Embedded UML Model Execution to Bridge the Gap Between Design and Runtime

Valentin Besnard[1(✉)], Matthias Brun[1(✉)], Frédéric Jouault[1(✉)], Ciprian Teodorov[2(✉)], and Philippe Dhaussy[2(✉)]

[1] ERIS, ESEO-TECH, Angers, France
{valentin.besnard,matthias.brun,frederic.jouault}@eseo.fr
[2] Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France
{ciprian.teodorov,philippe.dhaussy}@ensta-bretagne.fr

Abstract. The number and complexity of embedded systems is rising. Consequently, their development requires increased productivity as well as means to ensure quality. Model-based techniques can help achieve both. With classical model-driven development techniques, developers start by building design models before producing actual code. Although various approaches can be used to validate models and code separately, models and code are however separated by a semantic gap. This gap typically makes it hard to link runtime measures (e.g., execution traces) to design models. The approach presented in this paper avoids this semantic gap by making it possible to execute UML design models directly on embedded microcontrollers. Therefore, any runtime measure is directly expressed in terms of the design model. The paper introduces our UML bare-metal (i.e., not requiring an operating system) interpreter. Its use is illustrated on a motivating example, which can be simulated, or debugged, and for which message sequence charts can be generated.

Keywords: UML execution · Model interpretation
Embedded systems

1 Introduction

Embedded systems become more and more complex due to the emergence of new needs and applications (e.g., Internet of Things, autonomous cars, smart cities). This increasing complexity renders software programs more difficult to design, maintain, and evolve. One of the main consequence is that bugs and design faults are more difficult to detect and fix. To validate the system behavior, it becomes necessary to execute the system during early design phases, and to link design and runtime concepts together to ease the system analysis.

With model-driven engineering, a classical approach consists in simulating a model of the system under study on a desktop computer. Then, the application code is produced using code generation and executed on an embedded target. However, code generation creates a semantic gap between design models and executable code that makes it more complicated to link design models to execution concepts. Therefore, diagnosis activities (e.g., simulation, debugging) and runtime measures analysis (e.g., execution traces) can become complex. It is even more challenging to visualize the execution of a system running on an embedded target and to interact with its design model at runtime.

To partially address these issues, we introduce a model interpreter that can be used to execute UML models. This tool has been presented in [2] but in this paper, we will focus on interactions between design and runtime, which have not been presented yet. In our approach, the design model is directly loaded in our model interpreter for being executed. This technique avoids the semantic gap created by code generation and ensures that the same concepts (here UML concepts) are used between design and runtime. Indeed, the model execution can be directly visualized in terms of UML concepts through two kinds of interactions. Online interactions used during simulation and debugging can be employed to interact with the model during its execution. Offline interactions are also available to visualize the model execution through the generation of message sequence charts (MSC) from execution traces. These MSC are directly expressed in terms of the design model elements. This approach is a first step towards the goal of executing design models for complex embedded systems. This work shows that it is possible to do it on bare-metal for small embedded devices (e.g., Internet of Things) but this approach can be generalized to use operating systems for more complex embedded systems applications.

Our UML model interpreter shows encouraging results towards feasibility. It can be used to execute UML models on desktop computers and embedded microcontrollers using model interpretation. This interpreter can be connected to a simulator for simulating and visualizing the system execution using a dedicated communication protocol. It is also possible to print execution traces into a formalism for generating MSC diagrams from these traces. Experiments have been made on a level crossing system to illustrate these features. These improvements contribute to reinforce the link between design and runtime as well as reducing time-to-market and increasing both productivity and quality.

The remainder of this paper is structured as follows. Section 2 introduces our model interpreter and the technique used to interpret a UML model on a bare-metal target. Then, we describe multiple interactions modes between design and runtime in Sect. 3. In Sect. 4, we discuss advantages of this approach before reviewing some related work in Sect. 5. Finally, we conclude this paper in Sect. 6.

2 Interpretation of Executable UML Design Models

To link design and runtime concepts, our approach is based on a model interpreter that can execute the model of the system produced during the design phase. In this section, we will present the process used to serialize a design model into source code before being loaded in and executed with our prototype.

The first step consists in designing a model of the system under study in UML. This activity can be performed with either graphical editors (e.g., Papyrus

[11]) or textual editors (e.g., tUML tool [9,10]). Using these tools, the design model can then be exported into the XML metadata interchange (XMI) format. To be executable, this model must specify explicitly the system behavior. In our case, the behavior of active classes is specified using state machines composed of states and transitions. Each transition can have a guard and an effect encoded respectively in an opaque expression and an opaque behavior. To write these guards and effects, we use an action language based on the C programming language but with specific syntactic extensions to simplify access to UML instances (i.e., instances of UML classes). These extensions can be used to send events, get and set values of attributes, and access content of event pools in a relatively simple way. With this syntactic sugar, users do not need to know the internal structure of the interpreter to use the action language.

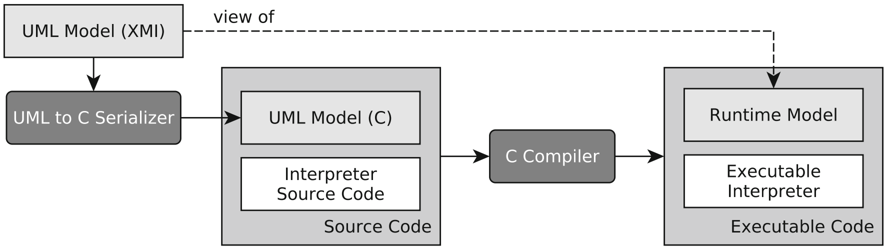

Fig. 1. Overview of the model generation process.

Once the executable model has been saved into XMI, it can be serialized into C source code using a transliteration, as shown in Fig. 1. This serialization can be seen as the way to load the model into our interpreter. In fact, it only adapts the syntax of the model to C programming language without performing any semantics change. The serializer is used to generate a C struct initializer for each UML element needed for model interpretation. With struct initializers the C compiler constructs the binary representation of the model in the initialized data section of the memory. This can be seen as compile-time model loading. Hence, this technique differs from classical code generation that generates both data and program required to execute the system. We only generate data that represents the static part of the model. The only exception concerns transition guards and effects that are serialized as C functions to which the C representation of transitions point (using function pointers). In fact, in UML with opaque expression guards and opaque behavior effects, the code of guards and effects is represented as strings stored into a body property. These C functions provide executable behaviors for these bodies without requiring to parse these strings or to perform expensive operations directly on the target. Apart from transition effects and guards, no code is generated from the UML model, only data. Moreover, this data is no more than an in-memory representation of a loaded UML model, similar to the result of EMF XMI loading.

Then, a C compiler is used to compile both the UML model in C language and the source code of the interpreter, which are then linked together into executable code. This executable code includes the runtime model composed of both the static and the dynamic part of the model. At this point, the reference model is the runtime model because this is the model really executed on the target. The design model is only a view of it. The resulting executable may be executed either on a desktop computer or on a microcontroller-based embedded system. The execution results in interpretation of the model using both the UML model of design (data) and the operational semantics implemented into the model interpreter (program). The implemented semantics tends towards the precise semantics for UML state machines (PSSM [13]) based on fUML [14].

3 Interactions with Design Tools

To reinforce the link between design and runtime, our approach is able to deal with two kinds of interactions between the runtime model and design tools. On the one hand, online interactions enable to interact with model execution for simulation or debugging purposes. On the other hand, our model interpreter is also able to generate traces at runtime that can be analyzed after execution. These interactions are supported by a choice of three possible interaction modes that are typical of embedded systems development process: simulation, debugging, and execution. The interpreter may be compiled with any of these features except the debugging loop which has not been implemented yet. In order to be deployed on the actual embedded system, the interpreter is compiled only with the execution loop. Therefore, interactions for simulation or debugging loops are not provided in the final product in order to avoid leaving a potential attack vector open.

3.1 Simulation

To interact with the model at design time, it is possible to use a simulator. The simulation mode enables making online interactions to explore the model and visualize its execution. Using our approach, model execution can be controlled through the interpreter running locally on a desktop computer or remotely on an embedded microcontroller. This second possibility can be employed to make hardware in the loop simulation directly on the board that will be used on the actual embedded system. To control model execution, our model interpreter provides the following application layer protocol:

Get configuration collects the current memory state of the interpreter.
Set configuration loads a configuration as the memory state of the interpreter.
Get fireable transitions collects transitions that can be fired on the next step.
Fire transition fires a transition of an active object's state machine.

Reset interpreter restarts the interpreter from the initial state of the model.

In this communication protocol, the memory state of the interpreter is called *configuration* and represents the dynamic part of the executed model. The configuration is composed of current states of state machines, contents of event pools (i.e., all events received by UML instances of active classes), and values of attributes. To get and set the configuration, we prefer the use of two global commands rather than multiple small and complex commands. This simplifies the protocol and gives the possibility to have an overview of the whole configuration at each simulation step. To improve performance, it is possible to use a *diff mode* that enables exchanging only bytes that are different between the current configuration and the previous one. For instance, if one wants to change the value of only one attribute, only the value of this attribute and its position into the configuration will be transmitted rather than the whole configuration. Virtual peripherals communicate with the model by directly reading or writing into event pools, which can also be performed in *diff mode*. With this simulation loop, execution flow is entirely controlled by the simulator. Therefore, it is possible to implement an execution loop in this tool to run model execution. Figure 2 presents the user interface of our simulator applied to a level-crossing model [2]. This interface shows the list of fireable transitions available in the selected configuration, the content of this configuration, and the part of the model state-space discovered since the beginning of the simulation.

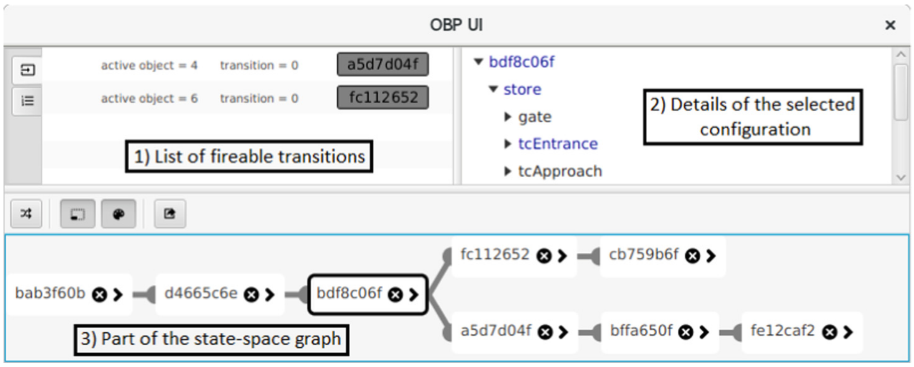
OBP UI
active object = 4 transition = 0 a5d7d04f
active object = 6 transition = 0 fc112652
1) List of fireable transitions
bdf8c06f
store
gate
tcEntrance
tcApproach
2) Details of the selected configuration
bab3f60b
d4665c6e
bdf8c06f
fc112652
cb759b6f
a5d7d04f
bffa650f
fe12caf2
3) Part of the state-space graph

Fig. 2. User interface of the simulator.

3.2 Debugging

Debugging is another kind of online interactions. It can be used to control model execution in the same way than previously introduced simulation purpose, or to automatically execute the model with the actual embedded system. For this reason, it is a mix between both simulation and execution modes. The debugging loop can be used to control model execution using the communication protocol and to observe the configuration of the model. This protocol enables injecting an event, changing the value of an attribute, or changing the value of the current state of a state machine using the *Set configuration* request. The *diff mode* can be used to reduce the cost and optimize communication performance. This communication protocol can also be used to make omniscient debugging [3,5,8] to go back-in-time. Indeed, received configurations can be stored and reloaded at any time as the current memory state of the interpreter. In debugging mode, it is also possible to execute the system using the execution loop implemented in the interpreter. The only difference with the execution mode is that it will check if there is a command (sent by the debugger) to process, so the runtime cost is small. In our prototype of model interpreter, these debugging interactions have some limitations. The first one is that execution of opaque behaviors and evaluation of opaque expressions cannot be debugged because they are implemented as C functions. It would become possible if we used UML activities to specify their behaviors, which we might explore in the future. The second limitation is that we do not support breakpoints for the moment. Hence, it is not possible to stop the execution automatically when reaching a given state of a state machine. This is the main feature that lacks in our interpreter for supporting this mode.

3.3 Execution

The execution mode is the main loop actually used on the deployed system. For offline interactions, we add the possibility to generate execution traces and to display them using messages sequence charts (MSC). MSC are a kind of diagram that captures interactions between active objects of the system. It is similar to a sequence diagram but enhanced with states of state machines, such that it is possible to know the current state of each active object at any time. MSC give an overview of a scenario and enable to visualize interactions between active objects (i.e., exchange of events) and their state machine progression. In our model interpreter, we have also added an optional feature to display attribute values changes.

In practice, we instrumented the code of our interpreter with C macros that are called each time an event is sent, a state machine updates its current state, or an attribute has its value updated. At compile time, the user can choose the MSC formalism to use for displaying the trace. This will replace C macros by calls to appropriate functions in charge of displaying the trace. If no trace is required, (e.g., on the deployed embedded system), these macros are replaced by no instruction to have no impact on execution performance of the actual system. At runtime, the trace will be printed either on the standard output stream (e.g., a serial port of the embedded target) or directly in a text file when running on a desktop computer. Afterwards, the trace can be loaded into a tool in order to generate a graphical diagram that gives a better visualization of it. In the current version of our model interpreter, we have chosen to display traces using the PlantUML[1] formalism. Traces are then converted into diagrams using

[1] http://plantuml.com/.

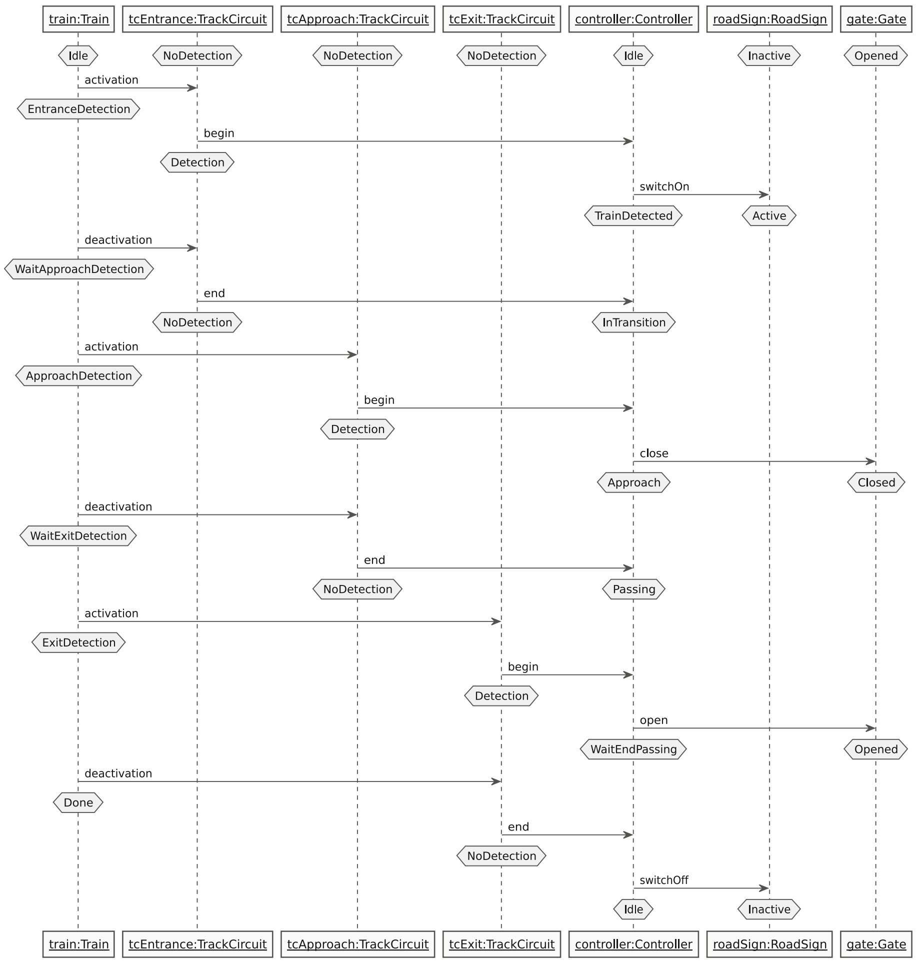

Fig. 3. Message sequence chart of a level crossing model.

the PlantUML tool. However, additional transformations towards different MSC formalisms can be easily added to our tool. Figure 3 shows an example of MSC diagrams obtained with our model interpreter. This example represents a trace of a level crossing model example introduced in [2].

4 Discussion

Our approach based on a model interpreter tackles issues to link design and run-time concepts. It thus should contribute to reduce time-to-market and increase both quality and productivity.

The first advantage of our approach is that our model interpreter can be deployed either on desktop computers or on embedded targets. Indeed, our prototype is adapted to be deployed on bare-metal microcontrollers without intermediate software layers like an operating system. This means that this model interpreter can run on embedded microcontrollers with relatively small memory size and relatively slow CPU. This possibility can be used to make simulation or debugging at the model level directly on embedded targets. For instance, this can be useful to detect some bugs linked to the hardware by making hardware in the loop simulation. In most of the classical approaches, the design model is specified and validated on desktop computers before being transformed into executable code through code generation, and executed on microcontrollers. In this case, the link between design and runtime is difficult to set up. With our approach, this link is easier to establish thanks to the possible interactions between design and runtime that it offers, and because we use the same model all along the development process. Indeed, our approach provides a continuum from design to runtime by applying simulation, debugging, and trace generation either on desktop computers or embedded microcontrollers.

The second point is that a single semantics implementation is used for execution and simulation. Indeed, the execution semantics used to interpret UML models is implemented in our model interpreter. In most of the classical approaches, the code generation step is a transformation that creates a semantic gap between design and runtime that may not ensure that simulation results are still valid at runtime. With our model interpreter, there is no problem of equivalence between the design model and the model used at runtime because all activities (execution, simulation, and debugging) are made through the interpreter. Hence, only one implementation of the semantics is used. This contributes to increase the development quality of the system.

The last key point deals with traces analysis as well as simulation and debugging results. In our approach, the design model is directly used for execution through a model serialization into C programming language. Therefore, the mapping between design and runtime concepts is straightforward. Simulation and debugging techniques can be used to simulate the model directly in terms of design concepts. This also facilitates execution traces analysis to inject feedbacks in the design model and fix design faults. As a result, we expect that this will increase productivity and reduce time-to-market.

5 Related Work

Other works have shown abilities to execute models and establish links between design tools and runtime measures through various kinds of interactions.

Multiple implementations of fUML [14] or PSSM [13] have been realized to execute models conforming to these executable UML standards. Moka [1] and Moliz [12] are two of these implementations that are able to support execution, simulation, and debugging of UML models. GEMOC Studio [4,7] is another tool that contains a modeling workbench to design models conforming to any domain-specific languages. This tool has four different execution engines and several add-ons can be used to perform simulation, debugging, and trace generation.

All these tools are well-integrated into modeling development environments. For instance, Moka has an Eclipse-based user interface and can be used with the Papyrus [11] editor to simulate UML models with graphical feedbacks over diagrams. The main drawback of all these tools is that they are not adapted to execute models on embedded targets. Indeed, these tools use too much memory for being executed on a small microcontroller. The generic approach used to build these tools also induces a lack of performance because they are not adapted for embedded systems execution.

In comparison to these works, some approaches aim at executing models on embedded targets with small memory footprints and good execution performance. UML virtual machine (UVM) [15] defines a runtime environment to execute bytecode in the binary UVM format generated from models and includes extensions for fine grained concurrency and precise timing. In the same way, a front-end, called GUML [6], has been defined for GCC to compile directly UML models into optimized binary code. Both tools have similarities to our model interpreter but they cannot be remotely controlled by diagnosis tools to analyze model execution in terms of design concepts.

6 Conclusion

To bridge the gap between design and runtime, this paper has presented our approach based on a UML model interpreter. This interpreter uses the same model for design and runtime to offer a direct link between design and runtime concepts. To take advantage of this link, we have put in place online and offline interactions between design and runtime. Simulation and debugging activities can be applied directly in terms of UML concepts. This eases the integration of simulation feedbacks and the correction of bugs into the design model. To facilitate the visualization of model execution, our approach also relies on execution traces generated at runtime to produce MSC diagrams with PlantUML. We expect that these improvements should help engineers to analyze model execution and fix design faults in the design model. In fact, this should reduce time-to-market and increase productivity because the model analysis will be easier.

Another significant key point of our approach is that this technique remains valid for embedded systems. Indeed, the same model interpreter can be used to execute models on bare-metal targets equipped with small embedded microcontrollers. For simulation, the interpreter can be remotely controlled through a communication protocol that is sufficient to get/set dynamic data of the runtime model, and control model execution by firing state machine transitions. Hence, the boundary between design and runtime virtually disappears and the transition from one to the other can be realized in a continuous way using multiple activities (e.g., simulation, debugging, execution).

To reinforce the link between design and runtime, we are currently investigating other possibilities offered by our approach. Indeed, the protocol used for simulation can also be reused to connect other diagnosis tools, such as a model-checker. This should offer a new kind of online interactions to make the verification of formal properties on models.

Acknowledgments. This work is partially funded by Davidson Consulting. The authors especially thank David Olivier for his advice and industrial feedback.

References

1. Papyrus: Moka overview. https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
2. Besnard, V., Brun, M., Dhaussy, P., Jouault, F., Olivier, D., Teodorov, C.: Towards one model interpreter for both design and deployment. In: Proceedings of EXE 2017, Austin, United States, September 2017
3. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of SLE 2015, pp. 137–148. ACM, New York (2015)
4. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: Proceedings of SLE 2016, pp. 84–89. ACM, New York (2016)
5. Bousse, E., Leroy, D., Combemale, B., Wimmer, M., Baudry, B.: Omniscient debugging for executable DSLs. J. Syst. Softw. **137**, 261–288 (2017)
6. Charfi Smaoui, A., Mraidha, C., Boulet, P.: An optimized compilation of UML state machines. In: Proceedings of ISORC 2012, Shenzhen, China, April 2012
7. Combemale, B., et al.: A solution to the TTC'15 model execution case using the GEMOC studio. In: 8th Transformation Tool Contest. CEUR, Italy (2015)
8. Corley, J., Eddy, B.P., Gray, J.: Towards efficient and scalabale omniscient debugging for model transformations. In: DSM 2014, pp. 13–18. ACM, New York (2014)
9. Jouault, F., Delatour, J.: Towards fixing sketchy UML models by leveraging textual notations: application to real-time embedded systems. In: Brucker, A.D., Dania, C., Georg, G., Gogolla, M. (eds.) OCL 2014. OCL and Textual Modeling: Applications and Case Studies, Valencia, Spain, vol. 1285, pp. 73–82 (2014)
10. Jouault, F., Teodorov, C., Delatour, J., Le Roux, L., Dhaussy, P.: Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. Génie logiciel **109** (2014)
11. Lanusse, A., et al.: Papyrus UML: an open source toolset for MDA. In: Proceedings of ECMDA-FA 2009, pp. 1–4 (2009)
12. Mayerhofer, T., Langer, P.: Moliz: a model execution framework for UML models. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards, MW 2012. ACM, New York (2012)
13. OMG: Precise Semantics of UML State Machines, February 2017. https://www.omg.org/spec/PSSM/1.0/Beta1/PDF
14. OMG: Semantics of a Foundational Subset for Executable UML Models, October 2017. https://www.omg.org/spec/FUML/1.3/PDF
15. Schattkowsky, T., Engels, G., Forster, A.: A model-based approach for platform-independent binary components with precise timing and fine-grained concurrency. In: Proceedings of HICSS 2007. IEEE Computer Society, USA (2007)