Modular Deployment of UML Models for V&V Activities and Embedded Execution

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

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

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

Ciprian Teodorov
Lab-STICC UMR CNRS 6285,
ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

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

Jérôme Delatour
ERIS, ESEO-TECH
Angers, France
jerome.delatour@eseo.fr

ABSTRACT

To design embedded systems, multiple models of their environments are typically required for different purposes such as simulation, verification, and actual execution. Some of these models abstract the actual physical environment to facilitate Verification and Validation (V&V) activities. Others capture the connection to hardware peripherals, necessary to deploy the systems on actual embedded boards. However, mapping a system to different environment models for different purposes remains a complex task for two main reasons. First, the environment is often tightly coupled with the system, and the board used for its execution. Second, formal properties verified during the design phase must be preserved at runtime. To tackle these issues, we propose an approach for designing UML models in a modular way and deploying them for V&V activities or embedded execution. This approach uses UML modularity mechanisms to specify the system in a generic way, and to connect it to a given (abstract or real) environment. This technique has been applied on several UML models of embedded systems to analyze their behaviors by simulation and LTL model-checking before deploying them on embedded STM32 boards.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification*; *Interpreters*; • **Computing methodologies** → *Model verification and validation*; • **Computer systems organization** → *Embedded software.*

KEYWORDS

Deployment, UML Execution, Model-checking, Embedded Systems, Model-Driven Engineering

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3419227

ACM Reference Format:
Valentin Besnard, Frédéric Jouault, Matthias Brun, Ciprian Teodorov, Philippe Dhaussy, and Jérôme Delatour. 2020. Modular Deployment of UML Models for V&V Activities and Embedded Execution. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3419227

1 INTRODUCTION

To face the increasing complexity of embedded systems, software programs are designed using models. With model-driven engineering (MDE), these models can be analyzed at early design stages with formal verification techniques. To that end, not only design models have to be executable but a model of their environment is also required for the verification step. Simply mocking inputs and outputs of the system model, as made for testing, is not sufficient here because the goal is to get appropriate interactions with the environment for different execution scenarios (not only one test case). Therefore, Verification and Validation (V&V) activities (e.g., simulation, model-checking) usually make use of *abstract* environment models. These models abstract the actual environment to consider a superset of all possible scenarios. Some refinements are usually applied to take into consideration assumptions about the physical world, thus removing some unrealistic scenarios. This can be particularly useful in model-checking to reduce the state-space explosion problem [14]. Moreover, to deploy the system model on actual embedded boards, abstract environment models have to be replaced by *concrete* environment models. These models describe how the system is connected to sensors and actuators through hardware peripherals of the embedded target. Such models do not specify scenarios but rather the interaction with the physical environment.

Moreover, environment models are not limited to one abstract and one concrete environment model per system. Indeed, analysis techniques may require the use of several abstract environment models (e.g., one for interactive simulation, one for formal verification). Engineers also need to explore several deployments on embedded boards to determine the best one, according to project constraints (e.g., cost, memory footprint). However, two main research challenges remain for connecting the system model to different environment models in a modular way. *(1)* The environment is

often target-specific and tightly coupled with the system model, hindering modularity, and making difficult to change the environment model for different V&V activities or runtime execution. *(2)* The deployment usually relies on transformations (e.g., code generation, model transformations) to refine the platform-independent design model into a platform-specific model. These transformations are usually not proven, which makes difficult to ensure that the executable code deployed on actual embedded boards preserves formal properties verified on the design model during V&V activities.

To address these issues, this paper presents an approach for deploying models of embedded systems in the context of UML [21], a language that emerges as the de-facto standard for the design of software applications in industry. The goal of our approach is to decouple the system from the environment such that the environment model can be changed at any time without having to update other parts of the model. Our work provides modeling guidelines and a reference architecture, based on several files, to improve the modularity of UML models. Using UML modularity mechanisms (i.e., *ElementImport* and *Package*) and XML Metadata Interchange (XMI) identifiers, we are able to split the design of a UML model in several files including one for the system and one for each environment model. The system is thus defined in a platform-independent way and connected to a given environment for verification or deployment on embedded targets. For *V&V activities*, an abstract environment model, designed using UML state machines, is used to close the system execution. A given environment abstraction is usually specific to the V&V technique applied on the model (e.g., simulation, model-checking). In simulation, the user wants to explore different execution traces while in model-checking, the whole model state-space has to be explored. For *actual execution*, a concrete platform-specific environment model is used for connecting UML models to hardware platforms through a low-layer interface. This interface enables the use of UML to configure and to use microcontroller peripherals. This work has been put into practice to extend the Embedded Model Interpreter (EMI) presented in [4, 5]. This tool defines a unique implementation for the UML operational semantics to ensure consistency between V&V activities and execution of UML models on embedded platforms. Prior research work on this tool shows its capabilities to perform simulation [4], model-checking of Linear Temporal Logic (LTL) properties [4], and monitoring [5], but it misses to address the deployment of modular UML models. In this paper, our work mitigates this shortcoming and shows that a uniform and modular UML-based modeling approach can ease the development process. It helps to preserve the verification results at runtime by deploying on embedded targets the same pair (system model + UML operational semantics) as the one used during V&V activities. The main contributions of this study are: *(1)* The same system model can be easily connected to different environment models using a modular architecture. *(2)* The reference architecture and its instantiation can be used as a modeling guideline for other design approaches. *(3)* This work enables to connect the embedded UML models to the hardware peripherals through a low-layer interface. *(4)* This transformation-free technique gives more confidence in the fact that properties verified during the design phase will also be verified at runtime when the embedded model interpreter is used.

This approach has been evaluated on three UML models of embedded systems. For each system, an abstract environment model has been designed and was used for simulation and model verification with the OBP2[1] model-checker [28, 29]. These system models have also been deployed on a STM32[2] discovery board using concrete environment models to make the link with the hardware peripherals of the board.

The remainder of this paper is structured as follows. Section 2 describes a simple model used as example while Section 3 gives an overview of our approach. In Section 4, we present how to design modular UML models and we show in Section 5 how such models can be linked to hardware platforms. In Section 6, we detail how these models can be deployed on the model interpreter. Section 7 applies our approach on multiple models of embedded systems. Section 8 reviews the state of the art and we finally conclude this paper in Section 9.

2 ILLUSTRATING EXAMPLE

To illustrate our work, we use a very simple Button-Led system that switches On (respectively Off) a led when a button is pressed (respectively released). This system is composed of only one active object, named *controller*, which is responsible for managing the system behavior. As shown in Figure 1, the state machine of this object sends the *lightOn* (respectively *lightOff*) event on the *light* port when the *buttonPressed* (respectively *buttonReleased*) event is received.

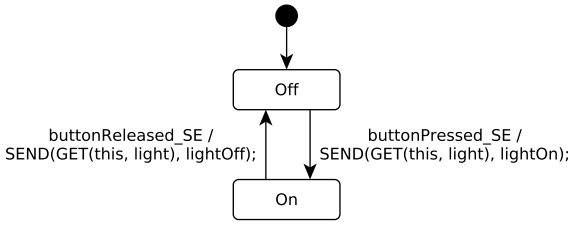


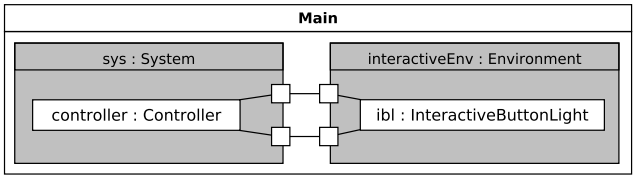
Figure 1: Controller state machine of the Button-Led system.

In this example, the system can be linked to four different environment models. They can all be executed by the model interpreter used in this project. Figure 2 shows how each environment is connected to the system *sys* through ports. Both the system and the environment are parts of the *Main* composite structure used for instantiation on the model interpreter. The following describes the four environment models used in this example.

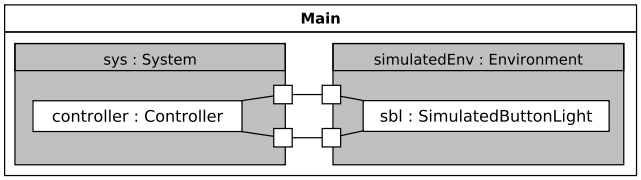
interactiveEnv: This first environment is an abstraction of the real environment and can be used for interactive simulation during early V&V activities. The physical environment is abstracted with only one instance of *InteractiveButtonLight*, called *ibl*. The state machine of this class is shown in Figure 3a. This state machine has only one state and one transition per sent or received event. For each received event, a transition is triggered each time an occurrence of this event (identified on figures by the name of the event followed by "_SE") is received. For sent events, corresponding transitions

[1] OBP2: http://www.obpcdl.org/

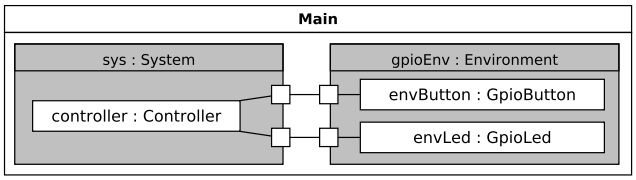
[2] STM32 discovery board is equipped with a 32-bits microcontroller ARM cortex M4, 1 MB Flash memory and 192 kB RAM.



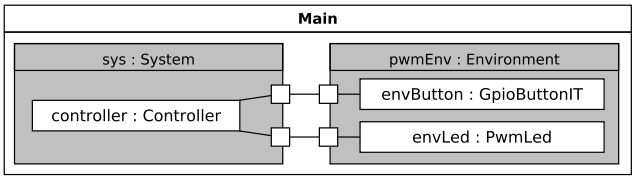
(a) Button-Led system with environment for interactive simulation



(b) Button-Led system with simulated environment



(c) Button-Led system with GPIO environment



(d) Button-Led system with PWM environment

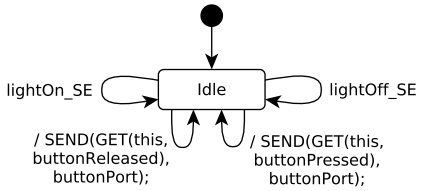
Figure 2: Button-Led system with different environments.

rely on completion events such that they can be fired at any time. This is the most generic environment that can be used because all event interleavings are possible.

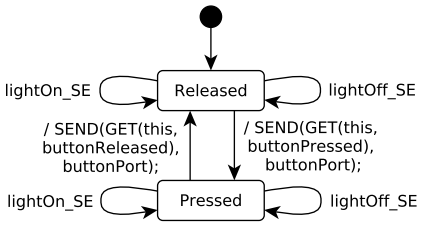
simulatedEnv: This environment model refines the previous one by taking into account an abstraction of the physical world: "The button cannot be pressed two times consecutively without being released". This abstraction can be used for model verification. The state machine of the *SimulatedButtonLight* class used in this model is depicted in Figure 3b. Two states are used to exclude all scenarios where two *buttonPressed* events or two *ButtonReleased* events are sent successively. However, it considers that *lightOn* and *lightOff* events can be sent at any time by the system.

gpioEnv: This environment can be used to deploy the system on an embedded target using General Purpose Inputs/Outputs (GPIOs). These peripherals can only be used to read or output all-or-nothing data. The *envLed* object is linked to an output pin while the *envButton* object is mapped to an input pin such that the physical state of the button can be read by polling (i.e., by checking actively the button state). State machines of *GpioButton* and *GpioLed* classes are depicted in Figures 3c and 3d.

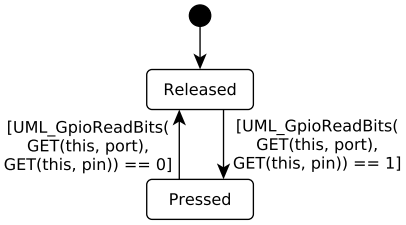
pwmEnv: This last environment can be used to deploy the system on an embedded target but with different hardware peripherals.



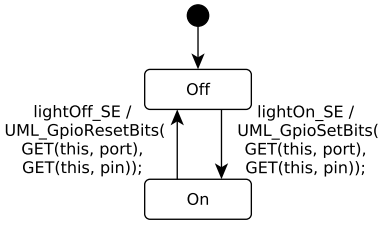
(a) State machine of the *InteractiveButtonLight* class.



(b) State machine of the *SimulatedButtonLight* class.



(c) State machine of the *GpioButton* class.



(d) State machine of the *GpioLed* class.

Figure 3: State machines of different Button-Led environment classes.

The *envLed* object is connected here to a Pulse Width Modulation (PWM) peripheral, which enables to control the light intensity of the led. The *envButton* object is here also linked to an input GPIO pin but this time the button state reading is triggered by an external interrupt. State machines of *GpioButtonIT* and *PwmLed* classes are quite similar to those of *GpioButton* and *GpioLed* classes, therefore there are not illustrated here.

To simulate, verify, and execute the behavior of this Button-Led model, we may require to link the system to each one of these environments at different phases of its development. Using the last two environments, this example will also enable to understand how the system can be linked to different hardware peripherals.

3 APPROACH OVERVIEW

To better understand the scope of this work, this section gives an overview of our approach for deploying modular UML models

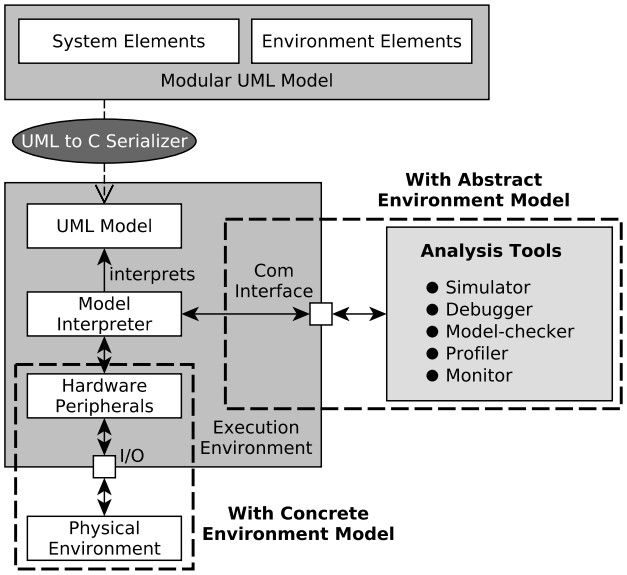


Figure 4: Approach overview for deploying modular UML models on embedded targets.

on embedded targets. Figure 4 illustrates our modular approach to decouple the system model from the environment model such that the environment model becomes an exchangeable component. First, the model is designed in a modular way such that elements of the system (*System Elements*) and elements of the environment (*Environment Elements*) are specified in different files and linked together through well-defined interfaces. All these UML elements create a *Modular UML Model* that can be loaded into the *Model Interpreter* at compile-time. This step uses a *UML to C Serializer* tool that serializes data of the model into C structure initializers for being loaded in the *Execution Environment* memory. For each element of the UML model, this means that each useful field of the UML metaclass instance is serialized as a field of a C structure initializer. Contrary to model transformations (e.g., package merge, code generation), our *UML to C Serializer* does not capture the semantics of the modeling language. The resulting *UML Model*, loaded in the execution platform memory, contains only data while the operational semantics of UML is defined by the *Model Interpreter*. In practice, the implementation of the UML semantics is made using the C language to keep efficient performance at runtime.

In case of an abstract environment model, *Analysis Tools* can be connected to the *Model interpreter* through a communication interface (*Com Interface*) to apply V&V activities on the interpreted model. In case of a concrete environment model, the interpreted model can interact with its physical environment through *Hardware Peripherals* of the board. The environment model can access these peripherals through a low-layer interface defined into C programming language, the native language of the interpreter. Using our approach, it is also possible to use a concrete environment model and connect the *Model Interpreter* to *Analysis Tools* such that it becomes possible to apply (hardware in the loop) simulation on the actual system execution.

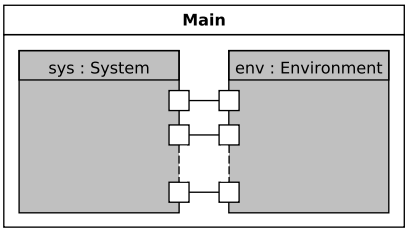


Figure 5: Generic composite structure.

In this work, the system model is directly deployed into its execution environment thanks to the embedded model interpreter of Besnard et al. presented in [4, 5]. Indeed, this tool can directly execute design models without additional transformations (e.g., code generation, model transformation). This model interpreter relies on a unique implementation of the modeling language semantics (here the UML semantics) to verify and execute UML models of embedded systems software. Therefore, unproven model transformations that capture the modeling language semantics must be avoided to preserve the unique implementation of the UML semantics in the model interpreter. Being transformation-free, this approach also helps to ensure that formal properties proven on the design model (e.g., properties to ensure the correctness of the system behavior) are still verified on the deployed model used at runtime.

The proposed approach offers multiple advantages. *(1)* The system model can be defined in a generic way and deployed as it stands for model verification and runtime execution. *(2)* It provides the possibility to easily link the system model with different environment models, and with hardware peripherals of embedded targets. *(3)* No model transformation is required to perform the deployment. This helps ensuring the preservation of verified properties at runtime by coupling modular UML models with the embedded model interpreter.

4 MODULAR UML MODELS

This section describes the reference architecture used to design models in a modular way as well as UML mechanisms used to support this modularity.

To decouple the system from its environment, we define a reference architecture that can be used as a modeling guideline. The *Main* composite structure of this modular architecture is shown in general terms in Figure 5. It contains two components: one for the system and one for the environment, respectively called *sys* and *env*. Both components have ports which enables to link them together with connectors such that they can interact with each other by sending signals through ports. Hence, these components do not reveal their internal structure but only the interfaces used by their ports. Each port has indeed *provided* and *required* interfaces that define events that can be respectively received or sent by the component to which the port belongs.

This reference architecture is also represented with a package diagram in Figure 6. This diagram contains four packages. The ***System*** package describes the system component that we want to verify and deploy. The ***Environment*** package defines the environment component which is used here to link the system with

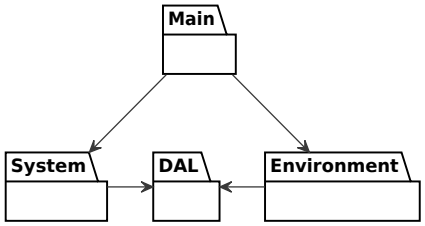


Figure 6: Package diagram of a modular UML model with environment decoupled from the system.

its actual physical environment or to specify an abstraction of it. We also need to describe the *events* exchanged by these two components as well as their port *interfaces*. This is performed by the ***DAL*** package that defines a Device Abstraction Layer (DAL) to abstract the environment from the system point of view. The goal of this package is to avoid defining the system in a specific way, but rather independently of the used environment, by relying only on interfaces. Finally, the ***Main*** package defines the *Main* composite structure in charge of connecting both the system (*sys*) and the environment (*env*) components together. This is the root composite structure of the model, which can be used for model instantiation following rules defined in the Precise Semantics for UML Composite Structures specification [22].

This reference architecture aims at decoupling the system from the environment such that the environment model becomes an exchangeable component. For this purpose, each package of Figure 6 is defined in its own file. To make reference to local elements, the concept used in XMI files is XMI identifiers (XMI IDs) because each element is identified by its owned XMI id. However, one main problem is to keep references to external elements (i.e., elements defined in other files) up-to-date. To address this issue, we use "stable" XMI IDs for all elements of the model. Indeed, we use the fully qualified name of each element as XMI id rather than common XMI IDs, which are usually not human-readable and meaningless. By analogy to program compilation, such XMI IDs can be seen as external symbols that have to be solved by the linker. These "stable" XMI IDs can be easily used by engineers to design models. To import an external element in a file, engineers only need the name of the UML file that contains the element as well as the fully qualified name of this element. These XMI IDs are said "stable" because they can be shared by several UML files. As a result, the use of the same qualified name in different files refer to the exact same element. To ensure consistency of the model, some OCL validation rules taken from the UML specification are checked before deployment. Using this approach, it becomes possible to only change the environment component to use a different one assuming that this component has the same name and the same ports. Despite the use of different environment models, other files (e.g., the one defining the *Main* composite structure) will be able to make reference to the environment component thanks to the use of "stable" XMI IDs. If XMI IDs were not "stable", engineers would have to update all references to this component in other files by changing (by hand) the value of its XMI id.

As shown in Figure 2, this architecture can be applied on the Button-Led example model to decouple the system from the environment. As a result, the environment component can be easily exchanged to make interactive simulation (*interactiveEnv*), simulation (*simulatedEnv*), or to explore different deployments on actual embedded boards (*gpioEnv* or *pwmEnv*).

5 LINK MODEL WITH HARDWARE

To deploy UML models on actual embedded boards, a concrete environment model must be defined to connect the system with hardware peripherals. In this case, the environment model is not an abstraction of the physical environment. It does not define scenarios based on environment assumptions but it makes the link to the actual system environment through sensors and actuators. In fact, the actual execution trace will depend on physical events that occurred in the environment and that are detected by the system.

The architecture used to deploy a modular UML model with a concrete environment is illustrated in Figure 7. The ***System*** is connected to the ***Environment*** through the ***DAL***, which defines interfaces between both components. In comparison to Figure 6, some additional packages are used to specify how the system is connected to the hardware of an embedded board. The ***DIL*** (Device Implementation Layer) is an implementation of the DAL. It defines UML classes, which can be seen as kinds of devices, the environment component can use. These devices enable to map events sent or received by the system with some actions (e.g., get value of a sensor, launch a timer, move an actuator) on microcontroller peripherals. This mapping is made using a UML state machine for each class of the ***DIL*** (e.g., state machines of *GpioButton* and *GpioLed* in Figures 3c and 3d). For each event defined in ***DAL*** interfaces, the ***DIL*** translates software UML events into physical signals and conversely. For an event sent by the system to the environment, the device specifies what has to be done each time this event is received. For instance, for the Button-Led system with the *pwmEnv* environment, the receipt of an occurrence of the *lightOn* event will launch a PWM channel with a given duty cycle. For an event sent by

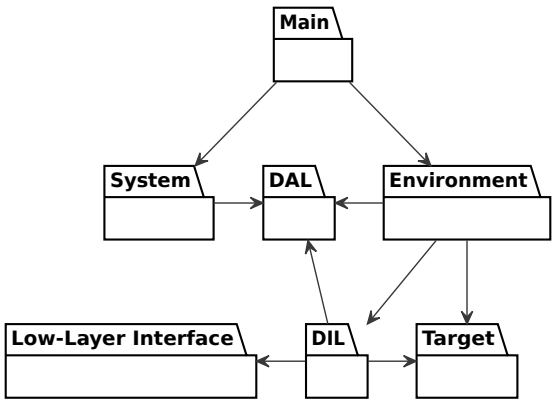


Figure 7: Package diagram of a modular UML model connected to the actual physical environment through hardware peripherals of a target.

the environment to the system, the device defines how the physical event associated to this UML event is captured (i.e., with which kind of sensors). For instance, for the Button-Led system, the event *buttonPressed* is detected by polling in the *gpioEnv* while this is done using an external interrupt in the *pwmEnv*. Each time a physical event is detected, it triggers the sending of the corresponding UML event to the system.

To implement these devices, the ***DIL*** uses two other packages. The ***Low-Layer Interface*** defines all functions that can be used to activate, configure, or run microcontrollers peripherals. These functions can then be called in guards and effects of UML state machine transitions using respectively *Opaque Expressions* or *Opaques behaviors*, which are standard UML concepts dedicated to the execution of non-UML pieces of code. *Opaque Expressions* and *Opaques behaviors* are thus used in UML models to make the link between UML concepts and low-level code. They are specified using the C-based action language of the model interpreter that provides some primitives to modify the model execution state and call functions of the ***Low-Layer Interface*** defined in C. For instance, in Figure 3c, the *GpioButton* state machine can read the value of a GPIO input pin by calling the function `UML_GpioReadBits()`. The ***Target*** package is also required to list all peripherals available on the board used to execute this modular UML model. For instance, for the STM32 discovery board used to deploy the Button-Led model, this target-specific package contains among others the list of GPIO pins and ports, the list of timers, as well as the list of PWM channels available on this board.

Once devices of the ***DIL*** have been modeled, the ***Environment*** can instantiate these devices and specify on which peripheral number each device is linked. For the Button-Led system with *gpioEnv*, the ***DIL*** defines a class *GpioButton* with two attributes: the GPIO port and the GPIO pin of the button. In the specific case of *envButton*, the *gpioEnv* composite structure (in Figure 2c) indicates that port `PORT_A` and pin `PIN_0` were used. Both `PORT_A` and `PIN_0` are UML elements defined in the STM32 ***Target*** package to which the ***Environment*** has to access to configure devices of the ***DIL***.

One main advantage of this approach is that different levels of abstraction can be used depending on the context of each project. Engineers can choose at which level it seems more relevant to stop modeling to use low-level code. In this work, we choose to model devices of the Board Support Package (BSP) but to access microcontroller peripherals with C code through a low-layer interface. It is possible to go further by modeling peripherals in UML and only accessing registers of the board using low-level code. However, we have decided to not do this because peripherals are specific to each microcontroller while devices (e.g., button, led) are generic concepts that can be more easily reused from one target to another. Using more abstraction levels also complexifies the model due to an increase in the number of UML objects, which may impact execution performance.

6 DEPLOYMENT OF MODULAR MODELS ON A MODEL INTERPRETER

This section explains how to deploy modular UML models on an existing model interpreter [4, 5] and what are the benefits of our approach in this context.

To deploy modular UML models, we use a model interpreter that defines a generic execution platform to execute UML models. In our approach, the system model is defined in a generic way (i.e., independently of the environment and of the execution platform). This is a Platform Independent Model (PIM) that can be executed by the model interpreter without refinement. As a result, the same system model can be used for execution and V&V activities. A classical technique to execute UML models is to refine a PIM into a Platform Specific Model (PSM), which contains details about the deployment and the hardware platform. Our approach is an alternative to model refinement for deploying models without model transformations. Knowledge about the environment and the execution platform are specified into an environment component and linked to the system component for deployment. This technique enables to deploy the same system with different environment models for all activities required during its development cycle. If engineers want to deploy the system differently (e.g., with different hardware peripherals), they just need to exchange the environment component.

To better understand how to apply this deployment technique in practice, Figure 8 presents the process used to deploy UML models on the model interpreter. Our modular UML models are split up into several files in the standard XMI format. Arbitrarily, four UML files are used in the figure but in practice we use one file per package (i.e., one file for the ***System***, one file for the ***DAL***, and so on). All these UML files are taken as inputs by the *UML to C Serializer* to serialize the model in C programming language. This process is directly applied on the resource set of all these XMI files. We keep relations between elements of these files using *Element Imports* and our "stable" XMI IDs. Resolution of these references to external elements is directly managed by the Eclipse Modeling Framework [27] using the `EcoreUtil.resolve()` method. With this technique, all these UML files are seen as only one model by the serializer. This tool serializes the UML model into C source code (*UML Model (C)*) such that the model can be loaded into the interpreter memory. In fact, the serializer performs a direct mapping of UML elements into C structure initializers. Unlike code generation, this tool does not capture the language semantics but only generates data (i.e., no functions are generated). Finally, both the *UML Model (C)* and the *Interpreter Source Code* are compiled and linked together to produce the *Binary Executable* of the system. This *Binary Executable* can

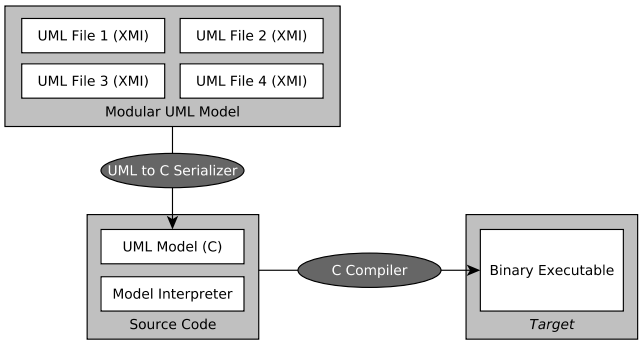


Figure 8: Deployment of a modular UML model on the embedded model interpreter.

then be used to apply multiple V&V activities on the runtime model with the OBP2 tool [28, 29]: trace-based simulation, LTL model-checking, and deadlock detection. It can also be executed on an embedded target if the environment model defines how to connect this model to hardware peripherals of the board.

An interesting advantage of deploying the reference architecture on this embedded model interpreter is to ensure the preservation of verified properties at runtime. These properties encode system requirements and are verified using model-checking to ensure the functional correctness of the system. To preserve the software correctness at runtime, the embedded model interpreter has the specificity to use the same combination (system model + operational semantics) for analysis activities and runtime execution. Only the environment model changes.

As a result, functional properties that are usually expressed only in terms of system objects will also remain verified at runtime on the condition that the abstract environment model covers all execution cases of the physical environment.

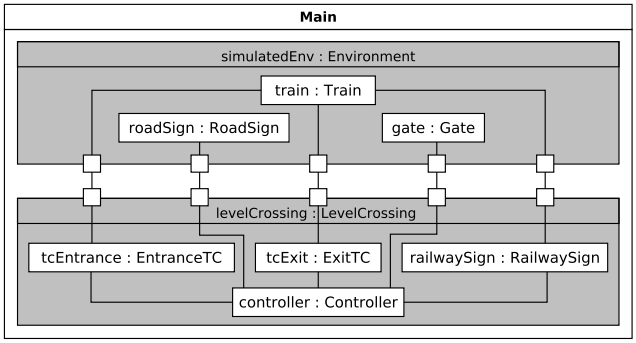
Environment models used for formal verification are an abstraction of the physical environment that theoretically considers a superset of all possible cases. However, to avoid state-space explosion, abstract environment models are usually refined by considering some assumptions about the physical world. By taking into account these assumptions, the model designer will remove some unrealistic scenarios in the design model. During this error-prone task, some realistic scenarios may potentially be removed too, which would break the condition that the abstract environment model covers all execution cases of the physical environment. In practice, it is possible to check that assumptions, considered during the design phase, still hold at runtime. For this purpose, the embedded model interpreter provides monitoring facilities [5]. Design assumptions can be encoded as monitors to check that such hypotheses are actually valid at runtime and, hence, that the current execution trace has been covered by model verification. Our monitoring solution is transformation-free and enables to trigger error recovery or fail-safe mechanisms in case of runtime failures.

Moreover, it is also necessary to check the behavior of software drivers used to run microcontroller peripherals (e.g., using testing). To that end, monitoring can also be used to detect bugs in these components as well as other external defects (e.g., deficient hardware components). For all these purposes, runtime monitors can trigger some fault recovery mechanisms when a failure is detected.

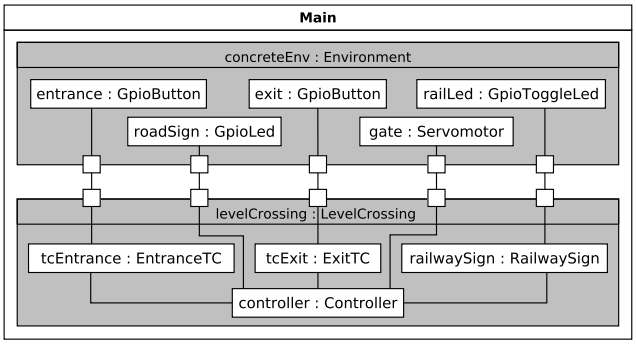
7 EXPERIMENTS

Our approach has been applied on the Button-Led model introduced in Section 2, and on two models of embedded systems: a level crossing system, and a cruise control user-interface. The goal of these experiments is to show that modular UML models can be deployed with abstract environment models to perform V&V activities, as well as concrete environment models for being executed on actual embedded targets. All these models have been designed with tUML [17], a textual syntax for UML. They conform to a subset of Eclipse UML[3] based on classes, composite structures, and state machines. Their fine-grained behavior is specified with the C-based action language of the model interpreter. These models have been

[3]Eclipse UML: https://www.eclipse.org/modeling/mdt/?project=uml2



(a) Level crossing system with a simulated environment.



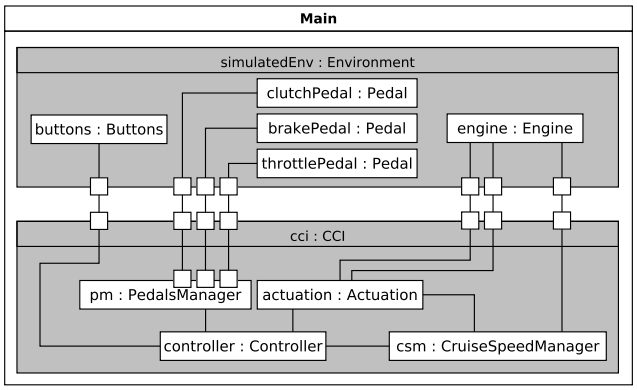
(b) Level crossing system with a concrete environment.

Figure 9: Level crossing system with different environments.

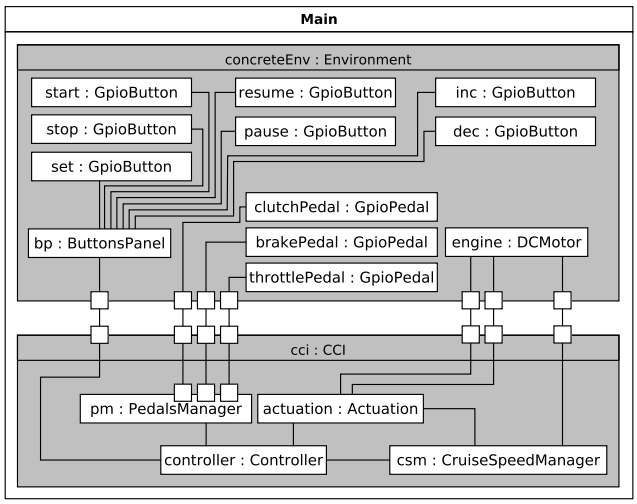
analyzed with the OBP2 model-checker that can be connected to the interpreter for model verification. They have also been deployed on STM32 embedded boards to be used in their actual physical environments.

Button-Led: This model has been deployed on the model interpreter with each one of the four environment models presented in Section 2. Both *interactiveEnv* and *simulatedEnv* have been used to perform trace-based simulation with OBP2. *gpioEnv* and *pwmEnv* environments have then been used to deploy the system on a STM32 embedded board. Both experiments show that the red led of the board switches on when the user button is pressed, and switches off when the user button is released.

Level Crossing: This approach has also been applied on the model of a level crossing controller presented in [4]. The goal of this system is to ensure the safety during the passage of the train on the level crossing. In [4], the model of the system has been closed with an abstract environment model tightly coupled with the system. This environment model relies on the assumption that there is only one train looping on the level crossing. In our experiments, we reuse this model and we decouple the abstract environment from the system as illustrated in Figure 9a. For this purpose, both the system and the environment have been encapsulated into dedicated components and linked together with connectors to obtain a modular UML model. Modularizing an existing model is not a difficult task. It mainly consists in splitting the model in different files and changing the model structure to add some composite structures,



(a) Cruise control interface system with a simulated environment.



(b) Cruise control interface system with a concrete environment.

Figure 10: Cruise control interface system with different environments.

ports, and interfaces. Using this model, we have succeeded to perform simulation, deadlock detection, and model-checking of four LTL properties from [4]. The state-space of this model is composed of 122 execution states linked together with 209 transitions, which gives an idea of the model complexity. In this paper, we also introduce a concrete environment model depicted in Figure 9b. Using this setup, the level crossing model has been successfully executed on a STM32 board.

Cruise Control Interface (CCI): To evaluate our approach on a more complex embedded system, we design a model of the user-interface of a cruise control system introduced in [5]. This subsystem aims at computing the cruise speed of the vehicle according to user interactions and the current speed returned by the engine. For this model, we have designed an abstract environment model illustrated in Figure 10a. This model makes the assumption that the system is independent of any other systems (e.g., Electronic Stability Program (ESP)). To consider a superset of all possible cases for model verification, we also consider that there is no cause-effect relationship between the value of the cruise speed setpoint sent to the engine and its current speed. Therefore, the current speed can go from 0 to 100km/h in one step. This cruise control interface model has been deployed with this abstract environment model on the model interpreter. The OBP2 model-checker has enabled to perform simulation, state-space exploration, and model-checking of three LTL properties introduced in [5]. These properties have been verified on this model having a state-space of 46,444,386 execution states linked with 82,734,350 transitions. For deployment on a STM32 board, we have also designed a concrete environment model (shown in Figure 10b) to connect the system to its sensors and actuators through microcontroller peripherals. With this concrete model, the CCI system has been executed on a STM32 board.

Conclusions: Based on these experiments, we can conclude that the approach is usable and that it provides interesting benefits. First, our technique helps to improve design quality of executable UML models by providing modularity at model-level and by avoiding code duplication of the system component. Stable XMI IDs have been really helpful to define models in multiple files. Our architecture simplifies the modeling task by designing several variants of the same UML model for different V&V activities only by designing different environment models. Second, we notice that using a modular UML architecture has no impact on results of V&V activities. We obtained the same size of model state-spaces and we verified the same LTL properties on our modular UML models than original non-modular UML models taken from the MDE community. This result was expected since our modular architecture only impacts the structural part of models, not their behavioral part.

During these experiments, the main difficulty has been the definition of UML components. For modularity, this approach requires components and ports with interfaces. Indeed, events cannot be directly sent to the target object but to a port that will forward this event to the final object or to another port. Hence, defining modular UML models require to use more UML elements. Due to this fact, UML models become more modular but also a bit more complex, especially for large systems.

In terms of threats to validity, our experiments do not consider models of industrial size to corroborate the approach scalability. The application of this modular architecture on a real-size model is kept as a future work. Another limitation is that we only consider the STM32 discovery board for model deployment with concrete environment models. To mitigate this problem, we have presented several concrete environment models using different peripherals to interact with the physical world.

8 RELATED WORK

Our work focuses on deploying modular UML models with different environment models to perform V&V activities as well as execution with hardware peripherals of embedded boards. In the context of MDE, different tools provide similar capabilities.

Modular Model Verification. Several other works aim at describing the system environment separately from the system for verification purpose. The approach [15, 28] used in the OBP model-checker (previous version of OBP2) is based on a Context Description Language (CDL). This domain-specific language is used to describe the context (i.e., the environment) using UML-like activity

and sequence diagrams, as well as temporal properties that must be verified by the system behavior. In comparison with CDL, our approach enables to specify the environment model directly in the design language (i.e., the language used for system modeling) and with the same design concepts (i.e., classes, state machines), which facilitates its usability by engineers.

Similar works use compositional analysis techniques [11, 13, 30] to verify system behaviors. These techniques aim at verifying each component of a software system separately from each other by determining which interactions the component can have with its physical environment. When all components have been individually verified, a dedicated composition operator is applied on these components to ensure that all properties are preserved on the whole system. In comparison with our work, compositional verification approaches help to reduce the state-space explosion problem as well as the amount of resources needed. However, they cannot ensure the preservation of verified properties during model deployment.

Model Deployment. More focused on model deployment, a typical approach, defined as a key concept of model-driven architecture in [23], is model refinement from PIMs to PSMs. This technique aims at designing a software system as a PIM and then to transform it into a PSM by adding target-specific rules for being able to deploy it on actual systems. In [20], model refinement is used in an educational tool based on an event-driven architecture. A set of model transformation rules enables to transform a PIM into a PSM, which can be used, almost as it stands, to generate the application code. In [25], a research project aims at verifying radio communication systems with a UML framework. This tool can be used to specify, in UML, the software application (PIM), the hardware platform, and the deployment (software + hardware). Using this setup, the framework enables to perform verification at different levels and to check if real-time and power constraints are satisfied.

Another technique for model deployment uses the package merge operator [31]. Such a technique has been applied in [2] to improve the modularity and the separation of concerns in architecture modeling. Despite these benefits, package merge has the drawback to imply "*a set of transformations, where the content of the merged package is expanded in the merging package*" [2].

Regarding deployment on actual embedded targets, Arduino Designer is a modeling tool that provides a graphical language to design software programs for Arduino. It provides the possibilities to connect these programs to hardware peripherals and deploy them on Arduino boards using code generation. Arduino Designer has also been enriched with simulation and animation capabilities [12] in Gemoc Studio [6] .

In comparison with model refinement, package merge, and Arduino Designer, our approach for deploying models is transformation-free. It enables to preserve properties, verified during V&V activities, on the runtime model executed by the model interpreter.

UML Model Execution. The study in [10] provides a systematic review of the state-of-the-art for executing UML models. Among the available tools, Moka [24] and Moliz [19] are two interpreters of fUML models that can be used for model execution, simulation and testing purposes. In comparison with our work, both interpreters cannot be used to execute embedded systems on embedded targets. GUML [7] and Unicomp [9] are two model compilers that can generate efficient low-level executable code for respectively UML state machines and UML activities. Papyrus-RT [16] is a tool that provides analysis capabilities and code generation for UML-RT models, a variant of UML models for real-time systems. These model compilers and code generators can be used to analyze and execute UML models but they rely on transformations to obtain the executable code. These unproven transformations do not ensure property preservation at runtime.

Property Preservation. Different tools are able to preserve the properties verified during the V&V phase on the executable code used at runtime. SCADE [3] is a synchronous approach for modeling, validating, and executing models of embedded systems and relies on a certified code generator to ensure the correctness of the generated code. CompCert C [18] is a verified C compiler that brings the proof that the generated executable code it produces conforms to the program taken as input. In a similar way, the work in [8] defines an automated round-trip approach to ensure the preservation of non-functional properties by using transformations with backward propagation facilities for going from code back to model. Other approaches (e.g., Event-B [1], UML-B [26]) use model refinement to ensure via proof obligations that a more refined version of a model conforms to its abstraction. All these approaches provides interesting techniques to ensure the preservation of properties on the executable code. In comparison with our approach, these works rely on translational approaches (i.e., model transformation, code generation, compilation) while our a work provides a solution for operational approaches (i.e., using interpretation).

9 CONCLUSION

The deployment of models is an essential feature of MDE. This paper has presented a model deployment technique for verifying and executing modular UML models of embedded systems.

Using this approach, engineers can define modular UML models by decoupling the environment model from the system model. The system model can be described in a generic way and linked with different abstract or concrete environment models. As a result, the same system component is used for all activities (i.e., simulation, model verification, and actual execution). Only the environment component has to be exchanged for deploying the system in different ways. This can be really helpful for engineers to explore different deployment configurations and identify the one that fits the best project constraints. To provide such modularity, our approach relies on a reference architecture, that provides modeling guidelines, as well as UML modularity mechanisms (e.g., ports, element imports) coupled with "stable" XMI IDs. This technique can be used to deploy abstract environment models with the system by taking into consideration assumptions about the physical world. These abstractions are useful to avoid state-space explosion and, thus, to apply analysis tools (e.g., simulator, model-checker). Our approach also provides facilities to link UML models with hardware peripherals of embedded platforms using concrete environment models. Such models enable to interact with the physical environment through sensors and actuators of embedded boards.

This modular deployment technique has been applied on multiple embedded system models of different complexity. These models have been designed in UML and deployed on an embedded model interpreter. This tool relies on a unique implementation of the

language semantics and provides a generic execution platform to execute and verify UML models. Abstract environment models have been used to analyze and validate these system models with the OBP2 model-checker before deploying them on embedded STM32 boards with concrete environment models. This approach has been successfully applied on UML models used as examples.

As a result, our approach makes a first step to address the two main research challenges mentioned in this paper. First, it provides a way to define and use different environment models for analysis activities and runtime execution. The system model is platform independent and entirely decoupled from the environment model. Second, the application of this technique coupled with the embedded model interpreter helps to ensure that the executable code deployed on embedded boards still preserve properties verified on the design model.

Further improvements of this work aim at applying software engineering techniques to better model the environment. Indeed, it is possible to have a lot of abstraction layers, like in operating systems (e.g., Linux). For instance, designing a generic model of the board support package (BSP) for the STM32 discovery board would be useful to better reuse low-level components. Moreover, we are also interested in applying our approach on an industrial case study to check its scalability on more complex systems.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Jean-Raymond Abrial. 2013. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA.

[2] Samir Ammour and Philippe Desfray. 2006. A Concern-based Technique for Architecture Modelling Using the UML Package Merge. *Electronic Notes in Theoretical Computer Science* 163, 1 (2006), 7–18. https://doi.org/10.1016/j.entcs.2006.07.005 Proceedings of the First Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005).

[3] Gérard Berry. 2007. SCADE: Synchronous Design and Validation of Embedded Control Software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and Prahladavaradan Sampath (Eds.). Springer Netherlands, Dordrecht, 19–33.

[4] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 2018. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*. Copenhagen, Denmark. https://doi.org/10.1145/3239372.3239395

[5] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. 2019. Verifying and Monitoring UML Models with Observer Automata. In *ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS '19)*. Munich, Germany, 161–171. https://doi.org/10.1109/MODELS.2019.000-5

[6] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. ACM, New York, NY, USA, 84–89. https://doi.org/10.1145/2997364.2997384

[7] Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet. 2012. An Optimized Compilation of UML State Machines. In *ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Shenzhen, China.

[8] Federico Ciccozzi. 2014. *From Models to Code and Back : A Round-trip Approach for Model-driven Engineering of Embedded Systems*. Ph.D. Dissertation. Mälardalen University, Embedded Systems.

[9] Federico Ciccozzi. 2018. Unicomp: A Semantics-aware Model Compiler for Optimised Predictable Software. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (Gothenburg, Sweden) *(ICSE-NIER '18)*. ACM, New York, NY, USA, 41–44. https://doi.org/10.1145/3183399.3183406

[10] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. 2018. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling* (10 April 2018). https://doi.org/10.1007/s10270-018-0675-4

[11] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. 1989. Compositional Model Checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. 353–362. https://doi.org/10.1109/LICS.1989.39190

[12] Benoit Combemale and Cédric Brun. 2015. Breathe Life Into Your Designer! http://gemoc.org/breathe-life-into-your-designer.html

[13] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Vienna, Austria) *(ESEC/FSE-9)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/503209.503226

[14] Philippe Dhaussy, Jean-Charles Roger, and Frédéric Boniol. 2011. Reducing State Explosion with Context Modeling for Model-Checking. In *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*. 130–137. https://doi.org/10.1109/HASE.2011.24

[15] Philippe Dhaussy, Jean-Charles Roger, Luka Leroux, and Frédéric Boniol. 2012. Context Aware Model Exploration with OBP tool to Improve Model-Checking. In *ERTS 2012*. Toulouse, France, xx.

[16] N. Hili, J. Dingel, and A. Beaulieu. 2017. Modelling and Code Generation for Real-Time Embedded Systems with UML-RT and Papyrus-RT. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 509–510. https://doi.org/10.1109/ICSE-C.2017.168

[17] Frédéric Jouault, Ciprian Teodorov, Jérôme Delatour, Luka Le Roux, and Philippe Dhaussy. 2014. Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. *Génie logiciel* 109 (June 2014), 21–27.

[18] Xavier Leroy. 2017. *The CompCert C verified compiler: Documentation and user's manual*. Intern report. Inria.

[19] Tanja Mayerhofer and Philip Langer. 2012. Moliz: A Model Execution Framework for UML Models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards* (Innsbruck, Austria) *(MW '12)*. ACM, New York, NY, USA, Article 3, 2 pages. https://doi.org/10.1145/2448076.2448079

[20] Geert Monsieur, Monique Snoeck, Raf Haesen, and Wilfried Lemahieu. 2006. PIM to PSM transformations for an event driven architecture in an educational tool. *Milestones, Models and Mappings for Model-Driven Architecture* (2006), 49.

[21] OMG. 2017. Unified Modeling Language. https://www.omg.org/spec/UML/2.5.1/PDF

[22] OMG. 2019. Precise Semantics of UML Composite Structures. https://www.omg.org/spec/PSCS/1.2/PDF

[23] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. 2005. Refinement via Consistency Checking in MDA. *Electronic Notes in Theoretical Computer Science* 137, 2 (2005), 151–161. https://doi.org/10.1016/j.entcs.2005.04.029 Proceedings of the REFINE 2005 Workshop (REFINE 2005).

[24] Sebastien Revol, Géry Delog, Arnaud Cuccurru, and Jérémie Tatibouët. 2018. Papyrus: Moka Overview. https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

[25] Samuel Rouxel, Jean-Philippe Diguet, Guy Gogniat, Nicolas Bulteau, Jonathan Carre-Gourdin, Jean-Etienne Goubard, and Christophe Moy. 2005. UML Framework for PIM and PSM Verification of SDR Systems. In *SDR Forum Technical Conference'05*. Anaheim, CA, United States.

[26] Colin Snook and Michael Butler. 2006. UML-B: Formal Modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.* 15, 1 (Jan. 2006), 92–122. https://doi.org/10.1145/1125808.1125811

[27] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[28] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. 2017. Environment-driven Reachability for Timed Systems. *International Journal on Software Tools for Technology Transfer* 19, 2 (01 April 2017), 229–245. https://doi.org/10.1007/s10009-015-0401-2

[29] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. 2016. Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability* 26, 7 (2016), 516–542. https://doi.org/10.1002/stvr.1611

[30] Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. 2003. Automated Environment Generation for Software Model Checking. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 116–127. https://doi.org/10.1109/ASE.2003.1240300

[31] Alanna Zito, Zinovy Diskin, and Juergen Dingel. 2006. Package Merge in UML 2: Practice vs. Theory?. In *Model Driven Engineering Languages and Systems*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–199. https://doi.org/10.1007/11880240_14