Unified LTL Verification and Embedded Execution of UML Models

Valentin Besnard  
ERIS, ESEO-TECH  
Angers, France  
valentin.besnard@eseo.fr

Matthias Brun  
ERIS, ESEO-TECH  
Angers, France  
matthias.brun@eseo.fr

Frédéric Jouault  
ERIS, ESEO-TECH  
Angers, France  
frederic.jouault@eseo.fr

Ciprian Teodorov  
Lab-STICC UMR CNRS 6285,  
ENSTA Bretagne  
Brest, France  
ciprianteodorov@ens-bretagne.fr

Philippe Dhaussy  
Lab-STICC UMR CNRS 6285,  
ENSTA Bretagne  
Brest, France  
philippe.dhaussy@ens-bretagne.fr

ABSTRACT

The increasing complexity of embedded systems leads to uncertain behaviors, security flaws, and design mistakes. With model-based engineering, early diagnosis of such issues is made possible by verification tools working on design models. However, three severe drawbacks remain to be fixed. First, transforming design models into executable code creates a semantic gap between models and code. Furthermore, for formal verification, a second transformation (towards a formal language) is generally required, which complicates the diagnosis process. Finally, an equivalence relation between verified formal models and deployed code should be built, proven, and maintained. To tackle these issues, we introduce a UML interpreter that fulfills multiple purposes: simulation, formal verification, and execution on both desktop computer and bare-metal embedded target. Using a single interpreter for all these activities ensures operational semantics consistency. We illustrate our approach on a level crossing example, showing verification of LTL properties on a desktop computer, as well as execution on a stm32 embedded target.

CCS CONCEPTS

• Software and its engineering → Formal software verification;
  • Computer systems organization → Embedded software;

KEYWORDS

UML Execution, Model Interpretation, LTL Model-Checking, Embedded Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MODELS '18, October 14–19, 2018, Copenhagen, Denmark. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239395

1 INTRODUCTION

Embedded systems and cyber-physical systems tend to become more and more complex due to the emergence of new needs and applications (e.g., Internet of Things, robotics, smart cities). Not only are these devices more difficult to make safe and secure, but these software intensive systems are also more exposed to both design and programming mistakes. Therefore, the implementation of embedded applications becomes a tedious and error-prone task. With model-based engineering, dedicated design methods have appeared to simulate, explore, and validate models at early design stages. Nowadays, the industry standard for using models in embedded systems is code generation that involves the use of two transformations. The first one provides a formal model required for verification at early design stages. The second one produces actual application code, and may be automatic, semi-automatic, or entirely manual.

While this classical approach enables users to both verify the models and execute the systems resulting from these models, three severe drawbacks remain. The first one is the semantic gap created by code generation, between models and their corresponding code. This gap makes it more difficult to link code fragments to concepts of the design model. The second issue is caused by transformations into formal models. These transformations complicate the understanding of diagnosis results because these results are not directly expressed in terms of design concepts. The last problem is the use of multiple separate definitions (i.e., at design model level, in formal model generator, and in executable code generator) of the modeling language semantics that are generally not proven to be equivalent. In fact, all of these issues are directly linked to the existence of multiple implementations of the language semantics. This is a result of using transformations into different languages.

To tackle these problems, we avoid having multiple definitions of the modeling language semantics. Instead, we rely on a single semantics definition that we specifically build to support all three activities: simulation, formal verification, and execution. In practice, this single definition takes the form of a model interpreter for UML (or a subset of UML). This interpreter can be remotely controlled by diagnosis tools to verify and validate the design model.
Therewith, it provides an execution platform running either with OS support on desktop computers or without OS (bare-metal) on resource constrained embedded targets. This approach improves diagnosis understandability and verification reliability because what is executed is really what has been checked.

To show the versatility of our approach, we have connected our interpreter to an extension of the OBP model-checker [36, 37] (https://plug-obp.github.io/) to enable formal property verification. Using this setup on a level crossing controller model, we have successfully verified a set of properties expressed with Linear Temporal Logic (LTL).

The remainder of this paper is structured as follows. Section 2 introduces our illustrating example. Section 3 shows an overview of the project before we describe the design of the model interpreter in Section 4. Then, we discuss the communication architecture used to control model execution, as well as its extension to support formal verification in Section 5. In Section 6, we present the results of applying the approach to our example. Section 7 reviews some related work, and we conclude in Section 8.

Figure 1: Overview of the level crossing system.

Figure 2: Composite structure diagram of the level crossing model.

2 ILLUSTRATING EXAMPLE

To illustrate our approach, we use a model of a level crossing controller as example (Figure 1). Level crossing protection systems are used at intersections of railways and roads to warn and protect road users from train traffic. These systems are typically equipped with sensors, that we will call EntranceTC and ExitTC (TC means Track Circuit), to detect trains, as well as actuators such as a Gate or flashing lights, that we will call RoadSign and RailwaySign.

A UML model of a level crossing controller has been designed using active objects with state machine behaviors [29] (Figure 2 and Figure 3). Guards and effects of the state machine transitions are expressed using an action language allowing arithmetical and logical operations, as well as sending and receiving events to other active objects.

The Controller coordinates the movement of the Gate as well as the activation of both RoadSign and RailwaySign according to signals received from EntranceTC and ExitTC. The entrance sensors warn the system of train arrivals by sending the entrance signal to the controller. In a same way, the exit sensor notifies the controller when the train leaves the level crossing by sending the exit signal. To process these signals, the Controller state machine has two loops. The first one is dedicated to closing the Gate and switching on the RoadSign before the train passage. The authorization is given to the train to pass by switching off the railwaySign. The second one is responsible for opening the Gate and switching off the RoadSign when no more trains are engaged.

Figure 3: State machines of the level crossing model.
In addition to simulation and state-space exploration on this model (as depicted in [5]), our goal in this paper is to illustrate that we can formally verify properties through model-checking. For this example, we have selected four properties to check the reliability of the system:

1. The Gate is closed when the Train is on the level crossing.
2. The light of the RoadSign is active when the Train is on the level crossing.
3. The Gate finally opens after being closed.
4. The light of the RoadSign finally shuts down after being activated.

3 OVERVIEW

To better understand the scope of this work, Figure 4 gives an overview of the UML interpreter project.

First, an executable UML Model (XMI) of the system under study must be designed and saved under a dedicated formalism (here XMI). This model is typically produced during the design phase of the system development. It can then be serialized into C programming language for being loaded at compile-time in the model interpreter. The serialization generates only data needed for model interpretation (UML Model (C)) and data types used to access instances of the UML model in our action language (Data Types for Action Language). Contrary to code generation, no functions are generated by this serializer for model execution (except for transitions guards and effects). In fact, the execution semantics is given explicitly by the Interpreter Source Code. The UML model and data types (data) as well as source code of the interpreter (program) are compiled and linked together to give the executable code of the Interpreter.

This executable code is composed of four important modules. The Runtime Model is the model used for execution. It consists of the static part of the model produced by the serializer and of its associated dynamic part used to store values of dynamic data (e.g., values of attributes, values of current state of state machines). The Interpreter module is in charge of the model interpretation of the Runtime Model. It must execute the system in conformance to the behavior described in this model with the execution semantics provided by this Interpreter. It is also possible to connect a Diagnosis Tool to this model interpreter using the Pilot module and the Converter software that converts TCP into the appropriate protocol used by the target. In case of model-checking, the Pilot uses services of the Checker to evaluate some boolean expressions of formal properties. This approach ensures that the same semantics definition is used for formal verification of the system and for its execution on the real embedded target.

The executed system can also interact with its Environment through inputs and outputs of the target, which can be a desktop computer or an embedded bare-metal target. To sum up, this project offers a solution for model execution and model diagnosis of UML models using a single semantics definition for various activities (e.g., simulation, model-checking, execution) typical of the embedded systems development process.

4 INTERPRETER DESIGN AND IMPLEMENTATION DETAILS

The design of the UML Interpreter is guided by the trade-off between the embedded system memory and performance constraints, and the need to provide powerful diagnosis facilities. To reach this goal, we present the process employed to generate the model used at runtime, the action language used to express the system behavior, as well as the implemented execution mechanism.

4.1 Generation of the Runtime Model

To deal with embedded constraints, a key point of our approach is the generation process of the UML model, used by the interpreter at runtime (Figure 6). First, a model of the system under study must be designed. This design model can be defined using either a

![Figure 4: Overview of the UML interpreter project.](image)

![Figure 5: Serialization of a UML class into C language.](image)
Figure 6: Process used for runtime model generation.

Figure 7: Class diagram of the interpreter design.

textual editor (Figure 5.A) [20], or a graphical editor (Figure 5.B) [21], and is generally saved under the XML metadata interchange (XMI) format. However, this format is cumbersome and not suitable for being loaded on an embedded microcontroller. To reduce the memory footprint and ease model execution, the UML model in XMI is serialized into C language, the implementation language of our interpreter. This serialization can be seen as the way to load the model into the interpreter. In practice, any element of the model is converted into a struct initializer in C language. For example, the instance of the Controller class of the level crossing system is serialized as depicted in Figure 5.C. Unlike code generation approaches that generate both data and program required to execute the system, here only data representing the static part of the model are generated. The whole program required to execute the system is provided by the interpreter source code (Figure 6).

Moreover, this UML to C serializer also generates model-specific data types used by the action language to access UML instances at runtime. At this point, all the source code is available and can be compiled with a C compiler to produce the executable code of the system (Figure 6). This executable artifact contains both the executable code of the interpreter and the runtime model that will be the reference for model execution and diagnosis. Hence, the UML model in XMI format may be considered a view of the runtime model.

4.2 Interpreter Design

To better understand the design of the model interpreter, Figure 7 introduces a simplified class diagram of its internal architecture. The different C modules of the interpreter are represented by classes in the class diagram. The Metamodel package provides the UML metamodel supplying class, composite structure, and state machine concepts. In practice, each class of the metamodel is defined into C language as a structured type and each attribute is represented as a field of that structure. The Model package contains the model and data types generated by the UML to C serializer. The UML model is encapsulated in the Model class and represents the static part of this model. Data type classes are used by the action language to access data of the model in a straightforward way. The Store class represents the dynamic part of the model composed of current state of state machines, event pools, and values of attributes. The Peers class provides links between active objects for model interpretation.

To interpret a model according to its execution semantics, an execution core is required. This is the purpose of the Interpreter package. The Interpreter class is the root class for model interpretation with the interpret() method as entry point. Its goal is to
coordinate the execution of the active objects. An active object is
defined for each part of the composite structure diagram with a
UML state machine that describes its behavior (e.g., Figure 2 and
Figure 3 in our example). An active object also makes the link be-
tween the static and the dynamic part of the model (e.g., to link an
attribute with a memory area where its value can be stored).

The active object state machine is executed according to the
events received by its event pool, events that trigger the state ma-
hine transitions. An event pool is composed of a dynamic part that
stores received events, and the meta information that contains its
characteristics (e.g., size, priority management). Events are added
in and consumed by the event pool during model execution.

We can notice that the order of event dispatching and the in-
terpretation of the reception of an event occurrence that does not
match a valid trigger are left undefined by the UML standard. For
these semantic variation points, we have implemented two algo-
rithms that both dispatch events following their arrival order. The
first one provides a first-in-first-out (FIFO) algorithm and ignores
events that do not match a valid trigger (from a current state). The
second one defers these kinds of events for being processed later.
In practice, the abstract EventPool is provided by a C header file.
The choice of the concrete event pool (FifoEventPool or OrderedList-
DeferredEventPool C module) used by the interpreter is done at
compilation time.

The execution of the state machine also needs to evaluate guards
and to execute effects associated to the state machine transitions.
The classes GuardEvaluator and EffectInterpreter provide required
mechanisms for these evaluations and executions through opaque
expressions and opaque behaviors, using an action language.

4.3 Action Language

Modeling languages often require action languages to ease the
description of model behaviors (e.g., for UML: Alf [27], OAL [18],
ASL [16], UAL [26]). As mentioned previously, in our design model,
an action language is employed to specify guards through UML
opaque expressions and to describe effects of state machine transi-
tions through UML opaque behaviors (Figure 3). To execute these
action language expressions, several solutions may be considered:
AST interpretation, bytecode interpretation, or native code exe-
cution. For a performance purpose, we have chosen to base our
approach on C native code execution, the native language of our
interpreter. We have then defined an action language that reuses the
syntax of the C programming language for arithmetic and logical
operations as well as conditions and loops expressions.

However, using C code, it remains difficult for users to express
domain-specific behaviors that are related to implementation-specific
concepts of the interpreter. To address this issue, our action lan-
guage based on C has been enhanced with a syntactic sugar for
sending events, for manipulating the event pools, and for accessing
the model elements such as attributes, states, and event identifiers.
All these expressions used in the UML design model are syntacti-
cally transformed into C functions during the model serialization.
These C functions are used at runtime to evaluate guards and exe-
cute effects (using GuardEvaluator and EffectInterpreter previously
introduced, Figure 7). Figure 8 shows an example of an opaque be-
behavior expressed in the design model, and its equivalent C function
generated during the model serialization. This opaque behavior,
extracted from the illustrating example, performs the sending of a
close event to the gate active object.

4.4 Interpreter Execution

Once the UML model has been set up on the embedded target, its
execution can be launched through an initialization step followed
by a sequence of interpretation steps.

At startup, all active objects are initialized to fire the initial
transition of their state machine from the initial pseudo-state to the
first state. When this initialization has been performed, the model
execution is divided into interpretation steps in charge of firing the
state machine transitions.

Each interpretation step begins by computing the set of fireable
transitions (i.e., transitions that have their trigger and their guard
satisfied). The step ends by firing the first fireable transition of each
active object. Note that if the model is deterministic at most one
transition is fireable at each execution point.

As a result, for each fired transition, the event matching the
trigger is consumed, the current state of the corresponding state
machine is updated, and the effect associated to the transition is
carried out.

5 CONTROL MODEL EXECUTION FOR
DIAGNOSIS PURPOSE

In order to ease the integration with diagnosis tools, our UML
interpreter provides a dedicated interface to connect them and to
control model execution. This section describes this interface, the
technique used to verify a set of propositions using the interpreter,
and explains how a model-checker can take advantage of these mechanisms to verify LTL properties on the runtime model.

5.1 Design of the Diagnosis Part

To perform model diagnosis (e.g., simulation or verification) directly on the executable code, our interpreter must be controlled remotely by diagnosis tools. The class diagram in Figure 9 focuses on the components needed for the diagnosis interface.

The Pilot class is the central element of this architecture. It coordinates the Interpreter execution with requests sent by the diagnosis tools through the communication stream. The target-specific Connection is responsible for connecting the model interpreter to the chosen diagnosis tool and the Communication class offers facilities to the Pilot for reading and writing on data streams. Furthermore, the Pilot can also use services provided by the Checker to evaluate arbitrary boolean-valued expressions on the runtime execution.

Using this architecture, every diagnosis may be performed on both desktop computer and embedded target. For instance, for a hardware-in-the-loop simulation the diagnosis tools can be connected to the embedded target, while for model-checking the desktop computer is usually more suitable.

5.2 Connecting the Diagnosis Tools

To facilitate the connection of diagnosis tools to multiple kinds of embedded targets or desktop computers, we have introduced a converter (Figure 10). The role of the converter is to translate TCP frames into the appropriate protocol available on the target (i.e., serial on embedded targets or TCP on desktop computers).

To connect to our model interpreter, diagnosis tools need to implement a TCP client and the application layer protocol used to exchange communication messages with the Pilot.

This protocol defines five requests for controlling model interpretation:

- **Get configuration** collects the current memory state of the interpreter (i.e., the store) which is the dynamic part of the model. The configuration is typically composed of the current state of all active objects, values of their attributes, and the content of their event pools.
- **Set configuration** loads a configuration as the current memory state of the interpreter. This request enables to set each active object in a desired state, to set values of their attributes, and to inject events in their event pools.
- **Get fireable transitions** collects transitions that have their trigger and their guard satisfied in the current configuration. These transitions could be fired on the next interpretation step.
- **Fire transition** fires a single transition of a given state machine active object. Only fireable transitions of the current configuration can be fired using this request. Events that may trigger the fired transition are consumed and the effect associated to this transition is executed.
- **Reset interpreter** restarts the interpreter from the initial state of the model. This resets all active objects in their initial states, resets attributes to their default values, and empties event pools.

This protocol is sufficient to control any step-by-step model execution with backward navigation facilities (as omniscient debugging [6]). It has been successfully used to perform state-space exploration of models on desktop computers and embedded targets [5]. In this study, state-space exploration has been enhanced with properties verification for model-checking.

5.3 Verification of Formal Properties

During state-space explorations, model-checking tools aim at verifying formal properties. A formal property consists of multiple expressions, called atomic propositions, linked together with logical operators.

These propositions are expressed in terms of the model concepts. However, the knowledge of the configuration is not sufficient for their evaluation that also depends on the model semantics, which is implemented into the interpreter. Therefore, the model-checker must involve the interpreter in the verification process for the evaluation of these propositions.
For this purpose, the communication protocol has been extended with one request to submit atomic propositions to the interpreter:

**Evaluate atomic propositions** sends a set of atomic propositions and returns results of their evaluations. The interpreter performs these evaluations on the runtime model using services provided by the Checker (Figure 9).

```
evaluatePropositions(stringProps)
```

**Figure 11: Sequence diagram for the evaluation of a set of atomic propositions.**

The atomic propositions are expressed in the same action language as the design model (cf. Section 4.3). These expressions are side-effect free and are evaluated directly on the target platform in the same way as the guards of the model. However, contrary to guards that are compiled at the same time as the model, atomic propositions are not known at compilation time. They require to be compiled and loaded on the interpreter afterwards. To achieve this, the converter (Figure 10) is involved in the process: (i) to generate the C code corresponding to atomic propositions, (ii) to compile it, and (iii) to transmit the executable code to the checker of the interpreter. Figure 11 details this process for the evaluation of atomic propositions with the interpreter.

For a **desktop computer** usage, the resulting executable code (`binProp`) is a dynamic library, that can be loaded dynamically by the host operating system. For performance purpose, generation and compilation steps are ignored if the atomic proposition has already been compiled. Then, the converter requests the interpreter to load the executable code of atomic propositions if it is not currently available in its memory space. This operation uses a `loadIndex` to identify each proposition on the interpreter. The interpreter performs propositions loading using the services of the Checker. When the checker receives the atomic propositions, it opens the corresponding C libraries, loads the appropriate C functions, and computes the function pointers needed to call the respective functions. Finally, these function calls will be performed by the checker when the converter requests to evaluate these propositions at the end of the process.

For the **bare-metal targets**, the dynamic libraries cannot be used for loading dynamically executable code due to the absence of operating system. However, through the same process, the executable code of atomic propositions can be compiled into binary code and transmitted by the converter to the interpreter. When received by the interpreter, this binary code is stored into volatile memory (RAM) and the function calls proceed, in the same way, using function pointers.

### 5.4 Application to Model-Checking

To show the versatility of our diagnosis module, we present in this section how our interpreter can be coupled with a model-checking engine, which provides native formal property verification. In our experiments, we have used an extension of the OBP model-checking engine [36, 37] but the integration process is general and can be used with other model-checking tools.

**Figure 12: Architecture used for formal verification with a model-checker.**

The properties to be verified are expressed using the linear temporal logic (LTL) language, which binds the atomic propositions, expressed using our action language, to boolean valued functions.
This decouples the temporal logic operators from the atomic propositions, enabling the use of standard model-checking algorithms for the verification. In practice the LTL property is converted to a Büchi automaton using the LTL3BA library [2]. The semantics of the Büchi automaton is obtained dynamically through the dedicated BA Runtime. The resulting Büchi automaton is synchronously composed with the Büchi interpretation of the model semantics. The verification algorithm, used in our prototype, is based on the "nested DFS" Büchi emptiness checking algorithm proposed by Gaiser and Schwoon in [14]. During the verification, the synchronization between the Büchi automaton representing the property and the Büchi interpretation of the model semantics is based on the valuation of the atomic propositions. This valuation is obtained by evaluating the propositional atoms with the Checker module in the configurations exposed by our interpreter. The coupling with the model-checking algorithm is realized through a ProxyRuntime, which acts as a wrapper translating the requests of the model-checking algorithm to network API calls on our UML interpreter.

The model-checker sets the current configuration of the interpreter using a Set configuration request. From this configuration, the model-checking algorithm requests the list of all fireable transitions from the runtime model using a Get fireable transitions request. Each fireable transition is then fired, using a Fire transition request, and the resulting configuration is collected, using a Get configuration request. All atomic propositions are evaluated in this target configuration, using the Evaluate atomic propositions request. Based on the resulting valuation the model-checker computes the synchronized transitions allowed by the Büchi property automaton. These synchronized transitions are fired, and the resulting composite configurations (consisting of the target model configuration and the target Büchi automaton configuration) are matched with the known set of configurations. If the new configuration is not already present in the known set, the configuration is added. This process continues until either a fix point on the known set is reached or a Büchi acceptance cycle, representing a counter-example, is detected. The interested reader should refer to the "nested DFS" algorithm, in [14], for a detailed presentation of this verification algorithm.

6 EXPERIMENTS AND RESULTS

Our approach has been applied to the level crossing example introduced in Section 2. To check the system reliability, we verify formal LTL properties on the runtime model through model-checking, as described in the previous section.

Our validation strategy involves an abstraction of the environment as shown in Figure 13. For this purpose, we integrate into the design model a Train active object that models the behavior of a train looping on the level crossing. We also introduce an authorization signal to synchronize the Train with the RailwaySign. The train takes into account the railway sign through the reception of this signal before passing on the level crossing.

The four properties that we want to verify on the system (introduced in Section 2) have been expressed into LTL formalism, using operators not (!), or (||), and (&&), globally ([[]]), eventually (<>), and implies (->):

1. "[] !((trainIsPassing && gateIsOpen)"
2. "[] !((trainIsPassing && roadSignIsOff)"
3. "[] (gateIsClosed -> <> gateIsOpen)"
4. "[] (roadSignIsOn -> <> roadSignIsOff)"

These expressions involve the following atomic propositions written in our action language:

- \texttt{trainIsPassing} = |train.state == PASSING|
- \texttt{gateIsClosed} = |gate.state == CLOSED|
- \texttt{gateIsOpen} = |gate.state == OPEN|
- \texttt{roadSignIsOn} = |roadSign.state == ACTIVE|
- \texttt{roadSignIsOff} = |roadSign.state == INACTIVE|

Model-checking of these four properties has been performed on both variants of the event pool: \texttt{FifoEventPool} and \texttt{DeferredEventPool} introduced in Section 4.2.

The \texttt{FifoEventPool} implementation ignores events that do not match a valid trigger from a current state. With that event pool, verification of the fourth property results in failure: road sign can stay on without switching off.

The message sequence chart (MSC) in Figure 14 helps us to understand the problem. This MSC has been generated using an execution trace facility of our interpreter, based on the PlantUML formalism (http://plantuml.com/). The diagram reveals that the execution stops suddenly just before the second passage of the train on the level crossing. During that second loop, the train sends activation signals to \texttt{tcFar} and \texttt{tcClose} that are then transmitted to the controller as \texttt{entranceDetection} signals. We can notice that these

![Figure 13: Environment used for model-checking.](image-url)
two signals are received by the controller while it has not finished processing all events of the first train passage. In accordance with our event pool semantics, these two signals are ignored by the controller that is currently opening the gate and switching off the roadSign. This results in a deadlock when the controller finally reached its Idle state (its event pool is empty and no other active object can evolve). This observation has been confirmed by the model-checker that detects 2 deadlocks on the model.

The second event pool implementation (OrderedListDeferredEventPool) does not ignore events, but may defer them for being processed later. With that event pool, the verification of the four LTL properties is successful. With this event pool semantics, even if entranceDetection signals are received too early by the controller, these events are deferred to be processed when the controller reaches its Idle state. This execution semantics avoids previously observed deadlocks and enables to satisfy the expected properties.

The state-space explored by the model-checker contains 173 configurations and 276 transitions with the first event pool implementation (FifoEventPool), and 122 configurations and 209 transitions with the second one (OrderedListDeferredEventPool). In terms of performance, the formal verification of the four properties on a desktop computer (Intel® Core™ i7-8550U CPU at 1.80GHz with 4 cores, 8 threads at 4GHz maximum frequency, 16GB DDR4 2400MHz RAM) takes, on average, 1.71 seconds for the first trial and 0.96 seconds for consecutive ones. The first trial is less efficient because the executable code associated to each atomic proposition has to be generated and compiled whereas consecutive trials ignore this step (as mentioned in Section 5.3).

Following the verification of the level crossing system, the interpreter and the runtime model have been deployed on an stm32 bare-metal embedded target (1 CPU core at 168MHz, 192kB RAM). The resource overhead of the interpreter has not been evaluated yet. A complete study needs to be carried out to determine how
much extra memory and execution resources are required by the interpreter infrastructure compared to other approaches.

7 RELATED WORK

The work presented in this paper focuses on model verification and execution in the context of embedded systems.

Several other works aim at verifying and executing models such as the modeling tools Rational Software Architect [22] and Rhapsody [15], the model compilers GUML [9] and UniComp [11], the language and modeling workbench GEMOC Studio [7], as well as the model interpreters Moliz [24] and Moka [33] usable with Papyrus [21]. A more complete review of tools based on execution of UML models can be found in [12].

Rational Software Architect and Rhapsody modeling tools enable UML model execution, simulation, and debugging through code generation capabilities, but provide neither model interpretation nor model-checking facilities.

GUML and UniComp are two model compilers that can directly compile UML models, into efficient executable code without translation into intermediate programming languages. UniComp is also part of an automated round-trip approach [10, 11] using back-propagation facilities to inject monitoring results in the design model. If both compilers enhance predictability of generated executables, they are not designed to apply model-checking on design models using the semantics they implement.

GEMOC Studio enables the design of a generic environment with execution engines [13, 25, 38] and domain-specific tools (e.g., graphical animator, trace execution manager, omniscient debugger). Moka and Moliz provides standalone UML model interpreters conformant to the rUML [28] standard, that may be integrated with modeling tools (e.g., Papyrus) for execution, simulation, debugging, and testing purposes. However, model interpreters of these tools do not fit embedded requirements for model execution and may result in a lack of performance on such systems. Moreover, these tools do not currently address model-checking based on the operational semantics implemented in their interpreters.

Using an approach more suited for embedded systems, some works suggest to transform UML design models into an intermediate formalism based on state machines (e.g., EHA [30, 31], ESM [34]). Even if the resulting models can be used for diagnosis and verification, they are not able to execute them on targeted embedded systems. Instead, these works involve code generation from these models. Another similar work for designing embedded systems is mbeddr [39]. The design model of the system is transformed into a formal model for being analyzed by the SMV model-checker, and into C code for being executed on an embedded target. However, all these approaches rely on separate transformations for verification and execution without proving the equivalence between the verified model and the executable code. Furthermore, the model designer may be exposed to the semantic gap introduced by these transformations, which implies the use of low-level tools (e.g., GDB).

Other tools provide support to analyze execution of embedded systems at model-level. The debugger introduced in [19] uses an embedded monitor to produce back annotated traces and build UML sequence and timing diagrams in real-time for visualizing the target behavior. MDebugger [3] uses a model-to-model transformation to

8 CONCLUSION

Model-based engineering facilitates the development of complex embedded systems through model verification and model execution techniques. This paper has presented an approach that unified model verification and model execution using a single operational semantics implemented in a UML model interpreter.

This UML model interpreter is designed for being executed on both desktop computers and embedded bare-metal targets. It can be remotely controlled through an application layer protocol (on top of TCP) to perform simulation and model-checking. The approach has been experimented on a level crossing controller, which has been verified by LTL model-checking before its deployment on the embedded target executing the same interpretation semantics.

That approach improves the continuum from design to runtime by applying simulation, model-checking, and execution on a single design model either on desktop computers or embedded microcontrollers. Using a single model and a single operational semantics implemented in a single interpreter avoids potential semantics gaps introduced by model transformations. Moreover, this approach avoids the equivalence problem between formal models used for verification and executable code: what is checked is what is executed. Diagnosis results are also expressed in terms of design concepts, which eases their understanding during the development process.

To improve this execution engine, we further plan to improve UML coverage by supporting more UML concepts, and to study the impact of UML semantics variation points on model execution through model-checking. To better align with industrial practices, we also consider improving our action language support, notably by making it possible to plug several action languages. Moreover, a dedicated study also needs to be performed to evaluate the resource overhead of this model interpreter in comparison with other related execution engines.

ACKNOWLEDGMENTS

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