Tooling the Dynamic Behavior Models of Graphical DSLs

Tihamér Levendovszky and Tamás Mészáros

Budapest University of Technology and Economics,
Department of Automation and Applied Informatics,
Goldmann György tér 3, H-1111 Budapest, Hungary
{tihamer,mesztam}@aut.bme.hu
http://www.aut.bme.hu

Abstract. Domain-specific modeling is a powerful technique to describe complex systems in a precise but still understandable way. Rapid creation of graphical Domain-Specific Languages (DSLs) has been focused for many years. Research efforts have proven that metamodeling is a promising way of defining the abstract syntax of the language. It is also clear that DSLs can be developed to describe the concrete syntax and the dynamic behavior. Previous research has contributed a set of graphical DSLs to model the behavior ("animation") of arbitrary graphical DSLs. This paper contributes practical techniques to simplify our message handling method, automate the integration process, and show where domain-specific model patterns can help to accelerate the simulation modeling process.

Keywords: Domain-Specific Modeling Languages, Metamodeling, Simulation.

1 Introduction

Domain-specific modeling is a powerful technique to describe complex systems in a precise but still understandable way. The strength of domain-specific modeling lies in the application of domain-specific languages to describe a system. Domain-specific languages are specialized to a concrete application domain; therefore, they are particularly efficient in their problem area compared to general purpose languages. Rapid creation of graphical Domain-Specific Languages (DSLs) has been focused for many years. Research efforts have proven that metamodeling is a promising way of defining the abstract syntax of the language. It is also clear that DSLs can be developed to describe the concrete syntax.

VMTS [1] is a general purpose metamodeling and model transformation environment. The visualization of models is supported by the VMTS Presentation Framework (VPF) [2]. VPF is a highly customizable presentation layer built on domain-specific plugins that can be defined in a declarative manner. The VMTS Animation Framework (VAF) [3] is a flexible framework supporting the real-time animation of models both in their visualized and modeled properties.

When we started using VAF, we encountered several simplification and support opportunities that make VMTS a more efficient and user-friendly tool that is able to visualize the simulation of third party tools with metamodeling techniques. This paper contributes practical techniques to simplify our message handling method, automate the integration process, and show where domain-specific model patterns can help to accelerate the modeling process.

1.1 VMTS Animation Framework

VAF separates the animation from the domain-dependent knowledge of the dynamic behavior. For instance, a 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-based communication. The architecture of VAF is illustrated in Fig. 1.

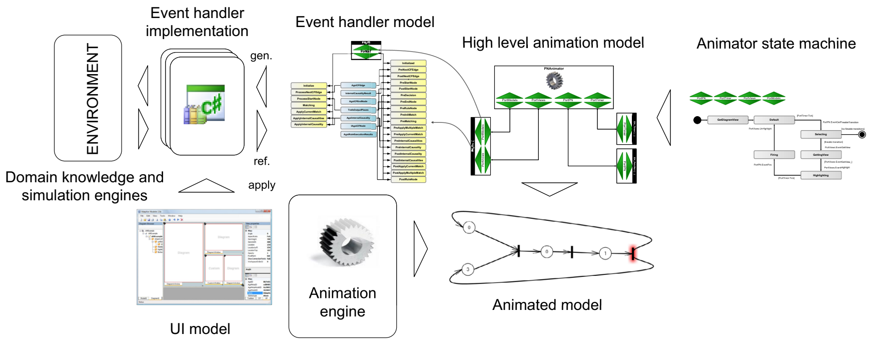


Fig. 1. Architecture of the VMTS Animation Framework

The animation framework provides 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 parameterizable 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 Fig. 1) are wrapped with event handlers, which provide an event-base interface. Communication with event handlers can be established using events. The definition of event handlers for custom components is supported with a visual language (*Event handler model* block). The visual language defines the event handler, its parameters, the possible events, and the parameters of the events - called *entities*. Based on the model, the skeleton of the event handler can be generated (*Event handler implementation* block). We have already implemented the elementary event handlers to wrap the functions of the modeling environment, however, in case of integrating a third-party component, the glue code between the component and the skeleton has to be written manually.

The animation logic can be described using an event-based state machine, called *Animator*. We have designed another visual language to define these state machines (*Animator state machine* block). The state machine consumes and produces events. The transitions of the state machine are guarded by conditions testing the input events and may fire other events after performing the transition. The input (output) events of the state machine are created in (sent to) another state machine or an event handler. The events produced by the event handlers and the state machines are scheduled and processed by a DEVS-based [4] simulator engine (*Animation engine* block). The event handlers and the state machines can be connected in a high-level model (*High level animation model* block). The communication between components is established through ports. Ports can be considered labeled buffers, which have a configurable size. On executing an animation, both the high-level model and the low-level state machines are converted into source code, which is executed after an automated compilation phase. In addition to model animation, another visual language is provided to design the layout of the user interface, which can be applied when executing a model animation (*UI model* block).

2 Automated External Tool Integration

A shortcoming of our previous solution was that the integration of external components (including simulation engines and third-party components) required too much work compared to using their API themselves. To integrate a component, we needed to create the event handler model including all the events for the components and we also needed to write the glue-code between the generated event handler and the component by hand. This is not a too intuitive task and is also error-prone. If the interface of the adapted component changes for some reason, the changes had to be propagated to the event handler model by hand, and the glue-code also had to be updated. To address this issue we decided to use reflection to load the metainformation of the component and to generate the event handler classes automatically, including both the event implementations and the glue code. There are several types of .NET class-members we can assign events to. The most trivial one is the .NET *event*. .NET events are language-supported solutions to implement some kind of notification service. One can subscribe delegates (roughly: function pointers) to the published events of an object, and they can be called by the publisher object using the Observer design pattern [5]. If a .NET event is fired, the event handler should catch it, and fire the appropriate animation event with the right parameters (received from the .NET event) through a port. If we consider the *MouseClicked* event of the *Control* object in the .NET Base Class Library, it has two parameters: the first is called *sender* and has a type of *object* – this parameter identifies the sender of the event –, the second one is called *e* and is of type *MouseEventArgs* – it identifies the location of the click, the pressed button etc. To wrap this .NET event we need to (i) create a modeled event with the same type of properties as the parameters of the event has (*object*, (ii) subscribe to the appropriate event of the inspected object with a callback method, (iii) instantiate the modeled event and fire it through a port if the callback method is called. A simple member method can also be considered both an event source and a sink as well: a method call can be initiated by firing an event to the event handler, the parameters of the event should correspond to the parameters of the called method. The possible return value of the method call can be propagated towards the animation logic using an event as well. For example we assume a *GetMatrix* method having a parameter of type *string* and a return value of type *double[,]*. The method can be wrapped with two events: *GetMatrix* and *GetMatrix_*. The *GetMatrix* event has a property of type *string* (as the corresponding method also has such a similar parameter), while *GetMatrix_* – the response event sent by the event handler – has a property of type *double[,]*, which corresponds to the return value of the wrapped method. .NET *properties* can be considered getter/setter methods for private fields or calculated data. However, instead of function calls, we usually access the hidden data with the syntax of accessing member fields. In fact, properties are implemented with special methods in the background, but the language hides them to the programmer. Thus, properties require three events to be created: (i) one to set the value of the property (similar to a method call without return value), (ii) one to query its value (sent to the event handler component) and (iii) another one to return the value in response to the query (sent by the event handler component).

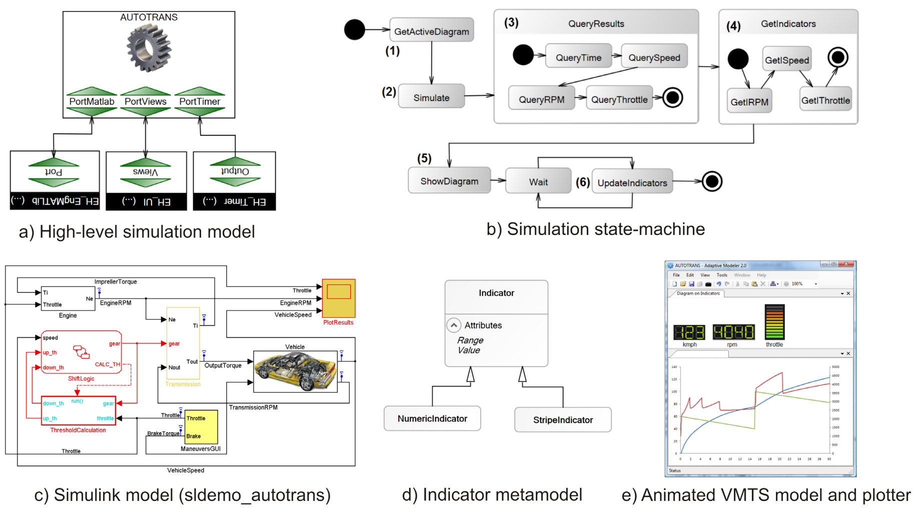
Instead of modeling the necessary events to wrap a component, we provide tools to automate the wrapping. In addition to define events in the event handler model manually, one can extend the event handler by adding *field*s to it. A field can be considered a member variable of the event handler, and the type of the field can be defined in an arbitrary external .NET assembly. Each field has the following attributes: *name*, *type*, *initField* and *members*. Setting the *initField* attribute to true indicates the instantiation of the field together with the instantiation of the event handler. The *members* attribute of the field defines the members of the field-type we would like to generate events and default event handlers for. Members are identified by name (*memberName* attribute), overloaded method names are distinguished with an order-id postfix. Each modeled member corresponds to an existing member of the type of the parent field. If a member is selected for generation, one, two or three events (depending on its type) are generated with parameters to wrap the properties of the wrapped member. The *portId* attribute of a *Member* identifies the port the generated events are sent or received through. Furthermore, we can also specify whether to generate the send/receive events using the *send* and *receive* attributes. This feature is useful if we do not want to take care of the return value of a method call, or, for example we would like to get the value of a property, but we would never like to set it directly. For example, if we would like to wrap a *Form* control, especially for its *ShowDialog* method and *MouseClicked* event, we need to add a *Field* of type *Form* to an event handler, add two *Member*s called *MouseClicked* and *ShowDialog* to the *Field*, set their *PortId* attribute to an existing port, and set their *Send* and *Receive* attributes to true – in fact, the *Send* attribute of members for .NET events is omitted. The presented procedure is also supported by a graphical user interface which automatically provides available type members and ports to create *Member* attributes

2.1 Integration of the MATLAB-Simulink Simulator

To illustrate the usage of our approach, we have integrated the MATLAB [6] API into VMTS, and we simulated a Simulink model and presented its results in VMTS. MATLAB does not have an official .NET-based interface, so we have used a free class library which wraps the most important MATLAB operations with a .NET class. The name of the component is *EngMATLib* [7]. The event handler model for *EngMATLib* contains a single *EventHandler* node which has a field called *matlab*, and its *type* is set to *EngMATLib.* We selected the *Execute* and *GetMatrix* methods of *EngMATLib* to generate events for. The *Execute* method is used to execute an arbitrary MATLAB command using its interpreter. It does not have a return value, and expects a single string parameter, thus the generator produces an event class called *Execute*, which has a string attribute. The *GetMatrix* method returns a floating point matrix identified by its *name* parameter. As the *GetMatrix* method has a return value, we generate both a *query* event (called *GetMatrix*) which has a *name* attribute, and a *response* event (called *GetMatrix_*) for the return value of the method call. Based on the performed settings the event handler implementation can be generated.

The simulated Simulink model (called *sldemo_autotrans*, Fig. 2 c) is shipped with the MATLAB installation, and models an automatic transmission controller. If we execute the simulation in Simulink, we can inspect the parameters of the car (including speed, engine rotation-per-minute, and throttle state) at different conditions. In the following example, we present these parameters on diagrams and also on monitors in VMTS. For this purpose, we have created a very simple visual language which can model indicators: numeric and stripe (Fig. 2 d). Each indicator has a *name* a *range* and a *value*, the modeled indicators are animated according to the results received from MATLAB.

Fig. 2 a and b present the high-and low-level (state machine) animation models. As it can be seen on the high-level animation model, the animator is connected to (i) the UI event handler (*EH_UI*), (ii) to the generated *EngMATLib* event handler and (iii) to a *Timer* which schedules the animation by sending a *Tick* event every 0.1 secs. The state machine model consists of the following steps: (1) query the active diagram, which contains the indicators to be animated (Fig. 2 e); (2) execute the simulation by sending an *Execute* event with a *"sim('sldemo_autotrans')"* command to MATLAB; (3) query the time, speed, RPM and throttle state vectors using *GetMatrix* on the signal logging structure called *sldemo_autotrans_output*; (4) obtain a reference to the indicators using their names; (5) present the data-vectors on a diagram; (6) animate the indicators by iterating through the vectors, and updating the *Value* property of the indicators based on the data-vectors. Note, that the loop is controlled by the *Ticks* of the timer event handler.



a) High-level simulation model
b) Simulation state-machine
c) Simulink model (sldemo_autotrans)
d) Indicator metamodel
e) Animated VMTS model and plotter

Fig. 2. Integrating Simulink simulations into VMTS

3 Hierarchical State Machines

Another shortcoming of our original solution was that in case of complex animation logics, the state machine became too large to be easily understood and to be modified. To overcome this drawback, we have extended the state machine formalism to support the modeling of hierarchical state machines. Fig. 3 depicts the metamodel of the actual state machine language. The containment loop-edge on the *State* node expresses the capability of building compound states. The execution of the extended state-machine is modified as follows: if a compound state becomes active, (i) its *Action* script is executed, (ii) the *Start* state of its internal state machine is activated, (iii) the internal state machine is executed, (iv) when reaching a *Stop* state in the internal state machine, the container state becomes active. If a nested state is active, both its and the container state's transitions are checked when the state machine is activated. If a nested state does not have any transitions which trigger the actual input, the transitions of the container

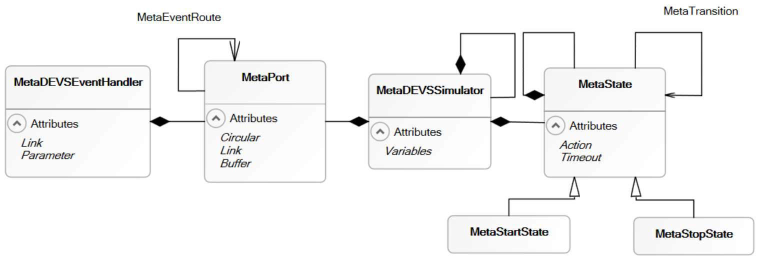


Fig. 3. State machine metamodel

state are verified, and - if any of them can be fired - the control is taken from the nested state, and returned to the container one. We have used nested states during the integration of Simulink as it can be seen in Fig. 2 b.

4 Domain-Specific Model Patterns

Software developers usually use a widespread UI framework and follow its guidelines when designing the user interface for their application. Therefore they usually met recurring cases when very similar components, or often exactly the same user interfaces, have to be designed. This situation is even more typical when someone is specialized for a certain kind of applications, e.g. mobile applications using a special platform. Using object oriented design techniques, one can handle these cases by applying design patterns [5]. We have extended [8] the idea of design patterns in VMTS to support not only UML based languages, but arbitrary metamodeled languages. VMTS also provides tool support for the rapid definition of design patterns, to organize them into repositories and to configure and insert patterns into an edited model. We have examined whether we can discover often recurring patterns in animation models and define them as animation design patterns. After creating numerous simulation models, we found that animation state machines are not quite appropriate sources to find patterns. This fact can be attributed to the language itself as well: the constraints of the transitions have to be defined in a textual way, and the firing of the events is also performed with action scripts. Furthermore, the integrated external components have their own type of events, and the specific state machines operate on these events. Instead of general patterns, one may discover recurring patterns during the animation of a specific domain.

The other two animation languages (the high-level animation language and the UI designer language - introduced in [3]) are better sources for pattern definition. Here we present only an outline of them. Two often recurring patterns for high-level animation models are presented in Fig. 4 a and b. The pattern in 4 a is the most elementary pattern which occurs in every animation model which provides visualization. The pattern contains an *Animator* and an *EventHandler* element, and their *Model* and *View* ports are connected with bi-directional *EventRoute* edges. Fig. 4 b can be considered the extension of 4 a with the notion of time by adding a *Timer* event handler to the model.

The VMTS UI designer language is used to describe the layout of the modeling environment during an animation. The designer language is very strict in the sense that most elements (windows in VMTS) have a maximal cardinality of 1. The two exceptions are the *DiagramWindow* and the *CustomWindow* components which model the presentation of a diagram or a custom edit control in the application. However, the number of the applied windows is limited by the size of the screen, as the more windows are presented, the smaller and less usable they are. Thus, the set of practically applicable layouts can be considered a small finite number. The two most common layouts are depicted in Fig. 4 c and d. In Fig. 4 c the most common windows (browser and property setter) are placed on the screen and one diagram is open. In Fig. 4 d the full screen is utilized: a diagram and the standard output windows are open. The usual scenario is that one can inspect the animation on the diagram, while the standard output presents trace information.

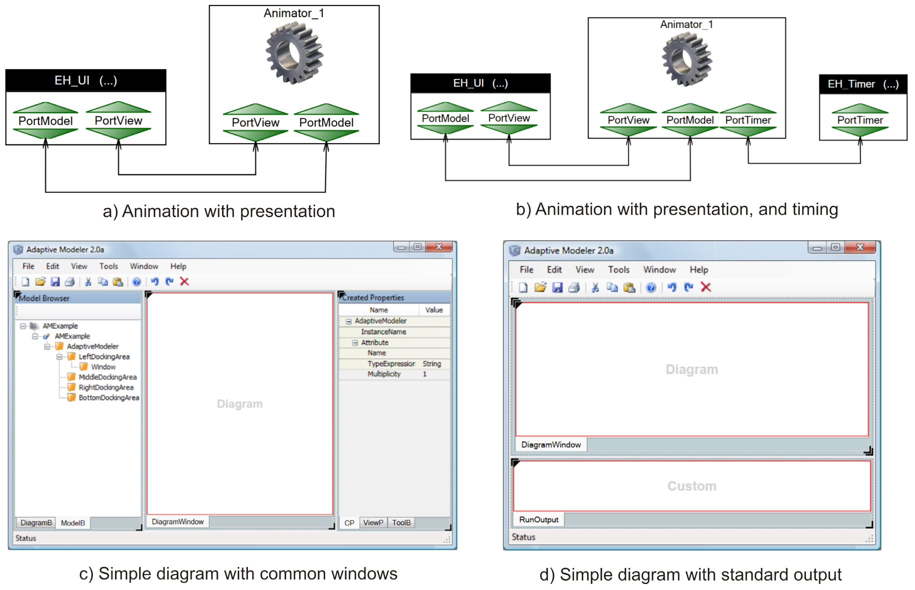


Fig. 4. Animation design patterns

5 Related Work

AToM3 [9] is a general purpose metamodeling environment with simulation and model animation features. AToM3 models can be animated either by handling predefined events fired during the editing of the model or by executing model transformation on the model and updating the visualization. Model elements in VMTS can also react to events similar to the ones defined in AToM3, 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. AToM3 provides language-level support for the integration of external components which can be called from Python. The main benefit of our approach is its 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.

The Generic Modeling Environment (GME) [10] 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. GME supports external component integration on binary level. One can embed components providing a COM interface into GME, however, the visual modeling of the integration is not supported, it has to be performed with manual coding without tool support.

MetaEdit+ [11] 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 up-dated. 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. By 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. The integration of external components is not supported by modeling or generative techniques.

Human Assisted Logical Language (H)ALL [12] is a domain-specific visual language for automatic derivation of user interfaces for critical control systems within the BATIC3S [13] project. Compared to VMTS, (H)ALL also applies hierarchical state machines to express the dynamic behavior of user interfaces. The state machines communicate via messages. The semantics of (H)ALL models is defined with Concurrent Object-Oriented Petri-Nets [14] (CO-OPN). The conversion between (H)ALL and CO-OPN is described with QVT [15] transformations. Compared to VMTS, (H)ALL does not deal with the integration of external components, it needs proprietary integration of components to the coordination engine.

6 Conclusion

The focus of our contribution is how to simplify the handling of complex messages, how to support the automatic integration to external tools, and how to optimize use of the existing DSLs with domain-specific model patterns. We found that the use of hierarchical state machines to describe the message handling simplifies the models to a great extent. The automation of the integration process accelerates the development time significantly. We outlined a few design patterns that can help reusing existing high-level animation and user interface models. In our experience, using these practical techniques helps to make the graphical DSL-based approach more efficient than manual coding. Future work includes creating more applications, optimizing the generators, enriching the DSLs, and identifying more specific model patterns for the animation paradigm.

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

References

1. VMTS homepage, `http://vmts.aut.bme.hu`
2. Mészáros, T., Mezei, G., Levendovszky, T.: A flexible, declarative presentation framework for domain-specific modeling. In: Proceedings of the working conference on Advanced visual interfaces (AVI 2008), Naples, Italy, pp. 309–312 (2008)
3. Mészáros, T., Mezei, G., Charaf, H.: Engineering the Dynamic Behavior of Metamodeled Languages, Simulation, Special Issue on Multi-paradigm Modeling (2008) (accepted)
4. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation, 2nd edn. Academic Press, London (2000)
5. Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
6. MATLAB homepage, `http://www.mathworks.com`
7. EngMATLib homepage, `http://www.thecodeproject.com`
8. Levendovszky, T., et al.: Supporting Domain-Specific Model Patterns with Metamodeling, System Special Issue on Metamodeling (submitted)
9. de Lara, J., Vangheluwe, H.: AToM3: A Tool for Multi-Formalism and Metamodeling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
10. Lédeczi, Á., et al.: Composing Domain-Specific Design Environments. IEEE Computer 34(11), 44–51 (2001)
11. Tolvanen, J.-P.: MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages. In: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Portland, USA, pp. 690–691 (2006)
12. Barroca, B., Amaral, V. (H)ALL: a DSVL for designing user interfaces for Control Systems. In: Proceedings of the 5th Nordic Workshop on Model Driven Engineering, Göteborg, Sweden (2007)
13. BATIC3S homepage, `http://smv.unige.ch/tiki-index.php?page=BATICS`
14. Buchs, B., Guelfi, N.: A formal specification framework for object-oriented distributed systems. IEEE Trans. Software Eng. 26(7), 635–652 (2000)
15. MOF QVT Final adopted specification,
`http://www.omg.org/docs/ptc/05-11-01.pdf`