Adaptation of an Executed Model and its Application to State Machines

Samson Pierre

University of Pau / LIUPPA research laboratory,
Avenue de l'Université, 64012 Pau, France
samson.pierre@univ-pau.fr
http://www.pauware.com/

Abstract. The execution of models is an important sub-part of the MDE (Model-Driven Engineering) field while the model-based software adaptation issue has gained more and more interest in the last few years. My Ph.D. works lean on the convergence of these two key aspects, giving rise to a further research question: how to deal with the adaptation of executed models? In that context, I am studying the case of state machines that are prominent examples of models able to be executed and for which adaptation scenarios might be sketched.

Keywords: MDE, model execution, adaptation, UML state machines

1 Problem of interest

The execution of models is an important topic for the MDE (Model-Driven Engineering) community. Indeed, it aims at substituting models for code. When we want to execute a model written using a DSML (Domain-Specific Modeling Language), we have the choice between compiling or interpreting it. Compiling a model consists in applying a transformation to it. It is translated into an other language (which is at a lower level of abstraction). We also speak about code generation to reference this approach. Unlike compiling a model, interpreting a model does not require to transform it. An engine is able to directly interpret the DSML execution semantics. We also speak about i-DSML [7] for this method.

The fact that a model conforming to an i-DSML can be executed before its implementation offers two advantages. The first advantage is that it is possible to discover and correct problems at the design-time in the software development process. The second advantage is that the implementation step in the software development process can possibly be skipped. An i-DSML is a 'self-contained' metamodel. It means that a number of elements are added to its structure. At first glance, this characteristic can be seen as a disadvantage because it pollutes the metamodel. But in MDE, we can consider this as an advantage because these elements in the model can be very useful (e.g., to create a trace at runtime to debug the system). So, using an i-DSML for the development of software applications is a good strategy to decrease efforts and costs.

Nowadays, the software adaptation has gained more and more interest. When we are building an adaptable software based on an i-DSML, the model has to be adaptable at runtime. So, its i-DSML must allow to define adaptable models.

In order to adapt a software at runtime, an adaptation loop must be implemented [4]. A first approach to implement this loop is the common models@run.time [1], depicted in Fig. 1 (a). In this case, a model is embedded at runtime and it is not executed but is kept in sync with the running system. Thus, this model is an abstract representation of the system and is used for a reasoning purpose on the necessity to adapt. An other way to implement this loop is the executed model adaptation, as depicted in Fig. 1 (b). In this case, a model is executed and directly corresponds to the running system. A last approach is to combine the two previous ones, and is depicted in Fig. 1 (c). In this case, a model is embedded at runtime and it is not executed (like in the first approach). In addition, there is an other model that is executed and corresponds to the system (like in the second way).

The main disadvantage of the first and the last solutions is that we have two distinct entities at runtime (the system and the model). This requires to be able to keep a valid correspondence between the elements of the two entities. So, we think that using the second solution is a good alternative to adapt a model at runtime. We also speak about direct adaptation to point at this approach.

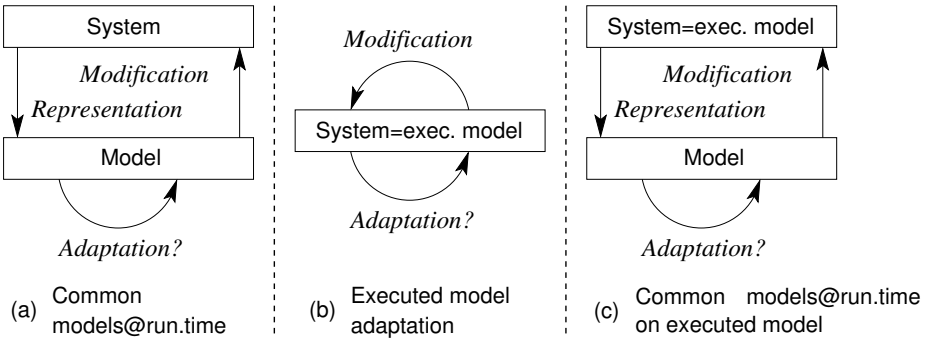


Fig. 1. The three ways to implement the adaptation loop

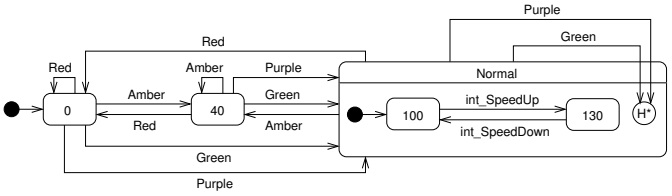
The problem of my thesis is to find an approach to adapt an executed model. The idea is to execute a model conforming to an i-DSML and to adapt it during its execution, based upon the direct adaptation approach. As an illustration, I am studying the case of state machines.

In Sect. 2, I give an example of a state machine that is adapted during its execution. In Sect. 3, I present the forthcoming contributions, both scientific and technical.

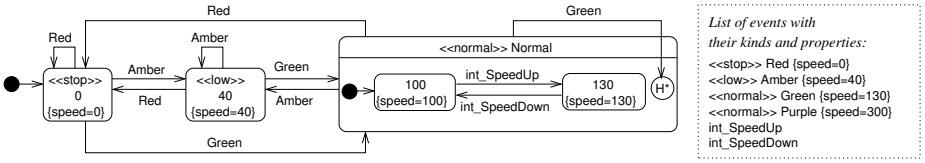
2 Working example

To illustrate the adaptation of an executed model, let us take the example of a train state machine [1] previously published in [2, 3]. The train can run on railroads at different speeds (0, 40, 100 and 130). The train can encounter signals of different colors (red, amber, green and purple) during the travel indicating that it must change its speed. When modelling this situation, each state of the state machine is associated with a speed and each event on the transitions of the state machine is associated with a color.

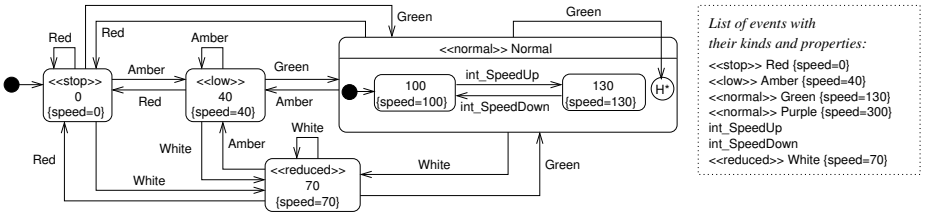
In Fig. 2 (a), we use an extended train state machine in which each state is able to treat all the events. So, there are a lot of transitions between the states.



(a) Extended train state machine: Explicit management of all events



(b) Extended train state machine: Properties and kinds for model elements



(c) Train state machine: Addition of a state and its associated transitions

Fig. 2. Adaptation of the train state machine

The i-DSML structure of this state machine is composed of three parts. The static part contains the main elements (the states, transitions, events, ...). The dynamic part contains the elements that can be changed during the execution (the current state, ...). The adaptation part contains the elements to manage the

[1] This example is not at all intended to be considered as a realistic specification of a train system

adaptation (the kinds and properties). In Fig. 2 (b), each state and event of the train state machine is associated with a kind (stop, low and normal) and a property (speed). These new kinds are written within guillemets (e.g., <<stop>>) whereas these new properties are written within braces (e.g., {speed=0}). For the events, these new kinds and properties are listed in the dashed box at the right of the figure.

We have different ways to adapt. Indeed, we can choose between the two adaptation modes below:

1. Exact mode: only the name of the event is checked.
2. Fail-soft mode: the kind of the event is checked.

Now, we are executing the train state machine. Let us consider that the train encounter a white color signal. Note that this event is associated with a kind (reduced) and a property (speed=70). Because the train state machine does not include any events with the reduced kind or with the white name, an adaptation is performed, whatever the adaptation mode selected. A state, transitions and events are added to the model structure. This new state is named '70' because the value assigned to the speed property of the event responsible of this adaptation is equal to seventy. This new state and these new events are associated with the same kind (reduced) and the same property (speed) than the event that triggered this adaptation. The current state is changed to this new state. In Fig. 2 (c), we can see the result of the adaptation.

3 Expected results

In this section, I describe the scientific contribution and the technical contribution. The scientific contribution is about model execution and model execution adaptation. The technical contribution is about tools enabling model execution and model execution adaptation for state machines.

3.1 Scientific contribution

I recall that the problem of my thesis is to find an approach to adapt an executed model with a special focus on state machines. I will execute a model conforming to an i-DSML and I will adapt it during its execution using the direct adaptation approach. The significance of this problem is not easy to understand because it raises important questions:

- What is the meaning of adapting a model during its execution?
- Why and how to modify its structure (i.e., its static, dynamic and adaptation parts)?
- Why and how to modify its behavior (i.e., its execution and adaptation semantics)?

To the best of our knowledge, the current state of the art has not solved this problem yet because adapting a model during its execution using an i-DSML has never been studied by the research community.

The goal of my research work is to give a precise answer to each of the previous questions. I am currently working on the description of the required elements for an adaptable i-DSML, to be sure that a model conforming to this metamodel can be considered as adaptable during its execution. Also, I am planning to create a DSL (Domain-Specific Language) which helps to create adaptation rules that will be taken in account during the adaptation loop. To that purpose, I planned to define the sense of an execution environment (or a context) from a model-based point of view.

As I am adapting executed models, I have studied the model execution and the model execution adaptation till now. I am showing below different ways that I found to execute models, then I explain the model execution adaptation.

Model execution There are different kind of approaches allowing us to execute models.

A first approach consists in using an action language (e.g., Kermeta, FUML or ALF) to implement an execution semantics. The action languages can have a graphical or textual syntax. For example, FUML has a graphical syntax (we can use activity diagrams to write an execution semantics as explained in [9]) while Kermeta and ALF have textual syntaxes.

An other way to execute models is to use a code generator (e.g., EMF) to transform a model into a general-purpose programming language (e.g., Java). In that case, the generated code contains empty parts that you must fill-in to add the missing execution semantics.

It exists a third approach which consists in using a transformation language (e.g., ATL) to do several endogenous transformations. Each transformation represents a step of execution. Doing so, at each step, the model may be serialized.

A fourth method to execute models is to use dedicated tools (e.g., PauWare Engine [2] for a state machines or Tina for Petri nets) where the execution semantics is directly embedded in the tools. PauWare Engine is a Java library, so the model is written in Java code. For his part, Tina has a graphical editor.

A last approach is to execute models using a tool that performs exogenous transformations (e.g., SCXML2PauWare). If there are currently no existing tools to run models written using the source metamodel, we can transform them to a target metamodel where tools are available. SCXML2PauWare enables to transform a state machine conforming to the SCXML metamodel to an equivalent state machine conforming to the PauWare Engine metamodel. In other words, it brings a translational semantics to SCXML state machines.

Model execution adaptation In Sect. 1, I say that an adaptation loop must be implemented. In the case of an executed model, this loop controls and fires

[2] `http://www.pauware.com/`

adaptation rules to adapt a model to its execution environment. These rules are composed of checking and actions.

There are two categories of actions [3]. The first is the CUD (Create, Update and Delete) whereas the second is the Substitution. The CUD is intended to be applied on the structure (i.e., the static, dynamic and adaptation parts) of the model. The Substitution is intended to be applied on the behavior (i.e., the execution and adaptation semantics) of the model.

Table 1 sums up the four actions applied on the elements of the model. We can see that when we are executing a model, only the elements of the dynamic part can be modified by the actions. When we are adapting a model, all the elements can be modified by the actions.

Element of the model	**Execution actions**	**Adaptation actions**
Static part	N/A	Create/Update/Delete
Dynamic part	Create/Update/Delete	Create/Update/Delete
Adaptation part	N/A	Create/Update/Delete
Execution semantics	N/A	Substitution
Adaptation semantics	N/A	Substitution

Table 1. The four actions applied on the elements of the model

In the example of Sect. 2, we have applied CUD adaptation actions on elements of the three structural parts. Indeed, the static part has been modified because we have added a state, transitions and events. The adaptation part has been modified because we have added kinds and properties associated with these new states and events. Finally, the dynamic part has been modified because the current state now references this new state.

The engine embeds operations corresponding to the execution semantics and the adaptation semantics. The Substitution adaptation action deals with these embedded operations. Using the DSL for the adaptation, we can weave these operations together in order to modify the two behavioral parts. So, in addition to create adaptation rules, this DSL will allow to orchestrate them (as does a language for workflows). The resulting orchestrations, as full-fledged executable models, will be in turn subjects to adaptation, as explained from the beginning of this article. We think it is a conceptually sound way to deal with reflexive adaptation (or meta-adaptation).

3.2 Technical contribution

I am studying the case of state machines. The state machines (or statecharts) have been described in [5]. Afterwards, they have been included into the UML [8] specification. Meanwhile, the SCXML [6] working draft was released.

Our research team has already developed a tool to execute state machines (PauWare Engine). For the example of Sect. 2, the execution semantics of state machines has been rewritten in Kermeta instead of using this existing tool,

for pedagogical reasons. However, in practice this way to do is error-prone and consumes unnecessary efforts and time. Consequently, it should be better to use PauWare Engine that is fully compliant with the UML specification. Because PauWare Engine does not support the adaptation yet, I am planning to extend the PauWare Engine API to add this feature.

In addition, I am developing a tool called SCXML2PauWare, to be able to run state machines initially described using the SCXML format. It is an Eclipse plug-in that does exogenous transformations (i.e., transformations from a source metamodel different from the target metamodel). The source metamodel is SCXML whereas the target metamodel is UML/PauWare. As such, SCXML2PauWare can be considered as a bridge since it increases the number of technological spaces supported by PauWare Engine.

In Fig. 3, we can see the big picture of this technical contribution. At the top of the figure, there are state machines written within different technological spaces (UML, SCXML, ...) and adaptation rules. These various state machines are all switched, through specific bridges, into equivalent PauWare state machines. At the bottom of the figure, there is the engine that is able to support both model execution and model execution adaptation for state machines. This engine takes as inputs the PauWare state machine and the adaptation rules.

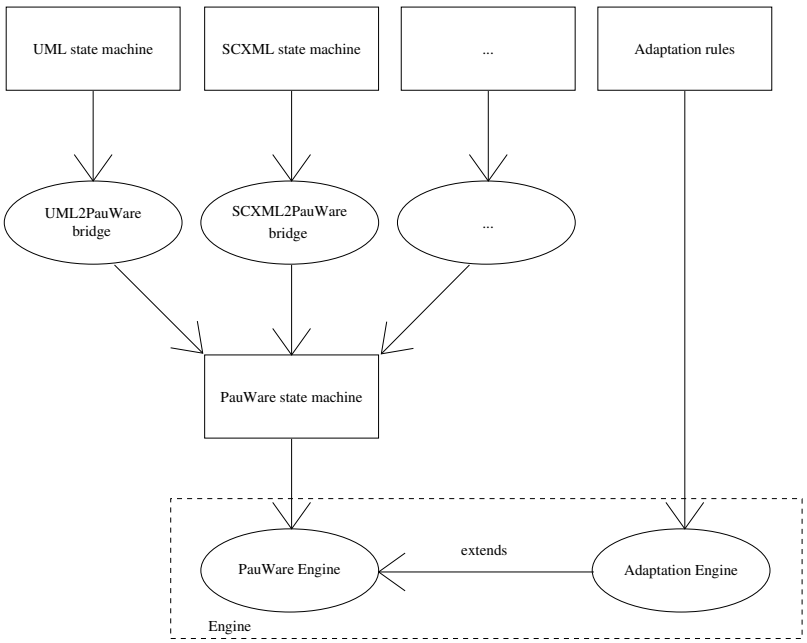


Fig. 3. The PauWare Engine adaptation extension and the SCXML2PauWare bridge

4 Conclusion

Building adaptable software based on the direct execution of models is a quite novel and promising approach. My thesis aims at defining a sound conceptual framework to study adaptable i-DSML, that is, a family of languages that allow to create models directly executable and adaptable through an engine. Moreover, since I pay a special attention on the adaptation issue, I expect to develop a specific language for writing adaptation rules and managing their orchestration during the adaptation loop. Beyond the scientific contribution, my research work ought to result in a tool chain dedicated to this question.

References

1. Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.
2. Eric Cariou, Olivier Le Goaer, and Franck Barbier. Model Execution Adaptation? In *7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012*. ACM Digital Library, 2012.
3. Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre. Characterization of Adaptable Interpreted-DSML. In *European Conference on Modelling Foundations and Applications (ECMFA 2013)*, volume 7949 of *LNCS*, pages 37–53. Springer, 2013.
4. Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.
5. David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.
6. Scott McGlashan, Michael Bodell, Jerry Carter, RJ Auburn, Torbjörn Lager, No'am Rosenthal, T. V. Raman, Daniel C. Burnett, Jim Barnett, Rafah Hosn, Mark Helbing, Rahul Akolkar, and Klaus Reifenrath. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Working Draft, W3C, October 2009. http://www.w3.org/TR/2009/WD-scxml-20091029/.
7. Marjan Mernik. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. Information Science Reference, Hershey, PA, September 2012.
8. OMG. OMG Unified Modeling Language (OMG UML), Superstructure. OMG Specification, OMG, August 2011. http://www.omg.org/spec/UML.
9. Mayerhofer Tanja, Langer Philip, and Wimmer Manuel. Towards xMOF: executable DSMLs based on fUML. In *Proceedings of the 2012 workshop on Domain-specific modeling*, DSM '12, pages 1–6, New York, NY, USA, 2012. ACM.