Towards one Model Interpreter for Both Design and Deployment

Valentin Besnard*, Matthias Brun*, Philippe Dhaussy†, Frédéric Jouault*, David Olivier‡, and Ciprian Teodorov†
*TRAME team, ESEO, Angers, France, *firstname.lastname*@eseo.fr
†Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France, *firstname.lastname*@ensta-bretagne.fr
‡Davidson Consulting, Rennes, France, david.olivier@davidson.fr

***Abstract*—Executable modeling of complex embedded systems is essential for bug discovery and safety validation at early design stages. A relatively large number of tools enable early design diagnosis and validation by transforming and analyzing the model into a formal setting. However, this transformation induces a semantic gap rendering diagnosis more difficult. Moreover, on the way to deployment, executable models are transformed into low level executable code. Existence of this second transformation similarly renders diagnosis of the deployed system more difficult, and also increases validation costs of the approach in the context of critical systems: a non trivial equivalence relation needs to be established between the formally analyzed model and the executable code. In this paper, we introduce a first step towards addressing these problems with a bare-metal UML interpreter, which uniquely defines the executable semantics for both design and deployment. To facilitate the diagnosis and validation process our interpreter offers a diagnosis interface through which the semantics is shared with diagnosis tools. The tools rely on this interface to interact with (observe and control) the executing model either locally on a PC during early design phases or remotely on the target embedded system after deployment. We illustrate our approach on a railway level crossing system ported to two embedded targets (at91sam7s and stm32), to which we connect a remote high-level simulator for interactive execution control and exhaustive state-space exploration.**

***Index Terms*—UML, Model Interpretation, Model Verification, Embedded Systems.**

I. Introduction

The new generation of embedded systems and CPS (Cyber-Physical Systems) provides more powerful features to meet emerging needs in numerous fields (e.g., automotive, avionics, robotics, smart cities). With the development of IoT (Internet of Things), they now tend to be connected and to collaborate with other systems on networks. This increasing complexity creates more difficulties for engineers to check and ensure safety, or simply to avoid bugs. Not only are behaviors of these systems more uncertain but they are also more vulnerable to cyber attacks, due to their network connections. To prevent the introduction of bugs during development of such systems, there is a need to execute, simulate, and verify models. In the industrial world, the common approach [1] consists of two transformations: one that converts the design model into various models (e.g., formal models) used by diagnosis tools during the design phase, and another one that generates deployable code from the user model.

This common approach has two main issues. On the one hand, the semantic gap created by transformations renders more complex the interpretation of diagnosis results. On the other hand, it is relatively difficult to prove the equivalence between all these models because each one is defined in terms of a different language with different semantics. In summary, we notice that the root cause of these problems is the use of multiple definitions of the language semantics defined by transformations towards different formalisms.

To solve these issues, we propose an approach based on a single semantics definition that overcomes the equivalence issue and guarantees the absence of semantic gap. Transformations that modify the semantics should be avoided to preserve the uniqueness of this definition. In fact, our solution consists of using a unified semantics for both design and deployment. This unique definition has been implemented in a bare-metal interpreter of models. To ensure executability of these models, the selected semantics should be complete and without inconsistencies. For this interpreter, we have chosen tUML [2, 3], a textual notation for a subset of UML, that fits these requirements. This interpreter enables to execute software applications at a higher level of abstraction. While this approach solves semantic issues, a lack of diagnosis tools appears. To fix this problem, our idea is to share the unified semantics of the language with diagnosis tools (e.g., simulator, debugger). A generic communication interface enables the connection of these tools to the interpreter to observe and control model execution.

Experimentations performed on a railway level crossing system show that our approach is on the way towards feasibility. Our UML execution engine can be deployed on a PC (to be used during the design phase) as well as on two embedded targets: at91sam7s and stm32. To control the interpreter, a communication link has been developed to connect different diagnosis tools. For the moment, we have a simulator that enables users to observe and control the execution of the model. We illustrate this feature by computing the exhaustive state-space of the level crossing model.

The paper is organized as follows. Section II presents our approach and our main contribution: a bare-metal UML model interpreter. In Section III, we show results of our experimentations on the level crossing example. Finally, related works are discussed in Section IV and we conclude in Section V.

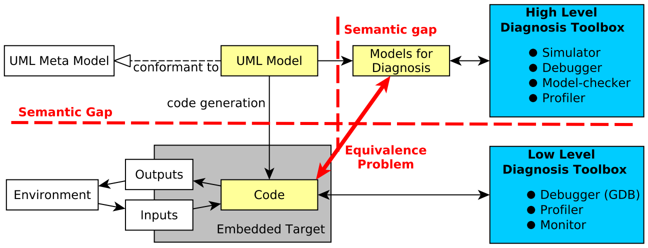


Figure 1. Diagram of the classical approach

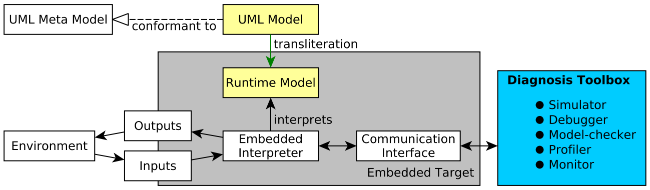


Figure 2. Diagram of our approach using a model interpreter

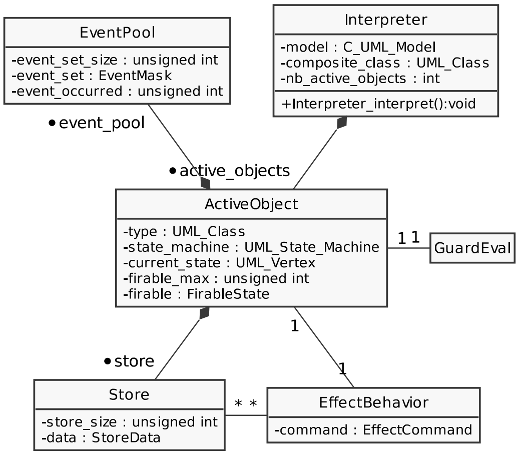


Figure 3. Class diagram of the interpreter

II. Design of a Bare-Metal UML Interpreter

A. Approach Overview

Our approach introduces an interpreter of UML models implemented in C language, which has the specificity to use a single definition of the semantics for both the design phase and the execution. To perceive the innovative aspect of this technique, it is important to understand the common approach used by engineers to develop software systems (Figure 1). First of all, engineers model the system using the semantics definition of dedicated modeling languages (e.g., UML, fUML, SysML). At this stage, many diagrams are typically produced to describe the system under multiple points of view. Some of them are dedicated to system execution while others are used for alternative purposes (e.g., testing, engineers understanding of the system). Diagrams required for execution are transformed into code using manual or automatic code generation. This transformation may change the semantics definition of the model and creates a semantic gap between the design model and the code. Hence, it is more difficult to identify an element of the design model in code and conversely. In parallel of this first activity, the design model is also transformed into other formalisms to be used by diagnosis tools for different purposes (e.g., simulation, exploration, formal verification). With this second transformation, a new issue appears. It is now more difficult to prove the equivalence between these models used for diagnosis and validation, and the deployed code.

Our approach (Figure 2) consists of modeling a system using the tUML language, that offers a clear and complete definition of a model without inconsistencies [2, 3]. With tUML, three views of the system are needed to define an executable model: class diagram, state machines, and composite structure diagram. Then, this model is serialized to C to be loaded into the interpreter. The serialization is applied only to elements needed for the execution (i.e., runtime data and runtime code) to adapt the syntax of the model without changing the semantics. This operation is a direct transliteration of each element of the model into C language as struct initializers, nothing more. Afterwards, this executable model can be executed by the model interpreter. One thing we still need to establish is the determinism of the interpreter execution (i.e., the absence of undefined behaviors [4]). This could be validated with static or dynamic analysis of the interpreter code.

This interpreter solves the three main drawbacks of the common approach. Firstly, the problem of equivalence between the design model and the code has been overcome thanks to the single definition of the semantics. Secondly, the semantic gap has been removed because no transformation, which alters or changes the semantics, is used. Thirdly, this unique definition facilitates the understanding of diagnosis results in terms of design model concepts. This solution has also other advantages. It may contribute to save time during the design phase by modeling only needed points of view of the system. Like code generation, the coding step requires no effort and the risk to introduce bugs is reduced. In our approach, this goal will be addressed by simulating and deploying the model directly on the interpreter embedded on the target.

Nevertheless, the execution of a DSL like tUML, introduced in [2, 3], creates another issue. Indeed, a lack of diagnosis tools (e.g., simulator, model-checker, debugger, or profiler) can be noticed for some specific languages. Rather than implementing an ad-hoc toolbox for each DSL, a solution (Figure 2) is to link the interpreter with existing tools [5]. Our model interpreter has been designed to be controlled remotely through a simple and generic API. This feature enables control of model interpretation either step by step or in a back-in-time way [6–8]. This interface is exactly what is needed to connect a model-checker, like LTSmin [9] or tools from ENSTA Bretagne [10, 11], for formal properties verification.

B. Interpreter Design

The model interpreter introduced in this paper has been implemented in C, a low level and general purpose language perfectly suited for embedded systems. The goal of this interpreter is to define a unique semantics definition that will be used for both design and deployment, and to execute models according to this semantics.

We will now give an overview of the interpreter architecture (Figure 3) rather than a detailed description that exceeds the scope of this article. The *Interpreter* class is composed of a collection of *ActiveObjects* and of the user model conformant to the tUML semantics. tUML basically corresponds to a subset of the class, state machines, and composite structure diagrams from UML. This last diagram defines a composite class, called SUS (for System Under Study), which goal is to specify the part played by each *ActiveObject* as well as links between them. Every *ActiveObject* has a state machine that defines its behavior, the current state of its state machine, and the list of all fireable transitions from its current state. It can compute fireable transitions, and fire a transition to make its current state evolve. An *ActiveObject* also owns an *EventPool* to store occurrences of events that it receives. The *EventPool* is currently designed as a bits field, which offers some advantages (e.g., a bounded representation of events) and defines a particular execution semantics. This specific semantics can be modified by using other data structures (e.g., FIFO) to implement different features (e.g., storage of multiple occurrences of a given event, preservation of events reception order). The *EventPool* has three principal operations: one to signal an event, one to consume an event, and the last one to check if an event has occurred. An *ActiveObject* also has a *Store* where values of its attributes are stored. The *Store* has only two operations: one to assign a value to an attribute, and another to get the value of an attribute. To have a generic representation of attributes, we used a generic pointer, which points to the suitable value of the attribute, and an integer to store the size of its data type. Furthermore, an *ActiveObject* needs to evaluate guards on transitions of its state machine. The class *GuardEval* has been designed for this purpose. Currently this class involves Flex and Bison to parse guard expressions, but future works will improve that evaluation using offline preprocessing to reduce the memory footprint and enhance execution performances. The last class is the *EffectBehavior* that executes effects associated to transitions when one of them is fired. Two kinds of effects are available yet with the simplified version of the ABCD language [3] used here. It is possible to send events to another *ActiveObject* or to assign a value to an attribute.

This interpreter has been designed to be portable on different targets. At the moment, this interpreter can be executed on a PC but is also supported on two embedded targets: at91sam7s and stm32. Other targets could be easily added because we designed the interpreter to be linked with target-specific libraries of the chosen target at build time. This specificity helps to fit embedded systems requirements, which involve limited computation and memory resources.

C. Communication Interface

To solve the problem of the lack of diagnosis tools with DSLs, we take into account the possibility to connect the interpreter with existing tools (e.g., simulator, debugger, profiler, model-checker). Our idea is to provide a generic API to be able to control remotely the execution of the interpreter. This simple interface is composed of four requests. **Get configuration:** collects the current configuration (memory state) of the interpreter. A configuration is composed of the current state, the *EventPool* and values of attributes, of each *ActiveObject*. **Set configuration:** loads a configuration as the current memory state of the interpreter. **Get fireable transitions:** gets transitions from *ActiveObject* instances that have their trigger and their guard satisfied in the current state. A fireable transition is identified by its id and the *ActiveObject* to which it belongs. **Fire a transition:** fires a fireable transition of an *ActiveObject*. When firing, the event of the trigger is consumed, the current state is updated, and effects attached to this transition are processed.

With only four requests, this simple interface offers several possibilities. Indeed, the model interpreted can be executed step by step. A specific execution path can be chosen with the ”Get fireable transitions” request. This feature enables to check a property or find a bug in a specific path. However, the most powerful functionality is the ability to make back-in-time execution [6–8]. The ”Set configuration” request can place or replace the interpreter in any configuration. This feature can be useful for debugging if we want to come back on previous configurations but also for the exploration of the state-space.

This generic API gives the possibility to use several kinds of tools in an easy way. Indeed, to connect an existing tool, you just need to add a TCP client and to implement the communication interface. In our point of view, it is better to use existing tools than implementing ad-hoc tools. One reason is that existing tools have been tested, used and approved for many years. Engineers are familiar with these tools and no formation will be required to teach them how to use them.

Our interpreter can support multiple kinds of connections even if only TCP connection for PC, and RS232 connection for at91sam7s and stm32 have been developed for the moment. With a strategy pattern [12], only the code of the chosen connection is compiled and loaded in the target, which contributes to reduce memory footprint. A new kind of connection can be easily added by implementing the connection interface. The interpreter should only know how to open, read, write and close the connection. It is important for us to offer this functionality because embedded boards have sometimes few, and specific kinds of serial ports available. Furthermore, to avoid implementing a serial communication within an existing tool, we have created a little application that converts TCP data frames into serial data frames. With this converter, only a TCP client plugin needs to be implemented into a tool so that it can control the interpreter over any serial connection.

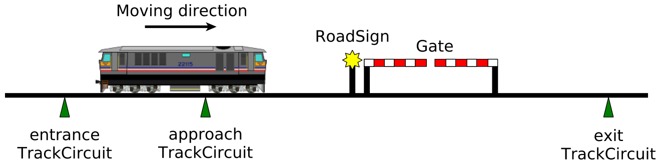


Figure 4. Schema of the level crossing example

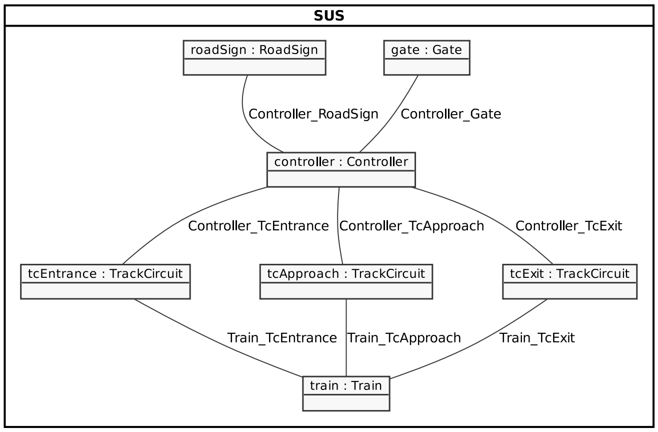


Figure 5. Composite structure diagram of the level crossing example

III. A UML Model on an Embedded Target

A. Case Study

To illustrate our approach we use a level crossing system, presented in Figure 4. A level crossing is a system built at the intersection of a railroad and a road. It is responsible for ensuring the safety of all road users during the passage of the train. In this case study, a UML model of a level crossing has been designed (Figure 5 and Figure 6). The *Controller* controls the activation of the *RoadSign* (e.g, lights, alarm) and the closure of the *Gate* according to signals received from *TrackCircuits*. *TrackCircuits* are sensors able to detect the *Train* on the railroad. Each one sends a signal when the *Train* is detected and another one when the detection ends. The level crossing counts three *TrackCircuits*: one for the entrance, one for the approach, and one for the exit of the level crossing.

All these objects are active, so their behaviors are described by state machines. A state machine is composed of states linked together with directed transitions. Transitions can be fired to change the state of an *ActiveObject*. A transition is fireable if the event associated with its trigger has been received by this state machine and if its guard is satisfied. When a transition is fired, its effect is processed. The language used to parse guards defined as *OpaqueExpressions* and effects defined as *OpaqueBehaviors* is a simplified version of ABCD [3]. The current version of the interpreter supports simple boolean expressions for guards as well as the sending of events and assignment of variables for effects. This is sufficient to execute a model and make *ActiveObjects* interact together.

We will now describe behaviors of *ActiveObjects* (Figure 6) to better understand these interactions. The *Train* state machine triggers sequentially the detection of the entrance *TrackCircuit* (tcEntrance), of the approach *TrackCircuit* (tcApproach) and of the exit TrackCircuit (tcExit). This linear state machine considers the passage of a single train on the level crossing. *TrackCircuits* state machines have only two states *Detection* and *NoDetection* that transmit signals to the controller at the beginning and at the end of train detection. State machines for tcApproach and tcExit are similar to the state machine of tcEntrance presented on Figure 6. The *Controller* state machine has the most important role in the system. When the train is detected by tcEntrance, it is in charge of sending a signal to switch on lights of the roadSign. Then, it sends a signal to close the gate when the train begins to be detected by tcApproach. At the end of the detection, we consider that the train is passing until the detection of tcExit, which sends a signal to open the gate. Finally, a signal is send to switch off lights of the roadSign when the train quits the detection zone of tcExit, and the controller returns in its idle state. The state machine of the RoadSign can alternate between *Inactive* and *Active* states thanks to *switchOn* and *switchOff* signal events.

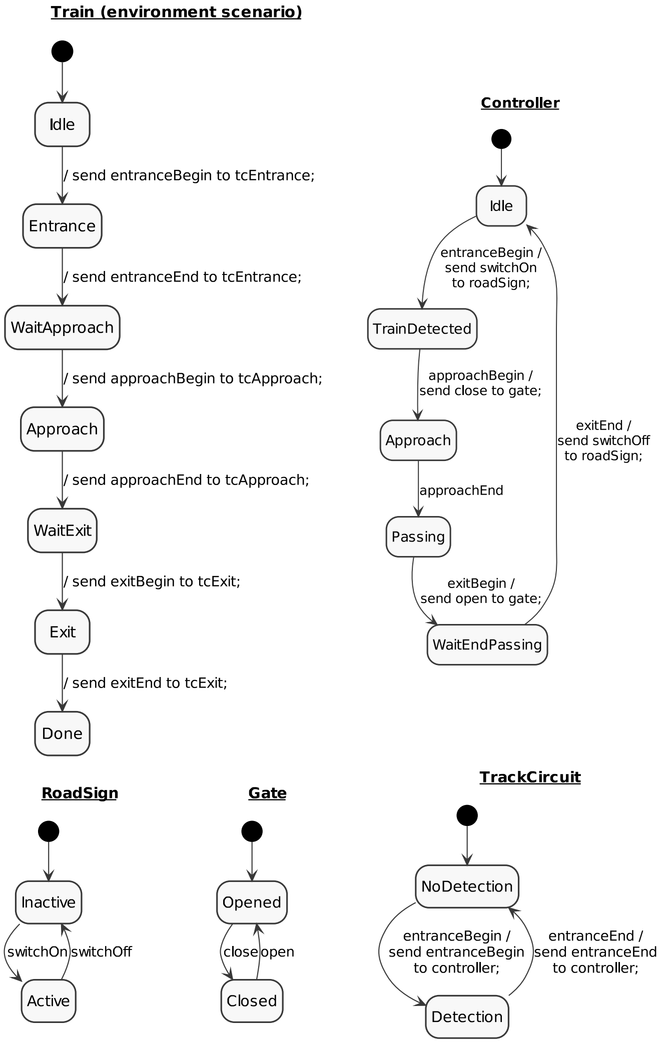
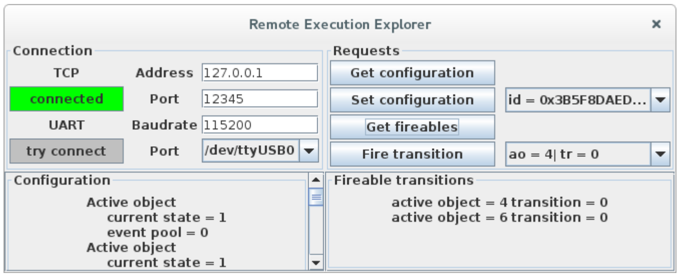

Train (environment scenario)
Idle
/ send entranceBegin to tcEntrance;
Entrance
/ send entranceEnd to tcEntrance;
WaitApproach
/ send approachBegin to tcApproach;
Approach
/ send approachEnd to tcApproach;
WaitExit
/ send exitBegin to tcExit;
Exit
/ send exitEnd to tcExit;
Done
Controller
Idle
entranceBegin / send switchOn to roadSign;
TrainDetected
approachBegin / send close to gate;
Approach
approachEnd
Passing
exitBegin / send open to gate;
WaitEndPassing
exitEnd / send switchOff to roadSign;
RoadSign
Inactive
switchOn
switchOff
Active
Gate
Opened
close
open
Closed
TrackCircuit
NoDetection
entranceBegin / send entranceBegin to controller;
entranceEnd / send entranceEnd to controller;
Detection

Figure 6. State machines of the level crossing example



TCP Address 127.0.0.1
connected Port 12345
UART Baudrate 115200
Port /dev/ttyUSB0
id = 0x3B5F8DAED...
ao = 4| tr = 0

```
Active object
    current state = 1
    event pool = 0
Active object
    current state = 1
```

```
active object = 4 transition = 0
active object = 6 transition = 0
```

Figure 7. User interface of the remote execution controller

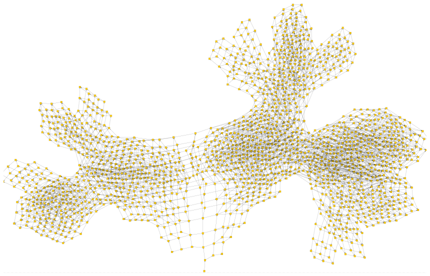


Figure 8. Graph of the state-space exploration of the level crossing example

In the same way, the state machine of the *Gate* is driven by *open* and *close* events to go in the *Opened* or *Closed* state.

This example model is not connected with its real environment because the link of the interpreter with inputs and outputs of the embedded board have not been implemented yet. However, the level crossing of our case study has sensors (*TrackCircuits*) and actuators (*RoadSign* and *Gate*) that have been integrated to the design model. This enables to simulate the level crossing example as if it was executing on the real system, which uses inputs and outputs of the board. In fact, only the controller will need to be deployed on the real system because other entities are active elements of the environment.

B. Deployment and Results

The level crossing case study has been deployed on our interpreter of UML models. This experimentation has been made on a PC with a Linux operating system, and on both at91sam7s and stm32 targets without operating systems (bare-metal). The level crossing model has been designed using tUML before being serialized into Eclipse UML, and finally into C. It has been successfully executed on the interpreter by firing the first fireable transition on active objects in turns. Without any optimizations, the memory footprint of the binary executable file containing the model, the interpreter, and specific libraries of the board for the stm32 is only 131 ko, which is sufficiently small to be embedded.

We have also connected two diagnosis tools to control remotely the execution of the level crossing model. A simulator has been developed to explore some execution paths of this model. It can be connected to the interpreter using either a TCP connection or a serial connection. Indeed, a serial port communication using a UART peripheral of the PC was developed to communicate directly with the embedded interpreter for debug purpose but this functionality has become useless with the communication converter. The simple user interface (Figure 7) provides four buttons to apply each request of the API and directly control the model execution. This gives the possibility to the user to get information on the execution and to make the model evolve by firing transitions. To observe the model execution, the simulator has also the ability to decode the configuration and to display its content in an understandable way for humans.

Another diagnosis tool has also been connected to the model interpreter for state-space exploration. For this purpose, the communication unit developed in the simulator has been integrated into this state-space explorer. This tool has computed the state-space of the level crossing model using a breadth first search algorithm connected with the interpreter via the TCP diagnosis interface. This results in a state-space constituted of 1,825 different configurations linked to one another with 5,793 transitions (This state-space exploration has been performed with 3 events preloaded in the *EventPool* of the controller: *entranceBegin*, *exitBegin*, and *exitEnd*.). The associated state-space graph shown in Figure 8 gives an overview of the extent of this model exploration.

To sum up, these experimentations show the ability of our interpreter to be remotely controlled by diagnosis tools for simulation and exploration purposes.

IV. Related Work

Some generic tools are able to build an interpreter, and execute models designed with a given DSL. GEMOC studio is an Eclipse plugin that integrates a language workbench, and a modeling workbench to define a DSML as well as design models conforming to this language. This environment can be used to build a model interpreter, and other diagnosis tools (e.g., a graphical model animator, a trace manager) [13]. Another example is the K framework [14], an executable semantics framework based on rewriting rules. It can be used to define programming languages or create tools. The possibility to target a large number of languages is a significant advantage of these frameworks. However, this also induces a lack of performance because the implementation is not specifically adapted to the application. Contrary to these tools, our interpreter has been designed in a classical way but with the goal to fit requirements of embedded systems. Our approach takes also into account that the interpreter has to be connected to existing diagnosis tools rather than generated new ones for each DSML. Some alternative methods (e.g., [1, 15–17]) focus on a semantics definition based on state machines. The transformation of these diagrams into intermediate formalisms like EHA enables the direct execution of these state machines. These tools are mainly used for simulation [18], exploration [19] and model-checking [1] purposes. However, they do not defined a generic interface of communication like our interpreter and it is not possible to control their execution on an embedded target. The key point of our approach is the use of a single semantics definition, which fixes the equivalence issue between design model and code. This could also be achieved using verified compilers or interpreters (e.g., the CompCert C compiler [20], and Jitk [21]). The formal language Coq enables to check and certify formal properties of programming languages. Correctness of generated or interpreted code is formally proven. It ensures this way the equivalence of the design model with the code. However, this kind of tools is not widely used yet due to its complexity.

Another essential problem for the execution of models is the lack of tools after deployment. This issue has not been completely addressed yet. For UML, some simulators and interpreters begin to appear without being widely used at the moment. For other DSLs, only generic frameworks or the use of a design pattern dedicated for monitoring [5] are able to generate this kind of tools. We differentiate our interpreter by the use of a generic interface to connect it with existing tools.

Numerous related works have also been achieved on real time virtual machines (VMs) [22–25]. Several concepts used by researchers on this topic have been applied to DSL execution engines. All these VMs interpret bytecode, so one important difference is that our model interpreter provides a higher level of abstraction. Some of these VMs have been implemented using specific languages (e.g., Estérel [26]). In comparison with our interpreter, UML is more adapted to model systems. Furthermore, UML is widely used during the design phase of systems in the industrial world. Using the same language for the design phase and the code is better because no semantic gap is created. Virtual machines for sensor networks (e.g., Maté [27], and Mote runner [28]) have similarities with our work even if our approach is more focused on industrial applications with both sensors and actuators. Nevertheless, VMs for sensors networks face several issues especially on communication speed and execution that we will also need to solve for making our interpreter scalable.

V. Conclusion

In this paper, we have introduced an interpreter of UML models with the specificity to use a single semantics definition for both the design phase and the deployment. This approach eliminates the problem of the equivalence between models, and the semantic gap created by model transformations. The use of tUML enables to define complete executable models that can be directly executed on the interpreter by serializing the model into C language. A first step to solve the lack of diagnosis toolboxes for DSLs has been made with the implementation of a generic communication interface that enables the use of existing tools. This enables to control remotely the execution of the interpreter step by step or in a back-in-time way. We illustrate this functionality by implementing a simple simulator that communicates with the interpreter running on an embedded board. This simulator can be used to simulate a model and to explore some execution paths. We have also confirmed that we can explore a full state-space.

Future work will focus on linking this engine with the LTSmin [9] model-checker and tools from ENSTA Bretagne [10, 11] to verify formal properties directly on models executed by the interpreter on embedded boards. We are also interested in interfacing other diagnosis tools for several purposes: monitoring, debugging, and profiling.

References

[1] G. Pintér and I. Majzik, "Program Code Generation based on UML Statechart Models," *Periodica Polytechnica*, vol. 47, no. 3–4, pp. 187–204, 2003.

[2] F. Jouault and J. Delatour, "Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems," in *OCL 2014*, A. D. Brucker, C. Dania, G. Georg, and M. Gogolla, Eds., vol. 1285, Valencia, Spain, Sep. 2014, pp. 73–82.

[3] F. Jouault, C. Teodorov, J. Delatour, L. Le Roux, and P. Dhaussy, "Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD," *Génie logiciel*, vol. 109, p. xx, Jun. 2014.

[4] J. Regher, "A guide to undefined behaviours in c and c++," 2010.

[5] Z. Drey and C. Teodorov, "Object-oriented design pattern for dsl program monitoring," in *Proceedings of SLE 2016*. New York, NY, USA: ACM, 2016, pp. 70–83.

[6] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xdsmls," in *Proceedings of SLE 2015*, New York, USA, 2015, pp. 137–148.

[7] J. Corley, B. P. Eddy, E. Syriani, and J. Gray, "Efficient and scalable omniscient debugging for model transformations," *Software Quality Journal*, vol. 25, no. 1, pp. 7–48, Mar 2017.

[8] S. V. Mierlo, Y. V. Tendeloo, S. Mustafiz, and B. Barroca, "Debugging parallel devs," 2014.

[9] G. Kant, A. Laarman, J. Meijer, J. Pol, S. Blom, and T. Dijk, "Ltsmin: High-performance language-independent model checking," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 692–707.

[10] C. Teodorov, P. Dhaussy, and L. Le Roux, "Environment-driven reachability for timed systems," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 2, pp. 229–245, Apr 2017.

[11] C. Teodorov, L. Le Roux, Z. Drey, and P. Dhaussy, "Past-free[ze] reachability analysis: reaching further with dag-directed exhaustive state-space analysis," *Software Testing, Verification and Reliability*, vol. 26, no. 7, pp. 516–542, 2016, stvr.1611.

[12] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[13] B. Combemale, J. Deantoni, O. Barais, A. Blouin, E. Bousse, C. Brun, T. Degueule, and D. Vojtisek, "A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio," in *8th Transformation Tool Contest*. l'Aquila, Italy: CEUR, 2015.

[14] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[15] T. Schattkowsky and W. Muller, "Transformation of uml state machines for direct execution," in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 117–124.

[16] G. Pintér and I. Majzik, "Automatic Code Generation Based on Formally Analyzed UML Statechart Models," in *Proceedings of the FORMS-2003 Conference*, G. Tarnai and E. Schnieder, Eds. Budapest, Hungary: LH́armattan, May 15-16 2003, pp. 45–52.

[17] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier, "Contracts for model execution verification," in *Proceedings of the 7th European Conference on Modelling Foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 3–18.

[18] A. Kirshin, D. Dotan, and A. Hartman, *A UML Simulator Based on a Generic Model Execution Engine*. Springer Berlin Heidelberg, 2007, pp. 324–326.

[19] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe, "The architecture of a uml virtual machine," in *Proceedings of OOPSLA '01*. New York, NY, USA: ACM, 2001, pp. 327–341.

[20] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart, "The CompCert Memory Model, Version 2," INRIA, Research Report 7987, Jun. 2012.

[21] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: A trustworthy in-kernel interpreter infrastructure," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014, pp. 33–47.

[22] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, “Design and implementation of a comprehensive real-time java virtual machine,” in *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*. New York, NY, USA: ACM, 2007, pp. 249–258.

[23] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, “A real-time java virtual machine with applications in avionics,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 1, pp. 5:1–5:49, Dec. 2007.

[24] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes, “A real-time java virtual machine for avionics - an experience report,” in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 384–396.

[25] D. Simon and C. Cifuentes, “The squawk virtual machine: Java$^{\text{TM}}$on the bare metal,” in *Companion to the ACM SIGPLAN OOPSLA ’05 Conference*. New York, NY, USA: ACM, 2005, pp. 150–151.

[26] B. Plummer, M. Khajanchi, and S. A. Edwards, “An esterel virtual machine for embedded systems,” in *Proceedings SLAP ’06*, vol. 126, Vienna, Austria, 2006, pp. 912–917.

[27] P. Levis and D. Culler, “Maté: A tiny virtual machine for sensor networks,” *SIGPLAN Not.*, vol. 37, no. 10, pp. 85–95, Oct. 2002.

[28] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, “Mote runner: A multi-language virtual machine for small embedded devices,” in *Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 117–125.