Formal verification and validation of embedded systems: the UML-based MADES approach

Luciano Baresi · Gundula Blohm · Dimitrios S. Kolovos · Nicholas Matragkas · Alfredo Motta · Richard F. Paige · Alek Radjenovic · Matteo Rossi

Received: 1 November 2012 / Revised: 13 February 2013 / Accepted: 28 February 2013 / Published online: 12 June 2013

L. Baresi · A. Motta · M. Rossi (✉)
Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milan, Italy
e-mail: Matteo.Rossi@polimi.it

L. Baresi
e-mail: Luciano.Baresi@polimi.it

A. Motta
e-mail: Alfredo.Motta@polimi.it

G. Blohm
Cassidian, Woerthstrasse 85, 89077 Ulm, Germany
e-mail: Gundula.M.Blohm@cassidian.com

D. S. Kolovos · N. Matragkas · R. F. Paige · A. Radjenovic
Department of Computer Science, University of York, Deramore Lane, York YO10 5GH, UK
e-mail: Dimitrios.Kolovos@york.ac.uk

N. Matragkas
e-mail: Nicholas.Matragkas@york.ac.uk

R. F. Paige
e-mail: Richard.Paige@york.ac.uk

A. Radjenovic
e-mail: Alek.Radjenovic@york.ac.uk

1 Introduction

The design and development of modern embedded systems is a complex and challenging activity that requires a careful and detailed analysis of all involved components. The system-to-be must be strictly aligned with the elements it needs to interact with, and with the context in which it will operate; strong timing constraints may further complicate the design. Early verification of design artifacts thus becomes an essen-tial activity to foster consistency, correctness, and integrity. However, the verification of such design models is challeng-ing: the adoption of formal methods in industry is far from widespread, and these models are often incomplete—which can introduce further challenges for automated analysis.

One of the main reasons for the scarce adoption of for-mal methods is the lack of the necessary mathematical back-ground, as well as the lack of robust and user-friendly tool support [45]. Following [8], many existing research efforts (e.g., [13,28]) have tried to shield mainstream model design-ers from the complexity of formal methods. The majority of these approaches provide an automatic translation mech-anism from models expressed in a flavor of UML (which is one of the most widely used system and software modeling languages) to a rigorous notation suitable for formal analy-sis. The main limitation of the aforementioned approaches is that each approach typically only focus on a single UML diagram or on a limited subset of the design notation.

The approach presented in this paper attempts to overcome the aforementioned limitations by supporting on-demand, automatic, and transparent model verification and closed-loop simulation of a wide range of UML diagrams, without the need for comprehending the complexities of the mathematical formalisms behind the approach. The aim is to provide designers with a tool that seamlessly extends their preferred design notation with formal verification capabilities. Automatic verification can be exploited at any stage of the development process, and works with the entire system model, a segment of it, or even a partial model implementation.

The proposed solution comprises three parts. As for the design notation, the MADES UML notation integrates a subset of UML and elements from MARTE [32] (the UML profile for Modeling and Analysis of Real Time and Embedded systems). This notation provides designers with a substantial and consistent set of modeling elements to model the system-to-be, and each element is ascribed with a formal semantics that enable formal verification. The formal representations of designed models are obtained automatically and allow the designer to verify designed models in isolation, through model checking, and to simulate (validate) the system within a Modelica [21] specification of the environment.

The proposed solution combines several existing and mature technologies. On the verification side, it exploits state-of-the-art model-checking technology to provide decision procedures tailored to the project domain. By exploiting domain abstractions and model fragments, designers can define properties at a level of abstraction close to their domain and they can ignore the underlying formalism. Model-to-text transformation via the Epsilon Generation Language [37]is an enabling technology that underpins our framework. The model-to-text transformations support the verification tasks by allowing system models to be mapped to the language required by $\mathbb{Z}$ot [35], the verification technology used by our solution. Counterexamples generated by the verification tool are mapped back onto the model elements to help identify potential sources of errors.

This paper builds on the work presented in [36]; it extends the latter by introducing new features of the MADES approach, most notably closed-loop simulation and the traceability of the results of the verification phase back to the original model; in addition, it illustrates some of the formal details underlying the MADES approach to Verification and Validation of UML models.

The paper is structured as follows: Sect. 2 presents the background of the proposed approach. Section 3 introduces the verification and simulation (validation) solutions fostered by MADES, while Sect. 4 illustrates them on a pair of examples. Section 5 summarizes the related work, and Sect. 6 concludes the paper.

2 Preliminaries

2.1 MADES project

The proposed approach to model-driven formal verification, developed in the context of the MADES project [4], aims to improve current practice in the development of embedded systems. The proposed approach is holistic in that it covers all phases of the development lifecycle, from design to code generation and deployment.

MADES makes several key contributions in the area of model-driven development of embedded and real-time systems. First, a dedicated modeling language was developed as a coherent subset of the OMG's UML and MARTE. Second, MADES supports the formal verification of key properties on designed artifacts as well as for closed-loop simulation based on detailed models of the environment. And third, code generation and code refactoring techniques have been devised, with features for compile-time virtualisation [22] of common and non-standard hardware architectures. A conceptual model of the inter-relationships between the key activities and their components in the MADES approach is shown in Fig. 1.

In more detail, the main contributions of MADES are the following:

1. The *modeling* component of MADES proposes a dedicated notation for modeling embedded systems.
2. The *verification* component of MADES focuses on providing formal verification and simulation support for the development of embedded systems; the verification of

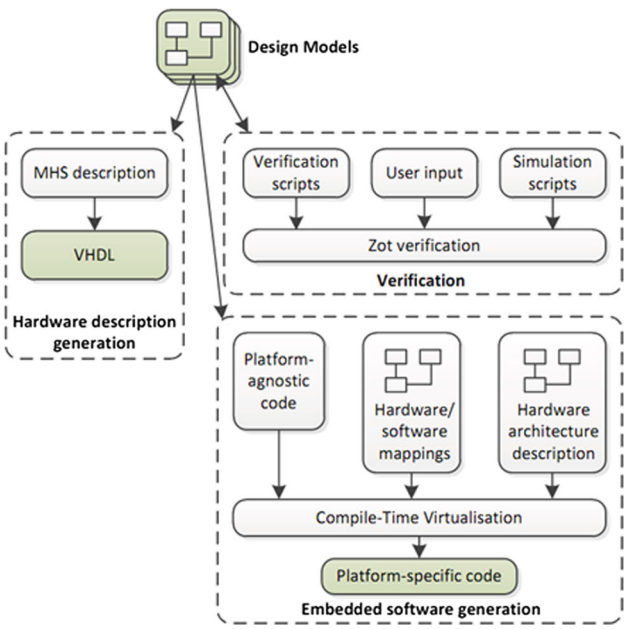


Fig. 1 MADES approach: overview of the key activities

key properties of the designed artifacts include, for example, whether a system will meet a specified deadline, or be able to support a specified volume of data; the simulation of the closed-loop system is based on detailed Modelica models of the environment for functional testing and early validation.

3. The *software generation* component provides the facilities for generating platform-specific embedded software from architecturally neutral software specifications; this is achieved using a novel technique of compile-time virtualisation to smooth the impact of the diverse elements of modern hardware architectures and cope with their increasing complexity.

Within the MADES project, all of the aforementioned aspects are fully supported by prototype tools integrated in a single framework. The intention is to have them thoroughly validated on real-life case studies in the surveillance and avionic domains.

More details about the MADES approach can be found at [29]. This paper mainly focuses on the verification and simulation aspects.

2.2 MADES notation

The MADES modeling notation was defined to simplify the complexity of UML and MARTE, making it consistent and usable for industrial users. In this section we provide a brief overview of the language, focusing on the parts that are most relevant for the verification and simulation activities.

MADES models can be built by assembling different diagrams. Class diagrams provide the static definitions of the elements in the system. Attributes and method parameters can be defined over the integers, the reals, and any of their finite subsets. For verification purposes there are limits on how these types are combined; more precisely, the decision procedures that are implemented in the tools on which the MADES verification approach is built cannot handle real numbers and unbounded integers at the same time. Class diagrams can also introduce clock types to constrain the timed behavior of components [2,32]. A class can be associated with a clock type to indicate that there is a single clock for all the instances of a class. Object diagrams contain all the instances of the classes and all the clocks. Dedicated clocks can be associated with specific objects; these associations override those defined in the class diagrams.

State diagrams are used to describe the behavior of the system objects to be analyzed. Each diagram comprises a set of states linked by transitions labeled with

$\langle trigger, guard, action \rangle$

The $trigger$ is an event, the $guard$ is a Boolean condition that is evaluated when the trigger occurs, and the $action$ is another event happening when the $trigger$ if the $guard$ is true.

Sequence diagrams describe partial behaviors of the system. They describe the messages exchanged among the objects defined in the object diagram and may also define time constraints, introduced through the ≪TimedConstraint≫ stereotype, which state metric timing relationships between events of the Sequence diagram. The messages should be instances of the operations defined in the objects' classes. Interaction overview diagrams (IOD) constitute a high-level structuring mechanism used to compose Sequence diagrams through standard operators such as sequence, iteration, concurrency or choice [6]. MADES UML uses Sequence diagrams to describe aspects of the system behavior and then composes these individual diagrams using IODs.

The different diagrams share a common set of events. Signals, beginnings and ends of messages, clock ticks, execution occurrences of Sequence diagrams, and states entered and exited are only some examples of all the events considered in MADES language; the reader can refer to [7] for a complete discussion. Shared events allow the different views to communicate. For example, a transition in a State diagram can be triggered by an event originating from a Sequence diagram, and the resulting action can be a *signal* that is sensed by an IOD. This is the case for the simple example system of Fig. 2. According to the IOD, the system starts from sequence diagram SD_1 of Fig. 2c. This diagram generates event $notify.start$, which is sensed by the State diagram and in turn activates the $Reaction$ (a signal) of the IOD, after remaining in state $process$ for at least the last 2 time units (including the current one). This signal is connected through a control flow to SD_2. In addition, the State diagram sends an acknowledgment back to object a after triggering the reaction. The execution of the system terminates after the end of SD_2.

Note that $notify$ is a method of class B that requires an integer (k) as parameter. The timing constraint of SD_1 states that the execution of the sequence processing on object b takes exactly 3 time units ($mNotify$ and $mAck$ are the names of the two messages in Fig. 2c).

2.3 Formal semantics

Verification and validation of MADES models is achieved by automatically translating them into a suitable formal representation. The underlying formalism used to enable formal verification is temporal logic. Temporal logic descriptions of systems consist of sets of predicates and axioms. The predicates correspond directly to the elements of the MADES models. The axioms describe how different elements are related to each other, effectively representing the actual semantics of MADES models.

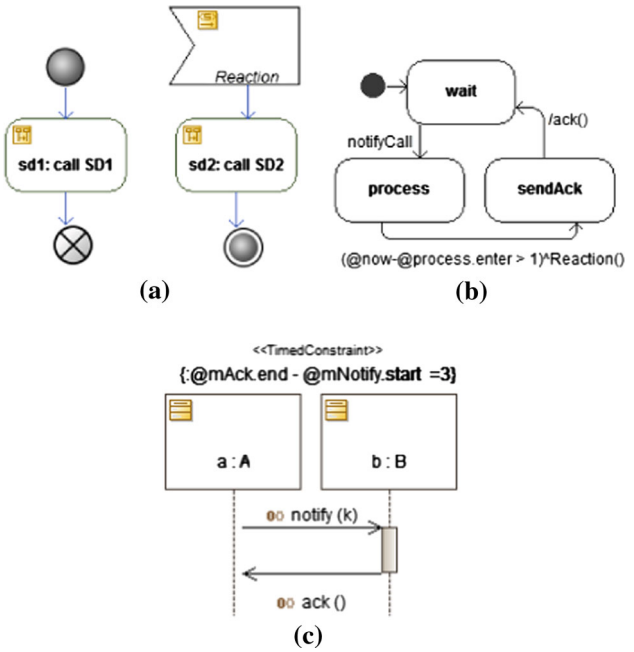


Fig. 2 An example model with three diagrams [an IOD (**a**), a State diagram (**b**), and a Sequence diagram (**c**)] communicating through shared events

The semantics is **decoupled** from the predicates that represent model elements: one can change the semantics (and experiment with different solutions and alternatives) whilst translation from UML to the predicates remains unaffected. The logic-based formalization can be **extended** in a straightforward manner. Adding a new diagram type only requires the definition of predicates that represent its elements, and their associated axioms. Introduction of new predicates can increase the amount of detail in the temporal logic representation of the elements. Adding more detail may result in specifications becoming too big for automated analyses. To avoid this scenario, the approach supports analysis of **partial models** by translating only diagrams of interest.

The temporal logic description of the models—completely hidden from the modelers—is specified in TRIO [14], a first-order linear temporal logic that supports a metric on time. TRIO formulae are built using standard first-order connectives, operators, and quantifiers. In addition, TRIO defines a single basic modal operator, Dist, that relates the *current time* (left implicit in the formula) to another time instant. Given a time-dependent formula F (a term that maps the time domain to truth values) and a (arithmetic) term t [indicating a time distance (either positive or negative)], the formula $\text{Dist}(F, t)$ specifies that F holds at a time instant whose distance is exactly t time units from the current instant. $\text{Dist}(F, t)$ is in turn also a time-dependent formula, as its truth value can be evaluated for any current time instant so that temporal formulae can be nested as usual. While TRIO can exploit both discrete and dense sets as time domains, MADES assumes the standard model of the non-negative integers $\mathbb{N}$ as a discrete time domain. For convenience, in specifying formulae, TRIO defines a number of *derived* temporal operators from the basic Dist through propositional composition and first-order logic quantification. Some of the more significant ones, including those used in this paper, are shown in Table 1.

Table 1 TRIO-derived temporal operators

Operator	Definition
$\text{Past}(F, t)$	$t \geq 0 \wedge \text{Dist}(F, -t)$
$\text{Futr}(F, t)$	$t \geq 0 \wedge \text{Dist}(F, t)$
$\text{Alw}(F)$	$\forall d : \text{Dist}(F, d)$
$\text{Lasted}(F, t)$	$\forall d \in (0, t] : \text{Past}(F, d)$
$\text{Lasts}(F, t)$	$\forall d \in (0, t] : \text{Futr}(F, d)$
$\text{WithinP}(F, t)$	$\exists d \in (0, t] : \text{Past}(F, d)$
$\text{WithinF}(F, t)$	$\exists d \in (0, t] : \text{Futr}(F, d)$
$\text{Since}(F, G)$	$\exists d > 0 : \text{Lasted}(F, d) \wedge \text{Past}(G, d)$
$\text{Until}(F, G)$	$\exists d > 0 : \text{Lasts}(F, d) \wedge \text{Futr}(G, d)$

TRIO specifications of systems consist of basic predicates and arithmetic temporal terms representing elementary phenomena of the system. The system's behavior over time is described by a set of TRIO formulae, which state how the predicates are constrained and how they vary over time, in a purely declarative fashion.

TRIO specifications can be analyzed through the Zot tool [35], a bounded model/satisfiability checker[1]. Zot can deal with numeric domains such as integers or reals [9], which can appear in MADES models. The input to the Zot tool is a script comprising a set of TRIO formulae expressed as Lisp statements, while its output is plain text. Given an input specification, Zot checks whether it is satisfiable or not. Using this basic mechanism, given a target model Zot can perform both verification of user-defined properties and simulation tasks. In fact, in the TRIO/Zot approach checking that property R holds for system S, where the latter is formalized by a formula Σ and the former by a formula ρ reduces to verifying that formula $\Sigma \Rightarrow \rho$ is valid or, equivalently, that $\Sigma \wedge \neg\rho$ is unsatisfiable. Zot encodes the temporal logic formulae to be checked into the input language of third-party solvers to achieve its task. Zot provides encodings both for SAT solvers and for Satisfiability Modulo Theories (SMT) solvers.

To present the proposed semantics we start with the predicates defined to encode UML models, and then we provide an overview of the axioms that formalize the meaning of the different concepts. A more comprehensive presentation can be found in [7].

[1] http://zot.googlecode.com.

2.3.1 Predicates

TRIO predicates are automatically generated from MADES design models using the tool chain described in Sect. 3. This section outlines the most relevant predicates associated with a MADES model.

To avoid clashes in the predicate names, we exploit the fact that every model element x has a unique identifier, which we indicate in the following as id_x.

For every clock c in a Class diagram, a temporal logic predicate $Clock_{id_c}Tick$ is declared, which holds every T time units, where T is the period of c. Similarly, given a signal s, we declare the predicate $Signal_{id_s}$, which is true in those instants in which s is triggered. For every operation (resp. attribute) x belonging to an object y we declare Boolean predicate $OBJ_{id_y}OP_{id_x}$ (resp. arithmetic temporal term $Obj_{id_y}Attr_{id_x}$). Predicate $OBJ_{id_y}OP_{id_x}$ holds when operation x is invoked on object y, while $Obj_{id_y}Attr_{id_x}$ represents the value of attribute x of y at the different time instants.

In MADES models, each State diagram is drawn in a class, but it describes the behavior of the objects of the type of that class. Hence, the predicates associated with State diagrams refer to objects, rather than classes. More precisely, for every state s in the State diagram of the class of object o, we declare predicates $Obj_{id_o}State_{id_s}Enter$, $Obj_{id_o}State_{id_s}Exit$ and $Obj_{id_o}State_{id_s}$ which hold, respectively, when o is entering, exiting, or is in s. In addition, if s has an invariant we declare predicate $Obj_{id_o}Invariant_{id_s}$, which is true when the invariant condition is true. Table 2 shows the complete list of predicates generated from a State diagram of an object o.

Given a Sequence diagram x, predicates $SD_{id_x}Start$, $SD_{id_x}End$, and SD_{id_x} are declared, which are true, respectively, at the beginning, at the end, and during the diagram execution. Also, predicate $SD_{id_x}Stop$ holds if the diagram terminates before reaching its end (e.g., because it is interrupted). For every message m we declare predicates $Msg_{id_x}Start$ and $Msg_{id_x}End$ that hold at the beginning and at the end of the message. Given an execution occurrence e we declare predicates $ExOcc_{id_x}Start$, $ExOcc_{id_x}End$, and $ExOcc_{id_x}$, with the obvious meaning. Finally, for every time constraint c associated with a Sequence diagram we declare predicate $Constraint_{id_c}$, which holds from the beginning until the end of the execution of the interaction.

Similarly, we generate predicates corresponding to the elements of IODs. Readers can refer to [7] for a complete presentation.

Finally, the $MADES\_System\_Start$ predicate defines the time instant in which the system starts. When $MADES\_System\_Start$ holds, all IODs start and all State diagrams enter their initial state.

Table 2 TRIO predicates associated with a State Diagram of object o

UML entity	TRIO item
State s	$Obj_{id_o}State_{id_s}Enter$, $Obj_{id_o}State_{id_s}Exit$, $Obj_{id_o}State_{id_s}$
Invariant (of s)	$Obj_{id_o}Invariant_{id_s}$
Transition t	$Obj_{id_o}Transition_{id_t}$
Trigger (of t)	$Obj_{id_o}Trigger_{id_t}$
Guard (of t)	$Obj_{id_o}Guard_{id_t}$
Action (of t)	$Obj_{id_o}Action_{id_t}$

2.3.2 Axioms

The presentation of the axioms defining the semantics of MADES diagrams is organized according to the different elements of the notation. For simplicity, we only present the elements that are relevant to the examples of Sect. 4. Further details can be found in [7].

Clocks For each clock c with period T the following axiom holds, which states that a clock ticks iff it did not tick during the last $T-1$ time units, which implies that it ticks at times $T, 2T, 3T, \ldots$[2]:

$$\text{Lasted}(\neg cTick, T-1) \Leftrightarrow cTick \tag{1}$$

Objects that are linked to a clock c either in the Class diagrams or in the Object diagrams run on a time base defined by c. All events that belong to these objects can only happen when c ticks. Given a clock c, we define the set of predicates running on this clock as $Events_c$. For every object o linked to clock c, if there is a Sequence diagram where o sends/receives a message, the predicate corresponding to the message start/end event is included in set $Events_c$. For every predicate e in $Events_c$ the following axiom holds, which states that event e can occur only in those instants in which clock c ticks:

$$e \Rightarrow cTick \tag{2}$$

The periods of clocks in a MADES specification are all expressed in relation to a uniform ideal discrete time that underlies the whole model. This abstract view of time is suitable for high-level specifications of timed systems, and it is not intended to capture implementation-level concepts such as the physical hardware clocks associated with computing devices. The notion of clock in MADES UML can be seen as an abstraction of such concepts, in that clocks can be used to introduce periodic behaviors in the specification.

[2] TRIO axioms are implicitly asserted for all time instants, hence formula (1) is implicitly interpreted as "Alw(Lasted($\neg cTick, T-1$) $\Leftrightarrow$ $cTick$)".

Sequence diagrams A Sequence diagram is defined as a set of lifelines, where every lifeline is an ordered list of events (e.g., message start/end, execution occurrence start/end).Given a lifeline l, we call $LifelineEv_l$ the set of its events. For every ordered pair of events $i, j \in LifelineEv_l$, if i holds at some instant, then j will follow in the future if the diagram is not stopped. This is formalized by the following axiom:

$$\begin{aligned} Ev_i \Rightarrow \text{Until}\big(\neg Ev_i \wedge \neg Ev_j, SD_x Stop\big) \\ \vee \\ \text{Until}\big(\neg Ev_i \wedge \neg SD_x Stop, Ev_j\big) \end{aligned} \tag{3}$$

where Ev_i and Ev_j are the predicates corresponding to events i and j. Also, the following formula defines that, if j holds at some instant, then i was true in the past and no stop occurred since then:

$$Ev_j \Rightarrow \text{Since}\big(\neg Ev_j \wedge SD_x Stop, Ev_i\big) \tag{4}$$

State diagrams The formalization in temporal logic of State diagrams is fairly standard; therefore, we will only hint at some of its features. Given a state s of the State diagram of an object o, we call $Incoming_s$ (resp. $Outgoing_s$) the set of its incoming (resp. outgoing) transitions. A necessary condition to enter the state is that one of the incoming transitions holds in the previous time instant:

$$\begin{aligned} Obj_{id_o} State_{id_s} Enter \\ \Rightarrow \text{Past}\left(\bigvee_{t \in Incoming_s} Obj_{id_o} Transition_{id_t}, 1 \right) \end{aligned} \tag{5}$$

Similarly, the necessary condition to exit from a state is that one of the outgoing transitions holds at the current time instant:

$$\begin{aligned} Obj_{id_o} State_{id_s} Exit \\ \Rightarrow \bigvee_{t \in Outgoing_s} Obj_{id_o} Transition_{id_t} \end{aligned} \tag{6}$$

Other formulae define that when object o enters state s predicate $Obj_{id_o} State_{id_s}$ holds and that if o is in s and s is not exited, it will also hold in the next instant. They are not shown here for simplicity.

An invariant i associated with state s of object o holds in all instants in which s is active, except the last ones before changing state (i.e., those in which predicate $Obj_{id_o} State_{id_s} Exit$ holds). This is captured by the following formula:

$$\begin{aligned} Obj_{id_o} State_{id_s} \wedge \neg Obj_{id_o} State_{id_s} Exit \\ \Rightarrow Obj_{id_o} Invariant_{id_i} \end{aligned} \tag{7}$$

An invariant is a condition on (some of) the attributes declared in the class of object o.

Note that, as highlighted by the Past temporal operator used in formula (5) we assume that one instant of time passes every time a transition is taken; hence, unlike other semantics of State diagrams/Statecharts [18], we do not allow an object to take multiple transitions of the same State diagram at the same time instant. Finally, users can introduce cross-State diagram constraints as a form of synchronization between diagrams.

Time constraints Both Sequence diagrams and State diagrams can include time constraints between pairs of events in the diagram (Fig. 2). A time constraint is an inequality between two events. For example, for the diagram of Fig. 2c the following formula holds:

$$SD_1 \wedge mAck \Rightarrow \text{Past}(mNotify, 3) \tag{8}$$

The time constraint of transition T_1 between states $wait$ and $process$ of Fig. 2b, instead, is formalized by the following formula, where $triggerT_1$ is a predicate that holds when the condition triggering T_1 is true:

$$triggerT_1 \Leftrightarrow \text{Lasted}(\neg processEnter, 1) \tag{9}$$

Time constraints are a powerful way to express the timed behavior of the system, as shown in the examples of Sect. 4.

Attributes State diagrams can include constraints (e.g., invariants) on the values of the attributes of classes. The constraints allowed are currently of the form $attr \gtreqless c$, where c is a constant. Class attributes can have unbounded domains such as the integers or the real numbers (though a MADES model cannot include both real-valued attributes and unbounded integer attributes, as the underlying decision procedures used for verification do not allow it). To deal with this kind of constraints, we use the features offered by SMT solvers that are able to handle decidable fragments of first-order logic such as linear integer/real arithmetic.

3 Verification and validation

The MADES approach addresses the twin issues of formal verification of properties of embedded systems designs and of validation of closed-loop models that include both the system under design and the (physical) environment with which it interacts.

Verification and validation (V&V) of system designs play a key role in the development of embedded systems from its early phases. A formal approach to these activities can help increase designers' confidence in the correctness of the developed system. A key to making formal V&V techniques appealing in the industrial practitioners is reducing the overall effort associated with the process. One way to achieve this is to hide the complexity of the formal models from the domain experts and allow them to specify the system of interest in a notation they are familiar with. To this end, the MADES approach uses model transformations to provide a seamless integration between the design and the V&V tools.

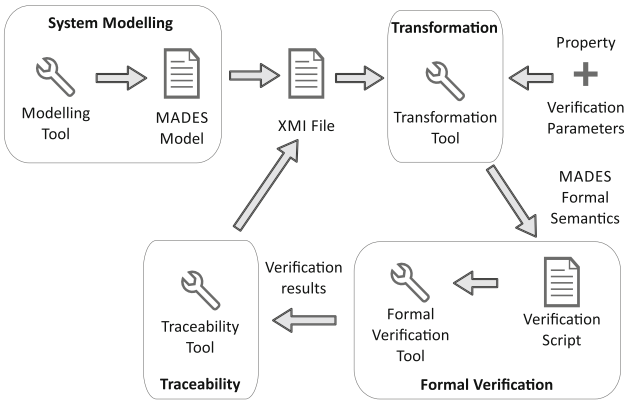


Fig. 3 Verification workflow

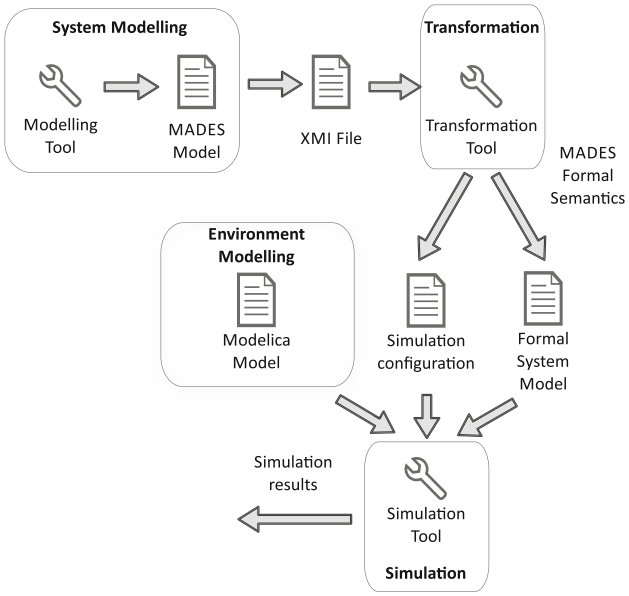


Fig. 4 Simulation workflow

We have accomplished this by means of a tool chain that supports two complementary workflows, one for the verification of system properties, and one for closed-loop simulation.

The verification workflow (Fig. 3) has four key components. *System Modeling* requires the specification of design models using the MADES notation (Sect. 2.2). *Transformation* produces a formal representation of the design models as a set of formulae expressed in the TRIO language [14]. *Verification* uses a model checker to perform formal proofs. In case of negative outcomes, the tool produces a counterexample. These counterexamples serve as input to *Traceability*, which enables the modelers to relate detected problems back to their source (e.g. a specific model element).

The simulation workflow (Fig. 4) is essentially based on the same components as the verification workflow. Modeling, in this case, comprises two parts: the embedded software system under design and the environment in which the system needs to operate. The latter is achieved through techniques and tools that are typical of control systems theory, and in particular through equations (both algebraic and differential) described in the Modelica language [21]. The transformation tool, in addition to producing the formal model of the system, sets up the MADES simulation tool with the necessary files and inputs to run the simulation. Finally, *Simulation*, through a co-simulation approach, produces a trace that satisfies both models, if one can be found.

Transformation, verification and traceability, and simulation take place under-the-hood, transparently to the users, who do not have to deal with anything beyond their domain. The simulation tool is external to the modeling one, but it is automatically configured and launched through suitable dialog boxes similar to those used to set up the verification tool. All technologies and tools are open source and available to the general public.

3.1 Transformation

Model transformations play a key role in the proposed approach and are used to shield system designers from the underlying complexity of the formal verification notations used under-the-hood. Figure 5 illustrates the internals of the transformation tools for the two aforementioned tool chains. The architecture of the two transformation tools is identical. In both cases the transformation tool consists of two components: $XMI2Java$ and the $Java2Lisp$, but their implementation as well as the input and output artifacts are different.

In the former case, $XMI2Java$ needs a system model (serialized in XMI), the configuration of the validation tool (i.e. Zot), as well as the system property to be verified. The output is an intermediate Java representation that holds the definition of the model along with its semantics. The second component ($Java2Lisp$) uses the representation generated by the first component to generate the formal model of the system, which is Lisp, the notation used by Zot. Then the formal model is passed to the verification tool to perform the verification activity.

In the latter case, $XMI2Java$ just needs the system model and the output is again the intermediate Java representation, then used by the $Java2Lisp$ component to generate the two outputs of the tool: the formal model and the configuration of the simulation tool. These two artifacts, together with the model of the environment, are then used by the simulation tool. In both cases, the transformation tool generates traceability information (Sect. 3.3).

The $XMI2Java$ component of both transformation tools is implemented using the Epsilon Generation Language (EGL) [37], a well-established and mature template-based model-to-text transformation language. EGL is part of the Epsilon model management framework [27], a family of consistent and interoperable model management languages.

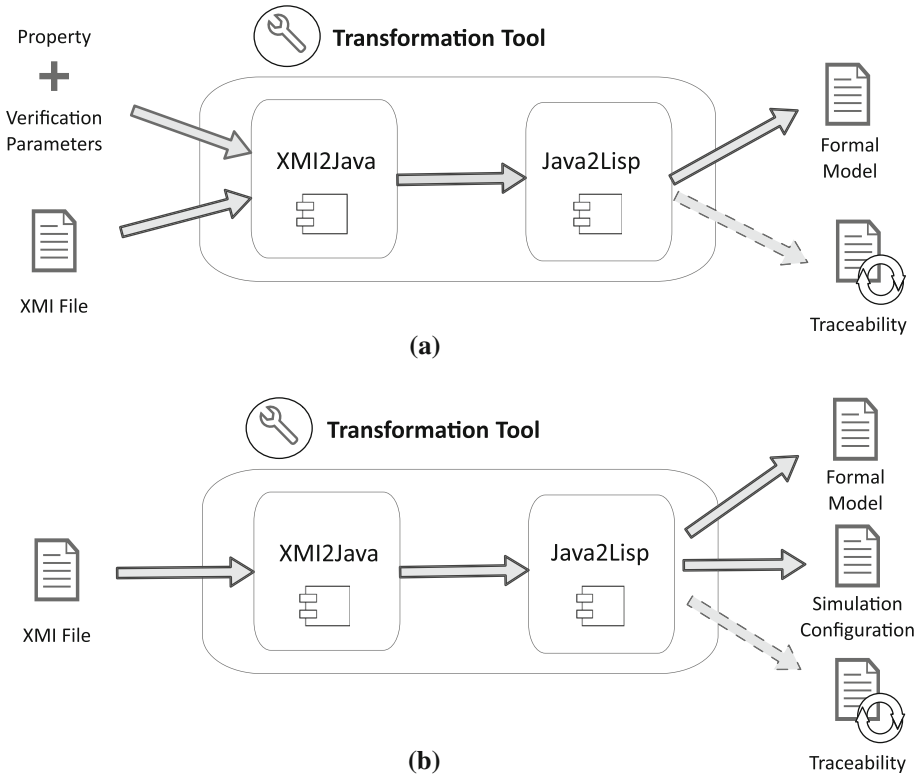


Fig. 5 Transformation tools and their inputs and outputs for the verification (**a**) and simulation (**b**) tool chains

Instead of utilizing the two aforementioned components to perform the transformation, we could have implemented the transformation directly from the system model to the formal ℤot model. However, the division of the transformation process in two steps provides a good degree of decoupling and independence between the UML metamodel and the Java primitives implementing the semantics. Thus, if one needs to build transformations for different semantics, for example to tailor them to different application domains, only the implementation of the Java methods executed by the $Java2Lisp$ component should be modified/extended, leaving $XMI2Java$ untouched. Moreover, this decoupling makes the implementation of the transformations simpler and easier to maintain.

The configuration information comes in the form of a number of parameters, which can be specified in the launch configuration of the tool. The user can specify the type of solver to be used by the tool (SAT or SMT), the encoding of the temporal logic to be used for a given solver (i.e. ℤot plugins) and the time bound for the verification (i.e. the maximum length of system executions to be analyzed).

The user must also define the property to be verified. If no property is provided, the system model alone is used instead. In this case, the verification tool looks for an execution trace of the system and if one exists it returns it.If instead a property is defined, the verification tool determines whether this property holds for the system or not. The property to be verified can be specified in the launch configuration of the verification tool as well.

Once the relevant information is provided to the tool, verification can be performed. Initially, the model expressed in XMI and the set up information are used by the $XMI2Java$ component to generate a Java representation of the model. Once this representation is generated, it is consumed by the $Java2Lisp$ component to generate the formal model expressed in Lisp.When the formal model is generated, ℤot performs the verification and the results are reported back to the user. Although this is a complex transformation chain, everything is hidden from the designer, who only has to interact with the modeling environment. The entire verification process is fully automated with minimal user intervention (i.e. providing a few configuration parameters).

3.2 Verification and validation

This section provides an overview of the verification and simulation activities, of the language constructs that have been introduced specifically for them, and of the theoretical framework on which the simulation tool is based.

First of all, while models of industrial embedded systems can be very large, only parts of them, typically those that have the most stringent safety-critical requirements, need to be verified and validated using formal techniques. This leads to so-called *lightweight* formal approaches [25], where for-

malisms are applied only where necessary. In this vein, we introduced stereotypes to allow designers to identify those parts of the system on which they want to focus V&V activities. These stereotypes are then taken into account during the transformation phase, which only produces a formal model for the tagged parts, instead of the whole MADES model.

More precisely, the ≪toBeVerified≫ stereotype is used to tag whole packages and single containers and classes; from ≪toBeVerified≫ elements the transformation takes the element types, operations, attributes, and State diagrams, which are, in a way, the backbone of the model. In addition, the ≪system≫ stereotype is used to tag classes collecting the objects (which are instances of ≪toBeVerified≫ classes) which form the actual system on which V&V activities are to be applied. From ≪system≫ containers the transformation takes, in addition to the objects composing the system (i.e., those depicted in the Object diagrams mentioned in Sect. 2.2), the diagrams (Sequence diagrams and IODs) representing the interactions among these objects.

On the slice of the model identified through the ≪toBeVerified≫ and ≪system≫ stereotypes designers can perform verification and simulation activities supported by the MADES approach.

Two types of verification activities are supported: consistency checks and property verification. When performing *consistency checking* the designer feeds the verification tool only with the slice of the system to be analyzed. In this case, the verification tool looks for an admissible run of the model. If it can find one, this means that the model is feasible, i.e., it has at least an execution. If no run can be found, then the model contains some inconsistency (e.g., two objects required to send messages at the same time to a third component that blocks when a message is received), which must be resolved before the system can be further developed.

Property checking is the classical verification activity where the user defines, using a suitable formalism, a property against which the model is to be checked (e.g. "every time message m is sent between objects $o1$ and $o2$, state s will be entered in object $o3$"). Then, the system model and the property are fed to the verification tool, which answers in two possible ways: either the property holds for the modeled system, or there is a run of the system that violates the property. In the latter case the tool returns a *counterexample*, i.e., a system run that witnesses the violation of the property. If the property does not hold, the user can navigate through the counterexample produced by the verification tool using the traceability feature of Sect. 3.3.

3.2.1 Simulation

The goal of simulation is, given models of the behavior of the system being developed and of its environment, to produce a trace that is "compatible" with both models, i.e., that

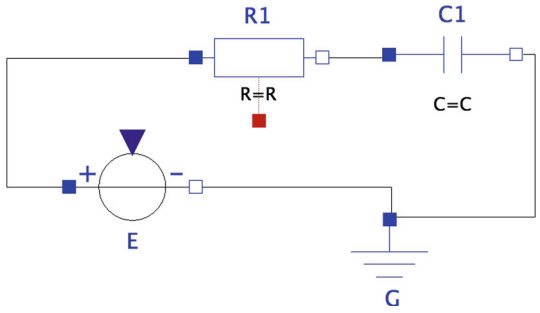


Fig. 6 RC circuit

does not violate either model. A *trace* is a sequence of values over a time interval, which correspond to the values assumed over that time interval by a set of quantities (or *variables*) of interest (physical quantities, such as temperature, velocity, altitude, or logical ones, such as a switch being turned on or off, etc.). Suppose, for example, that the variables of interest are the temperature Tp in a room and the state St on/off of a fan, and that the time interval is the discrete one $[0, 5]$; then a trace could be $[\langle \mathsf{Tp} = 25, \mathsf{St} = \mathsf{off}\rangle, \langle \mathsf{Tp} = 26, \mathsf{St} = \mathsf{off}\rangle, \langle \mathsf{Tp} = 27, \mathsf{St} = \mathsf{on}\rangle, \langle \mathsf{Tp} = 28, \mathsf{St} = \mathsf{on}\rangle, \langle \mathsf{Tp} = 27, \mathsf{St} = \mathsf{on}\rangle, \langle \mathsf{Tp} = 27, \mathsf{St} = \mathsf{on}\rangle]$. Usually, the model S of the system and the model E of its environment are expressed through different formalisms and using different notions of time. Model S is typically described through formalisms such as automata or, as in the MADES approach, logics, using a discrete notion of time (in which case interval $[0, 5]$ is, as in the example above, the sequence $[0, 1, 2, 3, 4, 5]$); the behavior of its physical environment E, on the other hand, is normally described through differential equations, using a continuous notion of time. Then, a notion of "hybrid" simulation is necessary, one that seamlessly mixes the concepts above.

The basic premise of the MADES simulation approach is that the system and environment models communicate with each other through shared variables. The reference model is essentially the one originally proposed in [23]: the system and environment models have private variables, whose values are not visible outside of the model itself, and also some shared variables that are accessible to both models and are used to communicate information between them. Each variable belongs to a model (e.g., shared system variables belong to the system model), which is used to compute the variable's value. In the MADES approach, variables can be real-valued quantities such as a speed or an acceleration, or discrete, finite (e.g., Boolean) signals such as a switch being turned on or off.

The dynamics of environment variables (i.e., model E) is typically governed by differential equations or algebraic laws, which, in the MADES approach, are expressed in the Modelica [21] language, and use a continuous notion of time. Model S, instead, is described through MADES diagrams, which are ultimately translated into a formal model expressed

as a set of metric temporal logic constraints according to the semantics presented in Sect. 2.3. The concepts that are needed to create a simulation model, such as the identification of what constitutes the "environment" and the shared variables, are introduced by designers through suitable MADES stereotypes.

To illustrate the features of the simulation mechanisms in MADES we will use throughout this paper a simple example of system that monitors and controls the RC circuit depicted in Fig. 6. The system reads the voltage on the capacitor, and, when it detects that the voltage has remained below a certain threshold for "long enough", it increases the voltage commanded by the generator and keeps it "high" for a certain amount of time. The elements of the MADES model are depicted in the Class diagram of Fig. 7; class *Env* corresponds to the RC circuit, which is the "environment" of the controller application that is being designed. The values of the voltages (the monitored one and the commanded one) are represented through attributes of the *Env* class. These attributes are of type double to indicate that they are real numbers. The controller application has two elements: a monitor component, which reads the voltage on the capacitor, and a controller component, which commands the voltage of the generator depending on the information it receives from the monitor and realizes the control strategy. As Fig. 7 shows, in this case the whole model *SimulationRCExample* is to be considered for the simulation, so it is tagged with the ≪toBeVerified≫ stereotype. To indicate which class contains the definitions of the relevant objects and interactions, class *System* is tagged with stereotype ≪system≫.

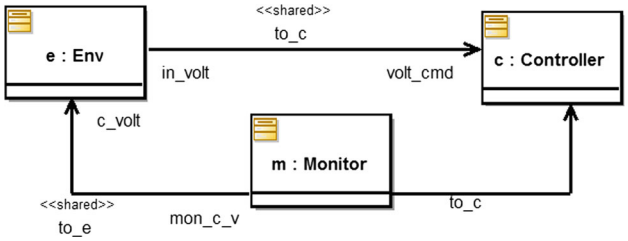


Fig. 8 Object diagram for the RC system

Class *Env* encapsulates the Modelica model, and as such it is tagged with the MADES stereotype ≪environment≫. Its attributes *c_volt* and *in_volt* are the variables that are shared between the system and the environment models; *c_volt* and *in_volt* are the names through which these variables are referred to in the Modelica model. Dually, *mon_c_v* and *volt_cmd* are the corresponding names used to indicate these variables in the system model (i.e., they are essentially synonyms used in the system model to refer to *c_volt* and *in_volt*). To state this correspondence in the MADES model, we use stereotype ≪shared≫ in the Object diagram. More precisely, since shared variable *c_volt* of object *e* of Fig. 8, which is an instance of class *Env* of Fig. 7, corresponds to variable *mon_c_v* of object *m*, a link between objects *e* and *m* is drawn; the link is tagged with the ≪shared≫ stereotype and its roles have the names of the variables that correspond to each other. The direction of the link, which points to the "owner" of the variable, indicates whether the variable belongs to the system (i.e., it is a *system* shared variable, as *volt_cmd*) or to the environment model (i.e., it is an *environment* shared variable, as *c_volt*).

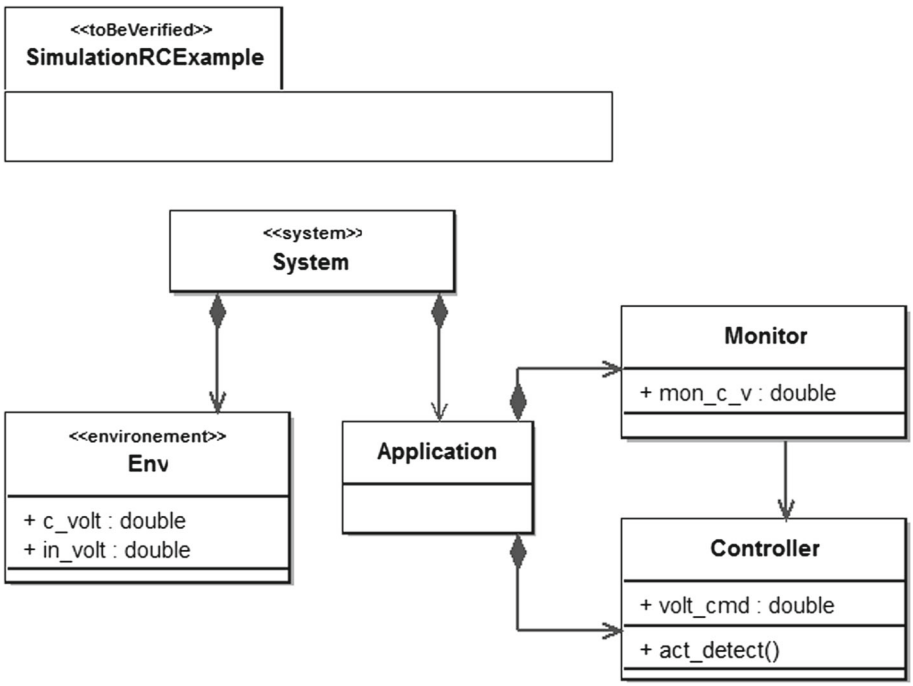


Fig. 7 Class diagram for the RC system

To illustrate how shared variables can be used in MADES diagrams, let us focus on the state diagram for the *Monitor* component, which is shown in Fig. 9. The diagram states that the component remains in state *High* as long as the value of the monitored voltage of the capacitor *mon_c_v* is > 6 V. This is captured by the invariant constraint *mon_c_v* > 6 associated with the state. As formalized in Sect. 2.3, the invariant must hold in all instants in which the component is in state *High*, except the last one, when the state is exited. When the voltage goes below the threshold, the component moves to state *Low*, as indicated by the trigger condition on the transition between the two states. One instant after having entered state *Low*, the component moves to state *Wait_Act*, where it stays for at most 2 instants, and then it changes to state *Act*. When transitioning to state *Act*, the monitor triggers the reaction of the system by activating Sequence diagram *Act Detect_Sq D* (see Sect. 4.2). Six time units after having been in state *Act* the monitor component moves to state *High* or *Low* depending on the value of the monitored voltage.

Section 4.2 shows the rest of the model of the RC system, and an example of simulation for the example system. The theoretical underpinnings and the algorithm of the MADES simulation approach can be found in [5].

3.3 Traceability

The transformation tool, which acts as the "glue" between the other components, generates traceability information, which can be used to check and maintain the consistency of the relevant artifacts. Moreover, such traceability information can be used to create a conceptual link between the different representations of the same concepts such as elements in the MADES UML model and in the formal model generated by the transformation tool.

In the approach presented in this paper, traceability support is provided between the following artifacts:

- Input MADES UML model and textual output of ℤot, that is, traceability information relates the result of the verification back to the input system model;
- Input MADES UML model and set of Java primitives generated by the *XMI2Java* component of the transformation tool. This information is captured mainly for debugging and accounting purposes.

Next, we provide a brief summary on how the traceability mechanism is implemented in MADES.

The traceability support of the proposed approach is illustrated in Fig. 10. The generated trace can be used to support backward and forward traceability between the original system model and the textual output of the verification tool. When the formal model is generated by the transformation tool, an additional internal file is generated, which contains tuples of the form ⟨ string id, XMI id ⟩. The first element of the tuple is the unique string identifier which is used in the formal model for the various model elements. This identifier is also used by the verification tool when it generates the results of the verification. The second element of the tuple is the XMI identifier from the source MADES UML model. The traceability tool of the tool chain links the two artifacts by accessing the list of tuples. When the user navigates from the textual output of ℤot to the source model, the tool finds the tuple which contains the string identifier of interest and then retrieves the XMI identifier of the model element. In a similar manner, when the user navigates from the MADES UML model to the textual output of ℤot, the tool finds the tuple, which contains the XMI identifier of interest and then it retrieves the string identifier of the textual output.

To visualize the various relationships between the two artifacts, we built a dedicated user interface (Fig. 11). This inter-

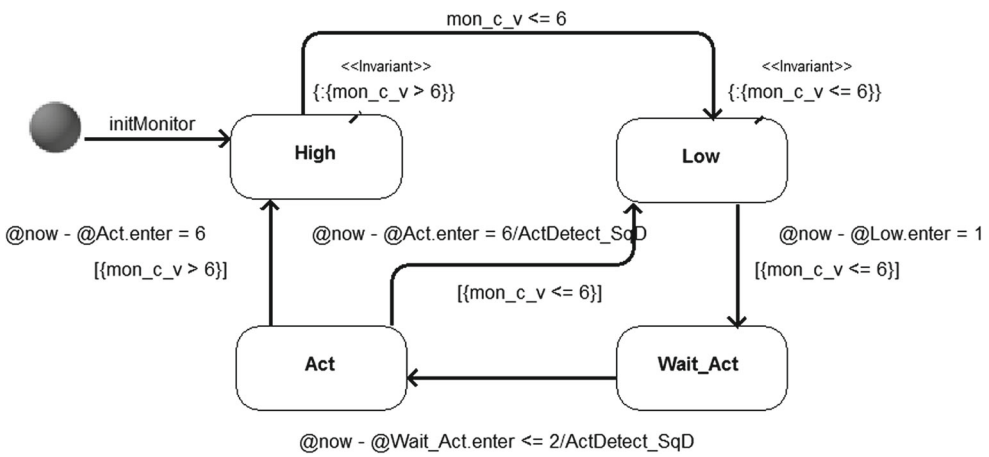


Fig. 9 State diagram for class *Monitor*

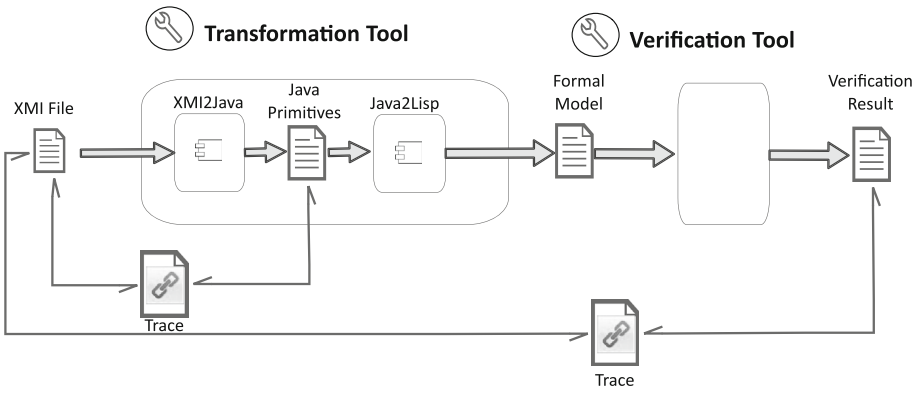


Fig. 10 Traceability support for MADES V&V tool chain

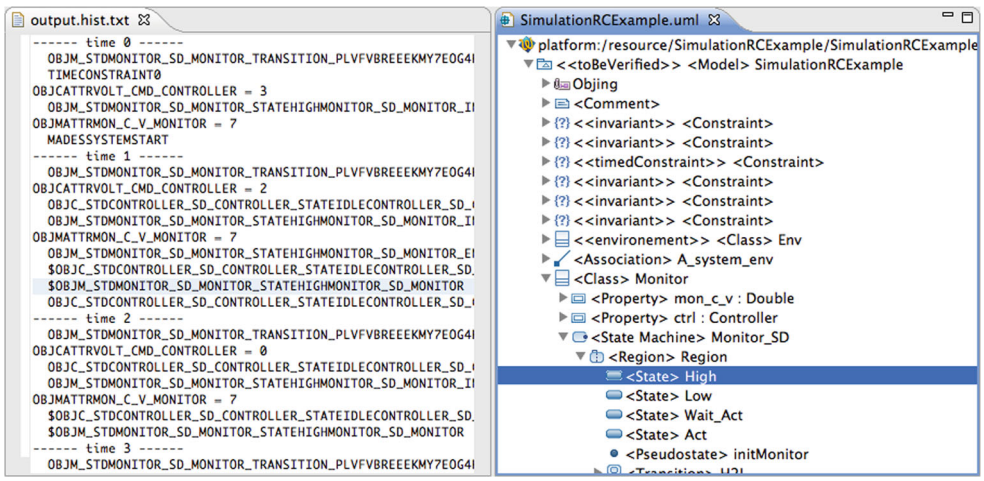


Fig. 11 Traceability interface

face is divided in two areas. The source model is shown on the right-hand side, while the textual output of the verification tool is shown on the left-hand side. The two areas are navigable through a hyperlink system. For instance, selecting a string identifier in the output file on the left highlights the corresponding model element in the source model.

In addition to the traceability support discussed above, experienced users can also have access to traceability information for the intermediate artifacts of the transformation chain. That is traceability information between the source MADES UML model and the intermediate Java representation, which is generated internally in the tool. This traceability information is generated automatically by the transformation tool utilizing the fine-grained traceability mechanisms provided by the model-to-text transformation language and tooling used, namely EGL and the Epsilon framework. This traceability information can be used by an experienced user of the tool to gain a deeper understanding of how exactly the formal model is created (i.e., which parts of the formal model are generated by specific parts of the MADES UML model) and therefore understand better the results of the verification.

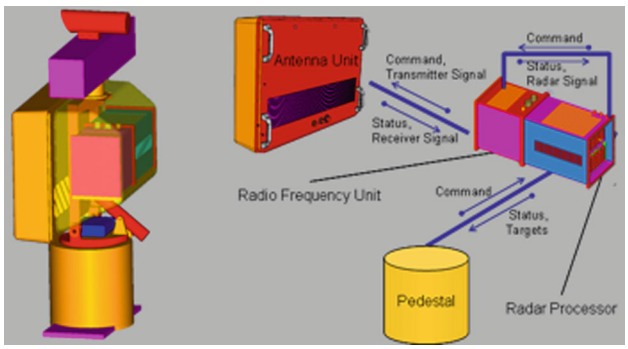


Fig. 12 Cassidian surveillance Radar

4 Assessment

This section presents the application of the MADES verification and validation solutions on two examples. The first demonstrates the use of formal verification (in the form of consistency checks) on an industrial case study provided by Cassidian[3]. The second demonstrates the use of simulation onto the RC system introduced in Sect. 3.2. We use two sep-

[3] http://www.cassidian.com.

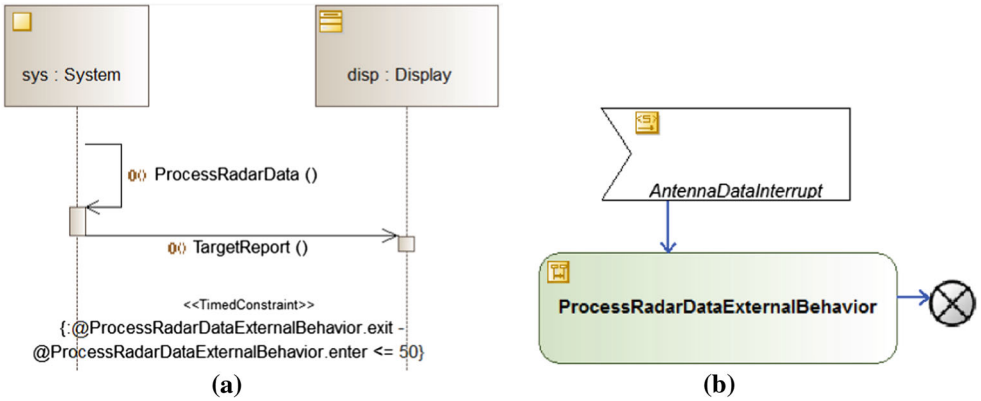


Fig. 13 External behaviour of the radar system

arate examples to illustrate the features of the MADES verification and simulation tools to better focus on the features of each technique. The two case studies witness the flexibility of the MADES solutions.

4.1 Verification

The case study addresses a ground-based surveillance radar system, presented in Fig. 12, and focuses on the Radar Processor.

The example system was chosen to compare the MADES approach against the current state of practice within the company. Since formal verification of UML models is not established as standard practice of the company, the case study showed how such an activity can be carried out, thanks to the flexibility of the MADES approach, without significantly impacting on the modeling style already in use. The case study also showed how the MADES verification approach can be used to enhance the current practice of the development of complex systems, where a common approach is the top-down refinement of the structure and allocation of functions to the resulting architectural elements (subsystems, software, hardware). In this approach, the development of the architectural elements is carried out by different teams, based on a functional specification. When integrating the architectural elements to sub-systems and systems, it happens regularly that the elements completely fulfill their functional specification, but they do not work together. Therefore, not only the structure, but also the behavior of the system, i.e. the interaction between the architectural elements, must be refined. When dealing with embedded real-time systems such as radar processors, it is essential that after each refinement step the behavior still respects the constraints that arise as a consequence of the available processing power, interface bandwidth, etc. The case study showed how the MADES formal verification approach can be used to support this step.

The pre-existing model, which was created using UML 1.4 according to the procedures described in [10], was re-engineered using the concepts and new stereotypes provided by the MADES notation [3]. The model was not significantly altered to make it verifiable by the MADES tools; instead, the MADES notation allowed designers to add precise timing constraints that were not previously expressible, and thus not analyzable.

The model includes many packages[4], each addressing a separate issue related to the development of the target system. Not all packages and classes are needed for the verification activity; those that are to be included in the formal model for verification were tagged with the ≪toBeVerified≫ and ≪system≫ stereotypes introduced in Sect. 3.2.

The radar system reacts to stimuli from its environment and in response to them performs respective computations and produces suitable outputs within prescribed time bounds. Cassidian's development process calls for defining a first, "external" view of the system through a Sequence diagram (Fig. 13a), which captures the reaction mechanism described above. To represent the fact that the computation is triggered by an external stimulus, we introduce the IOD of Fig. 13b, which describes it: when data are received from the radar antenna, the radar processor system reacts by performing its computation. The "external" behavior has a timing requirement, which is represented by the following constraint applied to the Sequence diagram of Fig. 13a:

```
@ProcessRadarDataExternalBehavior.exit -
@ProcessRadarDataExternalBehavior.enter <= 50
```
(10)

The high-level external behavior of Fig. 13 is realized through a sequence of messages exchanged by the components of the radar processor system. The details of this sequence are captured by a further Sequence diagram that refines the one of Fig. 13a. Figure 14 shows the elements of a possible software-based architecture of the radar processing system, and their interactions. As Fig. 14a shows, other solutions for the system have been designed, though they are

[4] Interested readers can refer to [11] for the details omitted in this paper.

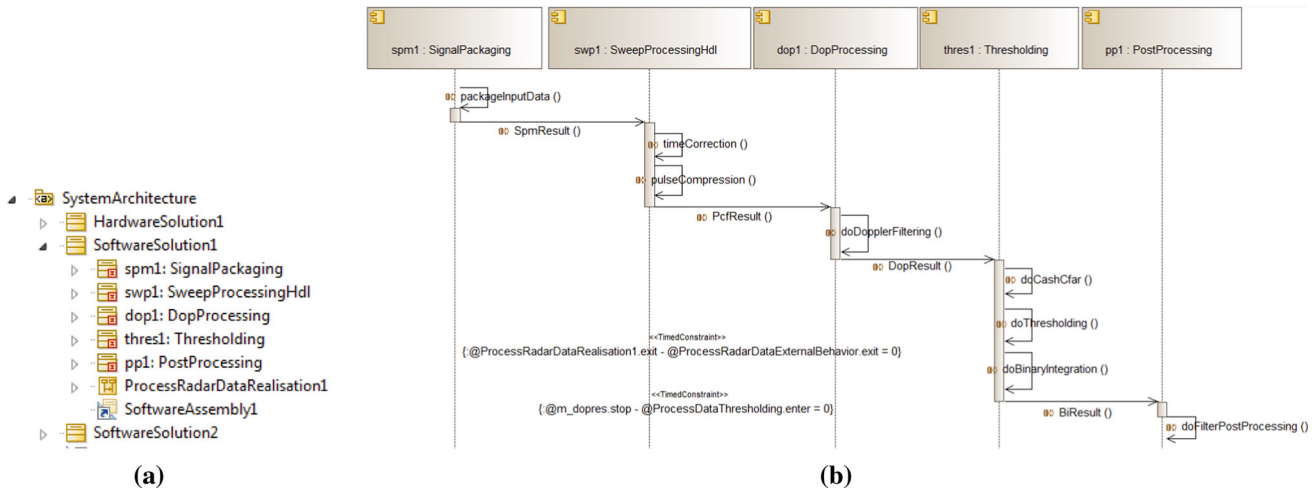


Fig. 14 Package containing the elements composing the system (**a**) and corresponding Sequence diagram (**b**)

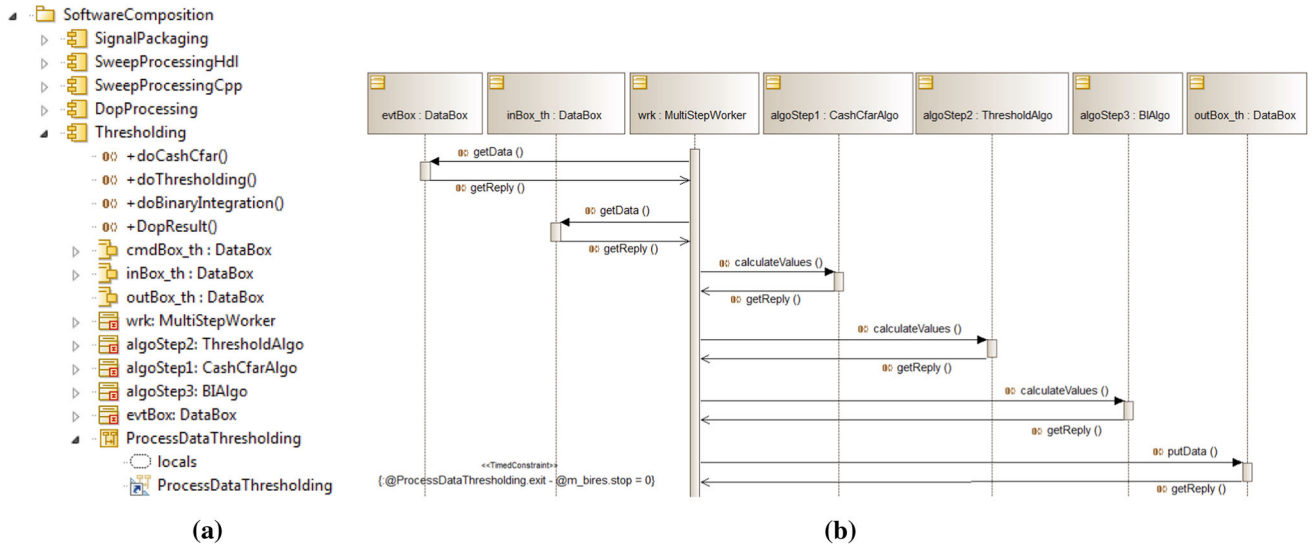


Fig. 15 Architecture of component *Thresholding* (**a**) and the interaction among its parts that achieves the thresholding computation (**b**)

not dealt with in this paper. We use MADES timing constraints to link the events of the high-level Sequence diagram of Fig. 13a to the corresponding ones of Fig. 14b. More precisely, the following constraint of the Sequence diagram of Fig. 13a states that the interaction of Fig. 14b starts when message *prd* (which corresponds to the invocation of operation *ProcessRadarData*) ends.

```
@prd.stop - @ProcessRadarDataRealisation1.enter = 0
```

Dually, the following constraint of the Sequence diagram of Fig. 14b states that it terminates exactly when the interaction of Fig. 13a also terminates [hence, the latter cannot last more than 50 time units, due to constraint (10)].

```
@ProcessRadarDataRealisation1.exit -
@ProcessRadarDataExternalBehavior.exit = 0
```

The architecture of Fig. 14a is further refined in the components shown in Fig. 15a. In particular, each of the computations performed by the objects of Fig. 14b is expanded in its own Sequence diagram, which represents the interactions occurring within the objects to achieve the computation. For example, Fig. 15b shows the interaction among the parts of component *Thresholding* that achieve the computation whose input and output messages are depicted in Fig. 14b. As before, suitable timing constraints are added to the various interactions to link their events. For example, the following constraint of the interaction of Fig. 14b states that the computation of Fig. 15b starts when message *m_dopres*, corresponding to the invocation of operation *DopResult* from object *dop*1 to object *thres*1, is delivered to the latter.

```
@m_dopres.stop - @ProcessDataThresholding.enter = 0
```

The model of the radar processor system outlined in Figs. 13, 14 and 15 has been checked for consistency through the MADES formal verification tool. More precisely, we used the tool to determine whether the system model admits some executions or not. If this does not occur, the most likely reason is that constraint (10) is too stringent given the sequence of computations that are necessary to process each batch of input data from the antenna. In fact, the semantics associated with Sequence diagrams is such that events along the same lifeline occur at separate instants of time, so an interaction such as the one of Fig. 15b takes a non-null time that, when considering also the time it takes to complete with the other computation steps depicted in Fig. 14b, creates a contradiction with constraint (10).

The check was performed in an incremental fashion, by exploiting tags ≪toBeVerified≫ and ≪system≫ to isolate the parts of the MADES model to be analyzed, starting from a basic model that included only the first level of refinement (i.e., the diagrams of Figs. 13a, 14b), then adding the details of the components of Fig. 15a.

Initially, the MADES verification tool was fed with the model consisting of essentially the diagrams of Figs. 13 and 14b, with the time bound in constraint (10) equal to 20 instead of 50. When only these elements were considered, the tool determined that the model was indeed satisfiable.

If, however, the details of the components of Fig. 15a were included, the limit of 20 time units to complete the processing was not feasible, because of the actual time to perform the various computations as modeled in the Sequence diagrams such as the one in Fig. 15b. In this case, setting the time bound of the whole computation to 50 as in constraint (10) was enough to make the model satisfiable.

Verification[5] took a time that ranged from few minutes (20, to check the consistency of the high-level model with time bound 20), to few hours (almost 3 for the check of the complete model with time bound 50). While the duration of the checks was at times considerable, the application of the MADES approach to the Cassidian case study gave encouraging results, as it allowed us to gain better insight into the modeled system. Improving the efficiency of the verification tool is part of our future work.

4.2 Simulation

This section shows the capabilities of the MADES simulation tool through the RC example system introduced in Sect. 3.2. Figure 16 shows the Modelica model of the circuit of Fig. 6. The Modelica model of the environment and the UML model of the system are created separately; the stereotypes described in Sect. 3.2 are then used to link the two models together for their co-simulation. The Modelica UML profile [39] may provide an interesting way to foster a tighter integration of the two models.

```
model RC
  Real c_volt(stateSelect=StateSelect.always);
  parameter Real in_volt = 10;
  parameter Real R = 10;
  parameter Real C = 0.5;
  Modelica.Electrical.Analog.Basic.Resistor R1(R = R);
  Modelica.Electrical.Analog.Basic.Capacitor C1(C = C);
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.Sources.SignalVoltage E;
equation
  // input voltage
  E.v = in_volt;
  // component connection
  connect(E.p,R1.p);
  connect(R1.n,C1.p);
  connect(C1.n,G.p);
  connect(G.p,E.n);
  // capacitor voltage (output)
  c_volt = C1.v;
algorithm
  when initial() then
    Init();
  end when;
end RC;
```

Fig. 16 Modelica model of the RC circuit

In addition to the diagrams shown in Figs. 7, 8 and 9, the MADES model of the software controller of the RC circuit includes the Sequence diagram of Fig. 18 and the State diagram of class *Controller*, which is shown in Fig. 17. The Sequence diagram of Fig. 18 is activated any time the monitor object, whose behavior is governed by the State diagram of Fig. 9, moves from state *Wait_Act* to state *Act*, as indicated by the action on the transition between the two states. The Sequence diagram describes the interaction between the monitor and controller objects, which consists simply of the monitor invoking operation *act_detect* on the controller. A ≪TimedConstraint≫ associated with the diagram imposes that the message is sent when the interaction starts (i.e., when the State diagram of the monitor object takes the transition from *Wait_Act* to *Act*). The State diagram of Fig. 17 shows that, before the controller receives an invocation of operation *act_detect* (i.e., while in state *Idle*), it keeps the voltage that is input to the RC circuit low (in the [0, 2] range, as stated by the invariant on the state). The invariant does not fix exactly the value of the commanded voltage, but it only sets a range within which the simulator can arbitrarily choose a value when generating the trace. Upon receiving an invocation of operation *act_detect*, the controller moves to the preparatory state *Wait*, where it stays until 1 time unit has elapsed since entering the state. While in state *Wait*, the controller keeps the commanded voltage in an intermediate

[5] The verification was carried out on a desktop computer with a 2.8GHz AMD Phenom™II processor and 8MB RAM; Zot was configured to use the SMT-based `smteezot` plugin, and the solver was Microsoft Z3 3.2. The bound on the length of the traces sought was 100.

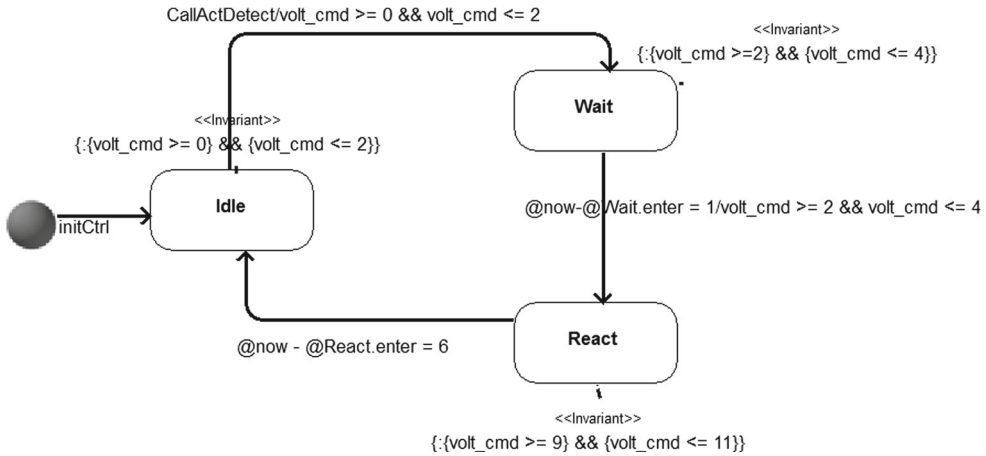


Fig. 17 State diagram for class *Controller*

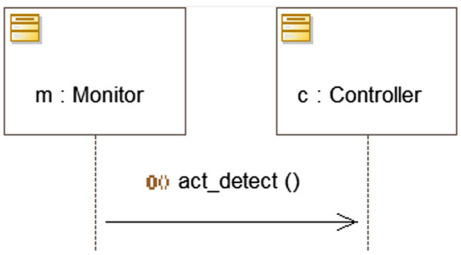


Fig. 18 Sequence diagram for the simulation example

range of [2, 4] volts. After the stated amount of time elapses, the component transitions to state *React*, where it stays until 6 time units have elapsed, and in which the commanded voltage is raised in the [9, 11] range. Then it moves back to state *Idle*, and the loop restarts.

Figure 19 shows the dialog box for launching the simulation on the MADES and Modelica models of the RC system and its environment. As the Figure shows, the user indicates which models are to be used, and the tool generates the formal model from the MADES one, sets up the simulation tool, then launches the simulation tool itself, and opens it in a separate window. Trough the simulation tool GUI the user can set up some simulation parameters, such as the sampling interval, the simulation horizon, and the backtrack depth (see [5] for further details), then launch the actual simulation. The tool implements the algorithm described in [5], and (possibly) produces a trace, such as for example the ones depicted in Fig. 20, which shows the traces of the monitored and of the commanded voltages. As the figure shows, the controller reacts in full force a few instants after the capacitor voltage has dipped below the threshold of 6 V; when it reacts, it increases the commanded voltage in the [9, 11] range for several instants; then, as the voltage has gone over the threshold again, it sets the commanded voltage back to a value in the [0, 2] range.

In this case the simulator was able to produce a trace that is compatible with both the system and the environment models. It can occur that the tool is not able, after exhausting the backtrack depth set by the user, to produce a combined trace. While this is not a proof that the closed-loop system is infeasible, as the simulator does not perform and exhaustive exploration of the space of the traces of the system, it is nonetheless an indication that there might be a problem with the designed system which ought to be investigated.

For example, in an earlier version of the model of the RC controller the Sequence diagram of Fig. 18 stated that the controller object remained blocked and unable to receive new operation invocations for some instants after an invocation of *act_detect*. At the same time the State diagram of Fig. 9 stated that the monitor component re-checked the value of the capacitor voltage immediately after invoking operation *act_detect*, which clashed with the blocking of the controller. In fact, as the capacitor voltage did not have time to grow past the desired threshold, this entailed a new invocation of operation *act_detect*, which was however not possible. In this case the simulator was not able to produce a trace of the closed-loop system, which led us to change the controller logic.

5 Related work

Our approach builds upon research conducted in different and heterogeneous areas such as semantic foundations of UML, model checking as well as Model-Driven Engineering (MDE). In the following, we will highlight the most influential work, which is related to the proposed approach.

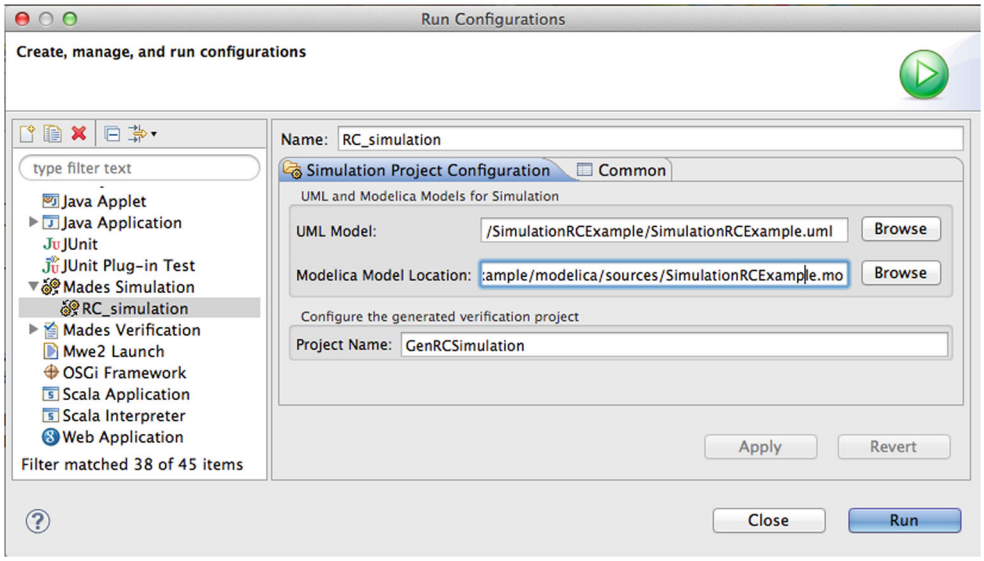


Fig. 19 Dialogue box for launching the simulation

5.1 UML semantics

Many earlier versions of UML explained the semantics of the language in English; more recent revisions (e.g., UML 2.5) as well as recent developments such as fUML are moving towards a more mathematical semantics, though largely these efforts are not focusing specifically on support for formal verification. As a result formal analysis of UML models is difficult, and ascribing formal semantics to UML to support formal analysis has been a very active area of research. The majority of the existing approaches propose the manual or automatic translation of various UML diagrams or elements to well-established formalisms such as Petri nets, Z or timed automata. For example, Evans et al. [19] translate UML class diagrams to the Z notation and then they utilize it to reason about the properties of the model. There are other examples (e.g. [17,42,44]) that propose the transformation of specific UML diagrams (e.g. class diagrams, state machines, etc) to a flavor of Z.

Another strand of research attempts to formalize aspects of UML by translating UML diagrams to Petri nets. Hammal [24] provides a formalization of UML statechart diagrams by translating them into Interval Petri Nets (ITPN), thus enabling the analysis of the performance and of the time properties of the system. Similarly, [40,41] propose a mapping of activity diagrams onto Petri nets. Saldhana and Shatz [38] use collaboration diagrams to connect statecharts and then they derive an Object Petri Net of the system. This is the enabler to apply standard techniques for Petri nets.

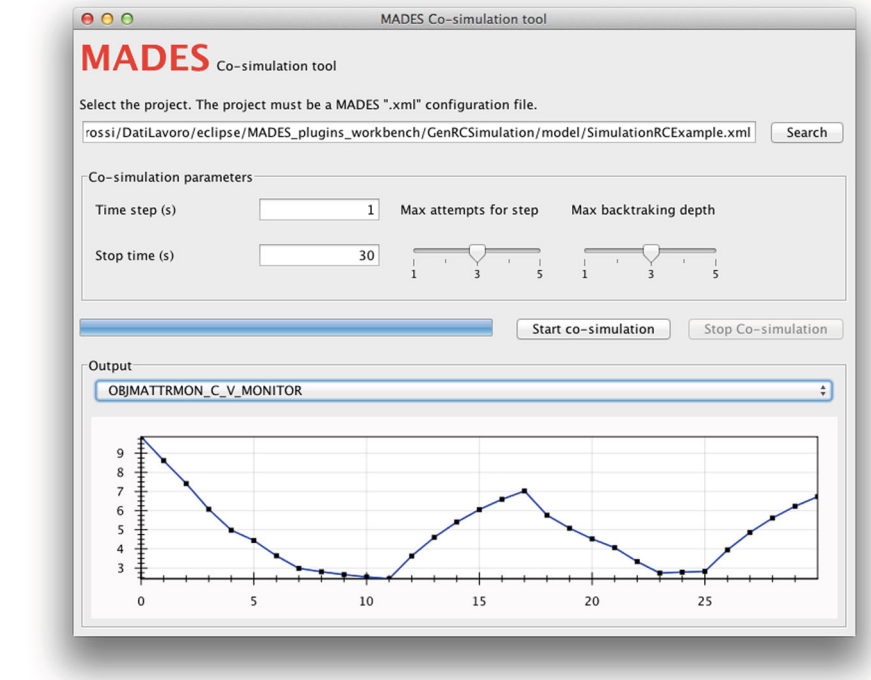
A third approach to the formalization of UML utilizes timed automata as underlying formalism. Vooduu [16] can be used to verify whether a set of statechart diagrams satisfy a time constraint expressed as a sequence diagram. To perform the verification both types of models are translated into timed automata. Again, [12] models the static and dynamic behavior of a system using statechart diagrams, then formalized in terms of hierarchical timed automata.

One major limitation of the aforementioned approaches is that they concentrate on a single type of diagram. Therefore, they neglect the problem of integrating different structural and behavioral diagrams to get a more holistic view of the system. Moreover, the approaches that utilize Petri nets suffer from the limitations of this formalism such as the explosion of resulting nets and the inability to distinguish between types and instances.

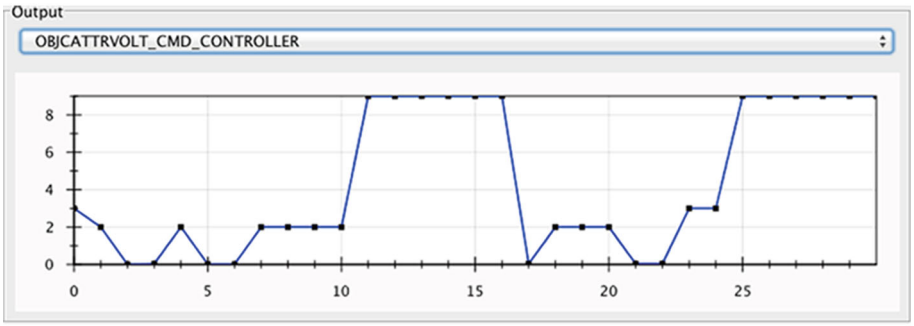
Also the Object Management Group (OMG) has recently released the fUML specification [31], which consists of a subset of UML2 as well as the execution semantics for this subset. The execution semantics of fUML is given in terms of first-order predicates or axioms over possible execution traces. Although the adopted formalism is appropriate for model execution, it is not appropriate for formal verification of user-defined properties. The semantics of time, concurrency, and inter-object communication is neglected, while many of the high-level diagrams of UML are not covered.

5.2 Model-driven transformation of UML models

With the advent of Model-Driven Engineering (MDE), the translation from UML models to formal representations has become easier, since novel technologies have been developed to assist the automatic transformation and manipulation of models.



(a)



(b)

Fig. 20 Results of the simulation of the RC system: monitored capacitor voltage (**a**) and commanded voltage (**b**)

The emphasis in MDE is on using standardized approaches for capturing models (e.g., UML [33], EMF/Ecore [43]), and task-specific languages (e.g., QVT, VIATRA [15], ATL [26], Epsilon [27], Kermeta [20]) for processing models to perform tasks such as internal and external model-to-model transformation, model-to-text transformation, model comparison, merging, refactoring, and migration in an automated manner.

The research into model-driven development of real-time and embedded systems has attracted substantive attention in recent years. For example, Gaspard2 [34] is an Integrated Development Environment (IDE) for System on Chip (SoC) development, whose objective is to provide an integrated environment that enables the modeling, simulation, code generation,deployment specification of SoC applications and hardware architectures, and the association of the two. The Gaspard2 framework uses MARTE for high-level system representation. The approach is based on MDE principles allowing the engineers to move from abstract system models to an executable platform.

The main focus of the MARTES [1] project is on the efficient and combined use of UML and SystemC for the systematic model-based development of real-time embedded systems. The project adopts ideas from MDA and aims to provide viable means for the definition, construction, experimentation, validation, and deployment of embedded applications on heterogeneous platforms.

Finally, the Quasimodo (Quantitative System Properties in Model-Driven-Design of Embedded Systems) project [30] aims to develop theory, techniques, and tool components for handling quantitative constraints in model-based development of embedded systems. Quantitative constraints involve the resources that a system may use, assumptions about the environment in which it operates, or the requirements on the services that the system has to provide.

Neither Gaspard2 nor MARTES support formal verification. In addition, the first two rely on UML dialects to model target systems, whilst Quasimodo uses timed, hybrid, and probabilistic automata.

6 Conclusions

One of the main limitations of UML is that its diagrams are hard to formally verify and validate due to inadequately defined semantics. This paper proposes the MADES solution to address this problem.

The proposed solution consists of three major contributions: a dedicated UML profile (MADES UML), a formal semantics ascribed to MADES UML and based on the TRIO metric temporal logic, and a modeling/verification framework that enables the verification and closed-loop simulation of designed systems. The resulting framework is entirely based on mature and open-source technologies.

The approach has been evaluated in close collaboration by the MADES industrial and academic partners on a number of representative case studies from the embedded systems domain. The proposed solution helped model all the problems and details of interest and the automatic verification helped reveal subtle problems since the very early phases of the design process.

As for the future work, the definition of the properties of interest, currently specified through a form-based approach, requires in-depth analysis and more domain-specific solutions. In parallel, we will continue the finer-grained improvement, optimization, and tailoring of the MADES
solution.

Acknowledgments This research was supported by the Seventh Framework Program (FP7/2007–2013) of the European Community, project MADES (248864), and by the Programme IDEAS-ERC, project SMScom (227977).

References

1. Andersson, P., Höst, M., Bergström, M.: UML to SystemC transformation in the MARTES project. In: Proceedings of the Work in Progress Session at Euromicro SEAA/DSD (2006)
2. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 4735, pp. 559–573 (2007)
3. Bagnato, A., Andrey Sadovykh, E.B., Matragkas, N., Rossi, M., Baresi, L., Morzenti, A., Motta, A., Crippa, M.C., Genolini, S., Audsley, N.C., Gray, I., Indrusiak, L.S., Kolovos, D., Paige, R.: D1.7 mades final approach guide. Technical report, MADES Consortium (2012)
4. Bagnato, A., Sadovykh, A., Paige, R.F., Kolovos, D.S., Baresi, L., Morzenti, A., Rossi, M.: MADES: embedded systems engineering approach in the avionics domain. In: 1st Workshop on Hands-on Platforms and Tools for Model-Based Engineering of Embedded Systems (HoPES), p. 5 (2010)
5. Baresi, L., Ferretti, G., Leva, A., Rossi, M.: Flexible logic-based co-simulation of modelica models. In: IEEE International Conference on Industrial Informatics (INDIN), pp. 635–640 (2012)
6. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: From interaction overview diagrams to temporal logic. In: MoDELS Workshops. Lecture Notes in Computer Science, vol. 6627, pp. 90–104 (2010)
7. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: Towards the UML-based formal verification of timed systems. In: Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 6957, pp. 267–286 (2012)
8. Baresi, L., Orso, A., Pezzè, M.: Introducing formal specification methods in industrial practice. In: Proceedings of the 19th International Conference on Software Engineering (ICSE), pp. 56–66 (1997)
9. Bersani, M.M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: Proceedings of the International Symposium on Temporal Representation and Reasoning (TIME), pp. 43–50 (2010)
10. Blohm, G., Bagnato, A.: D1.1 requirements specification. Tech. rep., MADES Consortium (2010). Available from MADES website. http://www.mades-project.org
11. Blohm, G., Eren, E., Bagnato, A., Bernardi, F.: D5.3 final evaluation report. Technical report, MADES Consortium (2012)
12. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 670–671 (2005)
13. Choppy, C., Klai, K., Zidani, H.: Formal verification of uml state diagrams: a petri net based approach. SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
14. Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., Morzenti, A.: From formal models to formally-based methods: an industrial experience. ACM Trans. Softw. Eng. Methodol. **8**(1), 79–113 (1999)
15. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA—visual automated transformations for formal verification and validation of UML models. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, pp. 267–270 (2002)
16. Diethers, K., Huhn, M.: Vooduu: Verification of object-oriented designs using uppaal. In: Proceedings of TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 139–143 (2004). http://link.springer.com/chapter/10.1007%2F978-3-540-24730-2_10
17. Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: An overview of RoZ: a tool for integrating UML and Z specifications. In: Wangler B., Bergman L. (eds.) Advanced Information Systems Engineering. Lecture Notes in Computer Science, vol. 1789, pp. 417–430 (2000)
18. Eshuis, R.: Reconciling statechart semantics. Sci. Comput. Programm. **74**, 65–99 (2009)
19. Evans, A., France, R.B., Grant, E.S.: Towards formal reasoning with UML models (1999)
20. Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in Kermeta. In: ECMDA Traceability Workshop (ECMDA-TW'06) (2006)
21. Fritzson, P.A.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley, London (2004)
22. Gray, I., Audsley, N.C.: Exposing non-standard architectures to embedded software using compile-time virtualisation. In: Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), pp. 147–156 (2009)

23. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. Softw. IEEE **17**(3), 37–43 (2000)
24. Hammal, Y.: A formal semantics of UML statecharts by means of timed petri nets. In: Proceedings of FORTE. Lecture Notes in Computer Science, vol. 3731, pp. 38–52 (2005)
25. Jackson, D.: Lightweight formal methods. In: FME 2001: Formal Methods for Increasing Software Productivity. Lecture Notes in Computer Science, vol. 2021, pp. 1–1 (2001). http://link.springer.com/chapter/10.1007%2F3-540-45251-6_1
26. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'06, pp. 719–720 (2006)
27. Kolovos, D.S., Paige, R., Rose, L., Polack, F.: The Epsilon Book. York, UK, Technical report, The University of York (2010)
28. Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., Pourzandi, M.: Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. Electron. Notes Theor. Comput. Sci. **254**, 143–160 (2009)
29. MADES: Model-based methods and tools for Avionics and surveillance embeddeD SystEmS (2012). http://www.mades-project.org/
30. Nielsen, B.: Quasimodo—quantitative system properties in model-driven-design of embedded systems (2007). http://www.quasimodo.aau.dk
31. Object Management Group: Semantics of a foundational subset for executable UML models (fUML). Technical report, OMG (2011). Formal/2011-02-01
32. OMG: UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems. Technical report, November, OMG (2009)
33. OMG: Unified Modeling Language—Infrastructure. Technical report, May, OMG (2010). http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/
34. Éric, P., Atitallah, R.B., Marquet, P., Meftali, S., Niar, S., Etien, A., Dekeyser, J.-L., Boulet, P.: Gaspard2: from MARTE to SystemC Simulation. In: Proceeedings of the DATE'08 Workshop on Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML Profile (2008). www2.linfl.fr/marteworkshop/proceedingsMarteWS08.pdf
35. Pradella, M., Morzenti, A., San Pietro, P.: Bounded satisfiabil-ity checking of metric temporal logic specifications. ACM Trans. Softw. Eng. Methodol. (2012, in press)
36. Radjenovic, A., Matragkas, N.D., Paige, R.F., Rossi, M., Motta, A., Baresi, L., Kolovos, D.S.: MADES: a tool chain for automated verification of UML models of embedded systems. In: Modelling Foundations and Applications. Lecture Notes in Computer Science, vol. 7349, pp. 340–351 (2012)
37. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: The epsilon generation language. In: Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA), pp. 1–16 (2008)
38. Saldhana, J.A., Shatz, S.M.: UML diagrams to object petri net models: an approach for modeling and analysis. In: Proceedings of SEKE 2000, pp. 103–110 (2000)
39. Schamai, W., Fritzson, P., Paredis, C., Pop, A.: Towards unified system modeling and simulation with modelicaml: modeling of executable behavior using graphical notations. In: Proceedings of the 7th International Modelica Conference, pp. 612–621 (2009)
40. Staines, T.: Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In: IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pp. 191–200 (2008)
41. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities (2005)
42. Than, X., Miao, H., Liu, L.: Formalizing the semantics of UML statecharts with Z. In: The Fourth International Conference on Computer and Information Technology (CIT), pp. 1116–1121 (2004)
43. The Eclipse Foundation: Eclipse Modeling Framework (EMF) (2012). http://www.eclipse.org/modeling/emf/
44. Williams, J.R., Polack, F.A.C.: Automated formalisation for verifi-cation of diagrammatic models. Electr. Notes Theor. Comput. Sci. **263**, 211–226 (2010)
45. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1–19:36 (2009). http://dl.acm.org/citation.cfm?id=1592436