Electronic Communications of the EASST
Volume 42 (2011)

Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010)

Active Model Patterns with Interactive Model Transformation

Tamás Mészáros and Tihamér Levendovszky and Gergely Mezei

13 pages

Guest Editors: Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Lazlo Lengyel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Active Model Patterns with Interactive Model Transformation

Tamás Mészáros[1] and Tihamér Levendovszky[2] and Gergely Mezei[3]

[1] mesztam@aut.bme.hu, [3] gmezei@aut.bme.hu
Department of Automation and Applied Informatics
Budapest University of Technology and Economics, Budapest, Hungary

[2] tihamer@isis.vanderbilt.edu
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN, USA

Abstract: With the proliferation of domain-specific languages, the generalization of OO patterns is a natural demand. Concepts and tools supporting pattern specification and execution for arbitrary domain-specific languages facilitate to meet these requirements. Our previous work introduced the Active Model Pattern Infrastructure and possible realizations for its static aspect. In this paper, we contribute a realization for the operational aspect of the framework. We propose graph rewriting-based interactive model transformation to describe and automate often recurring operational patterns in domain-specific modeling. We have extended a general transformation system with localized application of the rules and facilitate run-time customization possibilities for the domain engineer to influence the execution of the operations. We can specialize this approach to provide an implementation of the static aspect as well. We have realized our solution in the Visual Modeling and Transformation System.

Keywords: active model patterns, model transformation, VMTS

1 Introduction

Using OO patterns such as design patterns [GHJV95], architectural patterns [Bus96], refactoring operations [Fow99] has considerably simplified the design process of software systems. Patterns provide proven solutions for frequently recurring problems. They are usually described in an intuitive and informal way. For instance many design patterns can be described as an incomplete UML model fragment that needs to be inserted into the destination class diagram. Code refactoring operations, however, are usually represented with example code fragments and textual explanations. Nowadays, as domain-specific modeling is gaining an increased popularity, a noticeable amount of experience and knowledge have been collected among the experts of the different domains. A straightforward idea is to adapt OO patterns with automated tool support to the practice of domain-specific modeling as well. As Domain-Specific Modeling Languages (DSMLs) are meant to be used in arbitrary domains, thus, in accordance with [LLM09], DSML patterns are referred to as **model patterns**.

In our previous work [LK09], the basics of the Active Model Pattern (AMP) concept were introduced. An AMP can be regarded as the combination of design patterns and refactoring

operations in a domain-specific environment.

The AMP approach has three orthogonal aspects. The **static aspect** of AMP realizes the domain-specific static model pattern support. Static model pattern is the common name of domain-specific design patterns, architectural patterns, and patterns in general with arbitrary intention. They can be considered incomplete model fragments that are inserted into the target model. The AMP concept also includes universal design-time model manipulations, which are referred to as the **operational aspect** of AMP. Model refactorings or often recurring operation sequences during editing usually cannot be expressed with an incomplete model fragment with the static aspect. The operations can be considered on-demand, localized model transformations applied interactively. They can be realized either by a model transformation environment or by programming the modeling API directly (optionally supported with a proprietary DSL). 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.

[LK09] presented two approaches and their realization in the Generic Modeling Environment (GME) [La01] for the application of static model patterns. (i) The first one is based on the extension of the metamodel of the application domain. The pattern infrastructure generates an extended metamodel which is able to store pattern operations (insert/bind/ignore) in addition to the original properties of model elements. Pattern designers must use this specialized metamodel for the definition of patterns. (ii) The other approach uses the original metamodel and DSL to build the patterns, and assigns tags to the model elements to mark the model as a pattern. This solution requires the modeling environment to be able to tag its modeling elements with arbitrary information, even if the structure of the tag is not defined in the metamodel.

In this paper, we present an approach that supports the operational and the static aspects of Active Model Patterns with *interactive graph rewriting-based model transformations*. Using graph transformations [EEPT06], we could describe active patterns (both static and operational) without using the API of the modeling environment directly. In this case, graph production rules could describe the modifications in a precise way, thus facilitating further analysis on them.

In the following, we shortly illustrate an example domain and model patterns to realize (Section 3.1), then we collect the features that we expect an interactive model transformation environment to support AMPs (Section 3.1). In Section 3.2, we illustrate how we have realized the requirements in an existing system. In Section 4, we show how we can realize the operational pattern of the case study using the transformation system. Section 5 presents how the static aspect of AMPs can be implemented building on the operational aspect. Section 6 summarizes the related work and finally, Section 7 concludes.

2 Motivation

The motivation of our work is to aid editing Animation Framework (VAF) [MMC09][LM09] models in the Visual Modeling and Transformation System (VMTS)[VMT][LLMC07] with extensive tool support. The VMTS Animation Framework is a flexible framework supporting the real-time animation of models both in their visualized and modeled properties. The architecture of VAF is illustrated in Figure 1.

VAF separates the animation of the visualization from the dynamic behavior (simulation) of

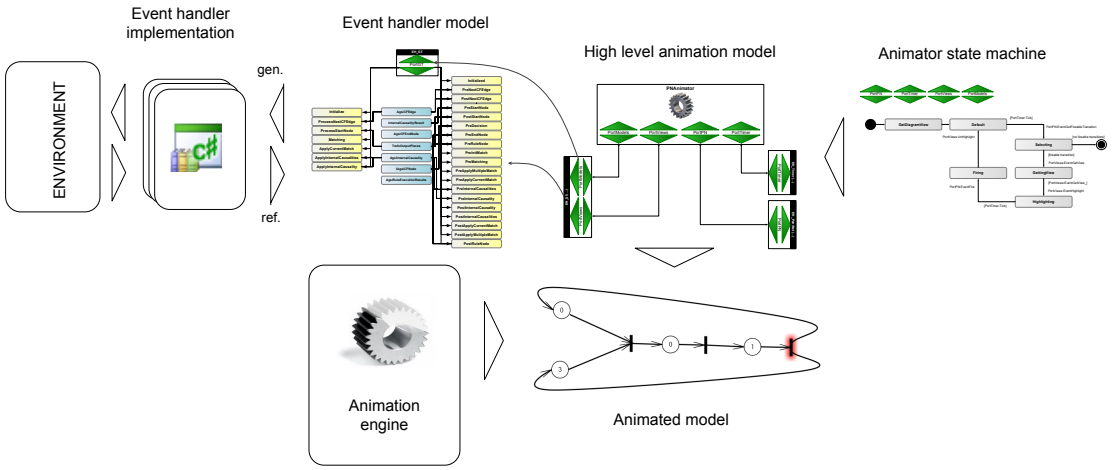


Figure 1: The Architecture of the VMTS Animation Framework

the model. In our approach, the domain knowledge can be considered a black-box whose integration is supported with visual modeling techniques. Using this approach, we can integrate various simulation frameworks or self-written components with event-driven communication. The animation framework provides three visual languages to describe the dynamic behavior of a metamodeled model, and their processing via an event-driven concept. *Events* are parametrizable messages that connect the components in our environment. The services of the presentation framework, the domain-specific extensions, possible external simulation engines (*ENVIRONMENT* block in Figure 1) are wrapped with *event handlers*, which provide an event-based interface. Communication with event handlers can be established using events. The definition of event handlers and the possible events is supported with a visual language. (*Event handler model* in the figure). The default implementation of an event handler can be generated [LM09] based on the interface of the wrapped objects (*Implementation* block). The animation logic can be described using an event-driven hierarchical state machine, called *Animator* (*Animator state machine* block). The transitions of the state machine are guarded by conditions testing the input events and fire other events after performing the transition. The input (output) events of the state machine are created in (sent to) another state machine or an event handler. The events produced by the event handlers and the state machines are scheduled and processed by a DEVS [ZKP00]-based simulator engine (*Animation Engine*). The event handlers and the state machines are connected through ports in a high-level model (*High level animation model*).

In this paper, we use the previously mentioned state machine language to illustrated our approach of AMPs. The metamodel of this language is depicted in Figure 2.

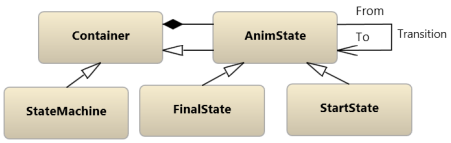


Figure 2: Metamodel of the target domain

State chart models may contain the usual start as well as end nodes and states (*StartState*, *FinalState*, *AnimState*), the states are connected with *Transition* edges. Each transition edge has a *Guard* and an *Action* property that describe the condition that must be satisfied to perform a transition and the actions to be executed on that transition. States and the *StateMachine*s have a common *Container* ancestor that may contain sub-states through containment edges. Thus, states may be composite as well.

2.1 An Operational Pattern

The example operation we realize with interactive model transformation is the unflattening of the state chart model. More precisely, the selected fragment of the state chart model is turned into a newly created composite state. Those outgoing transitions of the states that have a common *Guard* and *Action* property and a common target state are replaced with one leaving transition from the composite state. This process is illustrated in Figure 3.

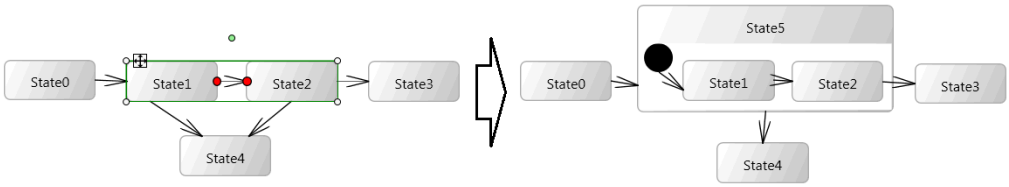


Figure 3: State chart unflattening

2.2 A Static Pattern

Figure 4 illustrates a static pattern for VAF state charts: a frequently used feature is the highlighting of a model element in a specific state of the state chart if the mouse hovers over the element. Occasionally, there may be multiple highlighted elements as well: [ML08] presents a visual model transformation debugger that highlights the match for a rule element under the mouse cursor after the successful match.

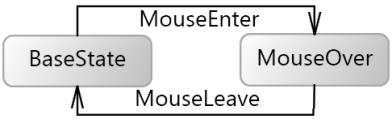


Figure 4: Static pattern to highlight element under mouse cursor

BaseState can be considered the default state without highlighting, and if a *MouseEnter* event is received, then a *Highlight* event is sent to the framework and the state machine proceeds to the *MouseOver* state. The corresponding guard condition is

PortPeripherals.PeekIsOfType< EHUI.EventMouseEnter>(),

and the action script is

Fire(new EHUI.EventHighlight(this, PortPeripherals.
PeekAs< EHUI.EventMouseEnter>().View) Color = Colors.Green , PortViews)

The target of the *Highlight* event in the action script is the *View* property of the received *MouseEnter* event: it identifies the visual object under the mouse cursor. Similarly, if a *MouseLeave* event is received, then the respective transition emits an *UnHighlight* event, and the *BaseState* becomes active again. Note, that *PortPeripherals* and *PortViews* are the conventional names of the ports to send and receive events to/from peripheral devices and the UI in VAF.

In the next sections we discuss the requirements and the conceptual as well as implementation-level solutions to realize tool-support for the introduced operational and static pattern.

3 An Interactive Graph Rewriting-Based Model Transformation Engine

We use interactive graph transformation as the execution framework for AMP. By *interactive* we mean that the domain engineer who executes the pattern application can visually trace and actively influence the behavior of the transformation. In the following we summarize the requirements and the realization of such a transformation system.

3.1 Requirement specification

We expect an interactive transformation engine to meet the following requirements to facilitate the implementation of the AMP approach.

1. The interactive selection of those model elements that should be involved in the transformation. The input of a usual transformation engine is one or several models. Therefore, we need to identify those elements of these models which may be used during the transformation. In case of the state chart unflattening this means the selection of those nodes that should be encapsulated in a composite state.
2. The engineer should be allowed to edit the rewriting rules at runtime: both their attributes and to perform structural changes on the rule.
3. The modeler should be allowed to modify the matches of the rewriting rules interactively. Requirement 2 and 3 is necessary to support static model patterns as shown in Section 5: thus we can exactly define how an inserted pattern should join the existing model elements.
4. If a complex model transformation (e.g. a model refactoring) is executed on the host model, it is possible that several branches of the transformation should be enabled or disabled according to the intentions of the domain expert. Here we assume that the execution order of the rewriting rules in the transformation is controlled by a control structure. For this purpose, the transformation control should be prepared to mark those branches where the user may have influence on the execution. For example in the state chart unflattening example we may ask the modeler whether to create an explicit start node in the composite state, or rather connect to the first child state of it directly. Depending on the selection of the modeler additional rewriting rules might be executed.
5. The transformation engine should provide a facility to ask for optional parameters from the domain engineer during the execution of the transformation: e.g. if the transformation creates new elements, several properties of the new elements (e.g. their name) should be asked instead of generating a default property value.

6. Finally, it is not enough to modify the underlying model repository, the changes should be reflected by the visualization as well: of course, deleted elements should be removed from the view, new elements should be displayed, and the changes in the attribute configurations should also be reflected in the visualization.

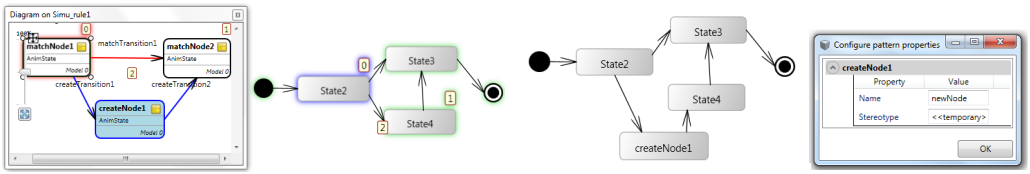
3.2 Realization of the AMP approach in VMTS

In the following, we guide through the requirements specified in Section 3.1, and illustrate with the help of the case studies how we have realized operational active model patterns in VMTS.

The interactive selection of those model elements that should be involved in the transformation. Recall that we suggest *localized* graph transformations to describe operational patterns. It is localized, because only several (selected) elements are involved in the transformation, and not the entire model. In VMTS, the model elements selected in the view are marked with a ”selected” flag. This flag can then be tested in the transformation, and the matcher can be restricted to find only the marked elements. In our case the flag is stored in a general *Tag* property, which is the member of each model element and is reserved to store arbitrary runtime meta information that is not persisted. In the textual constraints of the rules we may simply test this property. But, because this is a frequently used feature, we have extended the transformation specification language with a flag for the nodes and the edges: if the flag is set on an LHS element, then only the elements from the current selection are matched there.

The engineer should be allowed to edit the rewriting rules at runtime. Note that, because of the interpreted feature of the transformation engine, we may modify the production rules any time during the transformation (if the transformation is paused or waiting for user input), because the rule specifications are processed right when executing them.

The engineer should be allowed to modify the matches of the rewriting rules interactively. We facilitate the modification of the matching phase on two levels: (i) bind several elements of the rewriting rule to the elements of the host graph, and let the engine finish the match, (ii) and to modify a successful match manually. We provide a visual solution for both of them. Figure 5(a) illustrates a rewriting rule that matches two consecutive states connected with a transition edge ($matchNode1$,$matchTransition1$ and$matchNode2$), and inserts a new state between the two existing nodes.



(a) Visualizing and modifying a match (b) Editing attributes

Figure 5: Match visualization and attribute editing

The rule element $matchNode1$ is bound to the node $State2$ in the host graph. The binding is visualized with the ”0” tag next to the elements (a unique index starting from 0 is assigned to each pattern-host graph binding), and a blue outline highlights the matched element as well if one

hovers the mouse over the pattern item. An existing match can be modified, or an initial match can be defined with the help of the mouse: by clicking on the rule element while holding the Alt button, the pattern element (*_matchNode*1) receives the focus which is expressed with a red outline. Those elements of the host graph which have a type compatible with that of the pattern element are highlighted with green. By clicking on one of them, the match is changed. Note that even those elements of the pattern that define the creation of a new element (*__createNode*1 and the connecting edges in the figure) can be bound to existing elements of the host graph this way. Thus the matcher and the rewriting logic is configured to count on them when searching for a valid match, and not to create them in the rewriting phase.

Influencing the execution order of the transformation rules. We have two possibilities to influence the execution order of the rules interactively: (i) because of the interpreted feature of the engine, we can create/delete/reconnect edges in the control flow, provided that the execution is waiting for input or is paused. However, this is not a typical use case, as usually the transformation control should not be directly edited by the user. Instead, (ii) on the edges in the control flow one can create guard expressions and action scripts that prompt for user input and the engine proceeds its execution accordingly.

Asking for optional parameters. If a transformation creates new elements in the host graph, several properties of the new elements may be required to be customized by the user. E.g. the default name of new elements is typically changed right after creation. In order to support this requirement we have slightly modified the transformation definition language and the transformation engine. A new attribute called *PostConfiguration* is added to the possible attributes of the rewriting rule elements. Each item of *PostConfiguration* couples an attribute name and a default value. After performing a rewriting rule, the marked attributes are listed in a popup window where the user can set all the selected attributes at once (Figure 5(b)).

Changes in the model should be reflected in the visualization. As in case of numerous other model transformation environments, the model transformation engine of VMTS modifies the model objects only, but not their visualization. Therefore, we had to extend the interpreted transformation engine to (i) remove deleted edges and nodes, (ii) display newly created edges and nodes and (iii) to place the new elements in a way to follow the layout of the rewriting rule. We also made the coordinates of the model elements accessible in the textual constraints and the imperative code assigned to the rules, thus, they can be used both in the matching phase and edited in the rewriting phase. Furthermore, we apply the data binding features of the underlying UI framework (Windows Presentation Foundation - WPF) of VMTS to reflect attribute changes in the visualization as well.

4 Case study

In the following we present how we have realized the operational pattern outlined in Section 2 using the interactive model transformation system. Figure 6 illustrates the control flow graph of the transformation. It wraps the selected states with a composite state, and optionally creates a start state inside the new composite state, if exactly one selected element among all has incoming edges from outside the selection. All of those outgoing transitions of an arbitrary node within the selection that share their guard condition with one outgoing transition of each other node are

reconnected to the composite state. Those outgoing transitions of the other selected nodes that also share the action script and have the same target state are removed as well (i.e. the composite state already has those transitions).

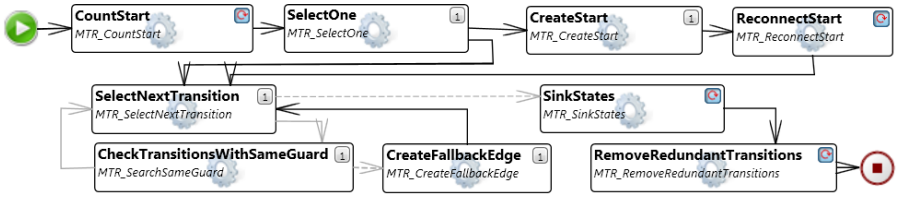
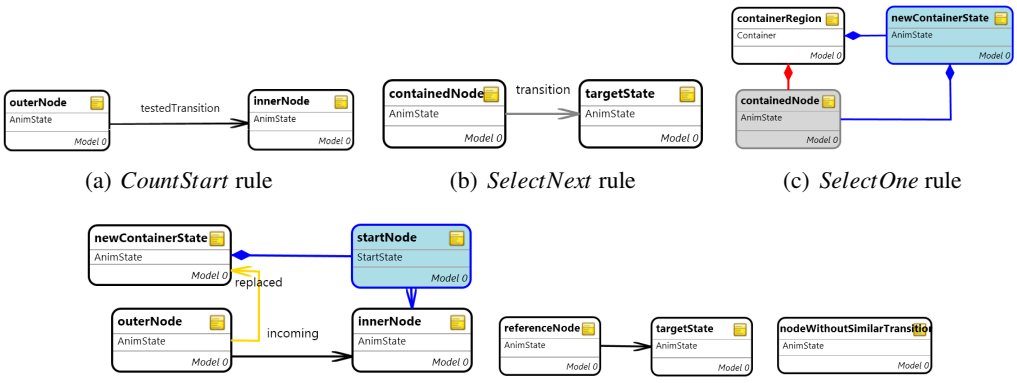


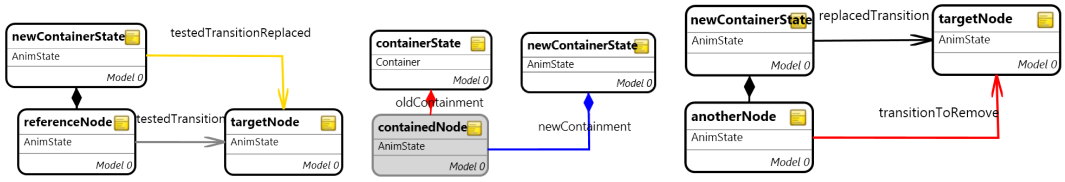
Figure 6: Control flow graph of the state chart unflattening transformation

The transformation works as follows. The *CountStart* rule (Figure 7(a)) matches all the states in the selection (*innerNode*) that have an incoming transition from a node outside the current selection, and puts those state nodes into a list. This is necessary, because in case the list contains only one element (i.e. one selected node has one incoming edge), we may offer to create a start node inside the composite node, and redirect the incoming edge to the composite node (as shown in Figure 3).



(a) *CountStart* rule (b) *SelectNext* rule (c) *SelectOne* rule

(d) *CreateStart* rule (e) *CheckTransitionsWithSameGuard* rule



(f) *CreateFallback* rule (g) *SinkStates* rule (h) *RemoveRedundant* rule

Figure 7: Rewriting rules to unflatten a state chart diagram

The *SelectOne* rule (Figure 7(c)) selects an arbitrary node (*containedNode*, called *reference node* from now on) within the selection, and places it into a newly created container (*newContainerState*). Both the *containedNode* and the *newContainerState* nodes are flagged

so that they can be identified later. In addition, the container state is positioned and sized to cover the rectangle of the selection, and *containedNode* keeps its original absolute position inside it.

The *SelectOne* rule is left by two flow edges: the guard condition of the one leading to *CreateStart* (Figure 7(d)) asks the user whether to create an explicit start state within the new composite state. But only if *CountStart* found only one selected node with an incoming edge from outside the selection. If there is more than one such node or the user answers 'No', then the execution proceeds to *SelectNextTransition*. Otherwise, *CreateStart* creates a new start state within the composite state, redirects the incoming edge to the composite state, and creates a transition within the start node and the state marked by *CountStart*.

The *SelectNextTransition*, *CheckTransitionsWithSameGuard* and *CreateFallbackEdge* rules iterate through the edges of the reference node, and if the guard expression of either outgoing transition is used for one outgoing transition for each other selected state, then that transition is pushed one level up to the composite state. This means that a transition can be pushed up without changing the semantics of the state chart, if all the selected states have an outgoing transition with the same guard expression. The *SelectNextTransition* (Figure 7(b)) selects and marks one non-marked outgoing transition of the reference node, then *CheckTransitionsWithSameGuard* (Figure 7(e)) tries to match another selected state without an outgoing transition with the same guard (phrased in the textual constraints of the rule). If it fails, then the transition can be pushed up (*CreateFallbackEdge* in Figure 7(f)). Otherwise, (and also after *CreateFallbackEdge*) a new transition is searched for by *SelectNextTransition*.

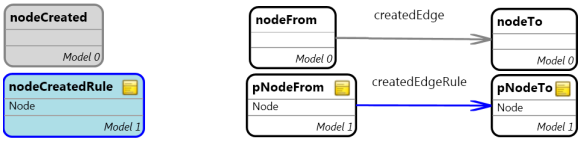
If there is not any non-marked outgoing transition for the reference node, then the remaining selected states are moved into the composite state (*SinkStates* - Figure 7(g)). Finally, *RemoveRedundantTransitions* (Figure 7(h)) removes those outgoing transitions of the selected nodes that have a corresponding outgoing transition on the composite node with the same guard and action expression and target state. Transitions with the same guard, but different action script or target node are left, as they override the behavior of the composite state.

5 Realizing static model patterns as operational patterns

By exploiting the features of the interactive model transformation environment, we can easily realize a static DSML pattern application framework. The key idea is to represent each static DSML pattern as a single rewriting rule (an operational pattern) that creates the elements in the target model, and to execute the rule. As presented in Section 3.2 we can coordinate the matching phase of the rule, and select those elements which already exist and those elements that should be created and attached to the existing elements. With the help of the *PostConfiguration* attribute, we can also select those properties of the inserted nodes and edges that may be customized at insertion time.

The generation of the insertion rule can be considered an operational pattern as well. The pattern consits of two rules (illustrated in Figure 8) that are executed exhaustively. *Rule_CreatedNode* matches all the selected nodes (*nodeCreated*) once, and creates (indicated by the blue background) the corresponding rule nodes (*nodeCreatedRule*) in the target model. The rule nodes are configured to perform the insertion of a node with the appropriate type. Similarly, *Rule_CreatedEdge*

matches all the selected edges once (*createdEdge*), and creates the related rule edges (*createdEdgeRule*) in the target model. *Rule_CreatedNode* stores the mapping between the elements of the source model (*nodeFrom*, *nodeTo*)and the elements of the target rewriting rule (*pNodeFrom*, *pNodeTo*), thus, *Rule_CreatedEdge* can identify the corresponding endpoints of the created rule edge. The imperative code of the resulting rule is also configured to produce the same attribute configuration than that of the source model fragment. The *nodeCreated* and *edgeCreated* elements are also flagged (indicated by the gray background) not to be matched again in the same transformation.



(a) *CreatedNode* rule (b) *CreatedEdge* rule

Figure 8: Rewriting rules to generate a static pattern insertion rule

As a result, the transformation generates a rewriting rule that is topologically isomorphic to the selected model fragment, and performs the insertion of the copy of that model fragment into an arbitrary model. This process is illustrated in Figure 9 for the highlighting pattern.

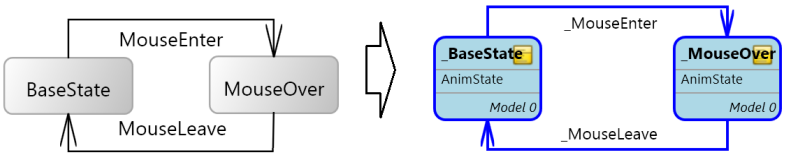


Figure 9: Composite state initialization pattern

The generated production rule contains one element for each corresponding pattern element: *_BaseState*, *_MouseOver* and the connecting transition edges. The *Action* property of each element in the rule is set to *Created* by default, meaning that each element of the pattern will be inserted as a new element in the target model if the rule is executed. However, as we can bind *Created* elements to existing model elements during the matching phase, we can skip the creation of one or several elements. E.g. if we bind the *_BaseState* node of the pattern to an existing host graph element, then it is not created, only the connecting elements.

A drawback of using plain rewriting rules to insert static DSML patterns is that the special syntax of the rewriting rules is displayed, and not the concrete syntax of the pattern. This may be uncomfortable for domain engineers being not familiar with the rewriting rule syntax. Therefore, we have extended the transformation engine to be able to present the pattern using its original syntax: we have added a *PatternTrace* attribute to the elements of the rewriting rules, which may point to the model elements of the source model of the pattern. This attribute is initialized by the transformation which generates the insertion rule. As this attribute defines an exact mapping between the elements of the pattern and the source model, instead of presenting the rewriting

rule during the transformation, the matches can be shown directly on the source model using the syntax of the source domain. This solution still has two drawbacks: (i) irrelevant parts of the source model are also shown, (ii) if the source model is modified, the mapping may become invalid. To overcome these limitations, we have extended the rule generator transformation to clone the selection, and - in addition to building the rewriting rule - to build a model which contains the selected model fragment only. Then, the *PatternTrace* attribute of the rule elements point to the elements of this clone model only.

6 Related Work

Since our research is related to DSML patterns, we do not consider UML-based solutions to be the related work of our research. Our previous work [LLM09] provides an extensive summary about UML-based design pattern approaches. To the best of our knowledge, besides VMTS, only GME has published its tool support for domain-specific model patterns to certain extent. A screencast about a promising ongoing research [Rei] on DSM refactorings became available recently, but we found the work unpublished otherwise. However, there are several tools that support some sort of interactive model transformation.

[ZLG05] introduces another GME solution: the Embedded Constraint Language is based on OCL [OMG06] but has imperative features as well. Using ECL one can describe model refactorings imperatively. The transformation framework also supports localized execution, however, this approach is rather close to direct API programming.

AToM^3 [LV02] provides a visual designer to build graph rewriting-based transformations, and facilitates the immediate visualization of the changes performed on the models. Furthermore, we can also access the UI through the API from the rules. However, AToM^3 does not support localized transformations, and one cannot influence the matching process directly either. The system does not provide a built-in support for static patterns either.

AGG [Tae04] also supports the visual specification of graph rewriting-based model transformations, furthermore, it facilitates the step-by-step execution of both the individual rules and the whole transformation. One can initialize or modify a match manually just like in VMTS, and manually select the rules to execute as well. An AGG based solution [Bie06] suggests graph rewriting to describe inplace model refactorings as well. As AGG has a layered, sequential control flow that does not support user interactions and conditional branchings, thus the engineer has much less influence on the execution order compared to VMTS. Static model pattern support is also a missing feature of AGG.

[TMM07] also suggests graph rewriting-based model transformation for refactoring operations. The solution generates executable code from the visual specification of the rewriting rules. However, it lacks the possibility to model the execution order of the rules, thus, it needs to be written in plain JAVA.

[EWL] The Epsilon Wizard Language (EWL) is a language tailored to interactive in-place model transformations on user-selected model elements. EWL is integrated with the Eclipse Modeling Framework (EMF) [BSM+03] and the Graphical Modeling Framework (GMF) [GMF10] and as such, wizards can be executed from within EMF and GMF editors. The drawback of EWL is that it is close to direct API programming, as refactoring operations are described using JAVA.

7 Conclusion

In this paper, we provide a complete framework to define and apply various model patterns with arbitrary intention in a domain-specific environment. We have presented the requirements and the realization in VMTS of an interactive graph rewriting-based model transformation framework. We provide possibilities to interactively influence the execution both on high (control flow) and on low (rule) level at runtime. The introduced transformation system serves as the execution engine for the operational aspect of 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. The presented approach is motivated and illustrated with a case study from the VMTS Animation Framework. Future work contains the design and realization of the tracing aspect of AMPs. Another open issue is how the operational pattern specifications could be generalized, and how the same transformation could be applied in different domains by the parameterization of the transformations.

Acknowledgements: This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project. This project is supported by the Hungarian Academy of Sciences - the Office for Subsidised Research Units and by the New Hungary Development Plan (Pr:TÁMOP-4.2.1/B-09/1/KMR-2010-0002)

Bibliography

[Bie06] E. Biermann et al. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST* 3, 2006.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003.

[Bus96] F. Buschmann et al. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EWL] Epsilon Wizard Language. http://www.eclipse.org/gmt/epsilon/doc/ewl/.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GMF10] Eclipse Graphical Modeling Framework home page. http://www.eclipse.org/gmf, 2010.

[La01] A. Lédeczi, et al. Composing Domain-Specific Design Environments. *Computer* 34(11):44–51, 2001.

[LK09] T. Levendovszky, G. Karsai. An Active Pattern Infrastructure for Domain-Specific Languages. *Proceedins of the First International Workshop on Visual Formalisms for Patterns*, 2009. in press.

[LLM09] T. Levendovszky, L. Lengyel, T. Mészáros. Supporting domain-specific model patterns with metamodeling. *Software and Systems Modeling* 8(4):501–520, 2009.

[LLMC07] L. Lengyel, T. Levendovszky, T. Mészáros, H. Charaf. Supporting Design Patterns in Graph Rewriting-Based Model Transformation. In *International Working Conference on Evaluation of Novel Approaches to software Engineering*. Barcelona, Spain, 2007.

[LM09] T. Levendovszky, T. Mészáros. Tooling the Dynamic Behavior Models of Graphical DSLs. In *In proceedings of the 13th International Conference on Human-Computer Interaction*. San Diego, USA, July 2009.

[LV02] J. de Lara, H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Metamodelling. In *FASE*. Pp. 174–188. 2002.

[ML08] T. Mészáros, T. Levendovszky. Visual Specification of a DSL Processor Debugger. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*. Pp. 67–72. Nashville, USA, 2008.

[MMC09] T. Mészáros, G. Mezei, H. Charaf. Engineering the Dynamic Behavior of Metamodeled Languages. *Simulation, Special Issue on Multi-Paradigm Modeling* 89:793–810, 2009.

[OMG06] OMG. Object Constraint Language, version 2.0. 2006. http://www.omg.org/technology/documents/formal/ocl.htm.

[Rei] J. Reimann. Generic Model Refactoring Based on Roles. http://emftext.org/index.php/Refactoring.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Application of Graph Transformations with Industrial Relevance (AGTIVE 2004)*. LNCS 3062 3062, pp. 446–453. Springer, 2004.

[TMM07] G. Taentzer, D. Müller, T. Mens. Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation. In *AGTIVE 2007*. Lecture Notes in Computer Science 5088, pp. 104–119. Springer, 2007.

[VMT] Visual Modeling and Transformation System. http://vmts.aut.bme.hu.

[ZKP00] B. P. Zeigler, T. G. Kim, H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[ZLG05] J. Zhang, Y. Lin, J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Volume II of Research and Practice in Software Engineering*. Pp. 199–218. Springer, 2005.