Engineering the Dynamic Behavior of Metamodeled Languages

Tamás Mészáros
Gergely Mezei
Hassan Charaf
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Goldmann György Tér 3, IV. em, H-1111, Budapest, Hungary
mesztam@aut.bme.hu

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. The aim of this paper is to contribute (i) an event-based conceptual architecture to support animation, (ii) a set of visual languages to describe the animation of the models and their execution. These concepts were used to implement the animation support in our tool called Visual Modeling and Transformation System (VMTS). The VMTS animation framework introduces novel languages to describe certain aspects of animation, while integrating the benefits of the existing approaches. Our solution clearly separates the domain knowledge and the animation description both on a conceptual and implementation level. Thus, the VMTS offers a concise and systematic solution to provide a highly customizable animation framework for metamodeled languages with strong integration support to external systems such as simulation engines. The efficiency of the approach is illustrated with a rather complex animation case study implementing a model transformation debugger.

Keywords: modeling and simulation environments, simulation languages, simulation system architecture

1. Introduction

Domain-specific modeling (DSM) has gained increasing 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 layout – usually needs additional modeling techniques.

Multi-paradigm modeling [1–3] is a special, straightforward way to apply DSM 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 orthog-

SIMULATION, Vol. 85, Issue 11/12, Nov./Dec. 2009 793–810

DOI: 10.1177/0037549709102812
Figures 1–4, 6, 8, 9, 11–13, 15 appear in color online:
http://sim.sagepub.com

onal directions of research [3]: (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 discussed in Section 2 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 (see Section 2 for details) 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.

The motivation of our work was to provide 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.

2. Related Work

The system that mostly influenced our approach is AToM^3 [4], which is a general purpose metamodeling environment with simulation and model animation features. AToM^3 models can be animated either by handling predefined events or by model transformation. Visual elements provide several events (e.g. CREATE, CONNECT, MOVE) which can be handled by implementing their handlers in Python. Using this approach, one can define the dynamic behavior of model elements. Event handlers are defined in the metamodels. The event handlers override the event handlers of the possible parent element in the inheritance hierarchy in the metamodel, thus, the inherited handler is executed in case of a missing child level implementation. Another possibility for model animation is executing graph transformation on the model and updating the visualization based on the modifications performed by the transformation. Note that graph transformations can also be executed as a response to the previously mentioned events. AToM^3 provides a visual designer to build graph-rewriting-based transformations. AToM^3 also supports user interface modeling to a certain extent: custom toolbars can be attached to metamodels, the toolbars are visible if an instance of the metamodel is edited. The buttons of the toolbar are modeled, and their event handler methods are implemented in Python. Compared to AToM^3, VMTS [5] can model the static layout of the editor application which can be applied at runtime. Model elements in VMTS can also react to events similar to the ones defined in AToM^3, but we do not define these event handlers in the metamodel but in the plugin which defines the concrete syntax. Our approach also focuses on the visualization and modeling of the animation logic, instead of implementing it by hand. VMTS users are not restricted to the application of a predefined set of events. The set of the applicable events is extendable, and complex behavior can be described with the help of event-based state machines (Section 4.3) instead of handling one single event at the same time. In contrast to AToM^3, VMTS separates attribute modeling and presentation: visualization can be animated independently from the underlying model, however, data binding [6] facilitates the connection of them as well. AToM^3 provides language-level support for the integration of external components which can be called from Python. The main benefits of our approach are flexibility and extensibility: external components can be adapted to the animation framework, the integration of external components and frameworks can be modeled, and they are wrapped into a unified event-based interface.

Ptolemy II [7] is a modeling and simulation environment with a history of more than 10 years. It provides numerous predefined models of computation to create the models in, including, but not limited to Discrete Event, Continuous Time, Distributed Discrete Event, Finite State Machine and Process Networks. The strength of Ptolemy is that models created in different domains can be integrated in a hierarchical way. Most domains in Ptolemy II (except for the Finite State Machine domain) have a dataflow characteristic. The results of a simulation can be visualized on plotters of several types. Plotters are controls that can visualize data in various formats including continuous and discrete diagrams as well as textual controls. The content of plotters is updated implicitly based on the values received on its input ports. Ptolemy II provides a flexible simulation framework for a wide range of domains and their combinations, however, the animation of the models must be written manually. In contrast, VMTS focuses on the animation of models and the modeling of the animation in a visual way. However, external simulation engines (such as Ptolemy) can also be adapted. In the VMTS Animation Framework, one may consider two different models of computation. One is used to describe the animation logic with communicating event-based state

machines; the other is the one specific to the animated domain implemented in event handlers (Section 4.2) or external simulation engines wrapped with event handlers. The base elements of Ptolemy models are the *actors*. Actors are communicating, concurrently executed components. Actors can also be implemented in JAVA by subclassing an existing actor class instead of modeling, thus, they can embed arbitrary components and provide their services through the unified interface of actors.

MATLAB/Simulink [8] is a complex modeling and simulation environment which also facilitates the combination of models from different domains. Most Simulink domains are based on either continuous or discrete time dataflow languages. MATLAB models can be animated using proprietary textual language. All the graphical elements in MATLAB have a handle, which can be used to reference the owner element. The properties of the referenced objects can be changed through the MATLAB API. The visualization of the elements is updated automatically on changing to the model properties. Similarly to AToM^3, nodes (*block*s) in Simulink models also provide a predefined set of editing events, which can be handled by writing *Callback Routine*s using the MATLAB Programming Language. The result of model simulations is usually visualized with the help of *plotters* (similar approach to the one presented in connection with Ptolemy II). However, MATLAB plotters have to be updated explicitly through the API without notification support when the underlying data changes. Compared to VMTS, MATLAB does not support the visual definition of model animations; furthermore, automatic change notification is not provided in all cases either. The MATLAB Programming Language also provides basic constructs to load and invoke methods of external components (binary libraries, and Windows COM components). Furthermore, custom applications can also use the MATLAB engine through its programming interface.

MetaEdit+ [9] 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.

The Transformation-Based Generation of Environments [10] (TIGER) tool generates visual editor plugins for Eclipse [11] from typed grammar-based visual language specifications. The generated plugins are based on the Eclipse Graphical Editing Framework (GEF) [12], which is an open source infrastructure for creating and using graphical editors based on Eclipse. In contrast to VMTS Presentation Framework (VPF) [13], the underlying architecture of GEF is the Model-View-Controller (MVC) [14] pattern. Both the manageable data (model), and the visualization (view) and the interaction features (controller) are separated into different classes. The model classes can be arbitrary Java classes (POJO), however, the controller classes should be derived from the common *EditParts* class. The visualization can be performed by an arbitrary Java class by following the restriction that it should implement a predefined interface (*IFigure*). GEF itself does not support automatic change-notification services, due to the underlying arbitrary model object. Notification support for model property changes is provided by the Eclipse Modeling Framework (EMF) [15] which is used by Tiger to define the model layer of the MVC pattern. Similarly to AToM^3, animation of models in TIGER can be achieved by performing graph transformation on the model, and due to the MVC architecture, the applied modifications are also reflected in the visualization. In contrast to VMTS, this approach requires the visualization properties to be modeled. The application of data binding in VMTS offers a finer grained approach compared to TIGER, as the notification service is implemented not only between the model and the view (and the controller) but also between specific properties of the model and the view, and these relations can easily be influenced with converters one-by-one. TIGER provides visual modeling techniques to define graph transformation rules, and uses Attributed Graph Grammar System (AGG) [16] to execute algebraic graph transformations. In contrast to TIGER, VMTS also focuses on the modeling of user interaction events, and the modeling of actions answered to these events.

The Generic Modeling Environment (GME) [17] is a general purpose metamodeling and program synthesis environment. GME provides two ways to animate models: (i) with the help of executing graph rewriting-based model transformations or (ii) by transforming models with custom traversing processors. Both approaches build on the fact that on updating model elements in the object space (MGA-MultiGraph Architecture) of GME, the presentation layer is notified about the changes, and the visualization of elements also updates. The GReAT toolkit [18] of GME is used to model graph rewriting-based transformations in a visual way and to execute the transformations. GReAT uses the Universal Data Model (UDM) data access layer over MGA to manipulate model elements. MGA is responsible for notifying subscribed components about changes. In addition to visually modeled transformations, arbitrary transformation can be implemented by hand using the UDM and the Builder Object Network (BON) interfaces. The graph transformation-based approach of GME to animate models is similar to

the one presented in connection with AToM^3 and TIGER, but – similarly to TIGER – GME does not support modeling user interactions. The visual modeling of animations is limited to the description of a closed graph rewriting algorithm. The visual modeling of adapting external components to GME is also not supported.

DiaMeta [19] is a framework for generating graphical diagram editors. It uses hypergraph grammars to specify visual languages. Editors generated with DiaMeta are capable of running online structural and syntactic analysis on the edited models. DiaMeta also employes EMF to define visual languages. Thus, it provides a similar approach to the one presented in connection with TIGER: change notification events are generated by the underlying EMF model on attribute changes, and the generated view and controller objects update the visualization as response to these events. In contrast with TIGER, DiaMeta uses custom classes for model visualization instead of utilizing GEF.

The presented tools use some kind of constraint checking, but for very different purposes. For example GME uses OCL constraints for model verification, GReAT uses constraints to refine rewriting rules (similarly to TIGER and VMTS). MetaEdit+ uses constraints to refine the user interface update mechanism (similarly to converters in VMTS). In addition, VMTS uses constraints as guard expressions in the animators. Table 1 summarizes the animation features of the previously mentioned tools.

3. Background

Visual Modeling and Transformation System [5] 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 illustrates the architecture of VMTS. Models are stored in a model repository (relational database), and can be edited through the Attributed Graph Supporting Inheritance (AGSI) interface. The AGSI layer provides the wrapper classes to create and manipulate models, model elements and their attributes. The VMTS Presentation Framework (VPF) [13] is a Document-View-based [14] presentation layer responsible for visualizing model elements and attributes.

The Object Constraint Language (OCL) [20] module is used for both model validation and for the refinement of the graph rewriting [21] rules for the general purpose Rewriting Engine (RE). The RE of VMTS can be used to describe transformations between arbitrary models. The

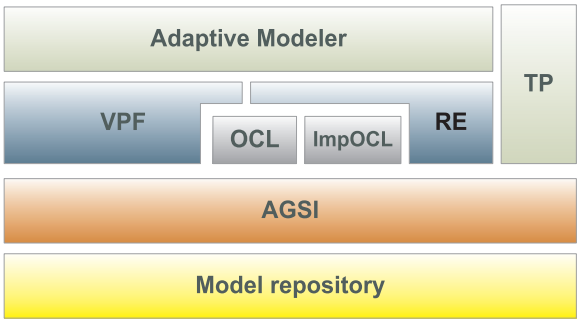

Figure 1. The architecture of VMTS

operations (insert/delete/update) performed by a rewriting rule are defined using the Imperative OCL language [22]. The Imperative OCL compiler transforms Imperative OCL code to executable C# code. Adaptive Modeler is the front-end application of VMTS and provides the services of the underlying layers in an integrated way. VMTS also facilitates the generation of metamodel-based class hierarchies to support building external tools for model traversing and processing via the Traversing Processor (TP) module.

3.1 Visual Language Definition

Figure 2 illustrates the process of visual language definition. The metamodel defines the possible elements and the relations between the elements. By instantiating the metamodel, we can create models using the defined language rules. VMTS also supports the concrete syntax definition using modeling [23]. The VMTS Presentation DSL (VPD) model assigns a specialized visualization to each model element, and it is able to model the presentation of simple attributes. Using the VPD model and the metamodel, a plugin can be generated that follows the metamodel rules and uses the defined concrete syntax at the same time. Using this approach, we are able to design new languages in an efficient way, however, we were not able to describe the dynamic behavior of each language element or that of the entire model.

4. An Animation Framework

As a result of our work, we provide an integrated solution for model and user interface animation, embedded in the VMTS modeling environment. The strength of our approach is that we support the definition of animations with visual languages on both low-level event definition and on high-level animation description.

Table 1. Comparison of modeling environments supporting animation

	VMTS	AToM^3	Ptolemy II	MetaEdit+	Simulink	TIGER	GME/GReAT	DiaMeta
Model animation	+	+	limited	+	+	+	+	+
Graphical notation of animations	proprietary DSL	graph rewriting rules	–	–	–	graph rewriting rules	graph rewriting rules	–
Simulation engine	Graph Tr. DEVS [19], under development	DEVS (DAE, DEV&DESS), Graph Tr.	Proprietary engine	–	MATLAB Simulation Engine	AGG (Graph Tr.)	GReAT (Graph Tr.)	–
Script language	C#	Python	Java, Ptolemy II expression language	SOAP XML, own script language	MATLAB Programming Language	Java	C++	–
Model-View synchronization	parametrizable with converters	+	–	parametrizable with constraints	limited	+	+	+
Supporting various framework integration	Visual models, Event wrappers, generating skeleton	binary level	wrapped with actors	binary level	binary level	binary level	binary level	binary level
Constraint checking	rewriting rules, guard expressions, UI update	–	–	UI update	–	rewriting rules	model verification, rewriting rules	–

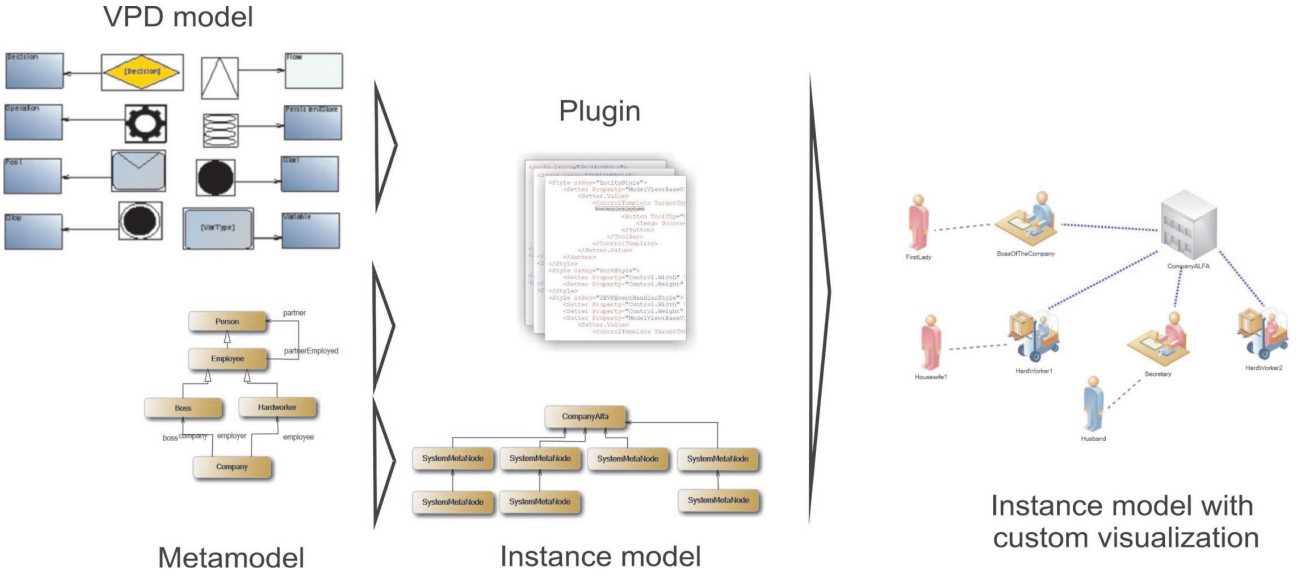

Figure 2. Visual language definition process

4.1 Architecture

Figure 3 illustrates the architecture of the animation framework. It highlights the elements of Figure 1 relevant for the animation framework (*AGSI*, *VPF*, *Adaptive Modeler*), and extends it with new components implementing domain-specific animations (*Animation engine*, *Animation Business Logic*). The *Modeling space* block represents the model repository and the data access objects for creating and editing models and model elements.

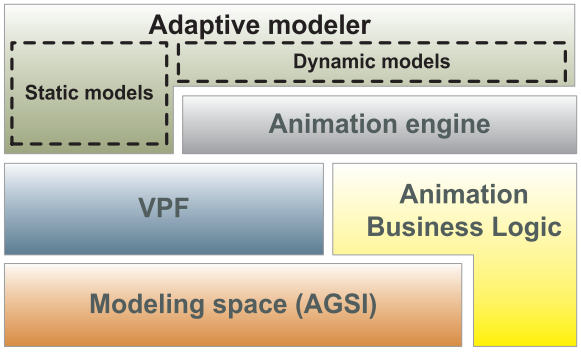

Figure 3. Animation framework architecture

The VPF is a graphical modeling layer that is based on the Windows Presentation Foundation [6] framework and applies its data-binding mechanism to keep the visualization and the underlying model synchronized. Any change performed in the model (e.g. made by model transformation) is immediately reflected in the visualization as well. Data-binding can be manipulated with the help of *converters*. Converters are used to transform the data represented by model element properties into another or an aggregated form before presenting them on the screen. The visualized data can be – but is not limited to – alphanumeric text, color, position information, arbitrary transformation or even visibility property. The domain knowledge, the business logic of the animated model, is implemented in the *Animation Business Logic* component (Section 4.2). Examples for this component are simulation engines and other services such as the Runge-Kutta methods for solving ordinary differential equations or the algorithm of performing algebraic graph transformation. The component may operate on VMTS models, and it may also communicate with external components and applications such as MATLAB. The *Animation engine* defines the reactions to events coming from the business logic and the VPF components. It updates model and visualization properties based on the messages coming from the business logic, and it may also instruct the business logic according to the user input events from VPF. *Static* and *dynamic* models are the concrete applications of the presentation framework extended with the animation engine.

We have designed three visual languages to model the animation of the models and the user interface. Figure 4 summarizes the languages of the animation framework and their application. The animation logic can be described with a state machine (called *Animator* in our terminology) that communicates via events (*Animator state machine* block). The environment of the state machine (referred to as *ENVIRONMENT*) including the user interface (VPF), the model repository (AGSI) and the domain-specific extensions (Animation Business Logic in Figure 3) are also wrapped with event-based interfaces, called event handlers (*Event handler model*). Event handlers convert the events of the environment to modeled events, therefore providing the input for the animation. An event handler model defines the events the handler can interpret and their parameters (*entities*). The default implementation of an event handler can be generated, but the handler methods of the specific events have to be implemented manually (*Event handler implementation* block). 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. The source code generated at runtime from the animation model relies on a DEVS-based (Discrete Event System Specification) [24] simulation framework (*Animation engine* in the figure), which schedules and processes events in a deterministic order. Due to the universality of DEVS, our approach is also capable of animating continuous systems and to adapt continuous simulation engines as well. In addition to manipulating the user interface with events, the static layout of the opened windows can also be modeled with a user interface modeling language (*UI model*). UI models can then be interpreted and applied programmatically.

4.2 Events and Event Handlers

The focus of our approach is the events. Events are the exclusive facilities to keep the connection between the animation engine and its environment, including the user interface and the underlying model. Events are generated when interacting with the user interface (e.g. pressing a mouse button) and events are also used to establish a connection between the animation engine and the domain-specific extension components for defining model semantics. Events can be either built-in – already provided by the animation framework – or custom. As it is shown in Section 4.3, the animation can be described using a graphical extended finite state machine formalism. The inputs and the outputs of the state machine are events.

Event handlers define the connection point between the animation logic and its environment. The purpose of event handlers is to wrap the environment of the animation logic into an event-based interface. The methods and properties of the wrapped object are hidden, and can be reached only through events. The event handlers and the events that they can interpret are tightly related. The definition of events and event handlers is supported by a visual language whose metamodel is depicted in Figure 5.

The meta-element of the event handler (*EventHandler*) defines two attributes for its instances: *CodeBehind* and *Parameter*. The *CodeBehind* attribute is used to reference a binary class library (dll). This class library contains the implementation of event handler methods for the events of this handler, the event interpreter and the possible initialization and destruction methods. The skeleton of the *CodeBehind* file is automatically generated based on the

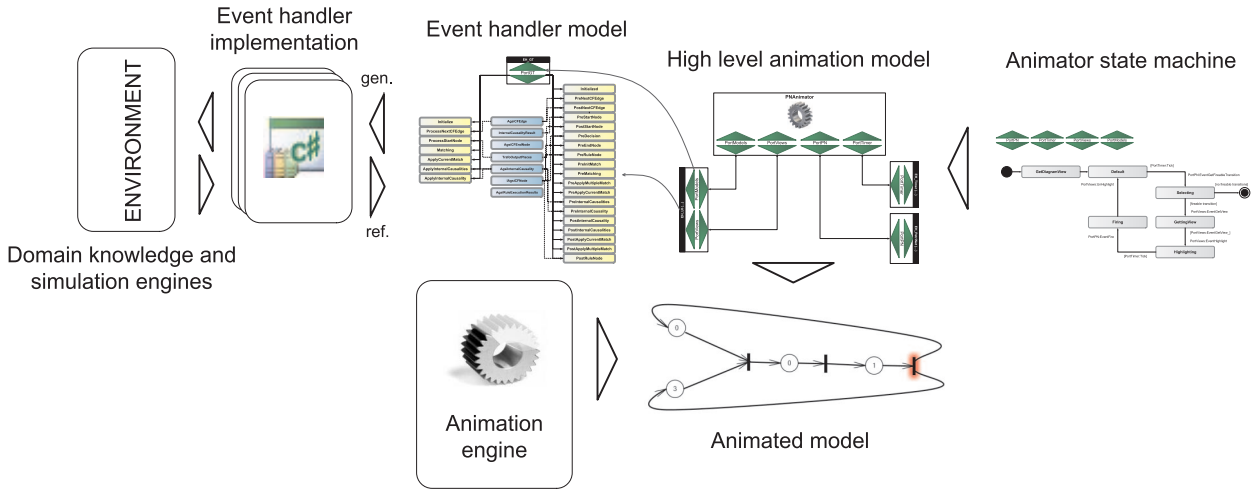

Figure 4. Overview of the animation languages

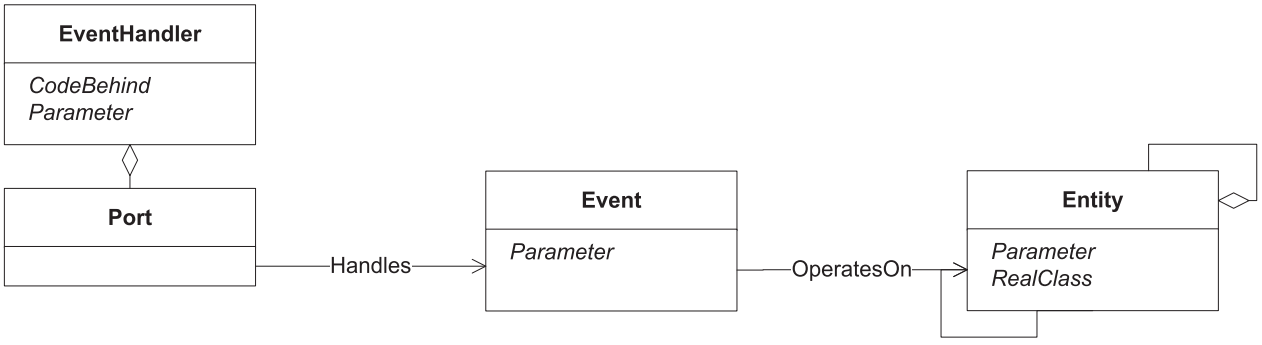

Figure 5. Event handler metamodel

event handler model; the developer only has to complete the body of the event handler methods. Architecturally, the implementation of these methods is the part of the Animation Business Logic (introduced in Figure 3). Custom parameters can also be assigned to event handlers using the *Parameter* attribute. Event handler instance models can only define the possible parameters and their default values, but not their concrete values, as they are determined during a concrete application of them (Section 4.3). For example, the *Timer* event handler fires a *Tick* event periodically, after a specified time elapses. This interval is the parameter of the event handler, and its value is not known by the design time of the event handler, because it depends on its application. The *Parameter* attribute is a compound attribute, meaning that it has subfields: *ParameterName*, *ParameterType and ParameterDefaultValue*. In case of the *Timer* event handler, *ParameterName* is set to '*frequency*', *ParameterType* is set to '*int*', and *ParameterDefaultValue* is set to 500 (msec).

The communication between the event handlers and the animation engine is *port*-based. Ports (the *Port* element in Figure 5) define the connection points for an event handler: we have the possibility to define several ports for the same event handler, and distinguish the handled events or the way in which we handle a specific event based on the receiver port. Ports and events (defined by the *Event* element) are coupled with the instances of the *Handles* edge.

Events can be customized with the help of *Entities*. Entities are supposed to be the parameters of events. After the code generation, both entities and events are mapped to real C# classes. The fields of the entity classes are modeled with the help of the *Parameter* attribute, which defines the name and the type of the referenced parameter. Instead of code generation, entities can also be bound to already existing types, which is useful if we have to interact with existing software components. Entities also support to be connected with other entities with the help of relations already known from UML class diagrams, in-

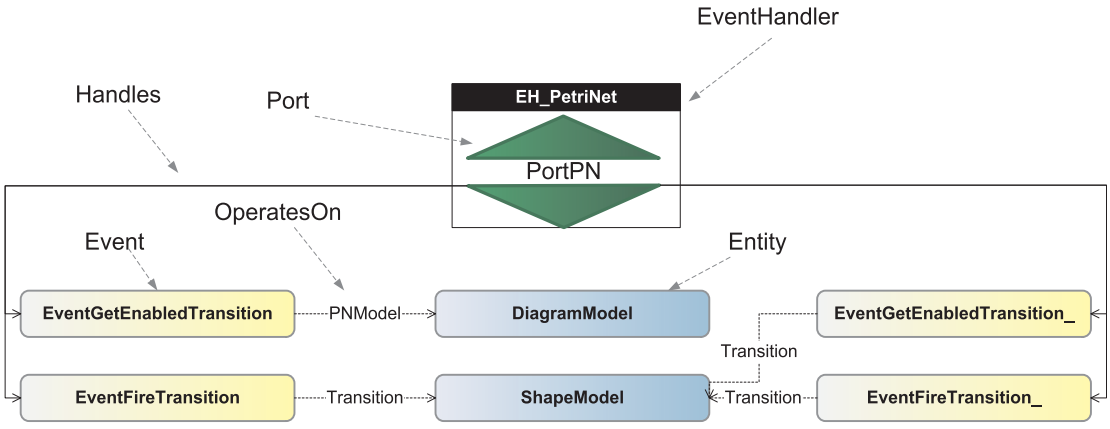

Figure 6. Petri-net event handler model

cluding containment, generalization and association. The generalization edge is mapped to class-level inheritance, while the association and containment edges result in the creation of member variables in the entities. Relations with *many* multiplicities are implemented with the help of generic lists. An entity can be assigned to an event with the *OperatesOn* edge. Entity references in an event are also implemented with member variables and generic lists. The name of the generated member variable matches the name of the *OperatesOn* edge instance.

An event handler instance model with a concrete syntax can be seen in Figure 6. The figure depicts an event handler for Petri-net models. The event handler provides one port (called *PortPN*) for sending and receiving events. *EHPetriNet* can handle four types of events, namely *EventGetEnabledTransition*, *EventFireTransition*, *EventGetEnabledTransition_*, and *EventFireTransition_*.

EventGetEnabledTransition is sent to the event handler to obtain a transition to be fired from a specific Petri-net model. The subject model has a type of *DiagramModel* (a VPF entity representing the diagram, this is an already existing type in VPF), and is referenced with the *PNModel* edge. This edge is mapped to a member variable during code generation. During the processing phase of *EventGetEnabledTransition* events, the event handler searches the subject model for an enabled transition, and responds an *EventGetEnabledTransition_* event parametrized by the transition found (it has a type of the external *ShapeModel* type, which is the document component in the Document-View architecture for shapes in VPF).

The event handler is manually implemented in the *CodeBehind* file, conceptually it belongs to the Animation Business Logic component, because it must incorporate the special knowledge how Petri-nets work. Similarly, the *EventFireTransition* event instructs the event handler to modify the container model of the referenced transition (*Transition* parameter) by firing the transition, and the event handler fires an *EventFireTransition_* event on the completion of this operation.

4.3 Animation Description Language

We have designed another visual language for the definition of the animation logic. This language describes which event handlers to use during the animation, and what actions to perform after receiving a specific event.

The animation description language is usually not intended to describe the simulation of a model in a specific domain – the simulation business logic is more likely to be defined in an external component wrapped with an event handler. The animation description language is rather used to define the transformation between events of the simulation business logic and the events of the user interface. The described intelligence converts complex simulation events into rudimentary user interface events or, on the contrary, collects events and produces complex events towards the simulation business logic.

Figure 7 depicts the metamodel of the animation language. The language can be used to create state machines which can react to events and fire new events. Recall that the state machine component is called *Animator*.

An animator may contain input and output ports (*Port* instances), which define the interface of the animator. The ports in our approach are buffers, which means that they may store a predefined number of events until processing them. The *Buffer* attribute defines the maximum length of the FIFO-based waiting queue. The *Circular* Boolean attribute specifies what the policy is, when the buffer is full and a new event arrives. If the *Circular* attribute is set to false, the incoming event is dropped. However, if it is set to true, the oldest element in the queue is removed, and the incoming one is enqueued into the buffer.

The applied event handlers are represented in the animation model as well. The *EventHandler* objects on this

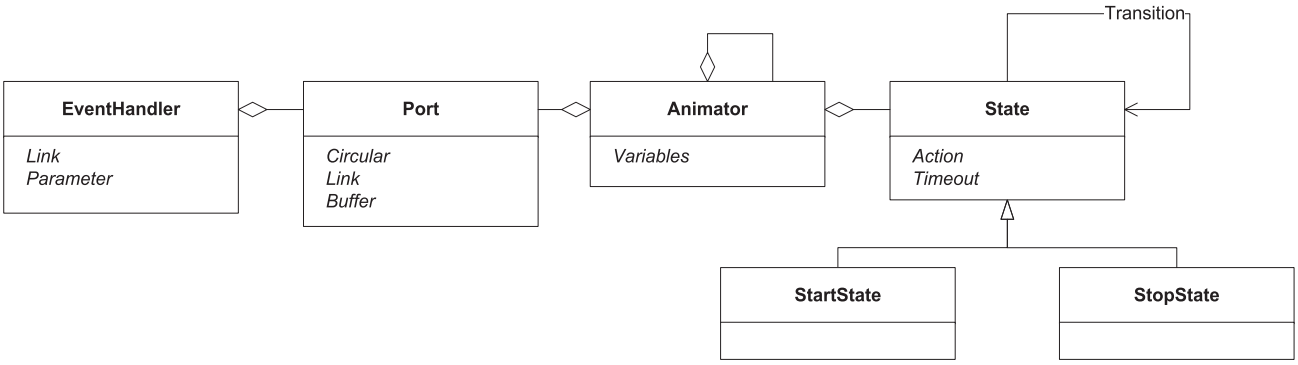

Figure 7. Metamodel of the animation language

level provide a *reference* to event handlers modeled in the event handler models. The coupling between the reference and the modeled event handler is established using the *Link* attribute of the element reference. The *Link* attribute contains the unique identifier of the event handler in the event handler model. Event handlers in the animation models also contain ports to facilitate bi-directional communication with them. Similarly to the event handlers, the ports contained by an event handler reference also serve as a reference to the real ports of the real event handler models: the *Link* attribute of the ports points to the ports in the event handler model. Using the event handler references and ports in the animation models, we can create customized instances of the event handlers already modeled. The *Parameter* attributes of the event handler reference elements are automatically generated after selecting the *referenced* event handler model, based on the same attributes of the referenced element. With the help of the *Parameter* attributes of the event handler reference, we can customize the predefined set of parameters of the event handler. The *Parameter* attribute defines the *ParameterName* and *ParameterValue* fields. The value of the *ParameterValue* field is filled with a default value set in the *ParameterDefaultValue* field of the original *Parameter*.

Two ports (contained by either an event handler or an animator) can be connected by *EventRoute* edges. The existence of an *EventRoute* edge between two ports indicates that the outgoing events of the port on the left side of the edge are forwarded to the port on the right side. If the *Direction* property of the edge is set to *BOTH*, the events from the right side are also forwarded to the left.

In Figure 8, an animator and three event handlers can be seen. The user interface event handler (*EH_UI*) and the timer event handler (*EH_Timer*) are provided by the framework, while the implementation of the domain-specific event handler (*EH_Petrinet*) is generated based on the model depicted in Figure 6, and the event handler methods are implemented manually. For example, the *EH_Petrinet* and *PortPN* model elements reference the elements with the same name in the model depicted in Figure 6. As the model in Figure 8 has an attribute reference to the underlying event handler implementation, the implementation can also be referenced from the code generated from the animation model. The ports of the animator and the event handlers are connected by *EventRoute* edges exactly specifying the way of the event streams.

Recall that an animator element wraps a state machine. The most fundamental element of this state machine is the *State* node (instance of the *MetaState* node). The transition between two states is described by the *Transition* edges. There are two special types of states: the *Start* state and the *Stop* state. The state-space of the state machine is defined not only by the contained state nodes. The animator may also have local variables (*Variables* attribute). Furthermore, the actual state of the ports also affects the number of possible states of the state machine.

A *Transition* edge between two states defines a guard and an action expression (*Guard* and *Action* properties), both implemented by program code currently. The guard condition must be able to be evaluated to a Boolean value. It may test the ports of the animator and its local variables. The guard conditions of the outgoing transitions of the actual state are tested every time a new event is enqueued in either port of an animator. If a guard condition can be satisfied, then the topmost events of all ports tested by the actual guard condition are removed. The contents of the non-tested ports are not modified, provided that they have a maximum possible buffer size larger than 0. Ports with a buffer of 0 length cannot store events: such ports are emptied even if no active transition has occurred. If a guard condition can be evaluated to *true*, the state transition is performed, and the next actual state will be the target state of the transition edge. When performing a state transition, the action expression of the transition edge is executed before removing the topmost element of the referenced ports. The action expression can contain three types of operations: the instantiation, customization and firing of an event on a specific port, or the manipulation of local variables, or the mixture of these operations.

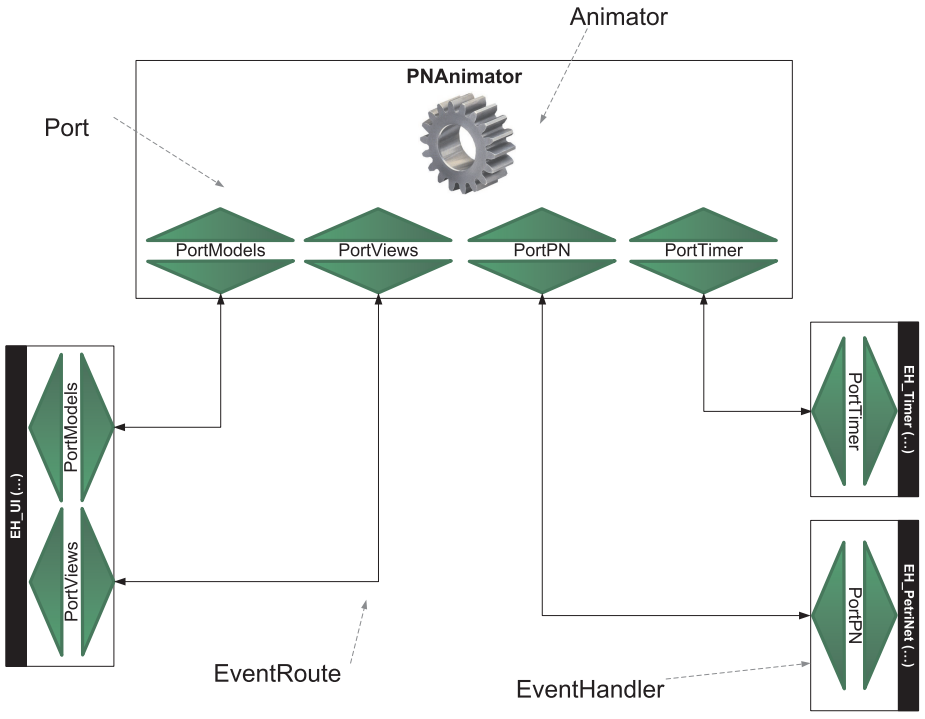

Figure 8. A high-level animation model

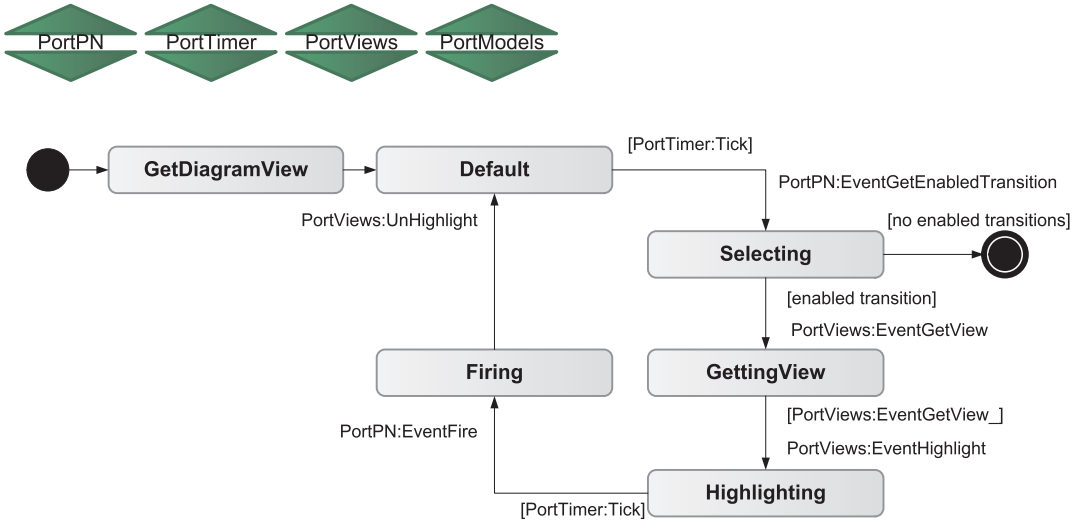

Figure 9. Petri-net animator

Each state has an *Action* property as well, which may contain a program code similar to the code stored in the same attribute of the *Transition* edges. This code snippet is executed after running the *Action* script of the active *Transition* edge, but still before removing the topmost elements of the tested ports. Figure 9 depicts an example animator which animates Petri-net models. The presented animator represents the internals of the animator depicted in Figure 8. In fact, the two figures visualize different aspects of the same model according to the metamodel illustrated in Figure 7.

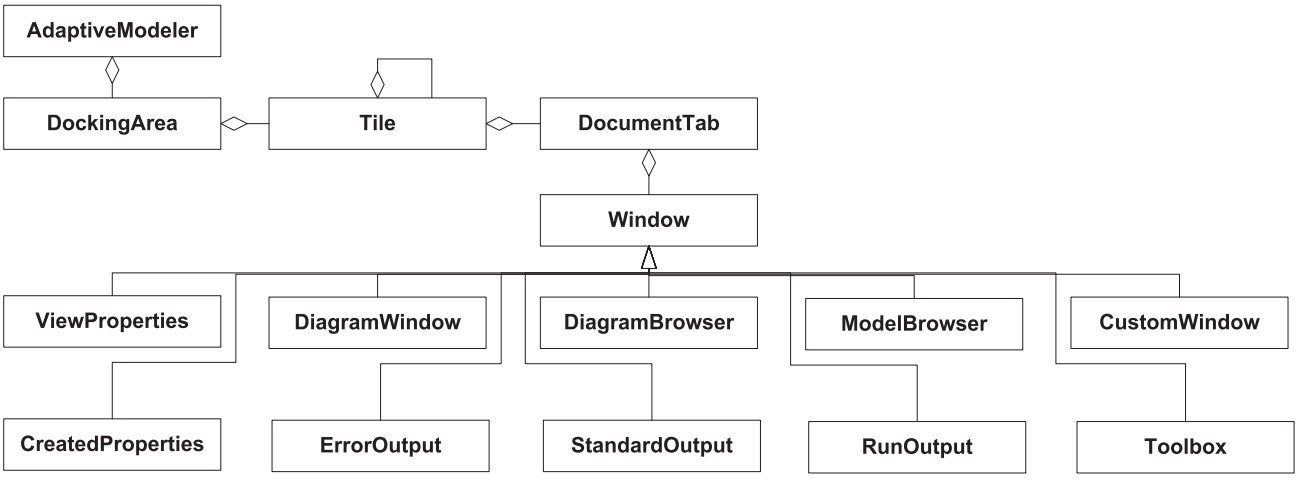

Figure 10. Adaptive Modeler user interface metamodel

4.4 The Animation Engine

The animation model introduced in Section 4.3 is transformed to executable C# source code by traversing the model. For this purpose, we use a handwritten model processor. The events fired by the animators and the event handlers are processed by a DEVS simulator. After firing events, they are enqueued into a global ordered event queue, and are processed in a synchronized way. Each event has a timestamp that represents the logical time to be processed at. This timestamp is used to keep the queue ordered. By default, the timestamp is set to the global simulation time, which equals to the timestamp of the last processed event. The timestamp can be set to an arbitrary value after instantiating an event. If two events have the same timestamp, they are ordered based on a unique identifier which is generated based on the instantiation order. Events defined by event handler models can be considered as *external event*s in the DEVS terminology. To provide a full DEVS simulator implementation, we also support *internal events*, which can also be modeled in the animation model. If no external event is received after a predefined time (*TimeOut* property of the *State* nodes) in a state – there is no enqueued event in the global event queue with a timestamp smaller than the current simulation time increased by the value of the *TimeOut* property – an internal event is fired, and the global simulation time is increased by *TimeOut*. An internal event does not cause any event to be enqueued in the ports of the animator, it only activates the animator, and instructs it to examine the transitions of the actual state again, but only the transitions whose *IsInternal* attribute is set to *true.* Internal transitions can have exactly the same guard conditions and action expressions as the external ones.

The presented state machine implementation provides a deterministic behavior in sense of producing the same output event sequence for the same start state and input event sequence. However, the determinism of the entire animation also depends on the determinism of the adapted external components. In case of real-time events (including timer events and user inputs) the deterministic behavior is not necessarily guaranteed.

4.5 User Interface Modeling

When executing a model animation, it is often important to apply a specific window layout on the modeling environment or to open several models as a startup operation. For example, when we execute a graph transformation in debug mode (detailed in Section 5), three models have to be opened by default (control flow model, host graph, output graph), and another window contains the actually executed rewriting rule. To support the visual modeling of the user interface (UI), we have designed a UI modeler language. Its metamodel is depicted in Figure 10.

The visual modeler application of VMTS is called Adaptive Modeler; it is represented by an *AdaptiveModeler* element. The Adaptive Modeler window is partitioned into four separate docking regions (*DockingArea*): left, right, middle and bottom. The contained visual elements are docked to one of the enumerated sides of the main window.

A *DockingArea* can be partitioned horizontally and vertically by adding *Tiles* to it. A *Tile* can contain either a *DocumentTab* element or two other *Tile*s which split the container *Tile* into horizontal or vertical slices.

The *DocumentTab* represents a tab group with several pages. A *DocumentTab* can contain *Window*s. There are several types of predefined windows (e.g. ViewProperties, RunOutput, Toolbox), and two special types: *DiagramWindow* and *CustomWindow*. The *DiagramWindow* models a view of a model; it can reference an arbitrary diagram in the model repository via its *DiagramID* attribute.

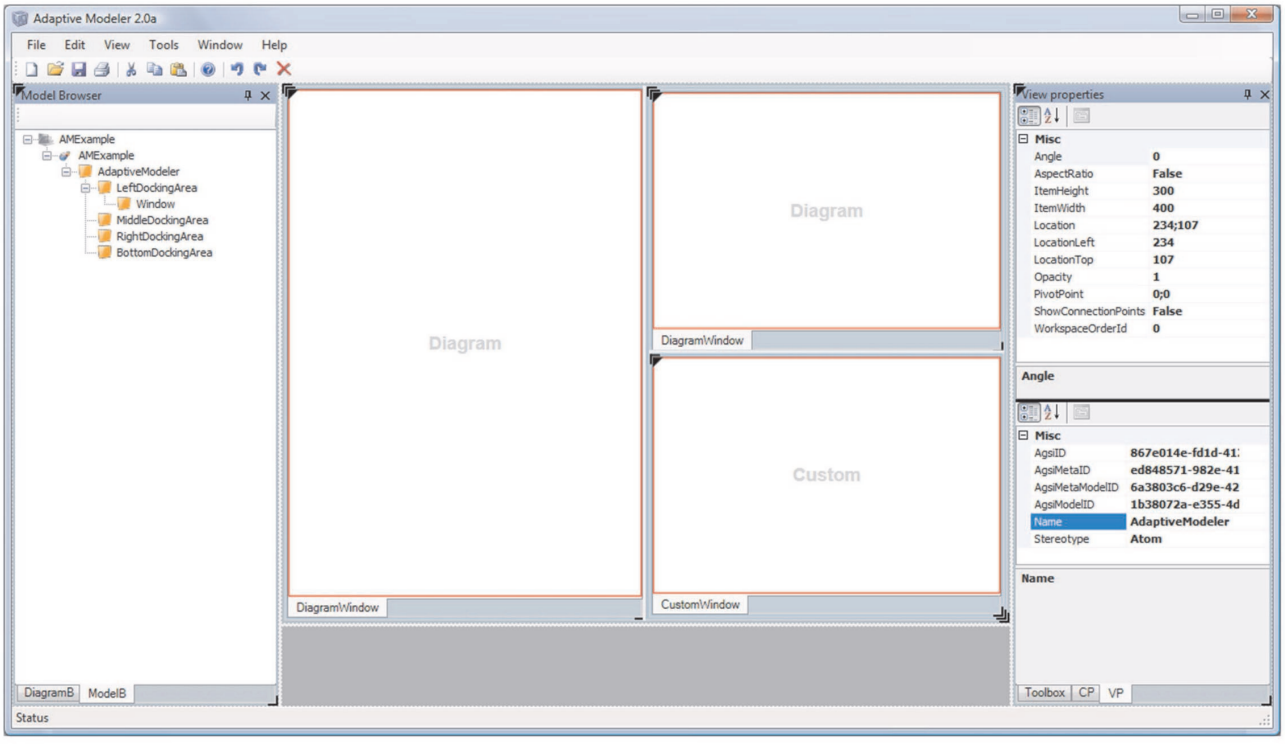

Figure 11. Adaptive Modeler user interface instance model

The content of diagram windows can also be changed using the *SetDiagramWindowContent* event of the UI event handler. *CustomWindows* are empty by default; their content can be set programmatically after executing a user interface model. Custom windows are the extensibility points of the UI model: new windows can be created, such as plotters, graph windows, setting panels. *Window*s have an attribute called floating, which indicates that a window is not docked to the main window, but is floating and freely repositionable.

User interface models can be applied at runtime by sending an *EventApplyLayout* event to the UI event handler. The event should be parametrized with the UI model. The event handler traverses the referenced model and applies the same layout to the running application instance.

Figure 11 illustrates the concrete syntax of the Adaptive Modeler UI model. The left *DockingArea* contains a *ModelBrowser* and a *DiagramBrowser* element, the right *DockingArea* contains a *Toolbox*, a *CreatedProperties* element, and a *ViewProperties* element, the bottom *DockingArea* is empty. The middle *DockingArea* contains a *DocumentArea*, and that contains a *Tile*. This *Tile* contains two other *Tile*s that partition it vertically, the second tile is partitioned horizontally again with two *Tile*s. Each *Tile* that does not contain other *Tile*s, contains a *DocumentTab* element. Finally, each *DocumentTab* contains either a *DiagramWindow* or a *CustomWindow*.

5. A Case Study

5.1 Background

In order to illustrate the usability of our approach, we have designed a model transformation debugger solution applying the presented animation framework. 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 is a powerful technique for applying graph transformations with a strong mathematical background. Graph transformation is based on 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: firstly the model items of LHS, which are not in the intersection of LHS and RHS are removed, then the model items of RHS, which are not in the intersection, are added (glued) to the input graph. VMTS has a graph transformation engine supported by visual languages to define transformations as well. LHS patterns can be defined using the metamodel of the models. Consequently, instead of an isomorphic match in the source model, we search

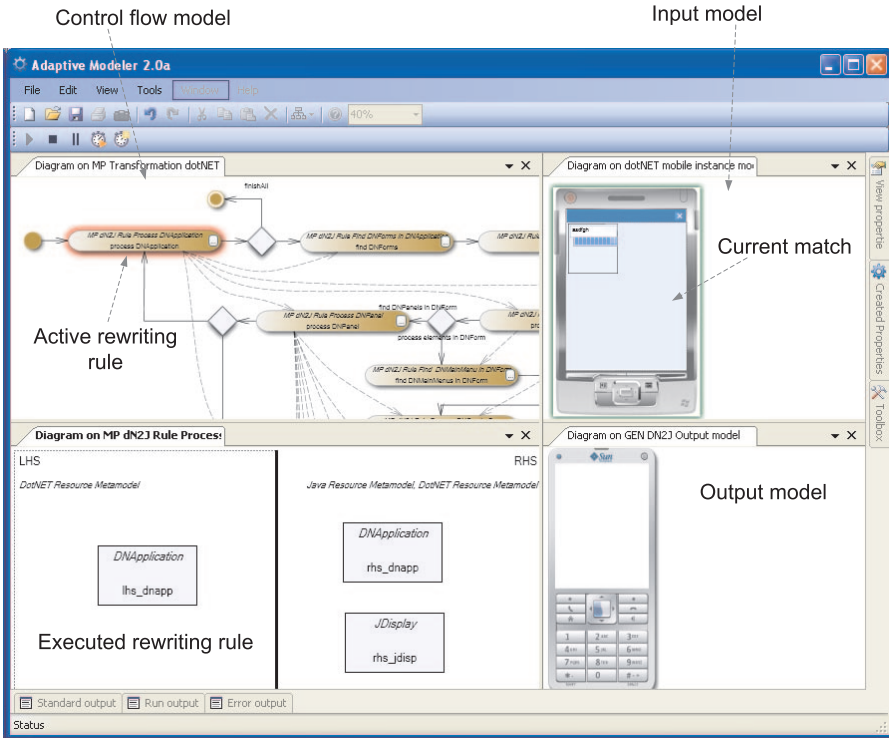

Figure 12. Window layout of the model transformation debugger

for a subgraph that can be created by the instantiation of the LHS. Editing graph rewriting rules is supported via the Rule Editor plugin of VMTS. With the help of that plugin, one can design the LHS and the RHS of a rule in a visual way. The execution order of rewriting rules can be defined with the help of the Visual Control Flow Language [25].

The motivation behind building a debugger for the model transformation solution in VMTS was the difficult traceability of the transformation process. A transformation described by a control flow model may contain numerous rewriting rule applications and decision points. The user requires feedback about the state of the transformation process. Our purpose was to visualize the entire model transformation process, including the animation of the control flow model to be able to trace the state of the transformation. Furthermore, the actually executed rewriting rule is also presented and the possible successful match in the host graph is highlighted as well. After the execution of a rewriting rule, the newly created elements are added to the view of the output model, and the deleted elements are removed. The animation can be executed automatically or step-by-step.

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 OCL condition. The *External causality* edge can declare that an element of the LHS of a rule matches another element of the RHS of another rule.

The operation described by a rewriting rule is called *internal causality* in our terminology. There are four types of causalities: *identify*, which declares that an element of the RHS of a rule matches an element of the LHS of the same rule; *delete*, which deletes a specified subset of nodes matched on the LHS; *create* and *modify*. The *create* and *modify* causalities are defined using the Imperative OCL language. These causalities are appropriate for creating new or modifying existing elements in the output model.

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, and (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 is applied. Figure 12 illustrates the model transformation debugger in operation, the executed trans-

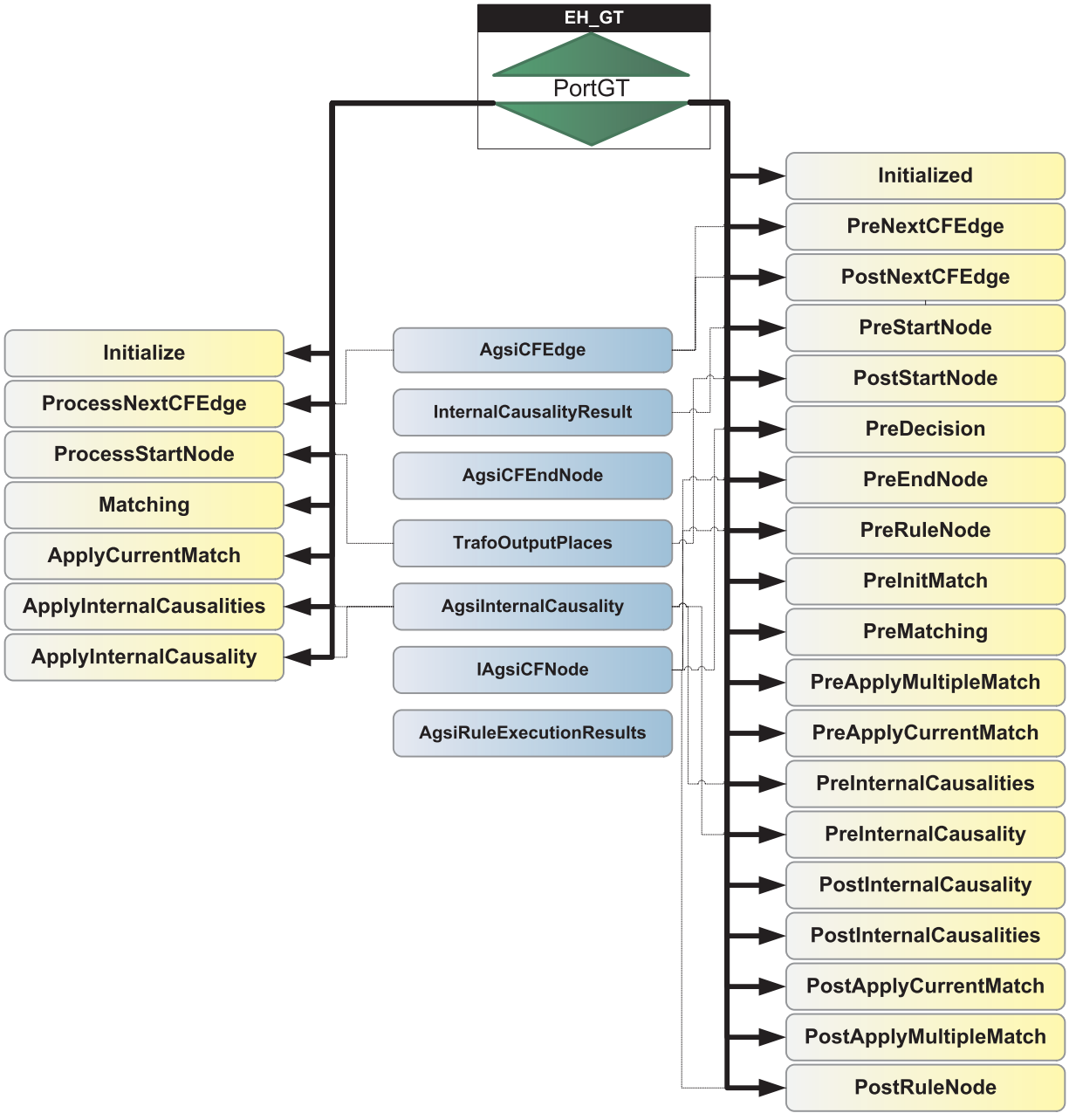

Figure 13. Graph Transformation event handler

formation transforms.NET mobile user interfaces to JAVA user interfaces. The control flow model, the rewriting rule, the output model, and the input model are opened.

The highlight around the first rule node in the control flow model indicates that the rule is currently executed. The rewriting rule in the lower left corner belongs to the highlighted rule node. The input model can be seen in the upper right corner, and the highlight around the mobile device represents a match for the actual rewriting rule. The output model in the lower right corner contains an element which is created by the already executed first rule.

5.2 Event Handler Model

The transformation engine is wrapped with the event handler depicted in Figure 13. On the left-hand side of the figure, the events sent to the event handler are depicted.

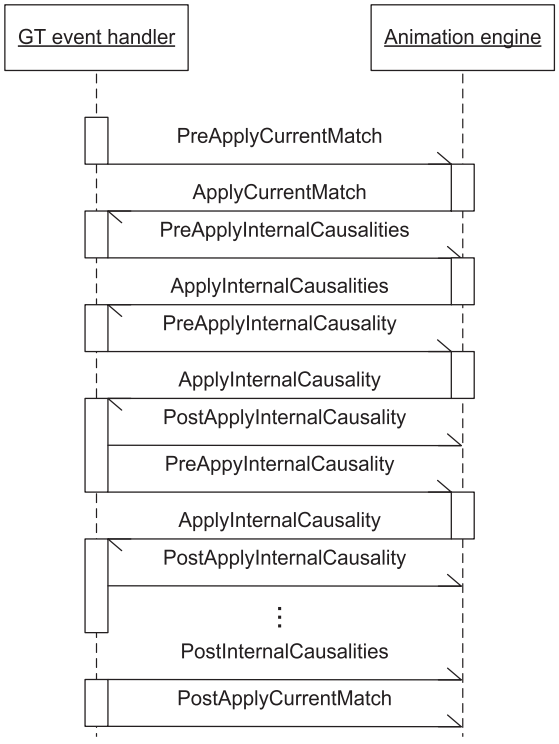

Figure 14. Causality-execution event flow

On the right-hand side there are the events that are sent by the event handler. In the middle, the entities used as parameters by the events can be seen. The events usually begin with the 'Pre' or 'Post' prefix (except for *Initialized*). These messages indicate the beginning and the completion of a sub-process in the transformation. For example, the *PreInternalCausalities* event is sent after the matching and before the application of the internal causalities. The event contains a list of the applied causalities. In order to continue the transformation process, an *ApplyInternalCausalities* event, which references the causalities to be applied, must be sent to the event handler. At this point, we can shorten or widen the list of causalities received from the *PreInternalCausalities* event. After sending the *ApplyInternalCausalities* event, a *PreInternalCausality* event is sent before processing each internal causality edge. The execution of a causality edge can be initiated by an *ApplyInternalCausality* event parametrized with the same *AgsiInternalCausality* entity as the *PreInternalCausality* event. After the execution of a specific internal causality, a *PostInternalCausality* event is sent by the event handler, and finally, a *PostInternalCausalities* event is sent after processing all causalities in the same rewriting rule. The event flow presented above is illustrated in Figure 14.

5.3 User Interface Model

A transformation usually operates on four models at the same time: (i) the input model, (ii) the output model, (iii) the visual control flow model, and (iv) the currently applied rule. Using the UI modeling language, we have designed the layout depicted in Figure 12 for visualizing the transformation process. The screen is divided into four segments for the four diagrams. Concrete models cannot be assigned to the segments at the time of creating the UI model, as the models are specified only when executing a transformation with a concrete configuration. The *EventSetDiagramWindowContent* event of the UI event handler can be used to assign a model to a window at runtime. This event references the model to be opened and the name of the window (*dwCF*, *dwTarget*, *dwRule*, *dwHost*) to be used to show the model. The layout itself can also be applied using the UI event handler, namely its *ApplyLayout* event. The parameter of this event is the layout model.

5.4 Animation Model

Figure 15 illustrates the logic that defines the animation of model transformations. On the left-hand side, the high-level animation model is depicted. The animator uses the services of three event handlers: (i) *EH_UI* (user interface manipulation), (ii) *EH_GT* (graph transformation business logic), and (iii) *EH_Timer* which fires a *Tick* event for every 1 sec. The tick event is used to give a real-time scheduling to the animation. If a *Timer* event handler is applied in the running animation, an additional toolbox appears in the Adaptive Modeler toolbar, which can be used to pause the real-time timer or to manually fire its *Tick* event. Thus, we can pause the continuous running of the animation and run it step-by-step. For the sake of simplicity, the capacity of the *PortModels*, *PortLayout*, *PortViews*, *PortTimer* and *PortGT* ports is set to 1, and their *circular* property is set to *true*. The buffer size can be 1, as we know that none of the event handlers produces two events as a response to a single query, or the events can be overwritten if they are not processed in the actual state. For instance, the *PostInternalCausality* event can be overwritten by the next *PreInternalCausality* event, as we do not handle the *PostInternalCausality* event at all. The *Action* property of the states of the presented state machine is used in none of the cases, firing new events and setting local variables are performed only on the execution of transitions.

The first transition fires an *Initialize* event through the *PortGT* port to initialize the transformation framework. The input parameters – including the applied control flow graph and the input model – of the transformation environment are set from procedural code. In the following four states (*ApplyLayout*, *OpenHostAndCF*, *GetCFDiagram* and *GetHostDiagram*), the layout definition shown in Figure 12 is applied, the control flow and input models

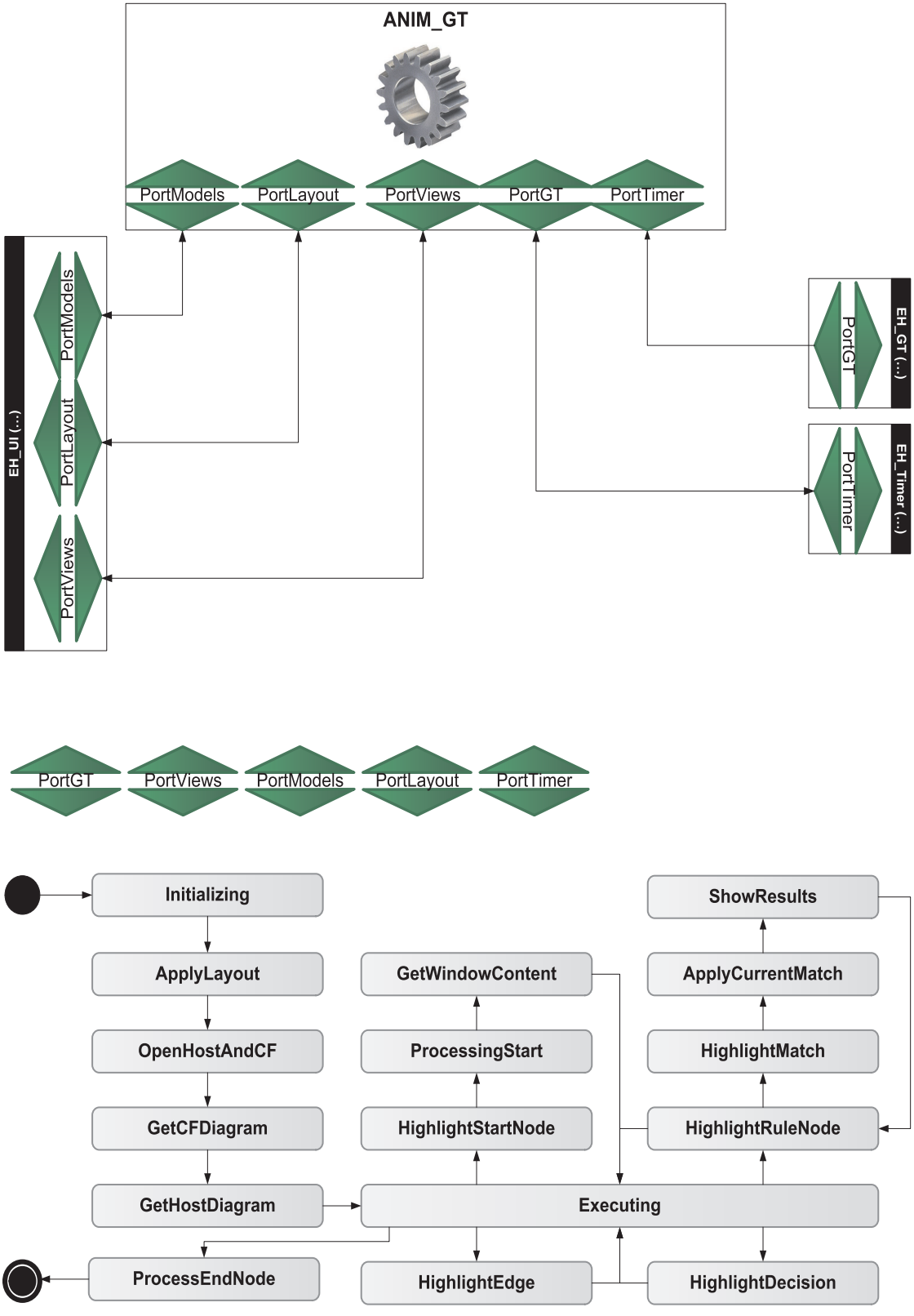

Figure 15. Animation model of model transformations

are opened, and the reference to the opened models is obtained and stored in local variables for further use. The *Executing* state can be considered as a base state, which is active before processing a new element of the control flow graph. The outgoing transitions of the *Executing* state trigger on the *PreXXXX* events of the specific model elements, namely on *PreStartNode*, *PreEndNode*, *PreDecision*, *PreCFEdge and PreRuleNode.* For example, the transition before the *HighLightStartNode* has the following guard condition:

```
PortGT.PeekIsOfType< EH_GT.PreStartNode>()
&& PortTimer.PeekIsOfType< EH_Timer.Tick>()
```

The condition above checks whether the first element in the *PortGT* port is of type *PreStartNode*, and the *PortTimer* port contains a *Tick* event.

The usual scenario behind processing control flow elements is highlighting them by sending an *EventHighlight* message to the UI event handler, then sending the appropriate processing event to the GT event handler, finally removing the highlight. In the case of the start node this scenario is extended by querying and opening the output model, as the output model is created after the processing of the *Start* node. The processing of *Rule* nodes is also divided into separate phases illustrated in Figure 14: (i) highlighting a rule node, (ii) highlighting a match, (iii) performing the rewriting (*ApplyCurrentMatch* state), and (iv) visualizing results (*ShowResults* state). The exhaustive execution of a rule may result in multiple matches for the same rule, that is why an edge points back from the *ShowResult* state to the *HighLightRuleNode* state, so a following match can be applied.

6. Conclusion

The key features of the animation environment of VMTS were presented in this paper. In contrast to other tools, we make considerable effort to model and visualize the animation logic as much as possible. We have provided an architecture concept, where the animation is separated from the domain-dependent knowledge. The event-based integration of the separated components incorporating the domain knowledge is supported via visual modeling techniques. We have contributed three visual languages to describe the dynamic behavior of a metamodeled model, and their processing via an event-based 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 and the domain-specific extensions are wrapped with event handlers. Communication with event handlers can be established using events. Event handlers can also be used to integrate external components, simulation engines into our environment. The definition of event handlers is supported with a visual language, and automatic generation of the event handler implementation is also supported. We have designed another visual language to describe a state machine which defines the animation logic itself. The state machine consumes and produces events. The transitions of the state machine are guarded by conditions testing the input events and fire other events after performing the transition. On executing an animation, the state machine is transformed into executable source code. The events produced by the event handlers and the state machines are scheduled and processed by a DEVS-based simulator engine. The modeling of the static layout of the editor application is also supported by a visual language. The interpretation and application of layout models can be achieved through events as well. We have illustrated the feasibility of the presented concepts by (i) implementing them in the VMTS Animation Framework, and (ii) building a Petri-net simulator and a graph rewriting-based model transformation debugger.

Our approach is particularly useful when describing the dynamic behavior of languages, which can be easily metamodeled. Languages hard to describe with metamodels, such as sequence diagrams, and languages which have a complex concrete syntax with, for example, complex layout algorithms and hardcoded visualization logic, are also hard to animate. These cases usually need the handling of an increased number of states and events. However, increasing the complexity of the animators is not recommended after a certain size because of the lack of hierarchical animators. Such cases can be handled by the implementation of the animation logic partially in domain-specific event handlers, increasing the amount of hand-written code. Our approach assumes that simulation engines to be integrated provide access or at least feedback about their relevant internal operations, otherwise the adaptation of such external components is difficult.

Future work includes the decreasing of C# code needed to be written when building a state machine, both when defining guard conditions and firing new events. The introduced concepts are language-independent, however, we would like to replace C# with imperative OCL where possible to become language independent on implementation level as well. Current research also focuses on the tighter integration of the model transformation engine into the animation framework to describe the operations performed by events with the help of graph rewriting rules. The described operations would include both model manipulation and the firing of response events. We would also like to search and collect common patterns in the state machines to build a design pattern repository for the animation domain.

7. Acknowledgement

The activities described in this paper were supported, in part, by Information Technology Innovation and Knowledge Centre.

8. References

[1] Mosterman, P.J., and H. Vangheluwe. 2002. Guest editorial: Special issue on computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation*, 12(4):249–255.

[2] Mosterman, P.J., and H. Vangheluwe. 2004. 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(9):433–450.

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

[4] de Lara, J., and H. Vangheluwe. 2002. AToM^3: A Tool for Multi-Formalism and Meta-modeling. In *Fundamental Approaches to Software Engineering*, *LNCS*, 2306:174–188.

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

[6] Nathan, A. 2007. *Windows Presentation Foundation*. Indianapolis: Sams Publishing.

[7] Brooks, C., E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. 2007. *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*, Technical Report No. UCB/EECS-2007-129, Berkley: University of California.

[8] MATLAB home page. http://www.mathworks.com

[9] Tolvanen, J-P. 2006. 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*, Portland, 2006, pp 690–691.

[10] Ehrig, K., C. Ermel, S. Hänsgen, and G. Taentzer. 2005. Generation of Visual Editors as Eclipse Plug-Ins. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, Long Beach, pp 134–143.

[11] Eclipse home page. http://www.eclipse.org

[12] Moore, W., D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. 2004. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks.

[13] Mészáros, T., G. Mezei, and T. Levendovszky. 2008. A Flexible, Declarative Presentation Framework For Domain-Specific Modeling. In *Proceedings of the 9th International Working Conference on Advanced Visual Interfaces*, 2008, Naples, pp 309–312.

[14] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester: John Wiley and Sons Ltd.

[15] Budinsky, F., D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. 2003. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional.

[16] Taentzer, G. 2004. AGG: A Graph Transformation Environment for Modeling and Validation of Software. *Application of Graph Transformations with Industrial Relevance*, 3062:446–453.

[17] Lédeczi, Á., Á. Bakai, M. Maróti, P. Völgyessi, G. Nordstrom, J. Sprinkle, and G. Karsai. 2001. Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11):44–51.

[18] Karsai, G., A. Agrawal, F. Shi, and J. Sprinkle. 2003. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, *Special issue on Formal Specification of CBS*, 9:1296–1321.

[19] Minas, M. 2006. Generating meta-model-based freehand editors. *Electronic Communications of the EASST Proceedings of the 3rd International Workshop on Graph Based Tools*, 2006, Natal.

[20] Warmer, J., and A. Kleppe. 2003. *Object Constraint Language: Getting Your Models Ready for MDA*, Second Edition. Addison Wesley Professional.

[21] Ehrig, H., K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. New York: Springer Verlag.

[22] Object Management Group. 2008. *MOF QVT Final Adopted Specification*. http://www.omg.org/docs/formal/08-04-03.pdf

[23] Mezei, G., T. Levendovszky, and H. Charaf. 2006. A Model Transformation for Automated Concrete Syntax Definitions of Metamodeled Visual Languages. In *Electronic Communications of the EASST*, *2nd International Workshop on Graph and Model Transformation*, 2006, Brighton.

[24] Zeigler, B.P., T. Gon Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation*, Second Edition. Orlando: Academic Press.

[25] Lengyel, L., T. Levendovszky, G. Mezei, and H. Charaf. 2005. Control Flow Support in Metamodel-Based Model Transformation Frameworks. In *Proceedings of the EUROCON 2005 IEEE International Conference on "Computer as a tool"*, 2005, Belgrade, pp 595–598.

Tamás Mészáros *is a PhD student at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics, Budapest, Hungary.*

Gergely Mezei *is a full-time Senior Lecturer at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics, Budapest, Hungary.*

Hassan Charaf *is a full-time Associate Professor at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics, Budapest, Hungary.*