Code Generation with the Model Transformation of Visual Behavior Models

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: There exist numerous techniques to define the abstract and the concrete syntax of metamodeled languages. However, only a few solutions are available to describe the dynamic behavior (animation) of visual languages. The aim of our research is to provide visual modeling techniques to define the dynamic behavior of the languages. Previously, we have created languages to describe animation. In this paper, we describe how these models can be processed by model transformation techniques. We elaborate the main steps of the transformation and show the details as well. We use graph rewriting-based model transformation, therefore we provide a highly generic solution which can be easily modified, and analyzed with the techniques borrowed from the field of graph rewriting. The termination analysis for the presented method is also provided.

Keywords: Metamodeling, Animation, Model Transformation, VMTS

1 Introduction

Domain-Specific Modeling (DSM) has gained increased popularity in software modeling. Domain-Specific Modeling Languages (DSMLs) can simplify the design and the implementation of systems in various domains. Domain-specific visualization helps to understand the models for domain specialists not familiar with programming.

A popular way to define DSMLs is metamodeling. Metamodels define a vocabulary of model elements for a specific language by describing the available model elements, their properties and the relations between the elements. This definition is often referred to as the abstract syntax of the language. However, metamodeling is not meant to describe the visual representation, the concrete syntax, or the editing behavior of modeling items. Based on the metamodel, a default concrete syntax can be generated, but the description of customized visualization - including colors, sizes and layouting - usually needs additional modeling techniques.

Multi-paradigm modeling [MV02][MV04][VLM02] is a special, straightforward way to apply domain-specific modeling to describe complex systems with different languages. Each language is efficient in its own domain and together, they can describe different aspects of the same system. Therefore, multi-paradigm modeling not only applies domain-specific languages, but it

offers a higher level of model composition techniques. The integration of orthogonal models is achieved by the modeling environment and the model processors. Multi-paradigm modeling addresses and integrates three orthogonal directions of research [VLM02]: (i) multi-formalism modeling, which addresses building models using different formalisms and designing transformations between them; (ii) model abstraction that covers the description of models at different abstraction levels; (iii) metamodeling, which means formally defining models as formalisms. Language engineering is a key factor in Multi-Paradigm Modeling (MPM). Since MPM strongly builds on metamodeling, the applied language engineering methods must also be generic enough to support various metamodels.

Besides the generic methods to build the abstract and concrete syntax of a visual language, only a few solutions are available to describe the dynamic behavior ("animation") of the models created by metamodeling. By animation, we mean both the visualization of automated model manipulation and the manipulation of the presentation without modifying the underlying model itself. Recent solutions usually bind visualization properties to model properties, and achieve animation by manipulating model properties. Model properties are usually modified with model transformation or direct API calls.

In [MMC09], we have already presented an integrated solution to describe the dynamic behavior of the models in a generic and visual way. In our approach, we separate the model and its animation logic, and provide visual languages to define the animation of either the model elements or only their visualization. We apply the multi-paradigm approach in the sense of separating the animation description into different domains: (i) framework integration, (ii) animation logic specification, and (iii) user interface description. The integration of the models is performed with both references between models of different domains and by the model processors. The integration of external components or frameworks into our environment is supported with a visual language and a code generator, thus, the animation logic can handle all components in a uniform way.

To be able to execute the visual behavior models with high performance, instead of the runtime interpretation of the models we generate executable source code from them and compile the source code into reusable dynamic linked libraries. We perform the code generation with graph rewriting-based [EEPT06] model transformation. In this paper, we present the transformation steps used to build the model of the source code from the animation models. The presented solution is not specific to VMTS, it can be adapted to any metamodeling environment which implements our animation framework.

2 Background

Visual Modeling and Transformation System [VMT09] is a general purpose metamodeling environment supporting n-level metamodeling. N-level means in this context that the instance models can be used as metamodels: they can be used to define model hierarchies such as meta class diagram - class diagram - object diagram. The maximum depth of these hierarchies is not limited; we can construct an n-level modeling chain. VMTS uses a proprietary modeling space. Models in VMTS are represented as directed, attributed graphs. In our approach, edges are attributed as well.

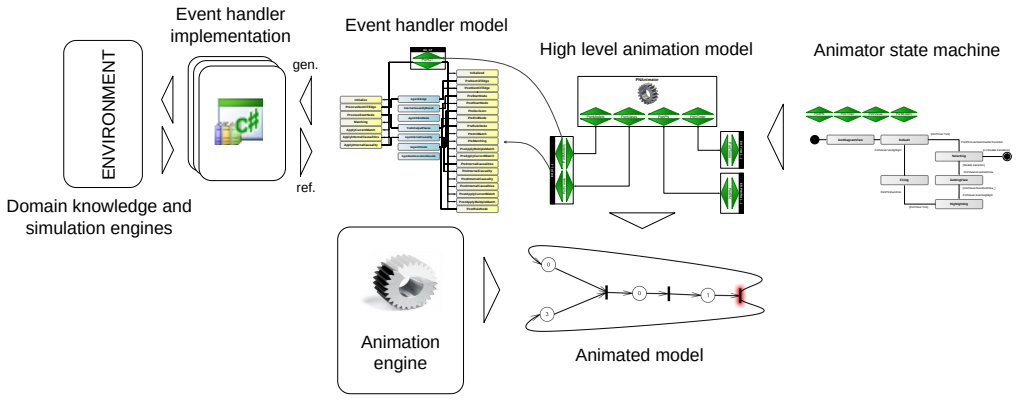


Figure 1: The Architecture of the VMTS Animation Framework

2.1 The VMTS Animation Framework (VAF)

The VMTS Animation Framework (VAF) [MMC09] 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 their parameters - called *entities* (*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). We have designed another visual language to define these state machines. The metamodel of this language is depicted in Figure 2. 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 state space of the *Animator*s can be extended using variables of primitive types. The input (output) events of the state machine are created in (sent

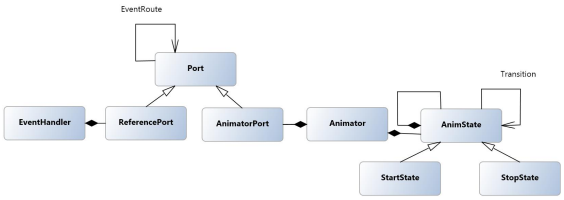


Figure 2: The metamodel of the VMTS animation description language

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 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 size. Note, that both the high- and low-level languages are defined by the same metamodel (Figure 2), however, based on their application they can be considered as two different languages.

2.1.1 Generated source files

Our implementation is based on the C# language, however, the presented solution can be easily adopted to arbitrary object-oriented programming languages. On executing an animation, both the high-level model and the low-level state machines are converted into source code, which highly builds on our DEVS-based simulation engine.

From each state-machine model (from each *Animator*) an individual class is generated, which implements the behavior described by the state machine. Furthermore, a *Configuration* class is also generated from the high-level model, this class wires the animator-classes and the event-handler instances together, and initializes the simulation framework.

The structure of the animation files is depicted is listed below:

```
namespace VMTS.VAF {
  public class <AnimatorName> : Simulator {
    //Ports of the animator implemented by properties
    public Port <PortName> {get; private set;} ...
    public <AnimatorName>(Coordinator coordinator)
      : base(coordinator) { }
    <Variables of the animator>
    public override void Init() {
      //Initialization of the ports
      <PortName> = new Port(this);
      <PortName>.Capacity = <capacity>;
      <PortName>.Circular = <isCircular>; ...
    public override void BuildUp() {
      startState = new VMTS.VAF.State(this, null, null);
      currentState = startState;
      //Initialization of the states
      State <stateName> = new State(this, <entering action>,
```

```
        <container state>); ...
      //Initialization of the transitions
      <fromState>.AddTransition(new Transition(this, <toState>,
        <guard condition>, <action>), <isInternal>);
  }}}
```

As one can see, the *Init* method initializes the ports, and the *BuildUp* method creates the states, and connects them with transitions. The *guard condition* and the *action* assigned to a transition is expressed using anonym methods.

The structure of the configuration files is the following:

```
namespace VMTS.VAF {
  class Configuration {
    public Configuration() {
      coordinator = new Coordinator();
      Init();
    }
    public Configuration( Coordinator _coordinator) {
      coordinator = _coordinator;
      Init();
    }
    private void Init() {
      //instantiation of generated animator classes
      <animator field> = new <animator type>(coordinator);...
      coordinator.Simulators.Add(<animator field>);...
      //instantiation of event handler classes
      <event handler field> = new <event handler type>(coordinator);
      coordinator.EventHandlers.Add(<event handler field>);...
      <setting event handler parameters>
      //registering connections
      coordinator.AddMapping(<from>.<fromPort>, <to>.<toPort>);...
    }
    <Animator field declarations>
    <Event handler field declarations>
  }}
```

The *Coordinator* class instantiated in constructor represents the interface towards the simulation engine. In the *Init* method the animators and event handlers are instantiated, and registered towards the coordinator, furthermore, the connections between the components (animators and event handlers) are also registered.

3 Transformation Models

The model of the generated source code uses the VMTS implementation of a C# DOM, which is very similar to the CodeDOM [COD] of Microsoft. Our DOM implementation builds on a general code generation and parsing technique presented in [AL09]. The control sequence of the transformation is depicted in Figure 3. The sequence can be departed into two well-separated parts: part (1) creates the individual animation classes for each *Animator* element and its contained state machine, while part (2) creates the configuration class.

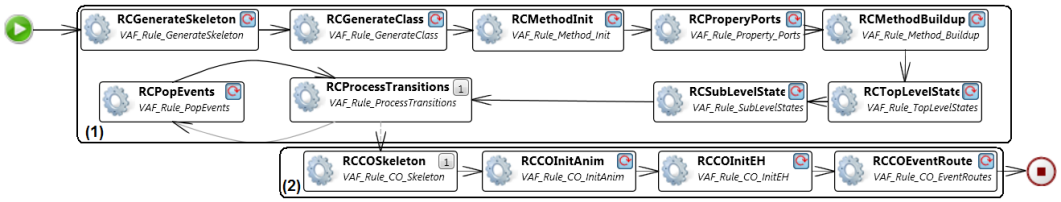
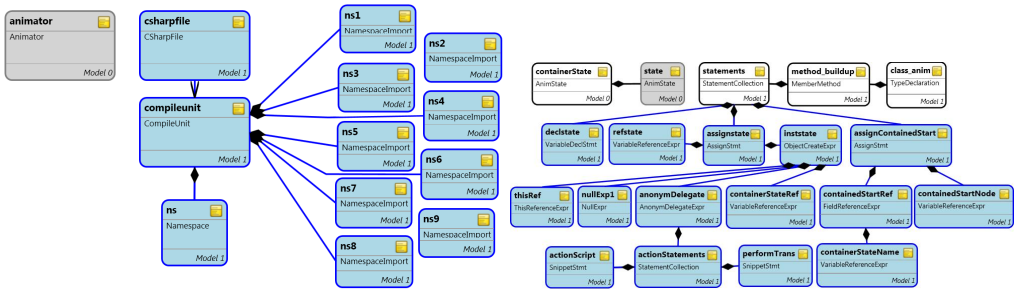


Figure 3: The control sequence of the transformation rules



(a) VAF_Rule_GenerateSkeleton (b) VAF_Rule_GenerateClass

Figure 4: Creating the skeleton of the animation source code

3.1 Processing the state machines

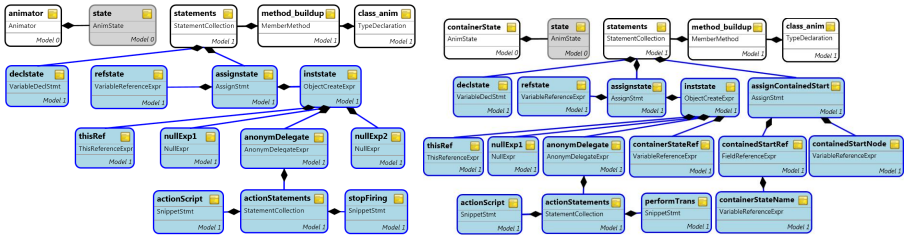
The *VAF_Rule_GenerateSkeleton* (Figure 4) rule creates the model for the source file, the default namespace (*VMTS.VAF*) and the necessary namespace references. The rule is executed *exhaustively*, meaning that it is repeated until a valid match (an unprocessed animator) is found. After finishing this rule, an output file with default content is generated for each animator.

The *VAF_Rule_GenerateClass* is illustrated in Figure 4. The rule matches an *Animator* and its connecting *namespace* element (the relation between the *Animator* and the *namespace* is expressed with not an edge, but with an attribute reference). Then the rule creates a class (*class_anim*) element in the output model, and adds a constructor with base constructor call and the skeleton of the *Init* method. The rule is executed in an exhaustive manner again.

Afterwards, the *VAF_Rule_Property_Ports* rule matches each *Port* inside an animator and generates a C# property for each of them within the appropriate class, furthermore, it also creates the initialization code for each port within the *Init* method of the class element. The *VAF_Rule_Method_BuildUp* rewriting rule matches each *Animator* and their connecting *class* again, and creates a *BuildUp* member method beginning with the initialization of the *startState* and *currentState* member variables.

3.1.1 Processing the states

The following two rules (*VAF_Rule_TopLevelStates* and *VAF_Rule_SubLevelStates*) are depicted in Figure 5. They are used to generate the model of the connecting code for the states contained directly by the *Animator* (*top-level)* and for the states contained by another states (*sub-level*). The *VAF_Rule_TopLevelStates* rule generates the model of the following code-fragment for each



(a) VAF_Rule_TopLevelStates (b) VAF_Rule_SubLevelStates

Figure 5: Processing states

top-level state:

```
State stateXXXX;
stateXXXX = new State(this, null, null);
```

or

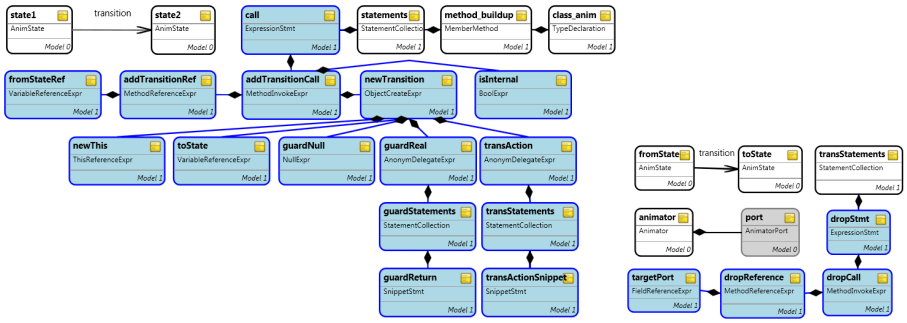
```
stateXXXX = new State(this, delegate() { <action script> }, null);
```

If an action is defined for the state, then the second version is selected, in case of which the action is specified with the help of an anonym method. The selection is ensured by assigning inverted application conditions to the *nullExp1* and *anonymDelegate* rule nodes: the affected elements are created only if the condition can be satisfied. The anonym method-version is selected in case of top-level *Stop* nodes as well: the simulation engine is stopped by sending a stop event to the framework (the *stopFiring* node is conditionaly created as well). Note, that the last *null* parameter in the constructor call of the *State* class denotes that the states do not have a container state.

The *VAF_Rule_SubLevelStates* is somewhat similar to the top-level version. One difference is, that at *Stop* states we do not have to stop the simulation, but to notify the framework to check the outgoing transitions of the container state as well (see the conditionaly created *performTrans* node). Another difference is, that the container state parameter is set (*containerStateRef*) in the constructor call. Furthermore, if a contained *Start* node is processed, the *ContainedStart* property of the container state object is set to the internal start state object (*assignContainedStart* branch in the figure). Both rules are executed exhaustively. When a *State* is processed, the rule creates an attribute-level reference from the *State* element to the *class_anim* element. The existence of this reference is set as a negative application condition for the rule, therefore, each *State* is processed exactly once, and the execution terminates. We also prescribe the existence of this reference as a positive application condition between the *containerState* and the *class_anim* element, thus we can ensure, that the code model for a contained state is generated only if its container state is already processed.

3.1.2 Processing transitions

Transition edges between two states are processed by the *VAF_Rule_ProcessTransitions* and the *VAF_Rule_PopEvents* rules (depicted in Figure 6).



(a) VAF_Rule_ProcessTransitions (b) VAF_Rule_PopEvents

Figure 6: Processing transitions

The *VAF_Rule_ProcessTransitions* rule matches transitions in an exhaustive manner, and generates the following code fragment for each of them inside the *BuildUp* method of the connecting class:

```
<stateFrom>.AddTransition(new Transition(this, <stateTo>,
    delegate(){ return <guard condition>;},
    delegate(){ <action script> }), false);
```

We can get the container class easily, as we have created a reference between each state and the connecting *class_anim* node. The last parameter (*isInternal* = false) denotes, that this transition is an internal one (triggered by a timer), or an external one, triggered by external events. At each execution of the rule, the selected *transition* edge is flagged, so that it cannot be processed twice. The *VAF_Rule_PopEvents* rule generates a *portXXXX.Drop();* command for each port triggered by the selected transition. The rule receives the transition to be processed as a parameter from the previous rule, and checks each port in an exhaustive way, whether it is used in the guard script of the actual transition. The *Drop* method call consumes the topmost event from the target port, and only from those ports, which were triggered by the transition.

As one can see, the *VAF_Rule_ProcessTransitions* and the *VAF_Rule_PopEvents* rule forms a cycle: they cycle exists, if each transition has been processed, including the verification of each port for each transition.

3.2 Generating the configuration class file

The configuration class is generated by the rules in the block (2) in Figure 3. The *VAF_Rule_CO_Skeleton* rule generates the skeleton of the class: type declaration, two constructors, and an empty *Init* method. The *VAF_Rule_CO_InitAnim* rule creates a member variable for each *Animator* element, and initializes them in the *Init* method. Similarly, the *VAF_Rule_CO_InitEH* rule creates member variables for the event handler elements, and initializes them. Finally, the *VAF_Rule_CO_EventRoutes* processes the *EventRoute* edges, registers the connections between the ports of the connected components:

```
coordinator.AddMapping(
  <sourceContainer>.<sourcePort>,
  <targetContainer>.<targetPort>);
```

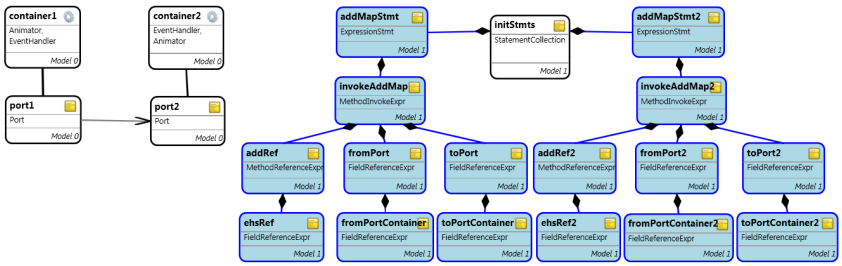


Figure 7: The VAF_Rule_EventRoutes rule

The *VAF_Rule_CO_EventRoutes* rule is illustrated on Figure 7.

Note, that on Figure 7 the *AddMapping* method call is modelled twice. As the *EventRoutes* can be bi-directional as well (depending on their *Direction* property), a reverse direction *PortMapping* can also be added. The reverse direction is created conditionally based on the settings of the *EventRoute* edge.

4 Termination Analysis of the Transformation

Except for the *VAF_Rule_ProcessTransitions* and the *VAF_Rule_PopEvents* rule-pair, there are no directed cycles in the transformation control flow graph. Therefore, we can examine the termination of the remaining rules separately.

In case of the *VAF_Rule_GenerateSkeleton* rule, the processing of an *Animator* is denoted by creating an attribute reference between the *Animator* (*animator* node) and the affected namespace declaration (*ns* node). The existence of such a reference is assigned as a negative application condition to this rule. Thus, during exhaustive execution, each *Animator* node is processed exactly once, and the rule terminates, as soon as there are not any unprocessed *Animator*s. The termination of the *VAF_Rule_GenerateClass* rule can be proven on a similar basis. That case the associated *Animator* and *Namespace* elements can be exactly matched using the attribute reference, and each *Namespace* element is flagged after processing it. The existence of this flag is assigned again to the rule as a negative application condition, thus the rule is executed exactly once for each *Animator*.

The termination of the remaining exhaustively executed rules can be proven based on the presented two cases. The *VAF_Rule_ProcessTransitions* rule is executed not exhaustively, but in a cycle. In each cycle, the rule flags the actually processed *Transition* edge, thus it cannot be matched again. As the following *VAF_Rule_PopEvents* rule does not change this flag (and that rule terminates as well), the cycle exits as soon as each transition has been matched exactly once.

As each rule in the transformation terminates, and the only cycle in the control flow exits after a finite number of steps, the entire transformation terminates as well.

5 Conclusion

In [MMC09] we have provided an approach and the languages capable of describing the dynamic behavior of visual languages. However, these languages alone are not enough to create a solution

providing dynamic animation for models. This paper presents a graph rewriting-based model transformation, which turns the behavior models into executable source code. By using this transformation, our final goal is now reachable. Moreover, by using a graph-rewriting based approach, we can analyze the properties of the transformation. Due to the limits of this paper we have analyzed only the termination properties of our solution, but with a little extension of the transformation we could also verify e.g. the liveness or other correctness properties of the input state machines. The presented approach is not specific to VMTS or to the C# language, but can be easily adopted to any other modeling environments using a similar DEVS-based simulation engine.

Bibliography

[AL09] L. Angyal, L. Lengyel. Synchronization of Textual and Visual Representations of Evolving Information in the Context of Model-Based Development. In *In Proceedings of the IEEE Eurocon 2009 Conference*. Pp. 468–443. St Petersburg, Russia, May 2009.

[COD] Microsoft CodeDOM website. http://msdn.microsoft.com/en-us/library/650ax5cx.aspx

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin, illustrated edition edition, 2006.

[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.

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

[MV02] P. J. Mosterman, H. Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. *ACM Trans. Model. Comput. Simul.* 12(4):249–255, 2002.

[MV04] P. J. Mosterman, H. Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation: Transactions of the Society for Modeling and Simulation International, Special Issue: Grand Challenges for Modeling and Simulation* 80:433–450, 2004.

[VLM02] H. Vangheluwe, J. de Lara, P. J. Mosterman. An Introduction to Multi-Paradigm Modeling and Simulation. In *In Proceedings of the 2002 Conference on AI, Simulation and Planning in High Autonomy Systems*. Pp. 9–20. Lisboa, Portugal, 2002.

[VMT09] VMTS Team. Visual Modeling and Transformation System website. 2009. 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.