Scope in Model Transformations

Māris Jukšs[1], Clark Verbrugge[1], Maged Elaasar[2], Hans Vangheluwe[3,1]

[1] {mjukss,clump,hv}@cs.mcgill.ca
School of Computer Science, McGill University
Montréal, Québec, Canada
[2] melaasar@gmail.com
Modelware Solutions
La Cañada Flintridge, California, USA
Department of Systems and Computer Engineering, Carleton University
Ottawa, Ontario, Canada
[3] hans.vangheluwe@uantwerp.be
Department of Mathematics and Computer Science
University of Antwerp, Belgium

Abstract. A notion of hierarchical scope is commonplace in many programmatic systems. 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. Previous work on scope, however, has focused on applying it to rule hierarchies, which reduces the number of matches performed, but not necessarily the cost of finding a single match. In this paper we define and explore a hierarchical scope formalism applied to the input graph, with associated modifications to the transformation rule definition. We then experimentally evaluate the benefits and challenges of our scoped model transformations in the state-of-the-art graph rewriting tool GrGen and our research-oriented, meta-modeling and rule-based model transformation tool AToMPM. We use a non-trivial "fire spreading" simulation transformation taken from distributed simulation community and a mutual exclusion transformation benchmark to demonstrate that integration of scope results in an elegant, intuitive, and efficient way of solving model transformation problems.

Keywords: scope, graph scoping, graph grammar, rule-based model transformations, search plans, efficient pattern matching

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[4] [1], as well as in representing the natural hierarchy or scoping that

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

exists within the underlying problem domain. In the context of model transformation (MT), 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. 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 [2]. 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 work 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 [3], or focused on domain-specific implementations of transformations [4]. 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 our work 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 [5] meta-modeling tool. Performance evaluation is performed on a forest-fire spreading simulation and a distributed mutex benchmark (from a model transformation benchmark suite [6]). In addition we map our scope concept to an efficient and highly optimized transformation tool *GrGen* [7]. 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) [8]. We demonstrate that adding scope to 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 paper is organized as follows. Sections 2 and 3 describe related work and background. Section 4 formally introduces our interpretation of scope and offers a running example. Section 5 investigates the use of scope in rule-based model transformations. Section 6 concentrates on implementation, experimental evaluation and interpretation of results. Section 7 concludes the paper and discusses future work.

2 Related Work

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 [9], 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 [10–12]. This is demonstrated, for example, in *GXL*—a graph transformation language with rule-based scoping and graph parameters [3]. GXL inherits greatly from TXL, a tree transformation language [13], 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* [14], 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 [4]. 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 subcollection, 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 approach is applicable to graphs irrespective of the hierarchy of the underlying model.

A more flexible representation of scope is found in other existing systems. The standard *QVT-Operational* (QVT-O) [15] 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. For example, a transformation using a metaclass *Employee* with a metamodel-defined container property "company" of type *Company* can define an intermediate property "director" of type Employee as a scope property. The value of such a property can be expressed in OCL as the first director in an employee's reporting chain. The property can consequently be used by a rule transforming employees under (in the scope of) a given director by simply referencing the new property (without having to use the possibly complex expression). If the transformation processes the employee model as a source model only (i.e., read only), then such an intermediate property value 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 [16]. A container housing several elements, for example, can be considered a scope over the enclosed elements. Mechanisms that ensure the containment (or 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,* [17] 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 [18], pivots are passed between transformation rules, and similarly, the tool GReAT [17] performs localized searches in the host graph using pivots, called *pivoted pattern matching.* T-Core, a collection of transformation primitives [19], 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* [20], 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 [21]. For example, in [22] the authors avoid matching in the whole input model by keeping track of the triple graph grammar correspondence nodes. Exploring application of scope to the bidirectional MTs is an interesting topic for future work.

A very fast pattern matching technique is based on incremental pattern matching, as discussed in [23, 24], and notably used in the VIATRA tool [25]. 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 an MT performs frequent updates. For such cases, an adaptive approach switching between incremental and search plan-based matching is presented in [26].

Generation of model dependent search plans from patterns was presented by Varro *et al.* [8], with GrGen used to demonstrate implementation [7]. A dynamic programming-based, generalized search plan algorithm was presented in [27]. 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. 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 for 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 will demonstrate with GrGen in Section 6. 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 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. A dynamic scope technique described in [28] complements this paper and aims at reducing the number of candidates for each match operation by discovering the scope or model element grouping automatically at runtime, based on model transformation statistics. In this paper, we formalize the notion of scope as a logically distinct part of the input model and investigate the use of scope as a first class citizen in model transformations.

3 Background

Our work depends on a number of specific technologies and tools. In this section, we give essential background information that is necessary for understanding our work. We start with a description of our tool AToMPM. We conclude this section with the description of search plans, an efficient pattern matching technique and the industrially relevant GrGen tool.

3.1 AToMPM

AToMPM [5], A Tool for Multi-Paradigm Modeling, is a multi-formalism/multi-abstraction meta-modeling and model transformations tool, developed 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.

The graph rewriting capability of AToMPM is powered by T-Core, 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 [29], contained within a Python implementation called Himesis [30]. 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.

3.2 Search Plans

We present a brief overview of the search plan-based graph pattern matching in order to explain the matching of scoped patterns presented later in this paper. In addition, we explain *primitive* match operations composing the SP and discuss the cost of an SP. For in-depth explanation of SPs consult the original works of Batz [31] and Varró et. al [8].

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 paper we assume the treatment of node label conformance is within the primitive operations and disregard attribute verification for brevity.

In Figure 1 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.

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

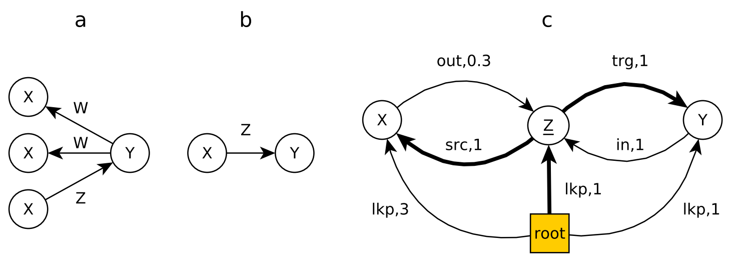

Figure 1. Input graph in column *a* with the pattern to match in column *b*. A corresponding (input sensitive) search graph is shown in column *c*, with its minimum spanning tree indicated by bold edges.

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 1.

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 [7, 8]. Model sensitive search plans are constructed from a weighted search graph representing the match pattern. In column *c* of Figure 1, for example, each edge is weighted with the cost of its operation.

There are several search plans possible for the pattern in Figure 1. 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. 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 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 (and also in Figure 1), 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 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 [32]. The minimum spanning tree is then used to produce the ordered search plan with the smallest cost.

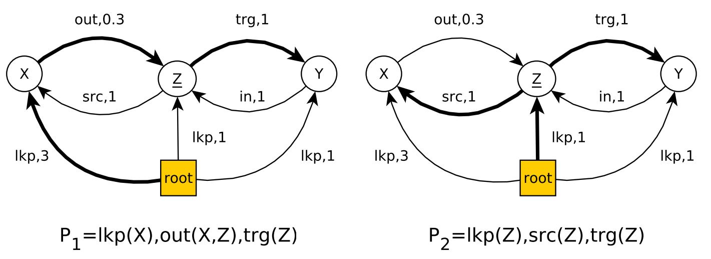

Figure 2. Spanning trees corresponding to the search plans P_1 and P_2.

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 1. 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 paper).

Therefore, the cost of the search plan $P{=}\langle o_1,\ldots,o_k\rangle$ 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.

As demonstrated in Section 5.4, our technique aims at reducing the cost of search plans by introducing early match operations with small branching factor. These operations are then prioritized in an SP and positively affect its overall cost.

3.3 GrGen

GrGen [7] 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 [33].

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. Rules and their execution sequences are specified. Then, the pattern matching backend generates model sensitive search plans.

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 implementation section.

- 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. Parallelization was purposefully omitted from the evaluation as the benchmarks implemented in our AToMPM tool do not execute transformations asynchronously.

4 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 shows a simple example of a *scoped graph* as used in this paper. 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 terms of connected, labeled circles, while the SF is represented using labeled and rounded-rectangles. In Figure 3, 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.

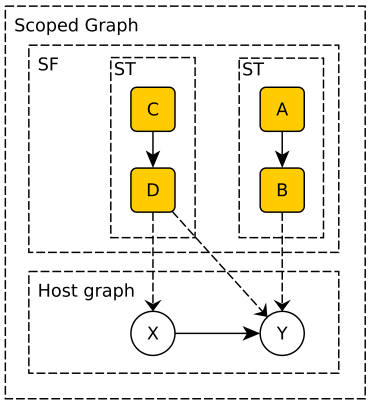

Figure 3. The scoped graph with the scope hierarchy forest (SF) containing two scope hierarchy trees (ST)

4.1 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,
- $L_G : V_G \rightarrow String$ 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,
- $L_S : V_S \rightarrow String$ 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. $\text{path}(s'', s) \wedge \text{path}(s'', s')$.

4.2 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 [34]. This overall geometric approach however is recognized as a standard way of modeling the dynamics of fire spreading scenarios [35]. Additionally, in [6] 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.

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 for 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 4 we thus introduce two STs 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.

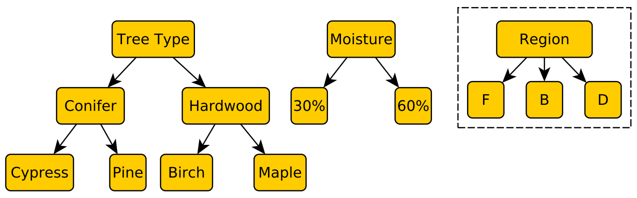

Figure 4. Extended forest-fire scope hierarchy

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 4). 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 5, 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.

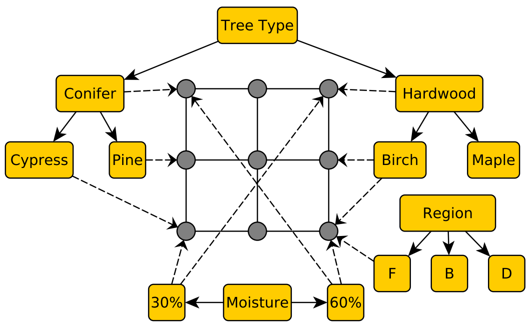

Figure 5. Forest-fire scope hierarchy applied to the forest grid

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.* [36, 37]. 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 [28], the active parts of the transformed model are considered as dynamic scope candidates.

4.3 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 6. This is the actual 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).

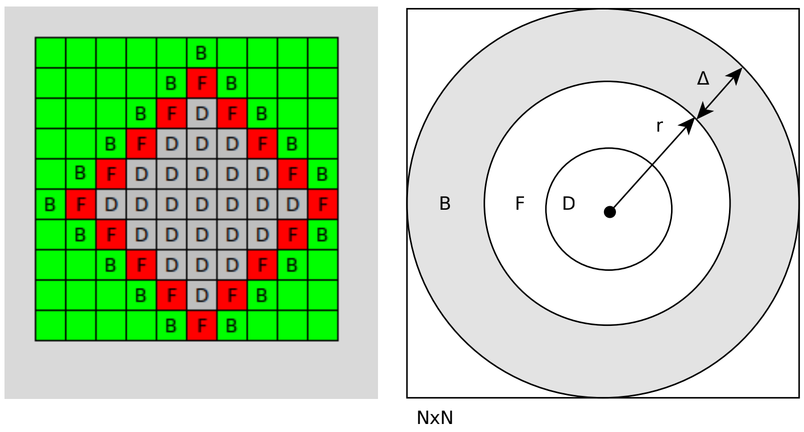

Figure 6. A screenshot of the forest-fire simulation and the model of the forest-fire spreading over the grid

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.

5 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.

5.1 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 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 7 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 [38]), 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.

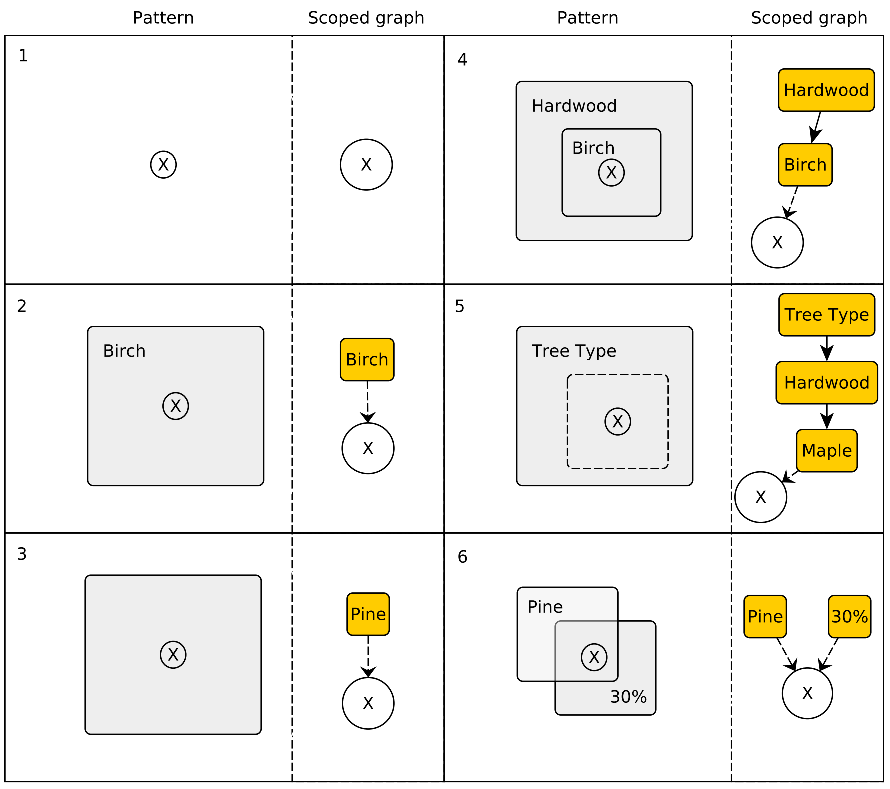

Figure 7. Six core scope patterns, each with the pattern on the left, and a possible matching scoped graph on the right. Panel 2 shows a *labeled scope* pattern, panel 3 contains an *anonymous scope* pattern, and nested and overlapping scope patterns are shown in panels 4 and 6 respectively. A *dashed scope* pattern is shown in panel 5.

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 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 $\textcircled{X}$ is placed inside a single, *labeled scope.* This indicates the innermost scope relationship for node $\textcircled{X}$. Here we match all birch cells in the forest.
- Panel 3. The node $\textcircled{X}$ 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, anonymous scope can also be used to determine the absence of scope for the node it contains.
- Panel 4. The node $\textcircled{X}$ 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 [39], 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 7) and intersection (panel 6 in Figure 7). 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.

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 labelled 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 7, 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 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.

5.2 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* [40] or *regular path query* [41] 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 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.

Table 1. Operators

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

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 $p \in$ SPE to an RE itself is then defined by translating p according to the mapping given in Table 1: $.*(p[A \to .][D \to .*])n$ (an arrow here means replacing construct on the arrow's left to 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 5.4. 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 7 to REs.

- Panel 5 is: $.*$ *Tree Type*$.*X$
- Panel 6 is a combination of two REs: $.*$ *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 8 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 8. 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 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 8 is allowed. The user should be aware of the resulting matches 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.

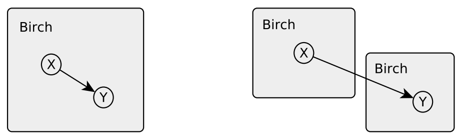

Figure 8. Flattening of a scope pattern on the left, with a result on the right.

5.3 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 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 9. 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 9. 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 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 10 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

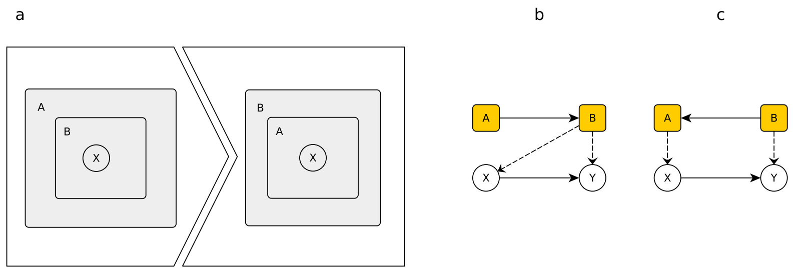

Figure 9. Ambiguity in modifying scope hierarchy in RHS.

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.

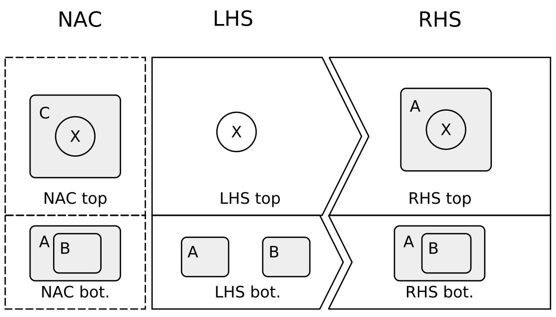

Figure 10. Extended model transformation rule.

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 7 are not allowed in RHS_T if they attempt to modify the scope hierarchy because of the hierarchy manipulation problem described in Figure 9.

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 5.6.

Returning to our example of the ambiguous intent in the transformation shown in Figure 9. We can now use the extended rule structure to more clearly express the transformation engineer's intent. Figure 11 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.

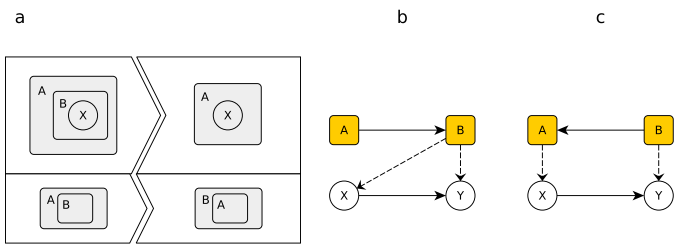

Figure 11. The extended rule applied to the ambiguity problem.

5.4 Semantics

In this section, we give the semantics of our scoped rule application a visual description.

In Figure 12 an extended rule application is shown using notation similar to that used in algebraic single-pushout (SPO) graph rewriting [42] (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. 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 $\textcircled{Y}$ connected to node $\textcircled{X}$. Both top and bottom patterns must have a match in the scoped graph for the rule to apply. That is $m\ top : LHS_T \to G$ and $m\ bot. : LHS_B \to G$ morphisms must exist.

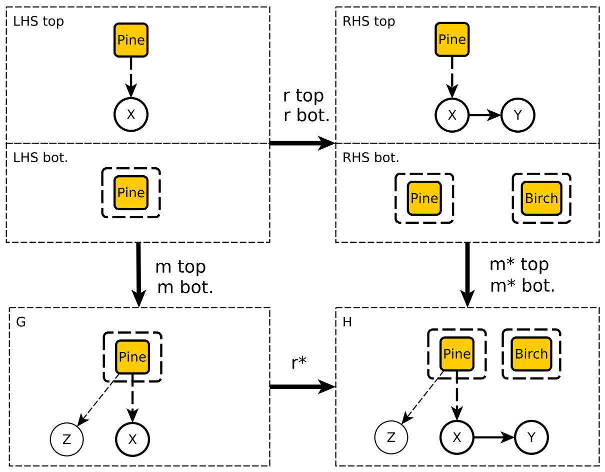

Figure 12. Scoped matching visualized (scoped graph syntax for patterns) using SPO notation.

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 5.6). 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.

5.5 Scope Matching Using Search Plans

We now present scoped matching in the context of search plans (SP). 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* [19]. 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. Some of the fine details of SP-based matching are not in the scope of this paper and are omitted.

To demonstrate how scope augments the SP-based matching we start with a non-scoped pattern and add scope. In Figure 13 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 shown after the comma, and a minimum spanning tree 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.

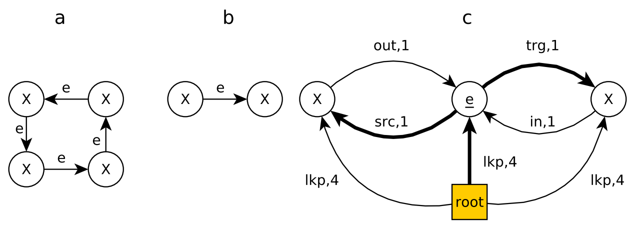

Figure 13. A minimum spanning tree over the search graph in bold edges on the right. The input graph is on the left, and the pattern in the middle.

Consider Figure 14 with the same layout as Figure 13. 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.

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

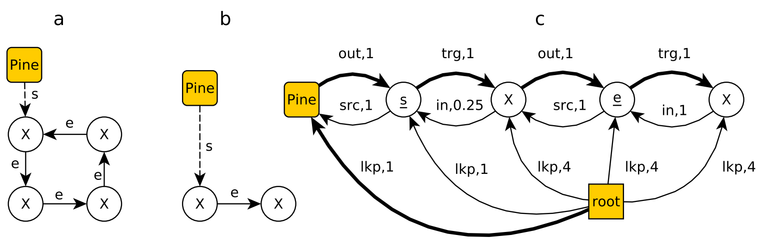

Figure 14. After addition of scope to the input graph and the pattern, the search plan generated from the search graph on the right has a cost of 5 as opposed to 12 without scope.

operations should accommodate returning a binding for any scope node in the *SF*. 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 14. 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 15 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 input graph node $\textcircled{X}$. The *up* operation on the contrary, is used when we match up the scope hierarchy from the input graph node. In Figure 15 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 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

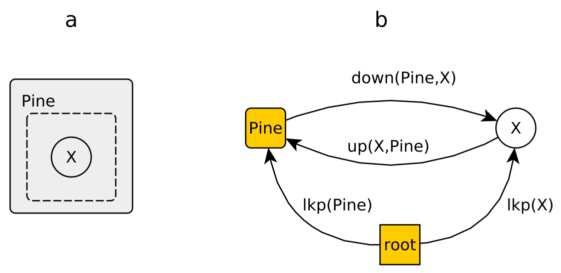

Figure 15. The search graph in column *b* displaying two new match operations targeting dashed scope constructs

semantics. In Figure 15 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 15. 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
        $ScopeEdge = \mathrm{out}(result, S)$
        $result = \mathrm{trg}(ScopeEdge)$
    end while
    return $result$
end function
```

```
function UP($Node, Scope$)
    $result = Node$
    while $result$ != $Scope$ do
        $ScopeEdge = \mathrm{in}(result, S)$
        $result = \mathrm{src}(ScopeEdge)$
    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.

5.6 Rewriting

As mentioned in Section 5.3, 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 paper.

Scope Well-Formedness is addressed next. This 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 [43], 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 [44] 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 absence of cycles. Another way to ensure consistency is by using a tool such as IncQuery [24]. This tool is used successfully in the 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 5.3, 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.

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.

6 Implementation

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 is 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)[5] implementations of the forest-fire and mutual exclusion benchmarks.

6.1 Mutual Exclusion

We wanted to evaluate the use of scope on a well known benchmark in the model transformation community. For this we chose the mutual exclusion benchmark in its as long as possible (ALAP) form, as introduced in [6] with a complete specification presented in [45]. In this benchmark processes are attempting to access shared resources. In order to ensure exclusive access to resources, the processes are interconnected to model a token ring. We use the non-scoped mutual exclusion implementation for AToMPM described in [28] and augment it with scope. On the left in Figure 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 corresponding to a process are linked with *held_by* association. We are concerned with the sequential execution

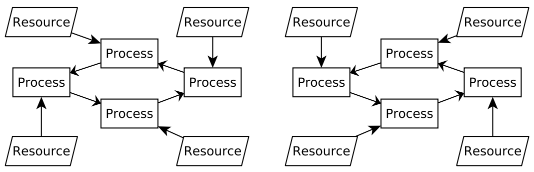

Figure 16. The initial mutual exclusion model on the left and the resulting model after transformation sequence execution on the right. Each resource is moved to the next process in the process ring.

of this benchmark because AToMPM does not support parallel rule execution. We place resource model elements into scopes, 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 places each resource into a single scope in the *SF*. In Figure 17 is the example of the *giveRule* transformation rule with the added scope labeled *Res* (other scoped and non-scoped rules are omitted for brevity).

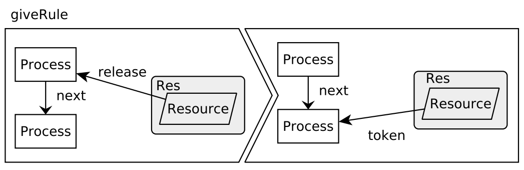

Figure 17. Scoped *giveRule*; resources in scope *Res* are used for matching the pattern in *LHS*.

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

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 paper we added the 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). The STS benchmark differs from the ALAP test in using a single resource.

6.2 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 [6]. There, the MT is executed with the aim of measuring the comb pattern matching performance (the model itself is not modified) over a grid of graph nodes. Note that in this paper 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 18 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

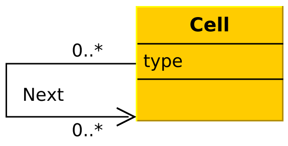

Figure 18. Forest-fire abstract syntax

the forest cell, healthy, on fire, smoldering or dead, is denoted by the integer attribute *type*, as inspired by Muzy *et al.* [37]. 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 22 on the right.

6.2.1 Implementation in AToMPM 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 19. 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 19 and Figure 20 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.

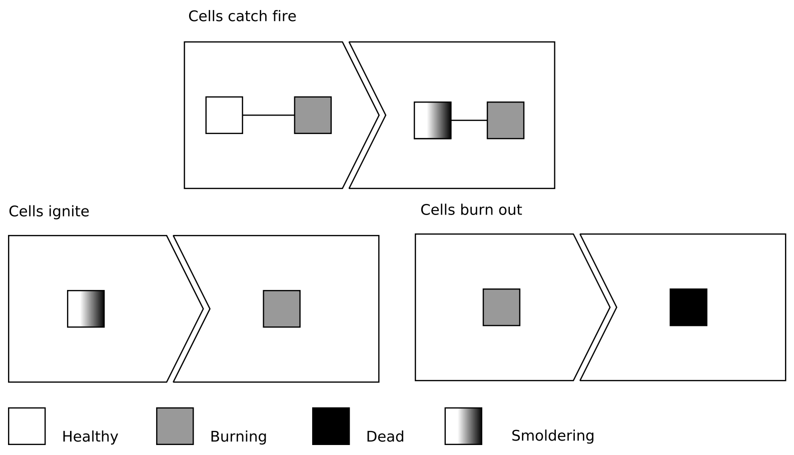

Figure 19. Core rules of the baseline forest-fire simulation shown using AToMPM syntax. Here and in the scoped transformation the cells are colored according to the *type* attribute value.

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 20. There are 4 rules in the core of fire spread simulation, as displayed in Figure 20. The first rule “Init F scope” is intended for 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

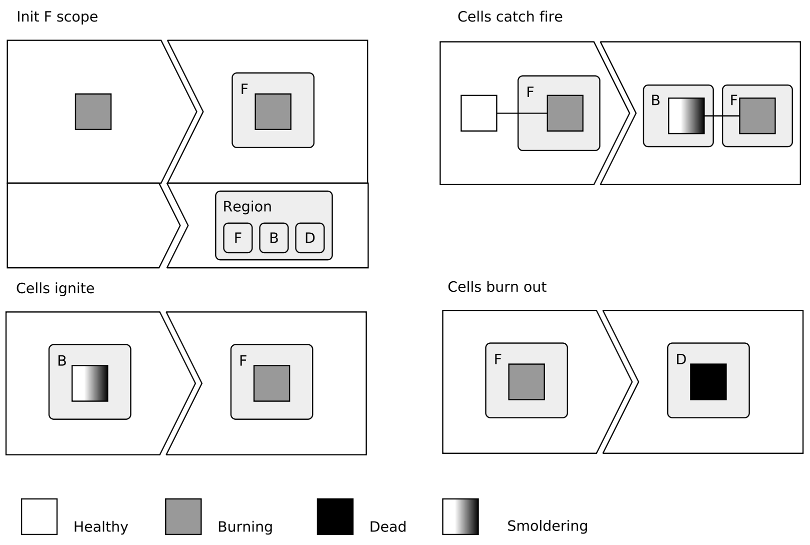

Figure 20. Core rules of scoped forest-fire simulation shown using AToMPM syntax.

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 given in Figure 21 (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).

AToMPM simulation screen shots of the results for both baseline and scoped transformations are shown in Figure 22. We chose to use Tkinter [46] 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.

6.2.2 Implementation in QVT We include excerpts of QVT-Relational (QVT-R) [15] and QVT-O transformations implementing our forest-fire simulation. Class *Cell* represents a single cell in the forest grid. The cell neighborhood relationship is denoted by the *next* property (similar to the *next* association in Figure 18). Property *type* is an enumeration mirroring the state of the cell described in Section 6.4. For simplicity, the transformation's intermediate property *Scope* is defined as a non-hierarchical enumeration of F, B, and D scopes from our running example. However, the intermediate property can be of any complexity and can mimic the abstract syntax of the scope formalism (in Figure 23) allowing for hierarchical scope constructs during transformation. The intermediate property is a good facility to implement scope as it does not exist outside the

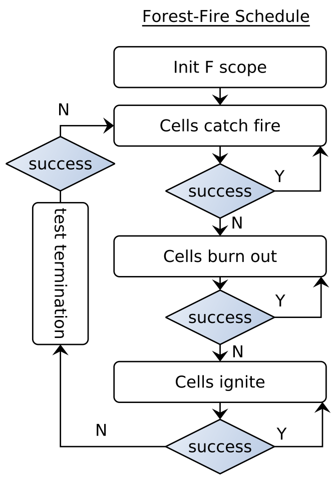

Figure 21. The sequential scheduling of the scoped and non-scoped rules.

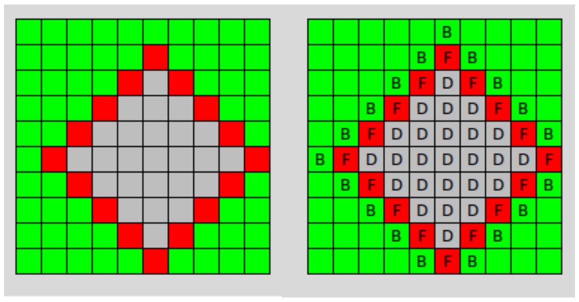

Figure 22. The Result of execution in both baseline (left) and scoped (right) cases in AToMPM simulation: the fire spreads uniformly.

context of the transformation that defines it. This ensures a non-invasive scope application as the MM of the language transformed will remain unchanged.

Below is the QVT-R transformation with the implementation of our "Cells catch fire" rule (in Figure 20). 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
    }
}
```

6.2.3 Implementation in GrGen We also consider the forest-fire simulation for scope evaluation using GrGen. We ported the non-scoped transformation from Figure 19 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 3) 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;}}}
```

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 6.5, the iteration may not always be beneficial.

6.3 Implementation of Scope in AToMPM

A prototype of scoped transformations is implemented in the AToMPM meta-modeling and graph rewriting tool [5]. Our design is intended mainly to establish additional proof of feasibility, and to serve as 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 23. 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 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 [47]. 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.

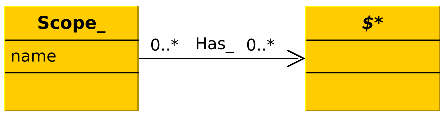

Figure 23. Scope abstract syntax used in AToMPM.

6.4 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 of 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 6, 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 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. 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. The evaluation was performed on an x64 i7 mobile quad-core processor with 16 GB RAM running Ubuntu 12.10.

6.5 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 24 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.

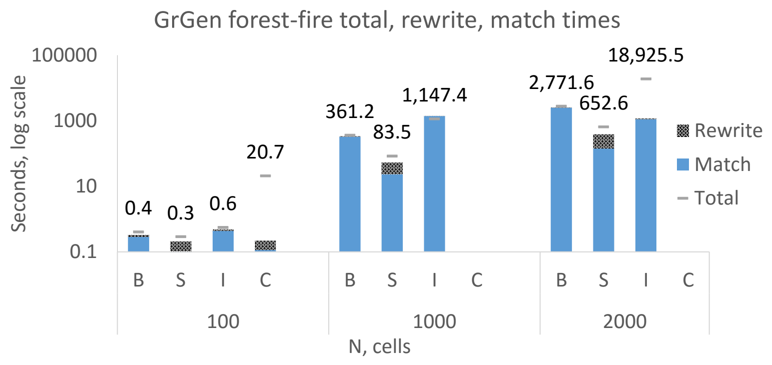

Figure 24. GrGen forest-fire total, rewrite and match times for Baseline (B), Scope Graph (S), Index (I), and Container (C) variations. Note the log-scale in time.

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 in 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.

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 25 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.

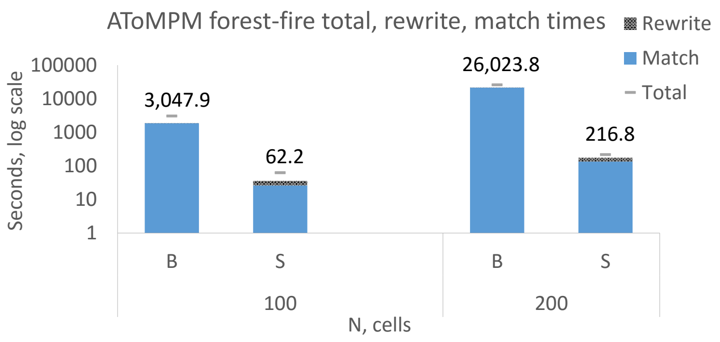

Figure 25. AToMPM forest-fire total, rewrite and match times for Baseline (B) and Scope Graph (S)

The results of the amalgamated forest-fire transformation demonstrate the effects of scope inclusion in GrGen. On average, the baseline amalgamated transformation on a 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. 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 26 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.

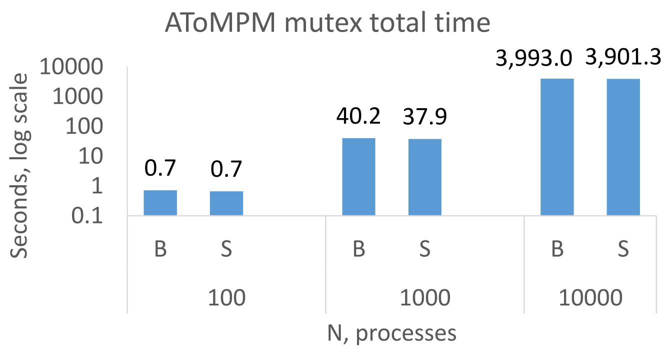

Figure 26. AToMPM ALAP mutual exclusion total time for Baseline (B) and Scope Graph (S)

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.

7 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 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 implements 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 is 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. 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.

References

1. Berner, S., Joos, S., Glinz, M., Arnold, M.: A visualization concept for hierarchical object models. In: Proceedings of the 13th IEEE International Conference on Automated Software Engineering. (October 1998) 225–228
2. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71 (1971) 151–158
3. Sarkar, M.S., Blostein, D., Cordy, J.R.: GXL - a graph transformation language with scoping and graph parameters (1998)

4. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational models for integrative and developmental biology (2002)
5. Mannadiar, R.: A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling. PhD thesis, McGill University (2012)
6. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation (March 2005)
7. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A fast SPO-based graph rewriting tool. In: Graph Transformations - ICGT 2006. Lecture Notes in Computer Science, Springer (2006) 383 – 397 Natal, Brasil.
8. Varró, G., Friedl, K., Varró, D.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. Electr. Notes Theor. Comput. Sci. **152** (2006) 191–205 Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).
9. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
10. Taentzer, G., Beyer, M.: Amalgamated graph transformations and their use for specifying AGG – an algebraic graph grammar system. In: LNCS 776, Springer (1994) 380–394
11. Rensink, A., Kuperus, J.H.: Repotting the geraniums: On nested graph transformation rules. ECEASST **18** (2009)
12. Boehm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations: A synchronization mechanism. Journal of Computer and System Sciences **34**(23) (1987) 377 – 408
13. Cordy, J., Halpern, C., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. In: Proceedings of the International Conference on Computer Languages. (October 1988) 280 –285
14. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundam. Inf. **69**(1-2) (July 2005) 123–178
15. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 (January 2011)
16. Guerra, E., de Lara, J.: Event-driven grammars: relating abstract and concrete levels of visual languages. Software & Systems Modeling **6**(3) (2007) 317–347
17. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. SoSyM **5**(3) (2006) 261–288
18. de Lara, J., Vangheluwe, H.L.: Using AToM^3 as a meta-CASE environment. In: 4th International Conference On Enterprise Information Systems. (2002) 642–649
19. Syriani, E., Vangheluwe, H.: De-/re-constructing model transformation languages. ECEASST **29** (2010)
20. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: 5th European Software Engineering Conference. Volume 989., Springer-Verlag (1995) 219–234
21. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: 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)
22. Giese, H., Hildebrandt, S., Lambers, L.: Bridging the gap between formal semantics and implementation of triple graph grammars. Software & Systems Modeling **13**(1) (2012) 273–299

23. Varró, G., Deckwerth, F.: A Rete Network Construction Algorithm for Incremental Pattern Matching. In: Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT Budapest, Hungary. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 125–140
24. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming **98, Part 1** (2015) 80 – 99 Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).
25. Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Viatra 3: A Reactive Model Transformation Platform. In: Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT, Held as Part of STAF, L'Aquila, Italy. Springer International Publishing, Cham (2015) 101–110
26. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: Efficient Model Transformations by Combining Pattern Matching Strategies. In: Proceedings of the International Conference on Theory and Practice of Model Transformations: ICMT Zurich, Switzerland. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 20–34
27. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An algorithm for generating model-sensitive search plans for pattern matching on EMF models. Software & Systems Modeling **14**(2) (2013) 597–621
28. Jukšs, M., Verbrugge, C., Varró, D., Vangheluwe, H.: Dynamic scope discovery for model transformations. In: Proceedings of the 7th International Conference on Software Language Engineering, SLE 2014, Västerås, Sweden, September 15-16. (2014) 302–321
29. Csárdi, G., Nepusz, T.: igraph reference manual (2012) `http://igraph.sourceforge.net`.
30. Provost, M.: Himesis : A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University (2005)
31. Batz, G.V.: An optimization technique for subgraph matching strategies. Technical report, Universität Karlsruhe, Fakultät für Informatik (April 2006)
32. Edmonds, J.: Optimum Branchings. Journal of Research of the National Bureau of Standards **71B** (1967) 233–240
33. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Graph Transformations. Volume 5214 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 396–410
34. Kourtz, P., O'Regan, W.: A model for a small forest fire...to simulate burned and burning areas for use in a detection model. Forest Science **17** (June 1971) 163–169
35. Finney, M.A.: FARSITE: Fire area simulator model development and evaluation. USDA Forest Service Research Paper, RMRS-RP-4 Revised (2012)
36. Muzy, A., Touraille, L., Vangheluwe, H., Michel, O., Traoré, M.K., Hill, D.R.C.: Activity regions for the specification of discrete event systems. In: SpringSim. (2010) 136:1–136:7
37. Muzy, A., Nutaro, J., Zeigler, B., Coquillard, P.: Modeling and simulation of fire spreading through the activity tracking paradigm. Ecological Modelling **219**(12) (2008) 212 – 225
38. Palacz, W.: Algebraic hierarchical graph transformation. Journal of Computer and System Sciences **68**(3) (2004) 497 – 520

39. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Typed Attributed Graph Transformation with Inheritance. In: Fundamentals of Algebraic Graph Transformation. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 259–281
40. W3C: XML path language (XPath) version 1.0. W3C recommendation (1999)
41. Mendelzon, A.O., Peter, Wood, P.T.: Finding regular simple paths in graph databases (1989)
42. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109**(1) (1993) 181 – 224
43. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars - a constructive approach. In: Proceedings of SEGRAGRA Graph Rewriting and Computation, Electronic Notes of TCS. (1995) 2
44. Biermann, E., Ermel, C., Taentzer, G.: Formal foundation of consistent EMF model transformations by algebraic graph transformation. Software & Systems Modeling **11**(2) (2011) 227–250
45. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: FASE. (1998) 138–153
46. Hughes, P.: Python and TKinter programming. Linux J. **2000**(77es) (September 2000)
47. Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Science of Computer Programming **76**(12) (2011) 1223 – 1246 Special Issue on Software Evolution, Adaptability and Variability.