SUPPORTING MODEL ANIMATION METHODS WITH GRAPH TRANSFORMATION

Ph.D. Thesis Booklet

Tamás Mészáros

Advisors:
Dr. Gergely Mezei, Ph.D.
Dr. Tihamér Levendovszky, Ph.D.

BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
DEPARTMENT OF AUTOMATION AND APPLIED INFORMATICS

Budapest, 2011
Tamás Mészáros

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

1111 Budapest, Magyar Tudósok körútja 2.

e-mail: mesztam@aut.bme.hu

Advisors:
Dr. Gergely Mezei, Ph.D.
Dr. Tihomér Levendovszky, Ph.D.
1 Introduction and Motivation

Modeling has become an essential tool in software engineering recently. With the help of models, we can raise the level of abstraction and improve the reusability of software components at the same time. Visual modeling can also help illustrate software systems in a conceivable way. It is a fact that modeling languages have a great role in the drastic improvement of software complexity in the last decades [Jéz03].

The Unified Modeling Language (UML) [UML10, MH06] has become the de-facto standard to describe software models. It provides languages to define both the static structure of the systems and also their dynamic behavior in a unified way. Due to the fact that UML provides the same set of languages to describe any system, it has many users and serves as a common interface among software engineers. Because of the same reason, however, it is often complicated to use, as the application logic or the components of the application-specific concept are often not that simple to express using general programming languages. The resulting models are often far from the natural language of the domain that they are intended to describe.

In contrast to UML, the other popular modeling approach is Domain-Specific Modeling (DSM) [KT08, SV06]. The essence of DSM is to use specialized modeling languages – called Domain-Specific Modeling Languages (DSMLs) – for each possible application domain instead of using general purpose languages such as UML. A DSML includes both an individual notation for the target domain and the topological constraints between the elements of the language. The notations of a good DSML should follow the notations (either graphical or textual) used by the experts of the target domain. Due to the fact that a DSML is specialized for one domain, it can be extremely efficient [KT08] compared to other solutions. Another advantage of DSM is that the tool support can also be customized for each domain, thus, further increasing productivity. These facts have also been recognized by the leaders of the software industry, and they have also built their own DSM environments [CJKCW07, Gro09]. The high customizability has drawbacks as well: the modeling environment must be built for each and every application domain along with the related generators.

One of the most popular forms to define visual DSMLs is metamodeling [Nor99]. Metamodeling means using a specific DSML to model the elements and constraints of a language. This model is called the metamodel of the language and the models created using this language are referred to as the instances of the metamodel.

Since metamodels are intended to specify only the structural properties (abstract syntax) of the language, other techniques are necessary to define the visual appearance (concrete syntax) of the specific elements. The most basic solution for this issue is to assign an icon to each element somehow, e.g. with an additional configuration file, or storing the path of the related image into the metamodel [LBM+01]. A more advanced approach is to hard-code the visualization of the elements, thus, one can customize and dynamically change the appearance based on the current properties of the model. The Generic Modeling Environment [LBM+01, GME10] also supports this approach using Decorators [MBL07]. In order to avoid hard-coding at least in many practical cases, AToM³ [dLV02] has improved the existing solutions, and facilitated the visual definition of the visualization: the language of their metamodel provides a graphical editor with primitives, and stores the visualization info in the metamodel. The Visual Modeling and Transformation System (VMTS) [VMT10, MLL06] and the Eclipse [GMF10] approaches can be considered the most advanced solutions at the moment: they provide a separate DSML for
the specification of the *concrete syntax* and to assign it to the elements of the abstract syntax.

Of course, domain-specific models are usually built not only with documentation purposes, thus, certain mechanisms are required to process them. Here we usually distinguish two different approaches: traversal-based model processors and graph rewriting-based model processors. Model traversing [VKLL09, VMT10] usually uses the API of the modeling environment to traverse the host model, and it generates some sort of output. This processes is often supported with script or template-based languages [To106, CJKCW07]. The other popular approach is the graph rewriting-based model transformation [EEPT06]. This technique searches for patterns in models represented as graphs, and replaces the found matches with other patterns. Thus, the mathematical model of graph transformation that has been researched since the ’70s [CER79] has found a practical application. Since graph transformation is intended to process models, a straightforward idea is to describe also the steps (rules) of the graph transformation system using modeling. Many tools (e.g. VIATRA [Var04], PROGRES [Züm96] or AToM3 [dLV02]) started to incorporate graph transformation engines with modeling support in the early times as well. Since the early attempts, many specialized kinds of graph transformation approaches have evolved (e.g. Pair Graph Grammars [Pra71] or Triple Graph Grammars [Sch95, Kön05] for model synchronization).

A domain-specific environment can be regarded as a special generative programming environment [CE00] where the input of the generators is a set of models. The most valuable asset in this infrastructure is the domain-specific generator as well as the domain-specific environment. This infrastructure must be built for each application environment separately. Thus, the aim of current research is to facilitate the rapid generation of domain-specific infrastructures – including DSMLs, model processors, and tool support.

Recall that there are numerous approaches to define the abstract and concrete syntax of metamodeled languages. However, there is no integrated solution to describe the dynamic behavior – *animation* – of the languages. As we have shown, the tendency is to solve DSM-related maintenance issues with modeling, and it is an open question whether we can also describe the dynamic behavior of the languages with modeling. Most tools (e.g. AToM3 or AGG [Tae04]) use graph transformation to animate models. However, similar approaches lack the capability to interact with their environment. Graph transformation can be considered a useful utility to describe dynamic behavior as it has been illustrated by international contests [Gra08, Gra09, TTC10] as well.

Since model animations are often demanding, it is important to optimize the transformations as much as possible to save time and resources. A straightforward idea is to achieve some kind of parallelism during the execution of the rules: this parallelism may be either multi-threaded or single-threaded, which means that similar parts of the different rewriting rules are evaluated at once and reused by all the related rules. For both approaches there are numerous related work [MLLC07, Tae96, VV04, Dör95], however, there are other possibilities, for which the optimization is still an open question.

Provided that such a system – incorporating model animation capabilities and a fast transformation engine – exists, designing animations still remains an engineering task. To accelerate this process, the software engineering community has a proven solution: defining patterns and solve recurring problems with the help of them. It is an open question if and how the animation environment can be prepared to support the application of patterns. This thesis presents theoretical and practical answers for the aforementioned questions.
The main topic of the presented research is supporting model animation with model transformation. The open issues related to the topic are as follows:

1. **Dynamic behavior definition of visual languages.** In order to facilitate the animation of visual languages, a fundamental concept should be elaborated, and a set of domain-specific languages that are capable of describing the dynamic behavior of metamodeled languages. The operational semantics of the new languages should be formalized, the expression power of the solution should be analyzed. Furthermore, transformations should be provided that make the automated execution of the animation models possible.

2. **Optimized execution of model transformations.** In order to optimize the performance of control flow-based model transformations, an overlapping technique should be elaborated that exploits the structural similarity of consecutively executed rules. The matching algorithm should perform the matching of the common parts of the different rules at once, thus reducing execution time. It should be examined whether and how the idea could be specialized to the exhaustive execution, i.e. the same rule is applied as long as possible. The semantic preserving property of the algorithms should be ensured. The achieved gains should be verified with measurements.

3. **Supporting model patterns.** The purpose is to support the definition and application of patterns describing often recurring editing operations. Thus, the fundamental concepts that facilitate the precise description of model patterns should be elaborated. The solution should be generic enough to express any possible operation. The approach should be generalized to domain-specific design patterns, i.e. the insertion of incomplete model fragments.

2 Methodological Summary

The outlined open issues have defined the direction of my research. As a formal background, I have investigated the mathematical background of modeling including typed and attributed graphs [EEPT06] and the related graph algorithms. To describe the dynamic behavior of visual languages, I have examined existing model animation approaches and languages. I have elaborated an animation framework and a set of visual languages. To describe the operational semantics of the visual models built with the languages, I used Abstract State Machines (ASMs) [BS03]. I have created model transformations to process the animation models and make them executable. To verify and analyze the properties of the transformations, I used the tools of category theory [Pie91]. To formalize the behavior of the implementation, I used ASM again. To prove the correspondence of the implementation to the operational semantics, I used bisimulation [Mil71].

The theoretical background of model transformations is covered by the mathematical background of graph transformations [Roz97] and the Double Pushout [EEPT06] (DPO) approach. Graph transformations used in my approach are strictly ordered. This strict order means that the transformation defines not only the rewriting steps themselves, but their order of execution as well. Before designing the related algorithms I have investigated existing algorithms for the sub-graph matching problem and their specialization for restricted classes of graphs.
To define model patterns, I used graph rewriting-based model transformations. To describe the semantics of the model transformation language used in our framework, I applied the ASM formalism. I have built a transformation that supports the application of static model patterns. I used category theory to prove the correctness of the transformation. I also used category and graph theory to verify the correctness of practical model patterns I built.

At the Department of Automation and Applied Informatics of BUTE we developed the Visual Modeling and Transformation System (VMTS) [VMT10] that is a metamodeling and model transformation framework. During my research, I have implemented the theoretical solutions in modules of VMTS in order to show the practical relevance of the solutions as well.

3 Novel Scientific Results

The scientific results of my research are summarized in three theses. The theoretical results are proved by mathematical and engineering methods, and are illustrated by applications to prove their industrial relevance.

The first contribution deals with the visual specification of the dynamic behavior (animation) of metamodeled languages. I provide (i) an event-based conceptual architecture to support animation and (ii) a set of visual languages to describe the animation of the models and their execution. A transformation is also elaborated that generates executable source code from the animation models. My solution clearly separates the domain knowledge and the animation description both on the conceptual and the realization level. Thus, my approach offers a concise and systematic solution to provide a highly customizable animation framework for metamodeled languages with strong integration support to external systems such as simulation engines.

In order to decrease the execution time of model simulations, the second contribution focuses on the performance optimization of graph rewriting-based model transformation systems. I provide an overlapped matching technique for control flow-based model transformation systems. My technique harnesses the similarity of consecutively executed rules by matching common parts of the patterns at once. In addition to the first-fit execution, I specialize the base algorithm for exhaustive and iterative execution as well. I provide another optimization technique for exhaustively executed rules, which technique reuses partial matches from the previous execution of the same rule. I introduce the combination of the two approaches as well.

In the third contribution, I provide a complete framework based on model transformation to define and apply various model patterns with arbitrary intention in a domain-specific environment. I provide the formal description of the execution logic of the transformation engine and analyze the correctness of the logic and the expression power of the solution. The introduced transformation system serves as the execution engine for the operational aspect of Active Model Patterns (AMPs) that describe frequently used complex model manipulations and refactoring operations in Domain-Specific Modeling. With the specialization of the operational aspect, we can easily realize the static aspect of AMPs as well. Here I provide the formal specification of the requirements against the static approach. I present a transformation realizing the static aspect and prove its correctness. The introduced approach is motivated by case studies from the VMTS Animation Framework and other domains.
Supporting Model Animation Methods with Graph Transformation

Thesis I
Defining the dynamic behavior of metamodeled languages

I have elaborated an event-driven concept and a set of visual languages to describe the dynamic behavior (animation) of metamodeled languages. I have formalized the execution semantics of the visual languages with Abstract State Machines (ASM), and provided graph rewriting-based model transformations to process the input models and generate the executable source code. I have formalized the behavior of the implementation and proved its correspondence to the specification.

Thesis I is contained by the third chapter of the dissertation. Related publications: [1][4][10][15][16][18][23][25][27][30]

An animation system is built up from animators that are connected through ports to each other. An animator contains states and the connecting edges, the state transitions are triggered by events that arrive from other components through ports. The corresponding formal structure is defined as follows.

Definition 3.1 (superuniverse of the model animation framework). The superuniverse \( \mathcal{A} \) of a state \( \mathcal{A} \) of the model animation framework is the union of the following universes.

\[
\begin{align*}
U_{\text{int}} &: \text{Universe of integer numbers} \\
U_{\text{real}} &: \text{Universe of real numbers} \\
U_{\text{bool}} &= \{ \text{true}, \text{false} \} \\
U_{\text{string}} &: \text{Universe of string literals} \\
U_{\text{ID}} &: \text{Universe of unique identifiers, exactly ordered items} \\
U_{\text{variable}} &: \text{Universe of states defined by global variables} \\
U_{\text{animator}} &: \text{Universe of animators (attributes: ID : } U_{\text{ID}}; \text{ Variables : } U_{\text{variable}}) \\
U_{\text{port}} &: \text{Universe of ports (ID : } U_{\text{ID}}; \text{ Owner : } U_{\text{animator}} : \text{ Container animator}) \\
U_{\text{stamp}} &: \text{Universe of timestamps (Time : } U_{\text{int}} : \text{ Logical/physical time; Index : } U_{\text{int}} : \text{ Additional index to make stamps unique)} \\
U_{\text{event}} &: \text{All the possible events handled by the animation logic (Stamp : } U_{\text{stamp}} : \text{ Scheduling time of the event; IsInternal : } U_{\text{bool}} : \text{ Indicates, if the event is internal; Type : } U_{\text{string}} : \text{ Defines the type of the event)} \\
U_{\text{statetype}} &= \{ \text{start, general, end} \} : \text{Possible state types} \\
U_{\text{state}} &: \text{Universe of animation model states (ID : } U_{\text{ID}} : \text{ unique identifier; Owner : } U_{\text{animator}} : \text{ Container animator; Type : } U_{\text{statetype}} : \text{ Type of the state; TimeOut : } U_{\text{real}} : \text{ if greater than or equals to 0, then an internal transition is performed after this timeout from the current state)} \\
U_{\text{edge}} &: \text{Universe of animation model transitions (ID : } U_{\text{ID}} : \text{ unique identifier (IsInternal : } U_{\text{bool}} : \text{Indicates if this is an internal transition; Right : } U_{\text{state}} : \text{ The edge is coming from this state; Left : } U_{\text{state}} : \text{ The edge is leading to this state; Guard, Action : } U_{\text{string}} : \text{ Guard and action expression assigned to the transition)} \\
\end{align*}
\]

Definition 3.2. The vocabulary \( \Sigma \) of the ASM formalism of the animation framework is assumed to contain the following characteristic functions (arities are denoted by dashes):

- **Monitored functions**: MappedTo/1, Animators/0, TestGuard/1, TestedPorts/1, PerformActions/1, ContainerState/1, Clone/1, FiredEvents/1, IsTransient/1
Controlled functions: \( \text{CurrentTime}/0, \text{currentState}/1, \text{LastTime}/1, \text{Element}/2, \text{LastIndex}/0 \)

Derived functions: \( \text{opset}/2, \text{opst}/2, \text{oplt}/2, \text{opeq}/2, \text{Peek}/1, \text{StartState}/1, \text{OutEdges}/1, \text{Top}/1, \text{StateInternalDelay}/1, \text{InternalDelay}/1, \text{NextInternal}/1, \text{NextExternal}/1, \text{MinExternal}/0, \text{MinInternal}/0 \)

Definition 3.3 (behavior of an animation specification). The behavior of a animation is specified by the vocabulary of the animation framework (Definition 3.2) and the ASM rules \text{Sort}, \text{Push}, \text{Pop}, \text{Fire}, \text{Step}, \text{StepExternal}, \text{StepInternal}, \text{GlobalStep}, \text{Animation}. By the rules we assume that the \text{choose} operator deterministically selects the element with the smallest ID.

Subthesis I.1

I have formalized the operational semantics of the animation framework following the definitions above, and proved that with the help of the animation framework one can simulate any system that can be defined using deterministic finite-state Discrete Event System Specification (DEVS) [ZKP00]. I have also shown that the languages of the animation framework can express any system specified using the Discrete Time System Specification (DTSS) [ZKP00] system, thus, it can simulate any continuous system with arbitrary small error.

Using a model transformation to process the models describing animation logic and to make them executable is a straightforward idea, because changes in the metamodel can be easily adapted. Furthermore, it facilitates the formal verification of the mapping between source models and the generated source code. I have realized a model transformation that generates executable source code from animation models. I have shown that the transformation detects topologic errors in the output, and the generated source code corresponds to the processed input model.

Subthesis I.2

I have proved that the model transformation mapping the animation models to source code always terminates, provided the input model is finite. I have shown that the transformation terminates without the modification of the input or output model if the input model contains topologically unreachable states. I have shown that executing the transformation on a G animation model, the generated source code builds up an object hierarchy not distinguishable from the model based on the Animators, ContainerState, MappedTo, TestGuard, PerformActions, FiredEvents and TestedPorts monitored functions, and the initial value of the \text{CurrentState} controlled function.

The generated source code uses the services of a DEVS-based simulation engine to animate models. In order to prove the equivalence of the expected operational semantics and the implemented behavior, I have formalized the execution logic of the implementation using ASM. Thus I have extended the formalization of the specification by the \text{newInternal}/1 monitored function, the \text{ElementGlobal}/1 and \text{LengthGlobal}/0 controlled functions, and the \text{RealLength}/1 derived function.

Definition 3.4 (vocabulary of the animation framework implementation). The vocabulary \( \Sigma^I \) of the ASM formalism for the implementation of the animation framework is assumed to contain the following characteristic functions:
**Monitored functions:** \( \text{MappedTo}^{1/1}, \text{NewInternal}^{1/1}, \text{Animators}^{0/1}, \text{TestGuard}^{1/1}, \text{TestedPorts}^{1/1}, \text{PerformActions}^{1/1}, \text{ContainerState}^{1/1}, \text{Clone}^{1/1}, \text{FiredEvents}^{1/1}, \text{IsTransient}^{1/1} \)

**Controlled functions:** \( \text{CurrentTime}^{0/1}, \text{CurrentState}^{1/1}, \text{LastTime}^{1/1}, \text{Length}^{1/1}, \text{Element}^{2/2}, \text{LastIndex}^{0/1}, \text{ElementGlobal}^{1/1}, \text{LengthGlobal}^{0/0}, \text{StartState}^{1/1}, \text{OutEdges}^{1/1}, \text{Peek}^{1/1}, \text{Top}^{1/1}, \text{StateInternalDelay}^{1/1}, \text{InternalDelay}^{1/1}, \text{NextInternal}^{1/1}, \text{NextExternal}^{1/1}, \text{MinExternal}^{0/0}, \text{MinInternal}^{0/0}, \text{RealLength}^{1/1} \)

**Derived functions:** \( \text{opset}^{2/2}, \text{opst}^{2/2}, \text{opleft}^{2/2}, \text{opeq}^{2/2}, \text{StartState}^{1/1}, \text{OutEdges}^{1/1}, \text{Peek}^{1/1}, \text{Top}^{1/1}, \text{StateInternalDelay}^{1/1}, \text{InternalDelay}^{1/1}, \text{NextInternal}^{1/1}, \text{NextExternal}^{1/1}, \text{MinExternal}^{0/0}, \text{MinInternal}^{0/0}, \text{RealLength}^{1/1} \)

**Definition 3.5** (behavior of an animation implementation). The implementation behavior of a VAF animation is specified by the vocabulary of the animation framework implementation (Definition 3.4) and the ASM rules \( \text{Sort}^{1}, \text{Push}^{1}, \text{Pop}^{1}, \text{Fire}^{1}, \text{Step}^{1}, \text{StepExternal}^{1}, \text{StepInternal}^{1}, \text{GlobalStep}^{1}, \text{Animation}^{1}, \text{PushGlobal}^{1}, \text{PopGlobal}^{1}, \text{RemoveGlobal}^{1}, \text{and} \text{EnqueueNextInternal}^{1} \). By the rules we assume that the \textbf{choose} operator deterministically selects the element with the smallest ID.

I have shown that the behavior of the underlying simulation system together with the generated source code corresponds to the specification, thus, the correctness of the implementation is proven:

**Subthesis I.3**

I have defined a \( \varphi \) mapping between the states of the animation implementation and the specification in a formal way. I have proved that the construction assigns a valid animation implementation state to each valid specification state and vice-versa, i.e. those states that can occur during the execution. I have shown that \( \varphi \) maps an external event in the implementation behavior for each external event of the specified behavior and vice-versa. I have proved that consequently the \( \varphi \) mapping is a bijection. I have shown that \( \varphi \) is a bisimulation between the animation specification \( A \) and its implementation \( A^! \).

**Subthesis I.4**

I have shown that the animation framework can be realized in practice. I have realized it in VMTS. I have proved that the animation framework is able to animate Petri-nets and model transformation definitions. Using MATLAB Simulink I have shown that the animation framework can integrate external simulation systems and use their calculation results.

**Thesis II**

**Compile-time performance optimization of model transformation systems**

In my approach, I assume a control flow-based model transformation engine where the application order of the rules is deterministic but the matches are not. I have elaborated algorithms (Overlapped Rewriting Algorithm - OLRA) to overlap the matching phase of graph rewriting rules reducing thus the execution time of the transformation. Overlapping the matching means that the isomorphic parts of the LHSs of the different rules are matched at once, and the common matches are extended with the remaining rule-specific parts for complete matches. Overlapping can be organized into a hierarchy as well: the
matching of the rules can be overlapped to a different degree. I have specialized the algorithm for three typical types of execution: first-fit, exhaustive and iterative. I have found satisfactory criteria when the algorithm can be used. I have introduced another optimization technique called deep exhaustive execution for the exhaustive execution manner. I have provided a hybrid solution combining the benefits of the two approaches.

Thesis II is contained by the fourth chapter of the dissertation. Related publications: [3][6][8][9][13][17][20][21][24][26][31][33].

The basic definitions of graphs, typed-graphs, attributed typed graphs and sequential independence are understood according to [EEPT06]. I also assign constraints to the production rules, thus, I have extended the basic definition of typed, attributed graph productions [EEPT06]:

**Definition 3.6** (typed, attributed graph production with constraints). Given an attributed type graph ATG with a data signature DSIG. A typed, attributed graph production with constraints \( p = (L \leftarrow K \rightarrow R, \mathcal{X}) \) consists of a typed, attributed graph production and a set of constraints \( \mathcal{X} \). Also, we have injective \( l, r \).

A production \( p = (L \leftarrow K \rightarrow R, \mathcal{X}) \) is applicable to a typed, attributed graph \( G \) via the match \( m \) if there is a context graph \( D \) such that (1) is a pushout and \( m \) satisfies constraints \( \mathcal{X} \), i.e. \( m \models \mathcal{X} \).

\[
\begin{array}{c}
L \leftarrow K \rightarrow R \\
\downarrow m \downarrow \downarrow (1) \downarrow \downarrow \\
G \leftarrow D
\end{array}
\]

I assume the following three ways of applying a rewriting rule on a host graph:

**Definition 3.7** (first-fit execution of a production rule). Given a \( G \) host graph and a \( p : (L \leftarrow K \rightarrow R) \) production rule. \( p \) is executed in the first-fit manner on the \( G \) host graph, if \( p \) rule is applied on one arbitrary \( m : L \rightarrow G \) match, provided that \( m \) exists.

First-fit execution simply means that after applying a specific rule, the control flow proceeds to the next rule.

**Definition 3.8** (exhaustive execution of a production rule). Given a \( G \) host graph and a \( p : (L \leftarrow K \rightarrow R) \) production rule. \( p \) is executed in the exhaustive manner on the \( G \) host graph if direct graph transformations with \( p \) production rule and arbitrary \( m_i \) matches are applied until no further match can be found. Formally: \( G \xrightarrow{p,m_1} G_1 \xrightarrow{p,m_2} \cdots \xrightarrow{p,m_n} G_n \), and \( \#m_{n+1} : L \rightarrow G_n \).

**Definition 3.9** (iterative execution of a production rule). Assume a host graph \( G \), a \( p : (L \leftarrow K \rightarrow R) \) production rule, an \( L^I \) subgraph with an \( i : L^I \rightarrow L \) morphism, and an \( M^I = \{m^I : L^I \rightarrow G\} \) initial match set. The \( p \) production rule is executed iteratively on the \( M^I \) initial match set, if it is applied for each \( m^I \in M^I \) initial match on an arbitrary \( m : L \rightarrow G \) match for that \( i \circ m = m^I \) (provided that \( \exists m \)).

Thus, a subgraph of the left hand side of the rule is iterated through an initial match set, the match is tried to be completed once for each iteration.

It is possible that the overlapped matching technique finishes the matching of the rules in a different order than the predefined application order. Given rules \( p_i, p_j \) suitable for
overlapped matching (or enabled for the overlapped rewriting algorithm), we expect that a match \( m_j : L_j \rightarrow G \) of \( p_j \) exists before and after the execution of \( p_i \). However, we do not expect \( p_i \) to preserve the attributes selected by \( m_j \). Only the attribute conditions \( \mathcal{X}_j \) on \( m_j \) have to be satisfied both before and after executing \( p_i \).

**Definition 3.10 (OLRA enabledness).** Given a typed attributed graph transformation system \( \text{GTS} = (\text{ATG}, P) \), \( P = \{ p_i : (L_i \leftarrow K_i \rightarrow R_i, \mathcal{X}_i) \} \). Two graph productions \( p_i, p_j \in P \) are overlapped rewriting algorithm (OLRA) enabled, if direct graph transformations \( t_1 : G \xrightarrow{m_i,p_j} H_1 \) and \( t_2 : H_1 \xrightarrow{m_j,p_i} X \) are sequentially independent for all \( G, m_i, m_j, \) and the status of \( \mathcal{X}_i \) and \( \mathcal{X}_j \) is not influenced by the prior execution of \( p_j \) and \( p_i \). That is, if \( m_i \models \mathcal{X}_i \) before executing \( p_j \), then \( m_i \models \mathcal{X}_i \) after executing \( p_j \) as well. Similarly, if \( m_i \not\models \mathcal{X}_i \) before executing \( p_j \), then \( m_i \not\models \mathcal{X}_i \) after executing \( p_j \) either \( (i \neq j) \) for all possible \( m_i \) matches.

**Subthesis II.1**

I have provided an algorithm that is able to overlap the matching phase of first-fit executed OLRA enabled rules that are applied consecutively according to the control flow of the transformation. I have proved that assuming a typed attributed graph transformation system \( \text{GTS} = (\text{ATG}, P) \), for all rules \( (p_i, \mathcal{X}_i) \in P \), if \( p_i \) has matches \( m_1^i \ldots m_n^i \), then the overlapped algorithm finds the same \( m_1^L \ldots m_n^L \) matches for \( p_i \) as well. I have shown that assuming a graph transformation sequence \( G_0 \xrightarrow{m_1^1} G_1 \xrightarrow{m_2^2} \ldots \xrightarrow{m_n^n} G_n \), if the overlapped algorithm finds \( m_1 \) for \( p_1 \ldots m_n \) for \( p_n \) as well, then after the overlapped execution \( (G^O_0 \xrightarrow{m_1^1 \ldots m_n^n} \xrightarrow{m_1^L \ldots m_n^L} G_n) \) the resulting \( G'_n \) and \( G_n \) are both structurally and attributewise isomorphic. Namely, the overlapped execution produces the same result as that of the non-optimized algorithm.

If we overlap exhaustively executed rules, we cannot preserve the execution order of the rewriting phase of the rules (as we cannot cache the matches like in case of the first-fit execution because of their large number). Therefore, we need a stricter condition compared to OLRA enabledness. We not only need the rules to preserve the matches of other rules, but also the commutativity of the attribute operations of the rules, i.e., executing them in an arbitrary order results the same final attribute configuration of the processed elements.

**Definition 3.11** (strong OLRA enabledness). Given a typed attributed graph transformation system \( \text{GTS} = (\text{ATG}, P) \) with OLRA enabled production rules \( p_i \neq p_j \in P \). If for all direct graph transformations \( t_1 : G \xrightarrow{m_i,p_j} H_1 \) and \( t_2 : H_1 \xrightarrow{m_j,p_i} X \) the transformations can be applied in the reversed order without changing the resulting graph then \( p_i \) and \( p_j \) are strongly OLRA enabled.

**Subthesis II.2**

I have provided an algorithm that overlaps the matching phase of exhaustively executed strongly OLRA-enabled rules that are executed consecutively according to the control flow of the transformation. I have proved that the depth of the recursive algorithm is at most the number of the overlapped rules. I have proved that if the individual rules terminate, then the overlapped execution terminates as well. Furthermore, when the overlapped execution...
terminates, there cannot be found a valid match for either of the overlapped rules. I have shown that the overlapped execution of exhaustive rules produces the same resulting graph as the non-overlapped execution. I have proved by measurements that the optimization technique is capable of reducing the runtime requirement of the overall transformation.

Subthesis II.3

I have elaborated an algorithm that overlaps the matching phase of iteratively executed strongly OLRA-enabled rules that are executed consecutively according to the control flow of the transformation. I have shown that the overlapped version of the algorithm produces the same result as the non-overlapped execution. I have verified the results with measurements.

To optimize the execution of exhaustive rules, deep exhaustive execution harnesses the partial matches of the previous execution of the same rule, and drops only those parts that are invalidated by the rewriting.

Subthesis II.4

I have proved that a match \( m \) found during the deep exhaustive execution of a \( p \) : \( (L_p \leftarrow K_p \rightarrow R_p, \chi_p) \) rule is a valid match, i.e. there is a subgraph isomorphic to \( L_p \), and satisfies the \( \chi_p \) constraints of rewriting rule. I have shown that if the deep exhaustive execution of a \( p \) rule terminates with a \( G \) resulting graph, then \( \exists m : L_p \rightarrow G \) match. I have provided satisfactory conditions to prevent the algorithm from restarting the matching to ensure the non-existence of further matches.

A straightforward idea is to combine the two optimization techniques for exhaustively applied rules, namely, overlap the execution of the individual rules and perform deep exhaustive execution on them at the same time.

Subthesis II.5

I have combined the overlapping technique and the deep exhaustive execution into one algorithm. I have shown that when the algorithm terminates, there is no match for either of the optimized rules in the host graph. I have shown that the resulting host graph of the combined algorithm equals to that of the non-optimized execution. I have verified the applicability of the presented techniques with measurements, and shown that the execution time can be decreased with orders of magnitude compared to the non-optimized version.

Thesis III
The Active Model Pattern Framework

The purpose of Active Model Patterns (AMPs) is to provide a solution for often recurring operations in domain-specific modeling. AMP has three aspects. (i) The static aspect can be considered the adaptation of the idea of UML design patterns into the domain-specific world: a static pattern is actually an incomplete model fragment that can be inserted into other models with the same domain. (ii) Model refactorings or often recurring operation sequences during editing usually cannot be expressed with an incomplete model fragment. The operations of the operational aspect can be considered on-demand,
localized model transformations applied interactively. (iii) The third aspect of AMP is the **tracing aspect**. It covers the detailed logging of model manipulations for certain operations, such as undo/redo purposes. In this research, I dealt with the static and operational aspects. I propose interactive graph rewriting-based model transformations to realize the operational aspect. Based on the operational aspect I have realized the static aspect as a special pattern as well.

Thesis III is contained by the fifth chapter of the dissertation. Related publications: [2][5][7][11][12][14][19][28][29][32]

In our approach, we realize static model patterns as an application of the operational aspect: we define an operational pattern that generates a rewriting rule from a specified model fragment. On executing the generated rewriting rule, a model fragment isomorphic to the source fragment is inserted into the target model. From the generated rewriting rule, we expect the properties defined below.

(The definitions of typed graphs, typed graphs with inheritance and inheritance clan – clan₁ – is understood according to [EEPT06].)

**Definition 3.12** (derived metatypes). Having a typed graph \( G = (N, E, s, t, type) \) typed over type graph with inheritance \( T = (G_T, I) \), the derived metatype of an \( n \in N \) node with an \( E' \subseteq E \) set of connecting edges is \( N_E'(n) = \{ n_T | n_T \in N_T^E(n) : \exists n' \in N_T^E(n) \text{ and } n \rightarrow n' \text{ edge in } I \} \)

where

\[
N_T^E(n) = \bigcap_{e \in E', s(e)=n} \text{clan}_I(s(type(e))) \cap \bigcap_{e \in E', t(e)=n} \text{clan}_I(t(type(e))) \tag{3.1}
\]

Namely, the metatypes are selected such that the connecting edges of the node can be connected to another node with one of those metatypes.

We expect an operational pattern consisting of a single rewriting rule the following to realize static pattern insertion on execution:

**Definition 3.13** (Valid static model pattern rule). Assume that a static model pattern is defined by marking one or several elements of the \( G = (N, E, s, t, type) \) host graph: \( L_N^\text{sel} \subseteq N \), \( L_E^\text{sel} \subseteq E \). And \( L^\text{sel} = L_N^\text{sel} \cup L_E^\text{sel} \). We say that the VMTS rewriting rule model \( p = (N_p, E_p, s_p, t_p, type_p) \) realizes the static pattern defined by \( L^\text{sel} \) if \( \exists \text{map}_E : L_E^\text{sel} \rightarrow E_p \) and \( \text{map}_N : \{ n | n \in L_N^\text{sel} \cup \bigcup_{e \in L_E^\text{sel}} \{ s(e), t(e) \} \} \rightarrow N_p \) injective functions that satisfy the following requirements:

\[
(\varphi_1) \forall n \in L_N^\text{sel}: \text{map}_N(n).\text{TargetTypes} = \{ \text{type}(n) \} \land \text{map}_N(n).\text{Action} = 'create'.
\]

\[
(\varphi_2) \forall e \in L_E^\text{sel}: \text{map}_E(e).\text{TargetTypes} = \{ \text{type}(e) \} \land \text{map}_E(e).\text{Action} = 'create'.
\]

\[
(\varphi_3) \forall n \in \bigcup_{e \in L_E^\text{sel}} \{ s(e), t(e) \} / L_N^\text{sel} : \text{map}_N(n). \text{TargetTypes} = N_T^L_E(n) \land \text{map}_N(n).\text{Action} = 'match'.
\]

\[
(\varphi_4) \forall e \in L_E^\text{sel} : s(\text{map}_E(e)) = \text{map}_N(s(e)) \text{ and } t(\text{map}_E(e)) = \text{map}_N(t(e)).
\]

**Subthesis III.1**

I have shown that a rewriting rule satisfying the \( \varphi_i \) properties inserts a model fragment isomorphic to the source fragment into the host model. For this purpose, I have defined the GraphMTG category for which I have shown that it satisfies the properties of a category, and I used it to formalize matching with multiple types.
Subthesis III.2

I have created an operational pattern that generates the static pattern insertion rule for a predefined model fragment and proved that the output of the transformation satisfies requirements $\varphi_1..\varphi_4$. I have shown that the realized static aspect of AMP is able to describe static patterns for mobile user interfaces, animation state charts, and electric circuit schematics.

I have provided operational patterns for the following problems: (i) animation state chart unflattening (encapsulating a set of states into a container state), (ii) transformation rewriting rule concatenation, and (iii) exhaustive execution of a set of rewriting rules.

Subthesis III.3

I have shown that Active Model Patterns realized using interactive model transformations is suitable to solve such engineering problems as animation state machine unflattening, rewriting rule concatenation, and exhaustive execution of a set of rewriting rules. With graph and category theory constructs, I have formalized the requirements of the mentioned operational patterns and shown that the transformations satisfy them.

4 Practical validation of the theoretical results

The theoretical results of our research – (i) visual model animation, (ii) optimized model transformation execution, and (iii) model pattern support – have been realized in the Visual Modeling and Transformation System and applied in industrial and research projects.

4.1 Framework

In order to prove the practical applicability, I have developed the following components in VMTS: (i) The VMTS Presentation Framework (VPF) together with the VMTS Animation Framework (VAF) facilitates the animation of visual models in VMTS. VPF provides the presentation logic and the API to visualize models, while VAF supports the definition of the model animations and their execution. (ii) The VMTS Transformation Framework (VTF) component is used to design and execute graph rewriting-based model transformations. It supports both the compiled and interpreted manners of execution as well as the overlapped application of OLRA enabled rules. (iii) I have implemented the operational and static aspects of the AMP concept in VMTS. I have prepared VTF to support interactive and localized execution, and built the transformations to generate the rewriting rules for the static aspect.

4.2 Applications

Based on the animation framework of VMTS, I have realized the animation of models in the Petri-net domain and the models using the languages of the transformation engine; thus, we facilitate the real-time tracing and debugging of model transformations. With the integration of the MATLAB Simulink [MAT10] simulation engine, we facilitated the animation of a simplified car model based on the simulation results of Simulink. At the same time we have also integrated the transformation engine of VMTS into Simulink, and
performed the model transformation of Simulink models based on VMTS transformation definitions. We have built operational patterns for numerous domains including model transformation languages and state charts. Furthermore, we have also defined and realized static patterns for domains like electric circuit schematic, mobile user interface and state charts.

5 Related Publications

Journals


Publications in International Conference Proceedings


6 Citations

[3] is cited by:


[13] is cited by:


[23] is cited by:


Nuno Oliveira. *Improving Program Comprehension Tools for Domain Specific Languages* (master thesis), University of Minho, Braga, 2009

References


Supporting Model Animation Methods with Graph Transformation


Supporting Model Animation Methods with Graph Transformation


