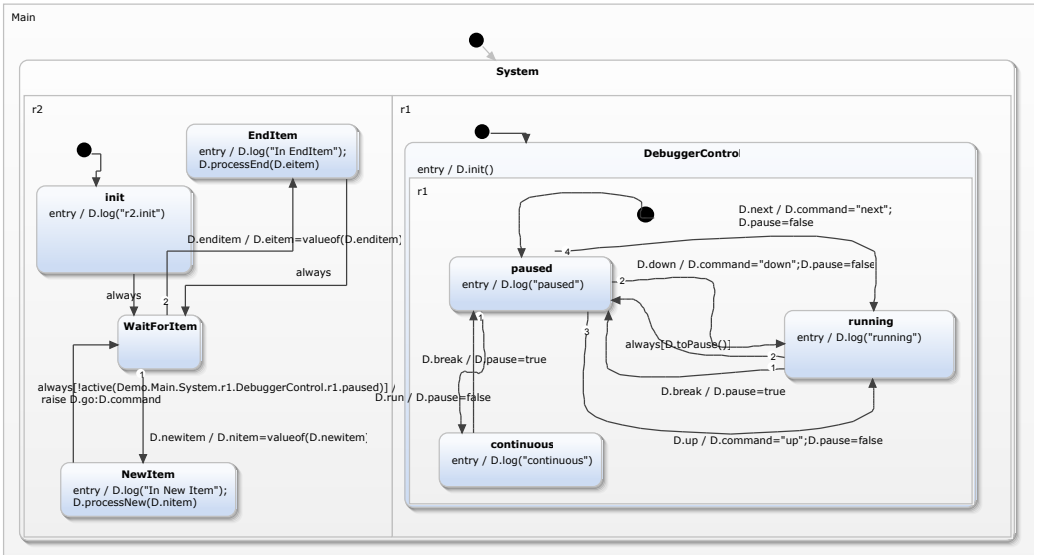

Figure 5–22: A *Statecharts* controller for ATL debugger.

contains all main components from the *Statecharts* model in AToMPM except the domain-specific state to handle extra information from the *NP* and perform custom user interface actions. This *Statecharts* model, however, allows for interactive control of ATL program execution, with stepping based on the *NP* generated from within the VM handling individual VM operations. The controller receives the navigation commands from the Eclipse toolbar in the ATL perspective.

We show a screenshot of the ATL tool with our debugging toolbar added in Figure 5–23. The toolbar buttons have the same behavior as described in the AToMPM

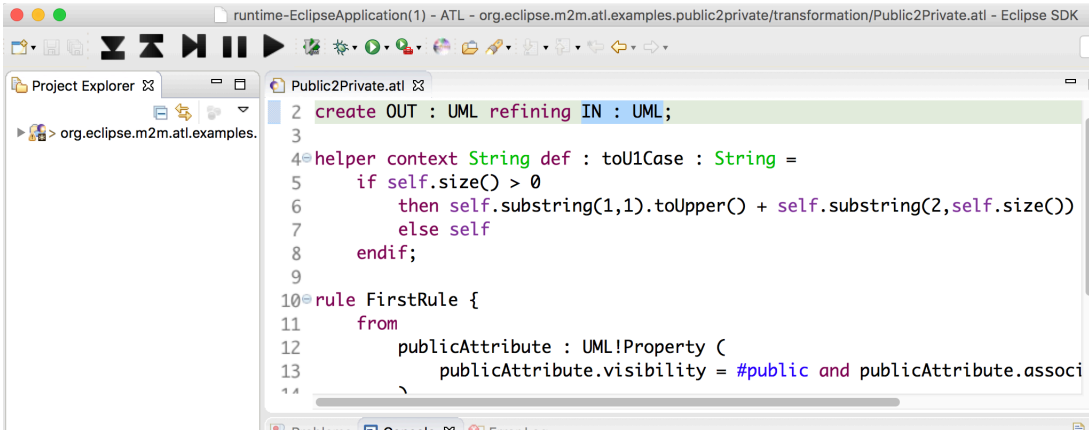

Figure 5–23: A screenshot of ATL debugger. Note a debugging toolbar similar to the one used in AToMPM.

implementation. We conclude this section by discussing performance implications of our debugging approach.

5.6.3 Efficiency Considerations

It is important for a debugger to have as little overhead as possible. In case of an interactive debugger we desire responsiveness, and in automated contexts, we are concerned with the increase in total runtime.

It is evident that the context switching from the MT to the debugger MT will impact performance. In our prototype design, each implicit navigation pointer change waits for a breakpoint evaluation, and this can become expensive depending on the number of breakpoints and their complexity. Additionally, and more generally, query evaluation in debugger rules is based on the same computationally expensive pattern-matching problem as in general MTs.

To better understance performance, we executed the ALAP MT from Section 3.4.1 in AToMPM, estimating the load of *NP* creation and communication with the *Statecharts* controller. The transformation was run with and without debugging enabled. More specifically, when debugging is enabled, the tool communicates *NP*s from all specified points in the target and the *Statecharts* controller processes them. We disable the action code *NP*s from being created in this test and concentrate on the MT schedule treatment consisting of transformations, rules, and their respective parts. From this we discovered that the implicit pointer evolution approximately triples the total runtime of the MT in question. Although an order-of-magnitude time increase is not unreasonable for a debugger, there is clearly a substatial performance impact.

A comprehensive performance evaluation is well beyond the scope of the basic design we introduce here (or our initial prototype). Different optimizations, however, are clearly possible that can mitigate many efficiency concerns. We notice, for instance, that while we may conceptually be checking rules at a fine granularity, debugger rules that are based on input model queries need only be actually re-verified after MT rule application (within/after CRUD operations), and even then only for

MTs which have potential to affect the debugger query results. Our use of scope offers another means of performance improvement. We have previously discussed the manual use of scopes to improve performance (end of Section 5.4). This process can be partly automated, either through (conservative) analysis or at runtime through the use of dynamic scopes [47]. As our debugging rules are fundamentally MT rules, many of the other MT optimizations also apply. Incremental pattern matching techniques [10], for instance, could improve complex queries, by enabling partial matches to be cached.

The particularities of our *Statecharts* controller implementation may also open avenues for exploring different controller models resulting in improved runtime performance. Different designs are possible, and custom solutions without the use of *Statecharts*-generated implementation code are also viable options to consider. This is motivated by the fact that the generated, executable *Statecharts* model itself is open to various optimizations. This situation is similar to compiler optimisation.

Finally, modern textual debuggers may benefit from hardware support for code and data breakpoints to improve performance. Our system is missing such facilities but we can imagine the performance benefits such support can provide in MT debugging contexts.

5.7 Conclusions and Future Work

In this chapter, we explored the design of a model transformation debugger based on model transformations themselves. The debugger allows for the specification of debugging scenarios using the syntax and semantics of MTs. This reduces the learning curve as the user is operating within the familiar domain of MTs. The

existing MT formalisms allow us to reuse, without any additional implementation effort, the DSL found in the target MT specification. We use the DSLs to define debugging rules with declarative queries, which facilitate the discovery of complex MT execution artifacts and allow us to reason about the MT specification as in higher-order transformations. The advantage of a debugging scenario, just like the MT itself, is that it can be left to run unattended and perform the desired tasks. This is particularly useful in considering model transformations with significant execution times where close interactivity is less desired, and also applies to cases where a combination of declarative queries and actions is part of an automated investigation.

A modeled solution to debugging has other benefits as well. Debugging scenarios can be exchanged between engineers, reused and analyzed, facilitating the repetition of investigation common to practical debugging. We envision that debugging scenarios can also be used to perform MT testing. In such case, the debugging transformation can be tailored to test at runtime alternative MTs.

Our approach was further based on a structured view over the general debugging process. This allows us to bring clarity to the notion of a step in the declarative context of model transformations. Our non-trivial prototype implementation, based on this view, allows us to evaluate the feasibility of our debugger. Many of the manual steps necessary to instrument the MT engine in order to support navigation pointers, commands, and pause/resume functionality could, however, be automated with the help of a suitable, and simple interface. This would enable us to explore means of generating such debuggers and applying the solution to other, existing MT tools with reduced development efforts. We do however evaluate our concept and

approach on two relevant MT tools AToM^3 and ATL. We discover that our structured view is portable to interactive debugger implementations and observe a substantial degree of *Statecharts* controller reuse in AToM^3. However, in case of ATL, we had to repeat certain *Statecharts* development activities because our *Statecharts* compiler did not support Java code generation.

For future work, we look to address the performance evaluation of debugging scenarios. We have tried to ensure our prototype is sufficiently responsive, but monitoring breakpoints necessarily slows down execution, which can be a concern when MT execution is time-sensitive, such as when processing real-time input or in cyber-physical systems. We expect, however, that the ability to debug model transformations in a way presented in this paper may outweigh the runtime effects on the whole system.

Finally, another important aspect of designing a debugger is verifying that the choices we made, including the proposed formalisms and granularity of the debugging steps are suitable for actual engineers and the debugging problems they encounter. Lack of a test or benchmarking test suite is a concern in text-based debugger designs, and we inherit that limitation here. Practically, of course debugger evaluation is best achieved with a user study. The intent of such study, would be to combine the feedback from multiple users involved at various stages of MDE and use it to improve the debugger functionality and capabilities.

Chapter 6
Related Work

This thesis discusses topics with broad underlying theories and technologies. Below is relevant related work grouped into two topics. We begin with a section on scopes and move to a section related to debugging in MTs.

6.1 Scope

Our focus on scope in this work explores an aspect of graph transformation that has not been deeply investigated in the past. Formal models of scope do exist [2], but the majority of scope applications in model transformation contexts are aimed towards using rule applications as the scope of subsequent rule productions, rather than incorporating scope directly into the host graph. In the graph rewriting community, rule-based scope is a variation on amalgamated rules [98, 83, 13]. This is demonstrated, for example, in *GXL*—a graph transformation language with rule-based scoping and graph parameters [84]. GXL inherits greatly from TXL, a tree transformation language [20], but operates on graphs rather than on trees. Scoping in GXL means that a scope produced by one rule application can be passed by value and used by other rules, and so on. To ensure unambiguous host graph segmentation into subscopes, selection of a match for a rewrite out of multiple available matches in GXL must be deterministic. Our extension to the transformation system does not impose a match selection strategy. In addition, we create scope hierarchies that can be transformed.

Scope in the host graph is most typically approached in terms of the natural structure of the host graph domain. A subtree of an abstract syntax tree (AST), for instance, defines scope in term rewriting systems. *Stratego/XT* [15], a program transformation, term rewriting language and a collection of tools, allows for scoping of dynamic rewrite rules by limiting their lifetime to a specific rewriting strategy, localizing application of a rewrite rule to a part of a program's AST.

A somewhat similar approach is taken by *MGS*, a domain specific language (DSL) aimed at simulating biological systems [34]. MGS was designed to express and manipulate local transformations of entities structured by abstract topologies. A set of entities organized by an abstract topology is called a topological collection, meaning that each collection type defines a neighborhood relationship of locality and subcollections as well. Transformation in MGS involves an identification of subcollections, followed by its rewriting and insertion back into the host collection. MGS explores the neighborhood relationships of collection types to define subcollections within the host collection. Both Stratego and MGS exploit the natural hierarchy of an underlying model. In contrast, our static scope approach is applicable to graphs irrespective of the hierarchy of the underlying model.

In the context of dynamic scope, topological activity [74, 80] computation in *MGS* explores the active and inactive regions in the model. Active regions are exclusively used to find the pattern matches. Nodes that are hot, in our application, can be considered an activity region in *MGS* language terms. However, our approach takes a temperature node coloring approach instead of computing active regions based on system state evolution. In addition, nodes in our concept "cool down" at

a specified rate, whereas in the MGS case topological areas become inactive at the next iteration.

A more flexible representation of scope is found in other existing systems. The standard *QVT-Operational* (QVT-O) [76] language, for instance, provides a feature that can also be used to implement a scoping mechanism, and indeed demonstrates how we can map our design into an existing transformation system. In QVT-O a transformation may define *intermediate properties* (with simple or complex types) in the context of the transformation itself (e.g., Transformation1::scope1) or a given metaclass referenced by it (e.g., Class1::scope1), in which case it dynamically gets added to the metaclass. Such properties can be used to dynamically define scopes for the model elements being transformed. The value of such a property can be expressed in Object Constraint Language (OCL)[1] . In addition, intermediate property values can also be cached and reused, potentially resulting in performance gains. Our approach here provides a less *ad hoc,* more formal integration of scope, directly exposing scope in the rule design, constraining the representation with an eye to efficiency, and making it an integral to the matching process. This allows the matching engine to more easily take advantage of the scope concept.

Scoping properties are also found in container-based approaches, where event-driven grammars have also been defined to manipulate the associated spatial relationships [37]. A container housing several elements, for example, can be considered a scope over the enclosed elements. Mechanisms that ensure the containment (or

[1] `http://www.omg.org/spec/OCL/`

association) relationship is maintained when the container is moved could then be repurposed to automatically maintain scoping relationships, and so provide similar functionality.

Our interest in scope was originally driven by a desire to improve the performance of graph transformation by reducing the size of the potential match set. Other techniques have also been applied to this problem. The idea of *pivots,* [1] for example, is to exploit the fact that subsequent rule matches may have dependencies that reduce the number of candidates. Initial partial matches (pivots) can be passed as parameters to the matching algorithm which then performs localized matching starting from and around the pivots. The approach of using pivots in model transformations has been implemented in a number of modeling tools; in AToM^3 [25], pivots are passed between transformation rules, and similarly, the tool GReAT [1] performs localized searches in the host graph using pivots, called *pivoted pattern matching*. T-Core, a collection of transformation primitives [94], also supports pivots. T-Core operates on graphs encapsulated in packets and pivots can be added to these packets. The packets are then exchanged between the matching and rewriting transformation primitives. An important difference between scope and pivots is that pivots are assumed to induce a valid binding produced from the previous rule application, while scope may not necessarily contain valid bindings, due to the heterogeneous concept of scope as a "bag" of host graph elements. We view scopes as complementary to pivots, providing another way to reduce the search space for graph pattern matching.

Other techniques attempt to prioritize parts of the matching process so as to reduce cost in practice. The high-level, multi-paradigm language *PROGRES* [87],

for instance, employs the technique of discarding graph pattern match candidates as early as possible. Restriction and attribute verifications are given priority, which, along with attribute indexing, improves efficiency. In our design, we prioritize scope verification on the host graph to achieve a similar result.

Incremental bidirectional model transformations is another area where efficiency of pattern matching is important [53]. For example, in [35] the authors avoid matching in the whole input model by keeping track of the triple graph grammar correspondence nodes. Exploring application of scope to bidirectional MTs is an interesting topic for future work.

A very fast pattern matching technique is based on incremental pattern matching, as discussed in [104, 100], and notably used in the VIATRA tool [7]. In essence, the incremental pattern matchers cache the matches as the input model is "consumed" during a warmup phase. Subsequent changes to the model are propagated to the engine and the matches are accordingly updated. This technique delivers matches extremely fast at the expense of memory and match update costs (when model changes are frequent). Therefore it is beneficial to use local search-based techniques, described in the next paragraph, when memory is at a premium or a MT performs frequent updates. For such cases, an adaptive approach switching between incremental and search plan-based matching is presented in [9].

Generation of model dependent search plans from patterns was presented by Varro *et al.* [107], with GrGen used to demonstrate implementation [33]. A dynamic programming-based, generalized search plan algorithm was presented in [106]. Search plans are an efficient way to match graph patterns as they incorporate a fail-first

matching strategy and heuristics to prioritize match operations. Match operations are given weights based on heuristics; in the simplest case, weights can be based on statistical information about the host graph, e.g., number of nodes and edges of a particular type. Operations are then sorted and executed, such that more expensive operations that can result in a large number of match candidates are executed after less expensive operations, where the number of candidates is as small as possible.

The search plan can be constructed before execution. An example of such functionality can be found in various tools [43, 115, 30, 68]. Dynamic search plan creation is demonstrated in [36, 19], and can also be selected *adaptively* from preprocessed search plan candidates [107, 105].

Heuristics used for constructing efficient search plans rely upon different sources of information. *Metamodel-specific* heuristics like [115, 30, 36, 45, 3] exploit the containment and cardinality constraints of a metamodel e.g. by navigating first along edges with at most one multiplicity. *Model-specific search plans* [107, 33, 105] evaluate statistical information of the underlying instance model (e.g. the number of edges of a certain type) to start the search from promising candidate nodes. In fact, initial bindings can be explicitly provided to the pattern matching process by *pivot* (or input) nodes as in [30, 43, 110, 105].

GrGen also provides facilities for scope implementation. The use of containers, such as sets and dictionaries in rules, constrains the search effort to a selection of the input model, and model attribute indices can also be used to filter the search space. The GrGen manual suggests using additional edges in the model to guide the search and thus improve the performance. These so-called *reflexive* edges are

well suited to the implementation of our scope concept. The SP-based techniques mentioned in this paragraph can be used in our approach without modification to the SP algorithm, as we demonstrated with GrGen in Section 3.4. This is possible due to the fact that the patterns augmented with scope are treated as regular patterns. However, in order to harness scope performance benefits the SP algorithm should be either model sensitive, or allow prioritization of bindings, such as in GrGen through the *prio* flags.

In the context of search plans our use of static scope aims at reducing the number of match candidates for selected match operations: scope information is used to produce a search plan with reduced costs.

Our dynamic scope technique is unique and complementary to the local search-based approaches in the sense that it aims to exploit the *transformation process* as heuristics by reducing the scope of the candidates to those elements touched by recent transformation rules. Our approach is also complementary (and thus applicable) to both *compiled* [103, 30, 5] and *interpreted* [3, 45, 49, 103] model transformation approaches. Finally, our scope can be viewed as way of caching relevant parts of the model during transformation. Caching is common in various programming contexts and is a way to reduce computation and facilitate resource access [90].

6.2 Model Transformation Debugging

In this section, we outline related work in the area of MT debugging and debugging in general. In particular, we are interested in approaches to automated debugging, originally applied to general programming languages. We first discuss the topics related to general debugging, followed by MT debugging related work.

General debugging. A variety of approaches exist for framing debugging designs. A prominent approach is to follow an *event-based* view of a debugging process, where a debuggee produces a stream of events that a debugger analyzes. Events can be inspected directly, combined to produce domain-specific events [17] or fully recorded for the purpose of forensic debugging [41]. Computation over time ordered event traces is presented in [4]. We have not focused specifically on events, but our work does rely on the implicit events exchanged between MT target and the debugging host. These events drive the debugging process automatically until the process is interrupted by a user or a breakpoint.

Another fairly generic approach is found in the moldable debugger (MD) [17]. This a flexible approach based on several primitive debugging events resulting from a debugging target operations. These include reading, writing attributes, method calls, sending messages and checking object state. These events are combined to produce domain-specific events/operations. It is a powerful approach that can be used to implement the event-based part of our debugger, and our view of a debugging language also builds on primitive debugging events. We, however, take a more structure-focused perspective, considering debugging as a process of navigating execution within and across hierarchical levels, and integrating that into automated debugging scenarios with declarative queries.

Query-based debugging [56, 79, 55] plays an important part in this paper. First used in general purpose language debugging, it allows the user to explore the program state based on explicitly constructed queries. *Whyline* debugger [48] on the other hand proposes to user the queries that can be applied to the recorded program

execution history. In this thesis, we employ declarative queries as natural in the MT context to explore relevant input/output models (including the MT models being debugged), as well as variables and data structures pertaining to underlying implementation details. Generic query designs can, of course, bring performance concerns. EMF-IncQuery tool [16] performs efficient, incremental declarative query evaluations and is used for MT verification. We can potentially utilize similar incremental pattern matchings to improve the performance of our query evaluation.

The GDL debugging language [21] was a significant source of inspiration for our work. GDL defines debugging operations that are embedded into a general programming language, and complex debugging scenarios can then be specified programmatically. Similarly, we embed the MT debugging language elements into the MT language. This allows us to use branch and loop constructs and create user-specified transformations for the purpose of debugging. Another example of debugging automation is the scriptable debugging tool *MzTake* [62, 61]. This tool, for debugging Java programs, operates on a stream of events and permits the creation of a variety of higher abstraction applications from primitive ones. Our approach aims at creating an executable model akin to a MT that works with complete MT stack from schedule down to pattern matcher implementation.

MT debugging. MT and model debugging have of course been explored in the past. AToMPM [95], for instance, already supports MT debugging at the level of the MT schedule and down to individual rules with pause, resume, and step functionality, as previously described [60]. In addition, AToMPM supports runtime, manual input model adaptation once the transformation is paused.

MT debugging solutions have also been described in other contexts. Stratego, for example, a textual, imperative transformation language is debugged using an event-based debugger [57]. Fujaba "story diagrams" are debugged at the model level using Eclipse and Java debugging facilities [32]. In this, however, there is no clear distinction between match/rewrite stages, nor is stepping through the pattern matching process at a fine-grained level of match discovery provided. A visual DSL processor debugging for VMTS is presented in [67]. The authors use event handlers to wrap the transformation engine, with visualization addressed by means of various events. Visualization of matches and the modified output model is available, although intermediate steps in the matcher or rewriter components that expose their functionality are not presented. Finally, GrGen also supports visual debugging and input model visualization [33], but again does not support stepping through the pattern matching process.

A stepwise, manual rule-level debugging and manual match selection can be observed in the AGG graph transformation tool [97]. A deeper exploration of rule application is not included. Fine grain debugging is also found in Tefcat [52] and ATL [45], which utilize Eclipse source level debugging. This permits the user to investigate MT through its implementation in terms of very low code level. We aim to raise the level of abstraction with our unifying concepts related to pattern matching and application.

Related work is also found in terms of the more specific elements of a debugger, such as in the process of stepping through the matching/application process. Visual MT debugging, for instance, includes the presentation of matching and execution

views demonstrated in [92]. Detailed stepping through the match process, however, is not available in their design. Tools for debugging of local search-based pattern matching itself have also been created [16]. Their approach allows for stepping through the search plan in its textual form, augmented by visualization of the input model and the match result. Our work here aims at taking a general view of pattern matching process through the use of sets describing that process.

Schoenboeck et al. (2010) present a DSL (based on colored Petri Nets) that explicitly models MTs completely. This allows one to visualize all inner workings of an MT, address non-determinism and use the formalism for formal verification of MT. The use of a single formalism, however, has some disadvantages in being complex to learn, and in terms of its ability to display complex transformations and large models. In [66], the authors describe debugging support in VMTS with a constant debugging step granularity, allowing for exploration of match, rewrite, *etc.* Our approach is more flexible, in that we introduce a variable notion of a debugging "step" depending on the level of debugging (rule granularity at schedule level, rule parts at the rule level). Finally, our approach aims to address potential non-determinism during the debugging of the matching phase, giving the user the ability to control the resulting pattern match binding, and thus closely explore the pattern matching process.

Chapter 7
Conclusions

In this thesis, we explore the avenues addressing model transformation efficiency and usability. Efficiency is still an issue in MTs, as pattern matching is the most expensive operation in rule-based systems. On the other hand, usability concerns, especially in terms of model transformation debugging is an active area of research due to the complexity of MT systems. Below is the summary of contributions made in this thesis.

In Chapter 3 we address scope in MTs. Scoping is a common concept of locality and grouping. In this chapter, we make scope a first-class citizen in MTs. The scope is no longer an abstract concept or an intermediate artifact of computation. The engineer can reason about and manipulate scope in a similar fashion to the way he or she manipulates models. Efficiency is a factor in our design, as is flexibility, and we build on a hierarchical representation, designed to be a part of the input model and intended to be applicable to any DSL used in the transformation. In order to manipulate scope, we extend the MT language to allow for scope hierarchy manipulation in conjunction with the regular input model modifications.

We demonstrate, with a non-trivial evaluation, that our scope concept can bring performance benefits, in particular in the context of SP-based matching. This is due to the fact that the introduction of scope to the input model creates favorable conditions for the SP matching.

Our scope concept is also applicable to scope-unaware systems. In such cases, we can model scope directly as a scoped graph or use the tool/language-specific facilities to simulate scope. For example, direct scope encoding within the graph was proven to be beneficial in the GrGen, already a highly efficient MT tool.

Static scope is intended to be used in MT rules. In Chapter 4, we propose a dynamic scope approach to address MT efficiency in cases when user-modification to MT rules is not available. This can happen when the MT is compiled or part of proprietary information. The dynamic scope technique is intended to discover scopes within the input model that are necessarily smaller than the whole input model. We employ a heatmap-based approach in order to discover areas of activity in the input model. These areas of activity represent scopes of interest and are used for pattern matching. In addition, we incorporate machine learning into model transformation process allowing us to reduce scope areas further. The evaluation of dynamic scope demonstrated a reduction in search space for search plan-based matching and favorable success rates for machine learning application.

In Chapter 5, we address debugging of MTs. First, we take the structured view of a general debugging process and apply it to the MT execution stack. The complexity of the MT stack is due to the hierarchical nature of the stack layers consisting of various formalisms at the different levels of abstraction. Our structured view aims at resolving this issue striving for a consistent debugging throughout MT stack, including debugging of action code.

We define a debugging language which allows for modeling interactions with a debugger. The language is based on the familiar MT rules. The rules when executed,

issue debugging commands, such as stepping, and control the debugging target execution. The rules can also contain declarative queries intended to investigate the state of the debugging target. The static scope formalism is used here to indicate the location of the query application within the debugging target. The reuse of DSLs from the MT being debugged is also beneficial as the engineer stays at the level of models and MTs themselves when debugging. The debugging rules can be combined, by means of rule scheduling, into debugging scenarios. These can encode complex interaction with a debugger and automatically monitor the state of the target program when human involvement is not required or necessary.

Finally, we model the heart of our debugger using the *Statecharts* formalism and investigate the applicability of our approach within three prototype implementations in different MT tools. We discover, that depending on the initial, underlying technology the degree of reuse can be high when porting to similar tools and reduced when porting to a dramatically different system. Regardless of reuse level, our approach seems to apply well to MT debugging.

There was a number of possible future work directions identified and described in each chapter. Here we provide a big picture summary. Certainly, it would be interesting to investigate the evolution of static scope syntax and semantics. To do that, a user study into scope use would provide invaluable input. The goal of the study would give us an evaluation of choices we made in our scope concept and give us insight into the possible scope syntax and semantics adaptations necessary to tailor scope for wider use. In a dynamic scope context, the application of machine learning within the MT process opened interesting opportunities for an array of applications

in MDE in general. We envision the increase of machine learning application in MDE in the near future. The collection of data from users in MT tools can be used for training ML algorithms, and it would be interesting to investigate recent advances in *deep learning* [54] for learning more from MTs.

We would like to investigate the use of MT syntax to define debugging scenarios in other MT contexts. This applies most notably to the MT languages used in other MT tools discussed in this thesis. Again, a user study for evaluating the choices we made here is an important future work direction and could also be used to validate our implicit assumption that a "deep debugging" approach, allowing exploration of the full MT is useful in practice. It is also interesting to explore the use of automated debugging scenarios to specify MT tests. This is motivated by the possibility of applying higher-order transformations to MT models. Another important aspect of MT debugging presented here is performance. The initial performance evaluation, as expected, indicated negative performance impact in our prototypes. The situation is akin to early designs for debugging injective languages, which have benefited tremendously from hardware support. The optimization of our debugging solution we leave for future work as that essentially represents an optimization problem outside the scope of this thesis.

In conclusion, in this thesis, we attempted to address the efficiency and usability of MTs. This is a vast topic to explore and there are certainly more things that can be done within the span of graduate students' theses. We hope that the research presented here finds application in the MDE and the industry in one form or another.

References

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

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems - 13th International Conference (MODELS 2010)*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[4] M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 217–222, 2002.

[5] András Balogh, Gergely Varró, Dániel Varró, and András Pataricza. Compiling model transformations to ejb3-specific transformer plugins. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1288–1295. ACM, 2006.

[6] Gernot Veit Batz. An optimization technique for subgraph matching strategies. Technical report, Universität Karlsruhe, Fakultät für Informatik, April 2006.

[7] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. *Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT, Held as Part of STAF, L'Aquila, Italy.*, chapter VIATRA3: A Reactive Model Transformation Platform, pages 101–110. Springer International Publishing, 2015.

[8] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer Berlin Heidelberg, 2008.

[9] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. *Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT Zurich, Switzerland.*, chapter Efficient Model Transformations by Combining Pattern Matching Strategies, pages 20–34. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 2009.

[10] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the VIATRA model transformation system. In Gabor Karsai and Gabriele Taentzer, editors, *Proc. Graph and Model Transformations (GRAMOT 2008)*. ACM, 2008.

[11] S. Berner, S. Joos, M. Glinz, and M. Arnold. A visualization concept for hierarchical object models. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 225–228, October 1998.

[12] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2011.

[13] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of graph transformations: A synchronization mechanism. *Journal of Computer and System Sciences*, 34(2–3):377 – 408, 1987.

[14] Steven Bosems. A performance analysis of model transformations and tools. Technical report, Enschede, The Netherlands, March 2011.

[15] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, July 2005.

[16] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, pages 275–282, 2015.

[17] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 102–121. Springer International Publishing, 2014.

[18] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, 1971.

[19] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004.

[20] J.R. Cordy, C.D. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Proceedings of the International Conference on Computer Languages*, pages 280 –285, October 1988.

[21] Richard H. Crawford, Ronald A. Olsson, W. Wilson Ho, and Christopher E. Wee. Semantic issues in the design of languages for debugging. *Computer Languages*, 21(1):17–37, April 1995.

[22] Gábor Csárdi and Tamás Nepusz. igraph reference manual, 2012. `http://igraph.sourceforge.net`.

[23] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.

[24] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*, chapter Typed Attributed Graph Transformation with Inheritance, pages 259–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[25] Juan de Lara and Hans L. Vangheluwe. Using AToM^3 as a meta-CASE environment. In *4th International Conference On Enterprise Information Systems*, pages 642–649, 2002.

[26] Jack Edmonds. Optimum Branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240, 1967.

[27] Hartmut Ehrig. *Tutorial introduction to the algebraic approach of graph grammars*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[28] Joost Engelfriet and Grzegorz Rozenberg. *Graph grammars based on node rewriting: an introduction to NLC graph grammars*, pages 12–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[29] M. A. Finney. FARSITE: Fire area simulator – model development and evaluation. *USDA Forest Service Research Paper, RMRS-RP-4 Revised*, 2012.

[30] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In *Proc. Theory and Application to Graph Transformations (TAGT'98)*, volume 1764 of *LNCS*. Springer, 2000. H. Ehrig and G. Engels and H.-J. Kreowski and G. Rozenberg.

[31] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[32] Leif Geiger. Model level debugging with Fujaba. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proceedings of 6th International Fujaba Days, 18-19 Sep 2008*, Dresden, Germany, 2008.

[33] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Graph Transformations - ICGT 2006*, Lecture Notes in Computer Science, pages 383 – 397. Springer, 2006.

[34] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Premyslaw Prusinkiewicz. Computational models for integrative and developmental biology. In *Actes du Colloque Modélisation et simulation de processus biologiques dans le contexte de la génomique*, page 43, Autrans, France, 2002.

[35] Holger Giese, Stephan Hildebrandt, and Leen Lambers. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software & Systems Modeling*, 13(1):273–299, 2012.

[36] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Improved flexibility and scalability by interpreting story diagrams. *Electronic Communications of the EASST*, 18, 2009.

[37] Esther Guerra and Juan de Lara. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software & Systems Modeling*, 6(3):317–347, 2007.

[38] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[39] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 138–153, 1998.

[40] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proceedings of SEGRAGRA Graph Rewriting and Computation, Electronic Notes of TCS*, page 2, 1995.

[41] Mark Hibberd, Michael Lawley, and Kerry Raymond. Forensic debugging of model transformations. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007. Proceedings*, pages 589–604, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[42] Ákos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. Experimental assessment of combining pattern matching strategies with VIATRA2. *The International Journal on Software Tools for Technology Transfer*, 12(3-4):211–230, 2010.

[43] Ákos Horváth, Dániel Varró, and Gergely Varró. Generic search plans for matching advanced graph patterns. *Electronic Communications of the EASST*, 6, 2007. Selected papers of GT-VMT 2007: Graph Transformation and Visual Modelling Techniques 2007.

[44] Phil Hughes. Python and TKinter programming. *Linux Journal*, 2000(77es), September 2000.

[45] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[46] Māris Jukšs, Clark Verbrugge, Maged Elaasar, and Hans Vangheluwe. Scope in model transformations. *Software & Systems Modeling*, pages 1–26, 2016.

[47] Māris Jukšs, Clark Verbrugge, Dániel Varró, and Hans Vangheluwe. Dynamic scope discovery for model transformations. In *Proceedings of the 7th International Conference on Software Language Engineering, SLE 2014, Västerås, Sweden, September 15-16*, pages 302–321, 2014.

[48] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[49] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008. Antonio Vallecillo and Jeff Gray and Alfonso Pierantonio.

[50] P. Kourtz and W.G. O'Regan. A model for a small forest fire...to simulate burned and burning areas for use in a detection model. *Forest Science*, 17:163–169, June 1971.

[51] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[52] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops Doctoral Symposium, Educators Symposium Montego Bay, Jamaica, October 2-7, 2005 Revised Selected Papers*, pages 139–150, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[53] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. In *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 67 of *Electronic Communications of EASST*. European Assoc. of Software Science and Technology, 2014.

[54] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[55] Raimondas Lencevicius. On-the-fly query-based debugging with examples. In *Proceedings Fourth International Workshop on Automated Debugging*, 2000.

[56] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 304–317, New York, NY, USA, 1997. ACM.

[57] Ricky T. Lindeman, Lennart C.L. Kats, and Eelco Visser. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 127–136, New York, NY, USA, 2011. ACM.

[58] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181 – 224, 1993.

[59] Raphaël Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill University, Montreal, Quebec, Canada, 2012.

[60] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 276–285, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss. A dataflow language for scriptable debugging. In *Proceedings 19th International Conference on Automated Software Engineering, 2004.*, pages 218–227, Sept 2004.

[62] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering*, 14(1):59–86, 2007.

[63] M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM*, 7(3):216–244, 1960.

[64] S. J. Mellor, K. Scott, A. Uhl, D. Weise, and R. M. Soley. *MDA distilled: principles of model-driven architecture*, volume 88. Addison-Wesley, 2004.

[65] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, December 1995.

[66] Tamás Mészáros, Péter Fehér, and László Lengyel. Visual debugging support for graph rewriting-based model transformations. In *Proceedings of Eurocon 2013, International Conference on Computer as a Tool, Zagreb, Croatia, July 1-4, 2013*, pages 482–488. IEEE, 2013.

[67] Tamás Mészáros and Tihamér Levendovszky. Visual specification of a DSL processor debugger. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 67–72, Nashville, USA, 2008.

[68] Tamás Mészáros, Gergely Mezei, Tihamer Levendovszky, and Márk Asztalos. Manual and automated performance optimization of model transformation systems. *Software Tools for Technology Transfer*, 12(3-4):231–243, 2010.

[69] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223 – 1246, 2011. Special Issue on Software Evolution, Adaptability and Variability.

[70] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[71] P. J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, September 2004.

[72] Jens Müller. Speeding up graph transformation through automatic concatenation of rewrite rules. Technical report, Universität Karlsruhe, Fakultät für Informatik, 2007.

[73] A. Muzy, J.J. Nutaro, B.P. Zeigler, and P. Coquillard. Modeling and simulation of fire spreading through the activity tracking paradigm. *Ecological Modelling*, 219(1–2):212 – 225, 2008.

[74] Alexandre Muzy, Luc Touraille, Hans Vangheluwe, Olivier Michel, David R.C. Hill, and Mamadou Kaba Traoré. Activity regions in discrete-event systems. In *Symposium On Theory of Modeling and Simulation - DEVS Integrative M&S Symposium (DEVS'10), Spring Simulation Conference*, pages 176–182. Society for Computer Simulation International (SCS), April 2010. Orlando, FL.

[75] Alexandre Muzy, Luc Touraille, Hans Vangheluwe, Olivier Michel, Mamadou Kaba Traoré, and David R. C. Hill. Activity regions for the specification of discrete event systems. In *SpringSim*, pages 136:1–136:7, 2010.

[76] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[77] Wojciech Palacz. Algebraic hierarchical graph transformation. *Journal of Computer and System Sciences*, 68(3):497 – 520, 2004.

[78] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[79] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Proceedings Australian Software Engineering Conference.*, pages 251–259, 2004.

[80] Martin Potier, Antoine Spicher, and Olivier Michel. Topological computation of activity regions. In *Proc. of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 337–342, New York, NY, USA, 2013. ACM.

[81] Marc Provost. Himesis : A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, Montreal, Quebec, Canada, 2005.

[82] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[83] Arend Rensink and Jan-Hendrik Kuperus. Repotting the geraniums: On nested graph transformation rules. *Electronic Communications of the EASST*, 18, 2009.

[84] Medha Shukla Sarkar, Dorothea Blostein, and James R. Cordy. GXL - a graph transformation language with scoping and graph parameters. Technical report, Kingston, Ontario, Canada, 1998.

[85] Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Catch me if you can – debugging support for model transformations. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 5–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[86] Johannes Schönböck. *Testing and Debugging of Model Transformations*. PhD thesis, E188 Institut für Softwaretechnik und Interaktive Systeme, 2012.

[87] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In *5th European Software Engineering Conference*, volume 989, pages 219–234. Springer-Verlag, 1995.

[88] Mirko Seifert and Stefan Katscher. Debugging triple graph grammar-based model transformations. In Uwe Aßmann, Jendrik Johannes, and Albert Zündorf, editors, *Proceedings of 6th International Fujaba Days, 18-19 Sep 2008*, Dresden, Germany, 2008.

[89] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September 2003.

[90] Alan Jay Smith. Cache memories. *ACM Computer Survey*, 14(3):473–530, September 1982.

[91] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, Wien, New York, 1973.

[92] Yu Sun and Jeff Gray. End-user support for debugging demonstration-based model transformation execution. In Pieter Van Gorp, Tom Ritter, and Louis M. Rose, editors, *Proceedings Modelling Foundations and Applications: 9th European Conference, ECMFA 2013, Montpellier, France, July 1-5, 2013*, pages 86–100, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[93] Eugene Syriani. *A Multi-paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, Montreal, Quebec, Canada, 2011.

[94] Eugene Syriani and Hans Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010.

[95] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A web-based modeling environment. In *Demos/Posters/StudentResearch@MoDELS*, pages 21–25. CEUR, 2013.

[96] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 653–669, Cham, 2014. Springer International Publishing.

[97] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 446–453, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[98] Gabriele Taentzer and Martin Beyer. Amalgamated graph transformations and their use for specifying AGG – an algebraic graph grammar system. In *Graph Transformations in Computer Science: International Workshop Dagstuhl Castle, Germany, January 4–8, 1993 Proceedings*, pages 380–394. Springer, 1994.

[99] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, pages 18–33, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[100] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80 – 99, 2015. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[101] Simon Van Mierlo. Explicitly modelling model debugging environments. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, pages 24–29, 2015.

[102] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD : SCCDXML exhended with class diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*, pages 1–6, 2016.

[103] Gergely Varró, Anthony Anjorin, and Andy Schürr. Unification of compiled and interpreter-based pattern matching techniques. In *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349, pages 368–383. Springer, 2012.

[104] Gergely Varró and Frederik Deckwerth. *Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT Budapest, Hungary.*, chapter A Rete Network Construction Algorithm for Incremental Pattern Matching, pages 125–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[105] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for emf models. In Zhenjiang Hu and Juan de Lara, editors, *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic*, volume 7307 of *LNCS*, pages 224–239. Springer, 2012.

[106] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software & Systems Modeling*, 14(2):597–621, 2013.

[107] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electr. Notes Theor. Comput. Sci.*, 152:191–205, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[108] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 79–88, 2005.

[109] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics, March 2005.

[110] Attila Vizhanyo, Aditya Agrawal, and Feng Shi. Towards generation of efficient transformations. In *Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *LNCS*, pages 298–316, Vancouver, Canada, October 2004. Springer-Verlag.

[111] W3C. XML path language (XPath) version 1.0. W3C recommendation, 1999.

[112] Ying Yang and Geoffrey I. Webb. Discretization for naive-Bayes learning: Managing discretization bias and variance. *Machine Learning*, 74(1):39–74, 2009.

[113] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[114] Andreas Zeller. Debugging debugging: ACM SIGSOFT impact paper award keynote. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 263–264, New York, NY, USA, 2009. ACM.

[115] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, 1996.