Visual Specification of a DSL Processor Debugger

Tamás Mészáros
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
mesztam@aut.bme.hu

Tihamér Levendovszky
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
tihamer@aut.bme.hu

Abstract

Graph rewriting-based model transformation is an essential tool to process graph-based visual models. If the execution of transformations is not supported by the continuous presentation of the modifications performed on the model, the traceability and the debugging of transformations becomes difficult. Recent modeling tools usually support the definition of rewriting rules based transformations in a visual or textual way, and only a few of them support visual debugging facilities. These debuggers are hand-coded at a price of a huge amount of work. This paper presents a model transformation debugger built on the top of the animation framework and the transformation engine of the Visual Modeling and Transformation System. The integration of the transformation engine and the animation of the user interface are described with visual modeling techniques.

Categories and Subject Descriptors I.6.2., I.6.3. [**Simulation and Modeling**]: Simulation Languages, Applications

General Terms Design, Languages

Keywords *Model transformation, Animation, VMTS*

1. Introduction

Domain-specific modeling is a powerful technique to describe complex systems in a precise but still understandable way. The strength of domain-specific modeling lies in the application of domain-specific languages to describe a system. Domain-specific languages are specialized to a concrete application domain; therefore, they are particularly efficient in their problem area compared to general purpose languages.

Models created with such languages usually need further automated processing methods to utilize the information expressed by the models in real, end-user applications. The processing may be similar to the source code compilers which convert human-readable source code to byte-code or machine code executed by the hardware or a virtual machine, but various model-to-model transformations are also frequent.

When developing a model processor for a language, it is important to be able to efficiently trace and debug the operations performed by the processor. It is not negligible how much effort is required to develop a visual debugger either. The motivation of our work is to provide a model transformation debugger solution built with the help of visual modeling techniques.

Visual Modeling and Transformation System (VMTS) [1] is a general purpose metamodeling environment supporting an arbitrary number of metamodel levels. Models in VMTS are represented as directed, attributed graphs the edges of which are also attributed. The visualization of models is supported by the VMTS Presentation Framework (VPF) [2]. VPF is a highly customizable presentation layer built on domain-specific plugins which can be defined in a declarative manner.

VMTS Animation Framework

The VMTS Animation Framework (VAF) [3] 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 the model. For instance, the dynamic behavior of a graphically simulated statechart is really different from that of a simulated continuous control system 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. The key elements in our approach are the *events*. 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 is supported with a visual language. The visual language defines the event handler, its parameters, the possible events, and the parameters of them - called *entities* (*Event handler model* in the figure). The default implementation of an event handler is generated based on the model, but the event handler methods which interact with the wrapped object have to be written manually (Implementation block).

The animation logic can be described using an event-driven state machine, called *Animator* (*Animator state machine* block). We have designed another visual language to define these state machines. The state machine consumes and produces events. The transitions of the state machine are guarded by conditions (*Guard* property) testing the input events and fire other events after performing the transition (*Action* property). States also define an *Action* property, which describes an operation that is executed when the state becomes active. 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 [4] based simulator engine (*Animation Engine*).

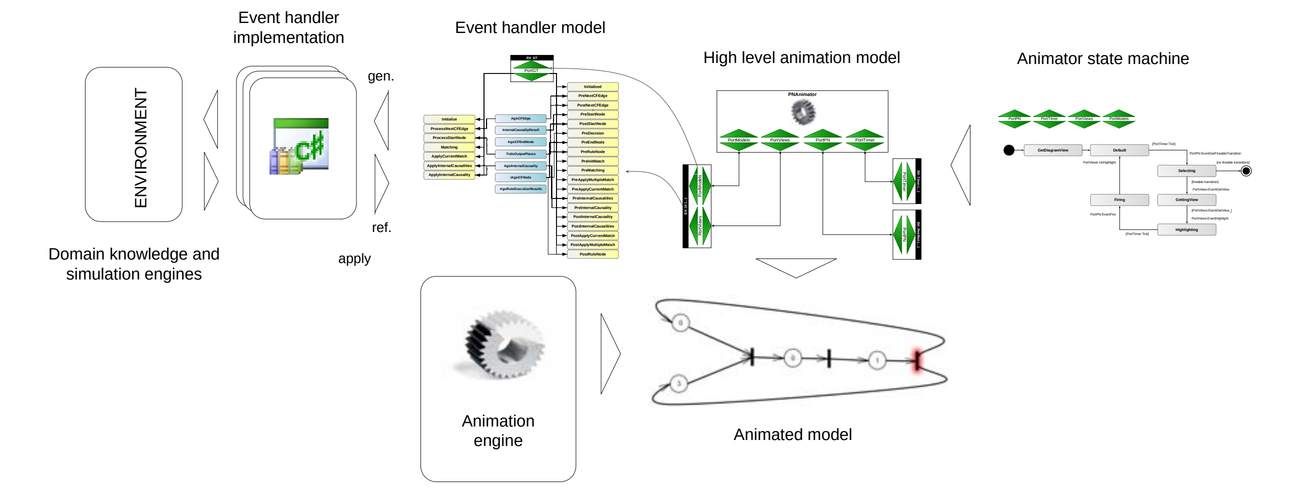

Figure 1. Architecture of the VMTS Animation Framework

The event handlers and the state machines can be connected in a high-level model (*High level animation model*). The communication between components is established through ports. Ports can be considered labeled buffers, which have a configurable but predefined size.

On executing an animation, both the high-level model and the low-level state machines are converted into source code, which is executed after an automated compilation phase.

Graph rewriting

Recall that in VMTS, models are represented as directed, attributed graphs. Model elements are represented by nodes and the connections between the elements are defined by the edges of the graph. This representation facilitates the applications of various graph transformation algorithms. Graph rewriting [5] is a powerful technique for applying graph transformations. Graph transformation consists of rewriting rules. Each rewriting rule has two parts: a Left Hand Side (LHS) and a Right Hand Side (RHS). The LHS defines a model pattern which has to be found in the input model, while the RHS describes a substitute pattern the match of the LHS has to be replaced with. Editing graph rewriting rules is supported via the Rule Editor plugin of VMTS. The execution order of rewriting rules can be defined with the help of the Visual Control Flow Language [6]. A Visual Control Flow model may contain six types of elements: *Start* node, *End* node, *Rule* node, *Decision* node, Flow edge and *External causality* edge. The *Flow* edge indicates the direction of the control flow. The *Start* node defines the entry point of the transformation, it also specifies the output model (if different from the input model). The *End* node indicates the end of the transformation. The *Rule* node means the application of a rewriting rule, which is defined in another model, and the *Rule* node only references that model. The *Decision* node is used to branch in the flow based on a predefined Object Constraint Language (OCL) [7] condition. The external causality edge can declare that an element on the LHS of a rule matches another element on the RHS of another rule. The operation described by a rewriting rule is called *internal causality* in our terminology. There are three types of internal causalities: (i) *create*, which is used to create new elements in the output model; (ii) *modify*, which is appropriate for changing the attributes of the matched elements and (iii) *delete*, which deletes a specified subset of nodes matched on the LHS. The *create* and *modify* causalities are defined using the Imperative OCL [8] language.

The application of a rewriting rule usually consists of two main steps: (i) searching a subgraph (match) in the input model that matches the LHS pattern of the rule, (ii) execution of the rewriting rules. If the *Exhaustive* attribute of the rewriting rule is set to true, then the same rule is applied until no match can be found. Otherwise, the next rule along the control flow is applied.

2. A DSL Processor Debugger

The aim of building a debugger for visualizing model transformations is to be able to trace the transformation process, and to have the possibility to intervene at runtime. Thus, we had the following objectives before beginning to design the debugger: (i) the input and output models should be visualized and should always reflect the current state of the models; (ii) the control flow model should be animated to be able to exactly trace the execution of the transformation; (iii) the actually executed rewriting rule should also be shown and in case of a successful match, the match should be visualized; (iv) the transformation should run step-by-step and continuously, the continuous running should be able to be interrupted by breakpoints, and the user should be allowed to perform jumps in the control flow, (v) it would also be welcome, if the models (at least the host and the target) could be edited at runtime.

Event handler model

The model of an event handler defines the events it can handle, the parameters of the events, and the interface of the event handler. The interface of a component is described by a set of *ports*: both event handlers and state machines provide their services through ports which can be connected to each other.

Before implementing the animation logic with event-driven state machines, we had to wrap our graph transformation engine (the “ENVIRONMENT” in this case) with an event handler, to provide an event-driven uniform interface for the animators. However, after performing the wrapping, we can use this event handler not only for the debugging solution, but also for various other simulations requiring graph rewriting-based model transformation.

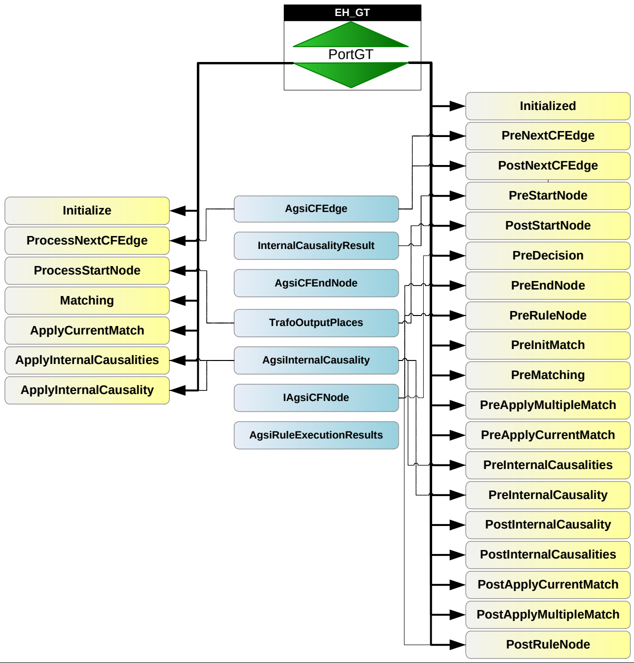

Figure 2. Model transformation engine event handler

Figure 2 illustrates the event handler model of the model transformation engine. The event handler (*EH_GT*) defines one port (*PortGT*) to send and receive events. On the left hand side the events received by the event handler can be seen, on the right hand side the events sent by the event handler are presented. In the middle, the entities (the parameters used by the events) are enumerated. The events sent by the event handler usually begin with "*Pre*" or "*Post*". The pre-events are fired before performing a specific operation, whereas the post-events are fired afterwards. After sending a pre-event, the event handler usually waits for another event to instruct the transformation engine to perform a step. Thus, we have the possibility to skip an operation or to modify its parameters. We have defined a pre-post event pair for each type of element in the transformation control flow: *Pre/PostNextCFEdge*, *StarNode*, *EndNode*, *Decision* and *RuleNode*. These events are also parametrized with the classes of the model transformation engine. The *Pre/PostNextCFEdge* events have a parameter of type *AgsiCFEdge* which points to the flow edge in the control flow. After sending a *PreNextCFEdge* event, a *ProcessNextCFEdge* event has to be sent to the event handler to follow the edge. The *ProcessNextCFEdge* event has a parameter of type *AgsiCFEdge* as well. This parameter should point to the edge to follow, thus, by pointing to an edge other than the one used by the *PreNextCFEdge* event, we can jump to an arbitrary edge in the control flow.

Rule nodes in the control flow are processed in the following steps: (i) The matcher algorithm searches for matches according to the LHS of the rule. If the rule node is configured for multiple matches, then several matches are found. Parts of the matches can also come from external causalities. (ii) The internal causalities of the rewriting rule are executed on the first match resulting in that several model elements may be deleted or created. (iii) In case of a multiple match, the following match is selected, and (ii) is performed again. (iv) In case of an exhaustive match, the complete process is repeated from (i) until no match can be found. The individual steps of this process are also wrapped with events, we have created the pre/post versions of *RuleNode*, *ApplyMultipleMatch*, *ApplyCurrentMatch*, *ApplyInternalCausalities*, *ApplyInternalCausality* events. The *PreMatching* event is sent before starting the matching phase, the *PreInitMatch* event is sent before initializing the match with the elements coming from external causalities. Influence on the matching and rewriting phase is also provided: the *PreApplyCurrentMatch* is fired before applying a match, however, one can override this match by sending an *ApplyCurrentMatch* with parameters different from the ones in the *PreApplyCurrentMatch*. We can also override the set of applied internal causalities and each internal causality as well with the help of the *ApplyInternalCausalities* and the *ApplyInternalCausality* events. The event flow of the rewriting phase is illustrated in Figure 3. The sequence diagram depicts the events fired between the transformation event handler and the animation engine when applying a rewriting rule including several causalities of it. This sequence diagram is included here for illustration purposes, not actually modeled, it is distilled from the state machine models, and only its implementation is generated.

Animation model

The animation can be described with the help of another visual language which can model state machines. These state machines communicate via events: the state transitions trigger the existence of a specific event on a specific port (or a specific event combination on a set of ports), and fire events when performing the state transition. The state machine is called *Animator* in our terminology. Animators are modeled on two levels: (i) on the high-level representation several animators and event handlers can be connected, and their interaction can be modeled, (ii) on the low level representation the individual states and transitions between the states of the state machine can be modeled.

Figure 4 illustrates the composition of animators and event

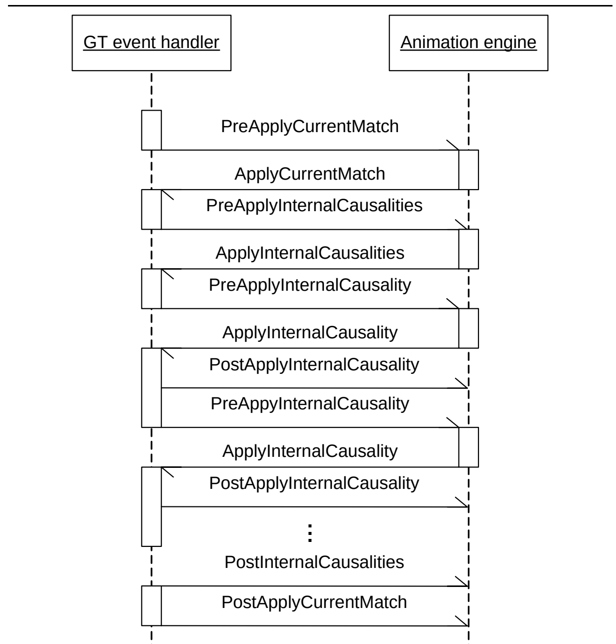

Figure 3. Event-flow of the rewriting phase

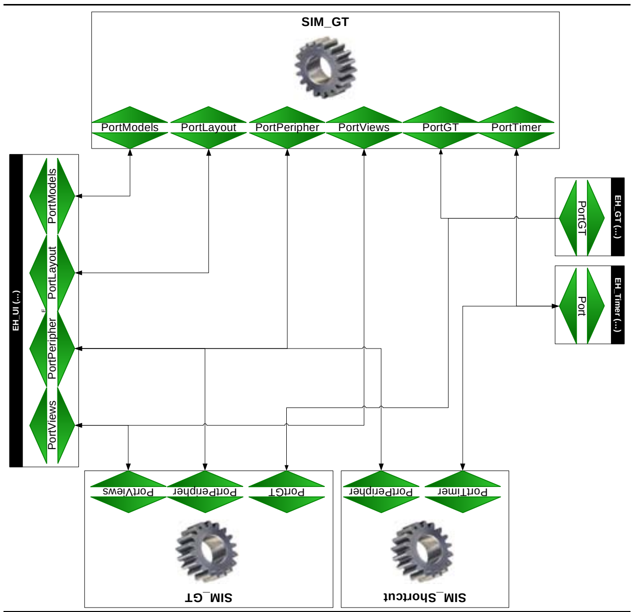

Figure 4. High level animation model of the debugger

handlers which implement the model transformation debugger. Event handlers (*EH_UI*, *EH_GT*, *EH_Timer*) can be seen on the left and right sides of the figure. The high level representation of three animators (*SIM_GT*, *SIM_MatchHighlighter*, *SIM_Shortcut*) is depicted on the top and the bottom of the figure.

The *EH_UI* element references the UI event handler which wraps the user interface API of VMTS and the model management methods. The *EH_Timer* element points to the event handler of a real-time clock, which fires *Tick* events with a predefined frequency. The frequency of the timer is set through the *Frequency* parameter of the event handler to 500 msec, thus one step is performed every half second in continuous execution of the transformation.

In Figure 4, one can see three animators: *SIM_GT*, *SIM_MatchHighlighter* and *SIM_Shortcut*. *SIM_GT* animates the control flow model, initiates the execution of the match and rewriting operations. *SIM_Shortcut* catches the key-presses, and instructs the *EH_Timer* to fire a *Tick* event if the F11 key was pressed. This feature is useful, if the timer is paused, and the user can execute the transformation step-by-step by hitting the F11 key. *SIM_MatchHighlighter* catches the mouse events, and highlights matched and created elements in the host and the output model of the transformation, if the mouse hovers over an element in the rewriting rule. Thus, we can check which elements were matched by which item in the LHS of the rule, and which new elements were created after the application of the rule. Using several animators to provide a solution, we can clearly separate orthogonal aspects of the problem space.

State machine models

Figure 5 presents the internals of the *SIM_MatchHighlighter* animator. Recall that this animator highlights those elements of the host model which are matched by the LHS element under the mouse cursor, and the elements of the output model that belong to the RHS element under the cursor. The default state of the animator is the *Matching* state. In case of a new match (*PreApplyCurrentMatch* event is received), the state machine stores the match in its *lastMatch* local variable, and resets the *lastResult* variable storing the newly created elements of the last rewriting. The appropriate guard condition is:

```
PortGT.PeekIsOfType<EH_GT.PreApplyCurrentMatch>() &&
PortGT.PeekAs<EH_GT.PreApplyCurrentMatch>().Match!= null
```

The corresponding action expression is:

```
lastMatch =
PortGT.PeekAs<EH_GT.PreApplyCurrentMatch>().Match;
lastResult = null;
```

The *PostApplyCurrentMatch* transition triggers an event with the same name, and stores the set of newly created elements in the *lastResult* local variable.

If the mouse hovers over a model item, a *MouseEnter* event is fired by the UI event handler. We have to filter it only for the nodes in the rewriting rules with the following guard condition:

```
PortPeripherals.PeekIsOfType<EHUI.EventMouseEnter>() &&
PortPeripherals.PeekAs<EHUI.EventMouseEnter>().View.
Model.AgsiMetaID.Equals(META_NODE);
```

The action expression which fires a *HighLight* event through the *PortViews* port for matched or created elements is listed below:

```
List<Node> match;
Node ruleNode =
(Node)PortPeripherals.PeekAs<EHUI.EventMouseEnter>().
   View.Model.AgsiItem;
if (lastMatch.TryGetValue(ruleNode, out match) ||
lastResult != null && lastResult.TryGetValue(ruleNode,
out match))
  foreach (Node n in match)
    Fire ( new EHUI.EventHighlight(this) { Element = n,
Color = Colors.Green }, PortViews);
```

A more complex scenario is implemented by the *SIM_GT* animator. It is responsible for (i) animating the control flow model, including detecting breakpoints and performing jumps, (ii) initiating the execution of rewriting rules, (iii) visualizing the changes of the output model. The internal structure of the animator is depicted in Figure 6. States and transitions in block (1) are used to initialize the transformation, to open the host and create the output model and to obtain a reference to the opened diagrams. The *Executing* state can be considered as a default state of the animation, the processing of the individual elements of the control flow model are initiated and finished in this state. Blocks (3), (4), (5) and (6), (7) are similar in the sense that they are responsible for processing and highlighting the elements of the control flow, namely the start node, edges, rule nodes, decisions and the stop node. Block (4) is entered after receiving a *Pre-*

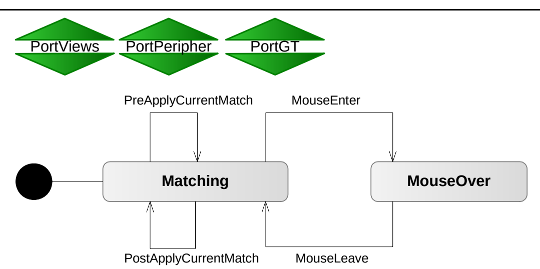

Figure 5. *SIM_MatchHighlighter* animator

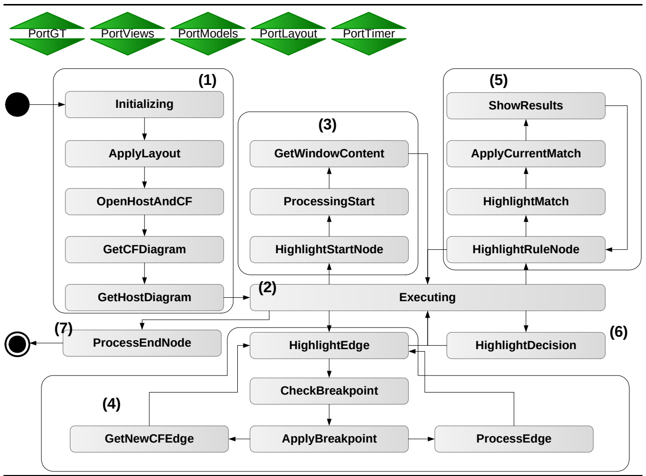

Figure 6. *SIM_GT* animator

NextCFEdge event. At this point we have the possibility to parametrize the *ProcessNextCFEdge* event with another edge, and step to an arbitrary point in the control flow model. This parameter can be set after testing whether an Alt+click event is received (Alt+click on an edge is used to jump to the edge). Consequently the guard expression before the *GetNexCFEdge* state is

```
PortPeripherals.PeekIsOfType<EHUI.EventMouseClicked>()
&& PortPeripherals.PeekAs<EHUI.EventMouseClicked>().
ModifierKeys == ModifierKeys.Alt && PortPeripherals.
PeekAs<EHUI.EventMouseClicked>().View.Model.AgsiMetaID.
Equals(META_EDGE)
```

In case of a successful evaluation of the guard condition, the processing of the control flow jumps to the new edge, otherwise the *ProcessEdge* will be the following state, and the execution continues in a normal way.

Due to the interpreted feature of the graph rewriting engine of VMTS, one can freely edit the control flow and also the host and output models during the debug process as well.

3. Related work

AToM[3] [9] is a general purpose metamodeling environment with simulation and model animation features. AToM[3] supports graph rewriting-based model transformations with a graphical editor for the definition of the rules; furthermore, the interactive debugging of transformations is also possible. The processed model can be animated according to the operations of the model transformation. The transformation can be executed in continuous mode or step-by-step. Compared to VMTS, AToM[3] does not support breakpoints or direct jumps between rewriting rules. Checking the result of a successful match is not possible either.

Graph Rewrite And Transformation (GReAT) [10] is a visual language and toolset to define and execute graph rewriting-based model transformations. GReAT has an approach similar to that of used in VMTS in the sense of graphical rule definition and the sequencing of the rules with a control-flow language. GReAT provides advanced debugging features including step-by-step execution of rules, and the application of breakpoints. The results of successful matches and the results of the rewriting rules are also logged in detail. However, inspecting the operations of the transformations in a visual way is not supported, although one can trace the transformation with the help of a textual interface.

The Attributed Graph Grammar System (AGG) [11] is an environment for developing graph rewriting based transformations. One can follow the execution of a transformation in AGG visually, including the applied rewriting rule and the host graph. The manual definition of matches is also supported by the environment. A transformation can run continuously or step-by-step, however, the process cannot be paused by predefined breakpoints in the rule-application sequence.

MetaEdit+ [12] is a general purpose metamodeling tool. It supports model animation through its Web Service API. Model elements in MetaEdit+ can be animated by inserting API calls into the code generated from the model, or by modifying the code generator to automatically insert these calls. If the attributes of a model element are changed, its visualization is automatically updated. The update mechanism can be influenced with constraints written in a proprietary textual script language of MetaEdit+. The modification of model attributes in VMTS also results in the automatic update of the presentation with the help of data binding. Applying converters to the data binding we can perform an arbitrary transformation on the presented data, this is a similar approach to constraints in MetaEdit+. Compared to VMTS, MetaEdit+ does not provide a graphical notation to define animation or for the integration of external components.

As of writing we are not aware of other visually modeled graph rewriting-based model transformation debuggers. Related work enumerated above provides hard-coded solutions for tracing and debugging transformations.

4. Conclusion

We have presented a visual debugger solution for model processors. The debugger is defined with the help of visual modeling techniques. Building on the VMTS Animation Framework, we could easily connect the animation of the user interface with the model transformation engine.

We have modeled the problem area on three levels. (i) The event handler model is used to wrap the model transformation engine with an event based interface. (ii) The high-level animation model connects event handlers with animators defining orthogonal aspects of the problem. (iii) The state machine models integrate the messages of the user interface and the transformation framework. They decompose the events of the transformation engine to a set of UI events (e.g. opening several diagrams after processing the start node), and also integrate messages of the event handlers into one or several new events (e.g. sending Event*HighLight* events after receiving timer *PreApplyCurrentMatch* and *MouseOver* events). The skeleton of the event handler implementation is generated based on the event handler model; the animation model and the low-level state machines are used generate the executable binaries implementing the debugger.

Future work includes the extension of breakpoints and jumps on further elements in addition to edges, and the improvement of breakpoints to stop the execution only if a predefined condition is satisfied. We would also like to provide a built-in OCL interpreter to evaluate OCL expressions on the transformed models at runtime. Similarly, we would also like to support the modification of causalities, especially the changing of their Imperative OCL code at runtime.

Acknowledgement

This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. The found of 'Mobile Innovation Centre' has supported, in part, the activities described in this paper.

References

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

[2] Mészáros, T., Mezei, G. and Levendovszky, T., A Flexible, Declarative Presentation Framework For Domain-Specific Modeling, Proceedings of the 9th International Working Conference on Advanced Visual Interfaces, 2008, Naples

[3] Mészáros T., Mezei G., Charaf H., Engineering the Dynamic Behavior of Metamodeled Languages, Submitted to Simulation: Transactions of the Society for Modeling and Simulation International, Special Issue: Multi-Paradigm Modeling: Concepts and Tools, 2008

[4] Zeigler B. P., Praehofer H., and Kim T. G., Theory of Modeling and Simulation, Second Edition, Academic Press, 2000.

[5] Ehrig, H. et al., Fundamentals of Algebraic Graph Transformation, Springer, 2006.

[6] Lengyel, L. et al., Control Flow Support in Metamodel-Based Model Transformation Frameworks, Proceedings of the EUROCON 2005 IEEE International Conference on "Computer as a tool", Belgrade, 2005, pp. 595-598

[7] OMG Object Constraint Language 2.0 Specification, http://www.omg.org/docs/ptc/05-06-06.pdf, last visited on 2008.07.16

[8] OMG MOF QVT Final Adopted Specification, http://www.omg.org/docs/ptc/05-11-01.pdf, last visited on 2008.07.16

[9] de Lara, J. and Vangheluwe, H., $AToM^3$: A Tool for Multi-Formalism Modelling and Meta-modeling. In ETAPS/FASE'02, LNCS 2306, pp. 174-188. Springer-Verlag

[10] Balasubramanian, D. et al., The Graph Rewriting And Transformation Language : GReAT, Proceedings of the Third International Workshop on Graph Based Tools, 2006, Natal

[11] Taentzer G., AGG: A Tool Environment for Algebraic Graph Transformation, LNCS 1779, pp. 333-341, Springer, 2000

[12] Tolvanen, J-P., MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages, In Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, Portland