Towards the UML-Based Formal Verification of Timed Systems

Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi

Politecnico di Milano
Dipartimento di Elettronica e Informazione, Deep-SE Group
Via Golgi 42 – 20133 Milano, Italy
{baresi,morzenti,motta,rossi}@elet.polimi.it

Abstract. This paper presents the approach to the formal verification of UML-based models of timed systems developed in the MADES project. The approach differs from many current ones in that it aims at (i) being inclusive in the range of diagrams considered when producing the formal model, and (ii) adhering to the UML notation as much as possible. The metric temporal logic-based semantics developed in the project is presented through an example system.

1 Introduction

UML, along with its dialects and profiles, is a widely utilized graphical, design notation. Despite the vast adoption, users only tend to agree on the interpretation of few well-known concepts, while the actual behavior of many parts of the notation is left open. The concrete syntax of the language is very rich, and offers alternatives to model the same concepts, but its semantics is only defined informally and imprecisely. This is enough if we think of UML as a pure modeling notation; it is not acceptable when one aims to detailed descriptions of the system-to-be, neither is it suitable for automated analysis and for deriving implementations that go beyond the frame of some classes.

In contrast, formal methods and tools (e.g., UPPAAL[1] or Alloy[2]), which would provide sophisticated analysis capabilities, have often demonstrated their inability to attract the masses: the required mathematical background hampers their adoption and many users privilege the “simplicity” of UML-like notations rather than more formal means. Many proposals (e.g., [10,11,18]) have already tried to bridge the gap between the two domains by attempting to provide (parts of) UML with a precise (possibly formal) semantics. The idea is to keep the positive aspects from both fields and provide the user with a well-known modeling notation, suitably augmented with a formal semantics behind the scene.

If we think of UML as design means for a well-known programming language (e.g., Java), the subset of the notation usually considered is very limited (mainly just class diagrams), and the actual semantics is assumed to be the same as

[1] http://www.uppaal.org
[2] http://alloy.mit.edu/

B.K. Aichernig, F.S. de Boer, and M.M. Bonsange (Eds.): FMCO 2010, LNCS 6957, pp. 267–286, 2011.
© Springer-Verlag Berlin Heidelberg 2011

the one of the target language. Petri nets have been widely used to explain the dynamic behavior of UML activity and state diagrams for years [19,12], but the immediate explosion of the resulting nets, along with the inability to easily distinguish between types (classes) and instances, hampered their adoption as underlying formal representation for UML models.

The heterogeneity and overlapping of the different modeling elements, the wide spectrum of the notation, and also the size of the resulting formal specifications, which are often too big for analysis, play against complete formalizations. A coherent, consistent, and complete formal semantics is a very complex and heavy task; things become even more complex when one considers some special-purpose extensions defined for particular domains.

These limitations have been our motivations, and challenge, for ascribing a formal dynamic semantics to a particular UML-based notation for timed systems, the MADES modeling notation [1]. This language borrows many concepts from SysML [15] and MARTE [14], but our interest is mainly in the timing aspects: we are interested in the *clocks* provided by MARTE, and in the UML diagrams that show time-related behaviors.

Being well aware of the difficulties inherent to the task, we decided to concentrate on a complete subset of the notation, which we called *verification* notation. This is limited with respect to the general modeling notation, but selected elements and diagrams are able to cover all the important aspects of a complete MADES model:

– The static parts of a system are covered through class diagrams. These diagrams, which can also be adopted to render components and objects, are used to define the terms (the alphabet) of the specification.
– The dynamic aspects and behavior of the different parts are rendered through: (a) State diagrams (and thus also activity diagrams), used to model the behavior of the different elements (components), (b) Sequence diagrams, used to model the "local" interactions among the different elements of the system, and (c) Interaction overview diagrams, used to relate different sequence diagrams. Sequence diagrams are adopted to describe limited scenarios that define how the system reacts to some particular conditions and/or inputs; interaction overview diagrams describe more complete interactions, and thus more general, and system-wide, properties.
– Clocks (and time diagrams) are used to add the time dimension to systems, constrain the behavior of components, and be able to predicate on it.

All these diagrams supply users with a complete, homogeneous set of concepts to render the system-to-be in a consistent way, offer a "simple" coherent notation and keep the verification phase simple.

The rest of paper is organized as follows. Section 2 provides an overview of the MADES approach. Section 3 presents the background concepts of the semantic notation. Section 4 defines the *verification* notation, while Section 5 uses an example system to introduce the formal semantics. Section 6 surveys some related approaches and Section 7 concludes the paper.

2 Modeling and Verification Workflow

The MADES workflow is designed to allow users to carry out formal verification activities while hiding from them the details of the creation of the formal model and also of the execution of the verification phase itself. Two issues are key in this approach: (i) the notation used for modeling the system to be verified must be one with which the user is familiar with (ii) the verification phase must be carried out without user intervention, in a "push-button" manner.

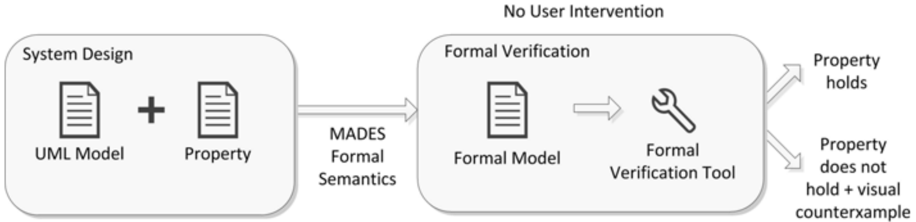

Fig. 1. Overview of the MADES workflow

Figure 1 provides an overview of the workflow. It starts with the definition by the user of a UML model of the system to be verified. By design, in the MADES approach the modeling notation conforms to the UML standard (including a pair of relevant profiles such as SysML [16] and MARTE [14]), with some restrictions that are needed to make the models verifiable in a fully automated way. In addition, the user provides as input the property to check the model against. We will discuss later how this property can be expressed.

Both the UML model and the property to be checked are translated automatically, without user intervention, into a formal model that is suitable to be input to a push-button formal verification tool. The translation is performed using the semantics described in Section 5. Then, the tool is run on the system/property combination, and its result is output. This can be either the notification that the property holds or, if the property does not hold, a trace of the system that violates it. In the second case, the counterexample trace is shown in UML form as feedback. The trace can be used to examine the behavior of the objective system.

Figure 2 shows a more detailed view of the MADES workflow, which highlights a distinguishing feature of the MADES approach, namely its inclusiveness with respect to the set of UML diagrams taken into account. In fact, unlike most existing approaches (see also Section 6), MADES allows users to use and combine a rich variety of UML diagrams to define the behavior of the system being designed. More precisely, the set of UML diagrams taken into consideration consists of class diagrams, object diagrams, sequence diagrams (SD), interaction overview diagrams (IOD), and state diagrams. Class diagrams and object diagrams provide an high-level overview of: (i) the types of the components of the system, (ii) the instances (i.e., the objects) of such types that are actually

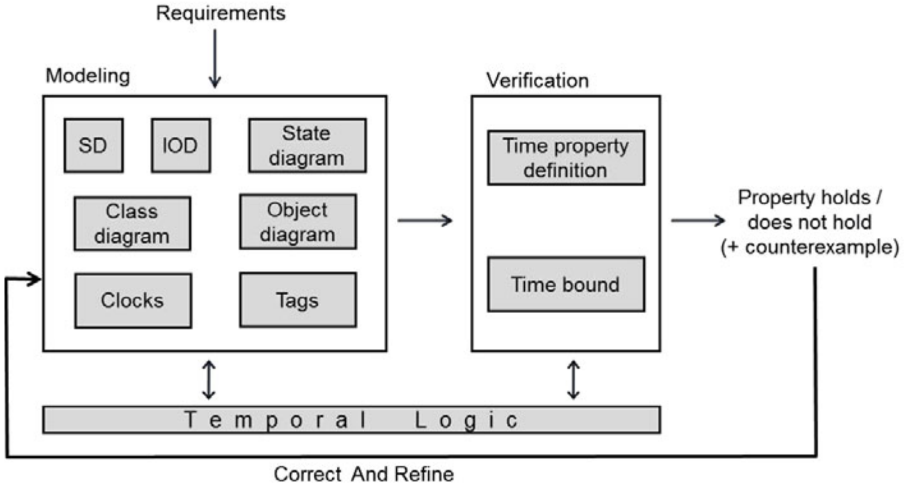

Fig. 2. Detailed workflow, with indication of available UML diagrams

present in the system, and (iii) their interconnections. Essentially, these diagrams provide the *alphabet* of the formal model, i.e., the basic items and events whose dynamics are described through the behavioral diagrams. State diagrams are used to describe the behavior of the components of the system taken one by one, in terms of their operations and of the effect that these have on their states (i.e., the attributes). Sequence diagrams, instead, define the basic interactions between the components introduced through class and object diagrams; these basic interactions are in turn composed into more complex ones through interaction overview diagrams. In addition, MARTE clocks can be used as reference to express timing constraints between events in the aforementioned diagrams. Finally, MADES allows users to add to diagrams some domain-specific tags that can be used to optimize the verification phase (these tags will not be discussed in this paper; an overview of them can be found in [1]).

Section 4 described the meta-model of the UML subset that designers can use to model systems to be verified in the MADES approach. To facilitate formal verification of MADES models, this subset of UML must be given a semantics based on a formalism that is at the same time *flexible*, to capture the meaning of heterogeneous diagrams, and *decidable*, to allow for fully automated verification. In addition, the formalism must be able to express the timing constraints described in the MADES notation through clocks. Given such requirements, the underlying formalism used in the MADES approach is the TRIO metric temporal logic described in Section 3, and in particular the decidable subset thereof that is supported by the $\mathbb{Z}$ot bounded model/satisfiability checker (hence the need to specify a time bound for the verification phase, as shown in Figure 2).

Finally, let us point out that, in the MADES approach, the property to be verified for the system can be expressed in two different ways. The first one is through the underlying formalism which is used to formalize the system (in our case, TRIO); this, however, would be against the idea that the user should never see the underlying formal notation. The second one is through a UML-based graphical notation that expresses the property of interest and that can be translated into the underlying formalism. The set of properties that can be expressed in TRIO is bigger than the set of properties that can be expressed through the UML-like notation. However UML hides the complexity of the TRIO language to the user. The UML-like graphical notation is still on-going research, thus this aspect of the work flow will not be discussed in this paper; some considerations in this regard can be found in [1][2].

3 TRIO and $\mathbb{Z}$ot

TRIO [7] is a first-order linear temporal logic that supports a metric on time. TRIO formulae are built out of the usual first-order connectives, operators, and quantifiers, as well as a single basic modal operator, called Dist, that relates the *current time*, which is left implicit in the formula, to another time instant: given a time-dependent formula F (i.e., a term representing a mapping from 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, in this paper we assume the standard model of the nonnegative integers $\mathbb{N}$ as discrete time domain. For convenience in the writing of specification formulae, TRIO defines a number of *derived* temporal operators from the basic Dist, through propositional composition and first-order logic quantification. Table 1 defines some of the most significant ones, including those used in this paper.

The TRIO specification of a system consists of a set of basic *items*, which are primitive elements, such as predicates, time-dependent values, and functions, representing the elementary phenomena of the system. The behavior of a system over time is described by a set of TRIO formulae, which state how the items are constrained and how they vary, in a purely descriptive (or declarative) fashion.

The goal of the verification phase is to ensure that the system S satisfies some desired property R, that is, that $S \models R$. In the TRIO approach S and R are both expressed as logic formulae Σ and ρ, respectively; then, showing that $S \models R$ amounts to proving that $\Sigma \Rightarrow \rho$ is valid.

TRIO is supported by a variety of verification techniques implemented in prototype tools. In this paper we use $\mathbb{Z}$ot [17], a bounded satisfiability checker which supports verification of discrete-time TRIO models. $\mathbb{Z}$ot[3] encodes satisfiability

[3] http://home.dei.polimi.it/pradella/Zot

Table 1. TRIO derived temporal operators

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

(and validity) problems for discrete-time TRIO formulae as propositional satisfiability (SAT) problems, which are then checked with off-the-shelf SAT solvers. More recently, we developed a more efficient encoding that exploits the features of Satisfiability Modulo Theories (SMT) solvers [3]. Through $\mathbb{Z}$ot one can verify whether stated properties hold for the modeled system (or parts thereof) or not; if a property does not hold, $\mathbb{Z}$ot produces a counterexample that violates it.

4 A Verifiable Subset of UML

This section presents, through a set of UML class diagrams, the meta-model of the verification notation. The meta-model is divided into the packages as shown in Figure 3.

The fundamental elements of the notation are grouped together in the *core* package, which includes different class diagrams. *core.diagrams* (shown in Figure 4) describes the set of diagrams used in the verification workflow and how they are related to one another. The MADES model is composed by Class diagrams, Object diagrams, and Interaction Overview Diagrams (IOD). State diagrams describe the behavior of the objects belonging to a certain class, and Sequence diagrams show the details of the interactions between objects.

Diagram *core.types* (Figure 5) gives an overview of the data types that can be used in the verification work flow. A *TypedElement* is an element of the system that has a type. A type can be one of the classes declared in the class diagram, or a *DataType*. A *DataType* can be a *PrimitiveType*, an *Array* or an *Enumeration*. Primitive types are *Boolean*, or *Integer*. An *Array* is an ordered list (of fixed size) of *Integers*. An *Enumeration* is a finite set of *Integers*.

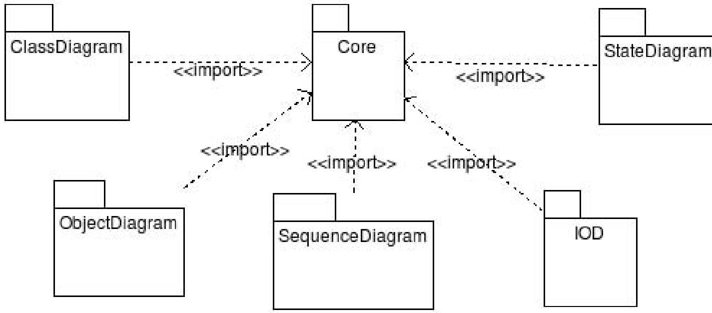

Fig. 3. Metamodel packages

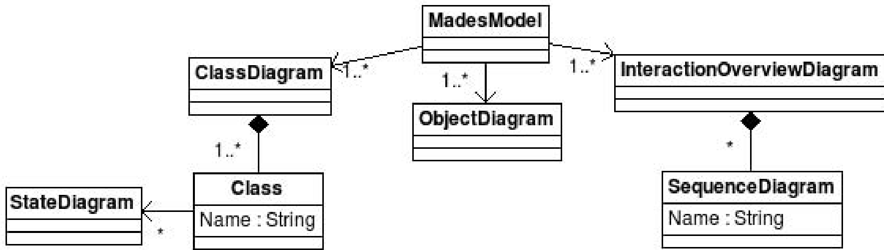

Fig. 4. MADES metamodel: core.diagrams

Diagram *core.events* (shown in Figure 6) defines what is an *Event*. Events are directly translated into temporal logic predicates and define how the system proceeds over time. Their temporal relationships will be precisely defined in Section 5; here, we simply list them with their informal meaning. *ActivityStart* and *ActivityEnd* occur in the time instants in which the IOD activity starts and ends. *DecisionPath* and *ForkPath* correspond to the time instants in which the IOD takes a certain path after a decision/fork operator. *JoinEnd* is the time instant in which all the diagrams preceding a certain IOD Join operator complete their execution. *SDStart* and *SDEnd* occur when a certain sequence diagram starts/ends. *SDAct* is the time instant is which a certain sequence diagram is ready to start. *SDAct* immediately precedes *SDStart*. *MessageStart* and *MessageEnd* occur when a certain message starts/ends. *ExOccStart* and *ExOccEnd* occur when a certain execution occurrence starts/ends. A *TimeEvent* is a *TimedInstantObservation* in a certain sequence diagram (for details see UML::CommonBehaviours::SimpleTime from [16]). The time note @$t1$ in Figure 12 of Section 5 shows an example of *TimeEvent*. Finally, *Interrupt* is the time instant in which a certain interrupt (i.e., an event that causes activities in an interruptible region of a IOD to exit, see [16] for further details) occurs. The rest of the metamodel describes how these events are associated to elements in the various behavioral diagrams, as shown, for example, in Figure 8.

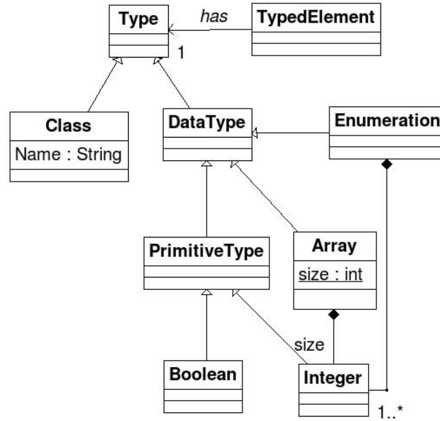

Fig. 5. MADES metamodel: core.types

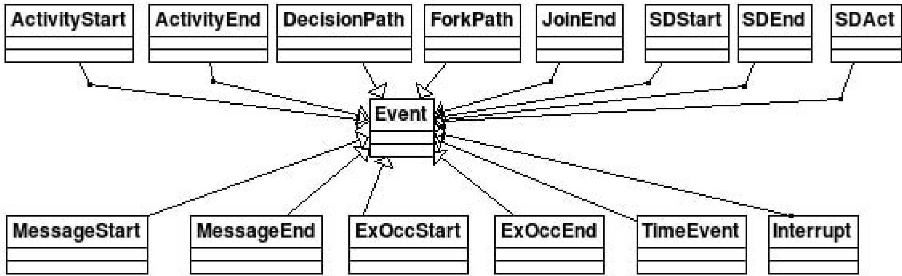

Fig. 6. MADES metamodel: core.events

Diagram *core.clocks* (Figure 7) defines the features of clocks. With respect to the UML/MARTE notion of clocks, for formal verification purposes we deal only with discrete clocks. Clock types are defined in the class diagram of a MADES model. Class *ClockType* has a set of attributes that define specific features of the clocks of that type (e.g., their period). A *Clock* has a *ClockType* and it can be attached to objects, classes, and sequence diagrams. When a clock is attached to an object (resp. class) the intuitive semantics is that the events related to that object (resp. to the objects belonging to the class) will proceed with the tick of this clock. When a clock is attached to a sequence diagram, all the events of the objects inside the sequence diagram will proceed with the tick of this clock (discrepancies between, for example, the clock of an object and that of a sequence diagram in which that object appears can be highlighted and sorted out during the verification phase).

The *core* package is completed with the *core.expressions* diagram (not shown here for the sake of brevity), which defines what is a valid expression. Intuitively,

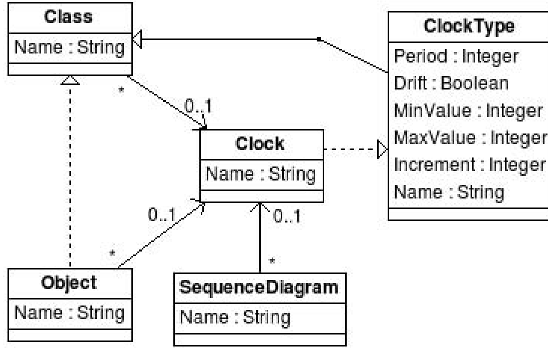

Fig. 7. MADES metamodel: core.clocks

the MADES verification admits three types of expressions: *MathematicalExpression*, *BooleanExpression*, *TimeExpression*. These expressions can be used in *Assignment*s, whose meaning is intuitive. More precisely, *MathematicalExpression*s can be used in *Assignment*s to variables of Integer type. *BooleanExpression*s can be used in *Assignment* to variables of Boolean type, and everywhere a boolean value is admitted (for example in the decision operator of the IOD). *TimeExpression*s can be used to define time constraints between events.

The other packages of the MADES metamodel describe the single diagrams in details. The MADES verification notation does not impose restrictions on the operators of Class diagrams, nor on those of Object diagrams, for which we refer to the UML specification [16]. As shown in Figure 8, a *SequenceDiagram* can contain: *Messages*, *ExecutionOccurrences*, *TimeEvents*, *CombinedFragments* and *StateInvariants*. *RecursiveMessages* are a special type of messages which are used not only for self invocations, but also for assignments to variables of *Integer* and *Boolean* type. Finally, *TimeConstraints* can be attached to sequence diagrams to define relations between the time events defined in the diagram. A *TimeConstraint* is a boolean expression made of *TimeInequalities* that relates two different events with some inequality operator. If the sequence diagram is inside an interruptible region of the IOD of the system, then *TimeConstraints* may also refer to those interrupts (as an example see Figure 12). The diagram of Figure 8 summarizes those concepts and shows the relations between the operators of sequence diagram and the events associated to them. Those events will be translated into temporal logic predicates together with the axioms that define their precise semantics.

*InteractionOverviewDiagram*s (whose class diagram is not shown here for the sake of brevity) can be seen as activity diagrams whose nodes are sequence diagrams. This very simple definition hides a number of details that are needed in order to define a precise semantics. In particular, in the MADES metamodel the *SequenceDiagram*s that are part of an *InteractionOverviewDiagram* have

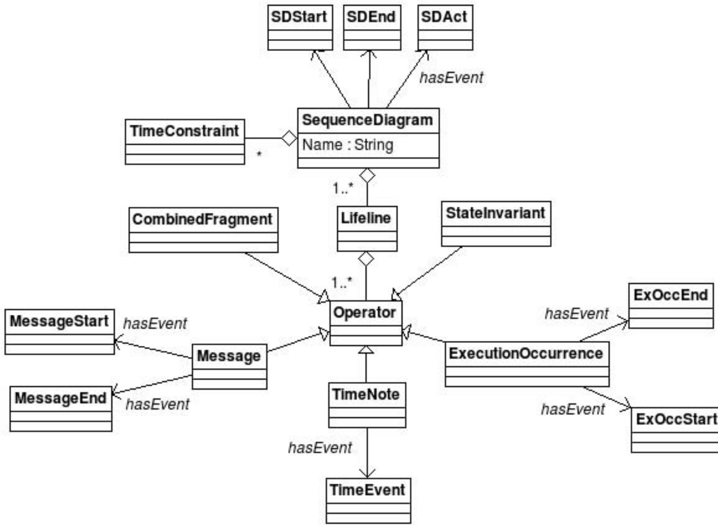

Fig. 8. MADES metamodel: The main elements of sequence diagrams

one incoming and one outgoing flow, and they can be grouped together through *InterruptibleRegions*. The *ExceptionEdge* going out from the *InterruptibleRegion* defines the name of the *Interrupt* associated with that region. This interrupt can be caught by different *InterruptNodes* that go into the *SequenceDiagram* which is responsible to continue the execution after that event. Figure 11 shows an example of *InterruptibleRegion*, which is associated with interrupt *connTimeout* that is caught by the *reconnect* sequence diagram through a suitable *InterruptNode*.

Finally, regarding *StateDiagrams* (whose metamodel is not shown here) we decided to keep the specification as simple as possible. In particular, a *StateDiagram* is a set of *States*. *InitialState* and *FinalState* are two particular kinds of *States*. The *Transition* from one state to another can be triggered by the *Events* of the system (i.e., those defined in Figure 6). Also, a *Transition* can be taken only if its *Guard* is true. The *Guard* is a *BooleanExpression* (whose features are described above). An example of state diagram is shown in Figure 13.

5 From UML to Temporal Logic Formal Semantics

This section presents the MADES semantics through an example system, which includes some desired properties to be verified. The system that is used to illustrate the modeling and verification features of the MADES approach is a simple telephone system. After a brief description of the system, this section shows how some meaningful UML diagrams are translated into their corresponding temporal logic form according to the MADES semantics.

5.1 Telephone System

The telephone system should provide the following features: At startup the system should connect to the remote server and initialize the graphical user interface (GUI). If the telephone is not correctly connected to the server, the GUI will not be shown. The connection is attempted 3 times with a timeout of 10 seconds. When the startup is finished, the system is ready to receive incoming calls and SMSs. Incoming calls may arrive at any instant. Incoming SMSs are checked on the server every 20 seconds by the telephone itself. If no reply is received by the server within 10 seconds, the attempt is not repeated. If the download is not completed within 10 seconds, the download is repeated. If the telephone is idle (e.g. it is not performing any call, nor an SMS composition) and the user presses a number, the number itself is shown on the screen and the telephone waits for the remaining digits until the green button is pressed. If the red button is pressed the system aborts the operation. If the SMS is not sent within 15 seconds, the operation is aborted. If the telephone is idle and the user presses the ok button, then a textual interface is shown to compose the SMS. When the ok button is pressed again the GUI waits for the telephone number and when the ok button is pressed again the SMS is sent to the recipient. SMSs are sent with tokens of 160 characters. The transmission time follows this formula: $trTime = length(SMS)/sigStrength * 10sec$, where $sigStrength$ may be [1..5].

5.2 UML Diagrams and Their Formal Semantics

Figure 9 shows the class diagram of the system. The diagram is not itself translated into temporal logic, though it is used to determine the *alphabet* of the formal model, that is, the actual predicates that appear in the formal model; for example, from the names of the operations the corresponding events are defined. The class diagram also contains some MADES-specific tags (e.g. the «TI» stereotype) that will not be analyzed in this paper. The diagram is also used to define the clock types of the system. For example, class *SMSClockType* defines a type of clock whose period is 20, which will be used for the periodic SMS retrieval. Figure 10 shows the object diagram of the telephone system, which contains the objects that are taken into consideration during the actual verification phase. The number of objects (i.e., instances of classes described in the class diagram) considered in the verification model must be finite, to allow for full automated verification. The role of the object diagram, then, is to precisely define what instances are present in the system model, and their (finite) numbers. For example, the diagram of Figure 10 defines that there are six *TransmissionThread*s. These objects are tagged as being a «set». The semantics that is given in the MADES approach to this stereotype is that, when an operation is invoked on one of the objects of a set, the actual identity of the object is irrelevant, as the objects all behave in the same manner. In the future, we will use such information to optimize the verification phase, but we do not delve into this issue any further in this paper. Figure 10 also shows an instance of clock *SMSClockType*.

The Interaction Overview Diagram of Figure 11 shows the startup of the system. Sequence diagrams are used to group together macro-operations which

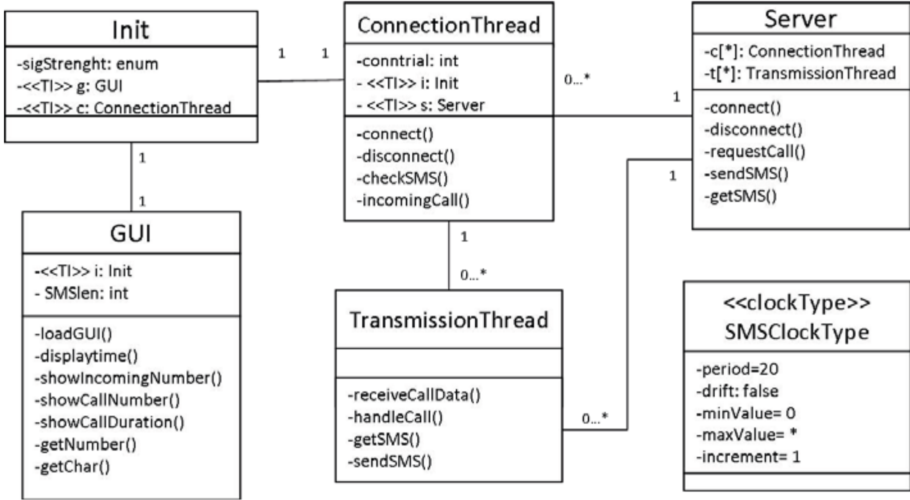

Fig. 9. Telephone System Class Diagram

are then combined together to obtain the complete system specification. Interruptible regions are used to stop the behaviors occurring inside a certain set of sequence diagrams and continue elsewhere. In this particular example we model the fact that while the telephone is performing the connection to the server it may happen that a connection timeout occurs. In that case the *connect* sequence diagram is stopped, and the *reconnect* sequence diagram continues the execution.

The temporal logic semantics of the diagram is generated as described in the following. Each sequence diagram D_x has three events, namely D_xAct, D_xStart, D_xEnd. Each event is translated into one temporal logic predicate, thus predicate D_xAct holds when the diagram is ready to start its execution, predicate D_xStart holds exactly one time unit later, and predicate D_xEnd holds when the diagram terminates. Depending on the operator that precedes diagram D_x the formula that defines D_xAct may change; for reasons of brevity, a presentation of the complete algorithm that manages all different cases is outside the scope of this paper (we refer the interested reader to [2]). In the following we focus on the definitions concerning some of the elements of Figure 11.

When the whole diagram starts (i.e., predicate *ActivityStart* holds), the first sequence diagram (i.e., *init*) is activated. In addition, diagram *loadGUI* is activated when *init* ends, as represented by the fork operator between the two diagrams. These properties are formalized by formulae (1)-(2).

$$InitAct \Leftrightarrow ActivityStart \tag{1}$$

$$loadGUIAct \Leftrightarrow InitEnd \tag{2}$$

$$connectAct \Leftrightarrow InitEnd \vee reconnectEnd \tag{3}$$

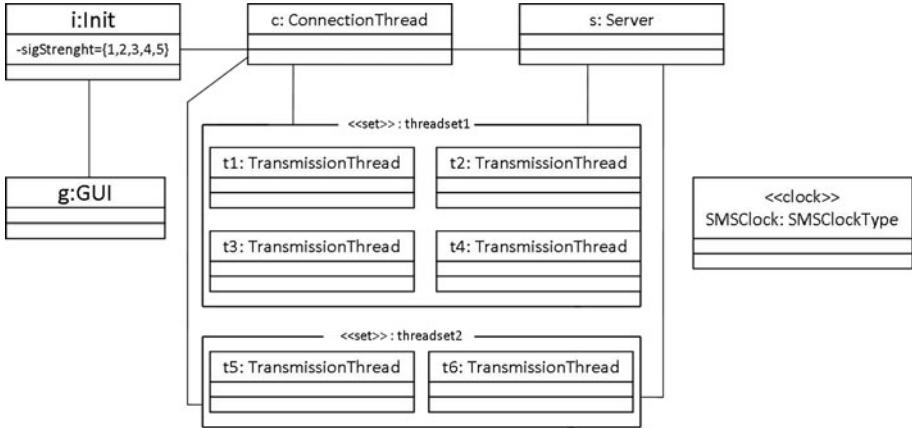

Fig. 10. Telephone System Object Diagram

If we focus on the *connect* sequence diagram of Figure 11, we notice that its activation condition holds at the same time instant in which either diagram *init* ends, or diagram *reconnect* ends. The reason is that *connect* is preceded by a merge operator, thus both paths entering the latter may activate diagram *connect*. One of those paths originates from a fork operator, but this is *transparent* to the semantics. This is all formalized by formula (3).

If, on the other hand, we analyze diagram *run*, its activation condition holds exactly when the nodes preceding the join operator have finished their execution. Formula (5) defines when the join ends (i.e., when the last between *connect* and *loadGui* ends). Formula (4) states that diagram *run* is ready to start exactly in the same time instant in which the join ends.

$$runAct \Leftrightarrow Join1End \tag{4}$$

$$\begin{aligned} Join1End \Leftrightarrow\ & (loadGUIEnd \wedge Since(\neg Join1End, connectEnd)) \vee \\ & (connectEnd \wedge Since(\neg Join1End, loadGUIEnd)) \end{aligned} \tag{5}$$

To conclude this part of the semantics, formulae (6)-(7) define the relations between the activation event and the start event of sequence diagram *connect*. More precisely, $D_x Act$ holds when the enabling conditions are true. Then, if the diagram is enabled and in the next time instant the activity has not ended, $D_x Start$ holds in the next time instant. Similar formulae hold for all other diagrams in the figure (they can be obtained simply by replacing *connect* with the name of the other diagrams, e.g., *loadGUI*).

$$connectAct \wedge \neg Futr(ActivityEnd, 1) \Rightarrow Futr(connectStart, 1) \tag{6}$$

$$connectStart \Rightarrow Past(connectAct, 1) \tag{7}$$

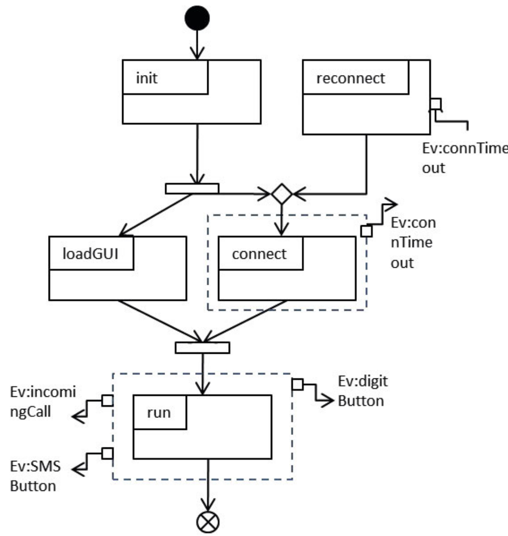

Fig. 11. Telephone System Interaction Overview Diagram

Figure 12 shows the *connect* sequence diagram in detail. The instance c of *ConnectionThread* calls its own procedure *connect()* to start the connection to the server. The time instant in which this connection procedure starts is marked with the time note @$t1$. Inside the *connect()* procedure the object c invokes the *connect()* procedure of the instance s of *Server*. After a while the reply message is received and the sequence diagram ends. According to the system specification the connection is attempted 3 times with a timeout of 10 seconds. To model the connection timeout of 10 seconds we added a *time constraint* to the sequence diagram. A time constraint relates two events with some temporal inequality operator. In that case the semantics is that if the timeout occurs, then it must occur exactly 10 time units after @$t1$ (more precisely, the constraint says that the difference between the time of the timeout and the time of event @$t1$ is exactly 10). Finally, the *alternative combined fragment* is added to specify that the connection is attempted only if variable *conntrial* (which is an attribute of class *ConnectionThread*) is strictly less than 3.

The temporal logic semantics starts from the events that are extracted from the diagram. In this case we have the following events: *connectSDStart*, *connectSDEnd*, which correspond, respectively, to the start and end of the whole diagram; *t1Event*, which represents the time instant in which event @$t1$ occurs; *c.connect1Start*, *c.connect1End*, which correspond to the start and end of recursive message *connect* (i.e., the invocation of operation *connect* being made by c on itself); *c.exOcc1Start*, *c.exOcc1End*, which represent the start and end of the execution occurrence that covers the lifeline of instance c; *s.connect1Start*, *s.connect1End*, which correspond, respectively, to message (i.e., invocation)

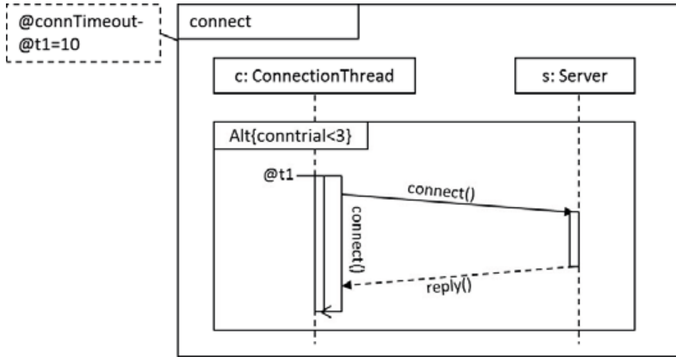

Fig. 12. Telephone System SD connect

connect being sent by object c to object s, and to the message being received by object s; *s.connect1ReplyStart*, *s.connect1ReplyEnd*, which correspond, respectively, to message *reply* (i.e., the reply to the invocation of operation *connect*) being sent by object s, and to the message being received by object c; *s.ExOcc1Start*, *s.ExOcc1End*, which represent the start and end of the execution occurrence that covers the lifeline of instance s. Notice that the events related to the execution occurrences and to the messages are labeled with some index. This is due to the fact that each object can have more than one execution occurrence in the system, and the temporal logic predicates must distinguish between them. The same holds for each method invocation. The events which are placed graphically on the same y-axis of the same lifeline hold on the same time instant, no matter what the other axioms state. This is defined by formulae (8)-(11).

$$t1Event \Leftrightarrow c.connect1Start \wedge c.ExOcc1Start \tag{8}$$

$$s.connect1End \Leftrightarrow s.ExOcc1Start \tag{9}$$

$$s.connect1ReplyStart \Leftrightarrow s.ExOcc1End \tag{10}$$

$$c.connect1End \Leftrightarrow c.ExOcc1End \tag{11}$$

Formulae (12)-(13), instead, enforce the ordering between the different events of the diagram. In particular, the fact that an event Ev_i is followed by another event Ev_j is stated by formula (12). On the other hand, formula (13) entails that we must have an occurrence of Ev_i in the past in order to have Ev_j now. In addition formula (12) defines that, if Ev_i holds now, then we must consider the following possibilities: either it exists in the future a time instant in which Ev_j holds and for all the time instants between Ev_i and Ev_j the sequence diagram is not interrupted, or it exist in the future a time instant in which the diagram is interrupted, and until that time Ev_j does not occur. The SD can be interrupted for two reasons: either because the interaction overview diagram

ends, thus $ActivityEnd$ holds, or because an interrupt associated to that diagram occurs. Formula (13) states similar properties for the past.

$$Ev_i \Rightarrow Until(\neg Ev_i \wedge \neg ActivityEnd \wedge \neg Interrupt_i ... \wedge \neg Interrupt_k, Ev_j) \vee Until(\neg Ev_i \wedge \neg Ev_j, ActivityEnd \vee Interrupt_i ... \vee Interrupt_k) \quad (12)$$

$$Ev_j \Rightarrow Since(\neg Ev_j \wedge \neg Interrupt_i ... \wedge \neg Interrupt_k \wedge \neg ActivityEnd, Ev_i) \quad (13)$$

Considering the diagram of Figure 12 formulae (12)-(13) are instantiated with the following events:

Ev_i	Ev_j
c.connect1Start	s.connect1Start
s.connect1Start	s.connect1End
s.connect1Start	s.connect1ReplyEnd
s.connect1End	s.connect1ReplyStart
s.connect1ReplyEnd	c.connect1End
c.connect1End	connectSDEnd

Moreover in this case we have that the set $Interrupt_i ... Interrupt_k$ is reduced to $connTimeout$.

Because the diagram includes an *Alternative combined fragment* the ordering between *connectSDStart* and the first event of the combined fragment is treated separately. Namely, we specify that *t1Event* follows *connectSDStart* only if *conntrial* is less than 3. This means that $conntrial < 3$ is added to the precondition of formula (12) instantiated with $Ev_i = connectSDStart$ and $Ev_j = c.connect1Start$. Finally, we instantiate again formula (12) with $Ev_i = connectSDStart$ and $Ev_j = connectSDEnd$ with $conntrial >= 3$ included in the sufficient condition as before.

The time constraint attached to the diagram is translated into formula (14) which states that *connTimeout* occurs exactly 10 time units after *t1Event*. Notice that according to formula (8) this is the same time instant in which the *connect* self-recursive message starts. Also notice that if the *connTimeout* occurs when the diagram is finished this does not affect the normal behavior of the sequence diagram according to formulae (12)-(13).

$$connTimeout \Leftrightarrow Past(t1Event, 10) \quad (14)$$

State diagrams complete the picture of what is taken into consideration by the MADES formal verification notation. The *ConnectionThread* state diagram is composed of five states (see Figure 13). The initial state goes into the *start* state. Here the object waits the beginning of the connection procedure. The self-message *c.connect*() moves it from *start* to *connecting*. The *s.connect*() procedure call moves it to the *Waiting* state. If the *connTimeout* interrupt occurs it goes to *reconnect* and then to *connecting* again with *c.connect*(). If *connTimeout* does not occur, the *c.connectEnd* event moves the object into

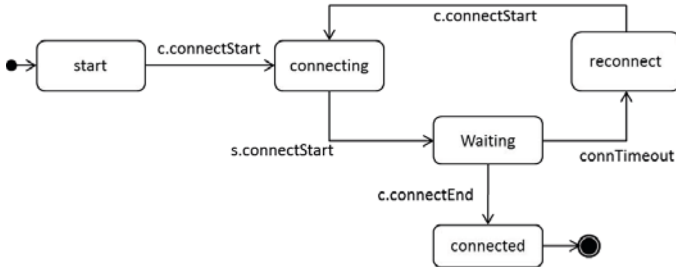

Fig. 13. Telephone System ConnectionThread State Diagram

the *connected* state. Let us focus on the definition of the behavior that makes the state machine enter and exit the *connecting* state. Predicate *c_Connecting* represents when the diagram is in state *connecting*. Then, formulae (15)-(16) state, essentially, that *s.connectStart* is the only event responsible for going out of the *connecting* state. Formula 17, instead, enforces the fact that the states are mutually exclusive. The same kind of formulae hold for the other states of the state diagram. Additional formulae, not shown here for the sake of brevity, define that in the first time instant the machine is in the initial state, then it non-deterministically enters the *start* state because no event is defined in the transition label.

$$c_Connecting \wedge Futr(c_Waiting, 1) \Rightarrow s.connectStart \tag{15}$$

$$c_Connecting \wedge Futr(c_Connecting, 1) \Rightarrow \neg s.connectStart \tag{16}$$

$$c_Connecting \Rightarrow \neg c_Start \wedge \neg c_Waiting \wedge \neg c_Reconnect \wedge \neg c_Connected \tag{17}$$

Finally, the complete formal model Σ that arises from the diagrams describing the system is given by the simple logic conjunction of all formulae. The result is a model that includes all features and constraints defined by the various diagrams, which allows us to formally perform various checks through tools such as the $\mathbb{Z}$ot bounded model/satisfiability checker.

A first kind of verification that can be carried out is a check of the consistency of the model obtained by the combination of the various diagrams. This corresponds to determining whether there exists at least an execution that is compatible with the system model, a task that can be carried out through the $\mathbb{Z}$ot tool simply by checking whether formula Σ is satisfiable or not.

As depicted in Figure 2, the MADES approach also allows users to define temporal properties of interest of the system, to be checked through formal verification techniques. For example, in the case of the telephone system presented in this section, a property we may be interested in is the following:

"The telephone start-up lasts less than 30 seconds"

To express this property formally in a graphical, user-friendly, way, we are currently working on a visual notation that is inspired by sequence diagrams. The

definition of the precise syntax and semantics of such notation is left for future work. Here, we simply note that, since the semantics of the MADES verification notation is given in terms of a formal language, properties of interest can always be expressed using this underlying formalism, in our case the TRIO metric temporal logic. In the case of the temporal property mentioned above, its formalization in the TRIO language could be the following

$$initSDStart \Rightarrow WithinF(runSDStart, 30) \tag{18}$$

where *initSDStart* and *runSDStart* are, respectively, the events associated with the beginning of the *init* and *run* sequence diagrams.

A wide range of temporal properties, of which (18) is just a simple example, can be defined by using the full set of features of the TRIO temporal logic to predicate on the events and attributes of the system modeled through the MADES notation. The properties that users are interested in verifying, however, are often a small subset of those that can be expressed through the full power of the TRIO language. For this reason, the graphical notation we are developing within the MADES project to express properties to be verified will trade the full expressive power of the TRIO language for a higher degree of simplicity and intuitiveness of the representation of properties of interest.

6 Related Work

The vast majority of works that ascribe UML with a formal semantics usually concentrate on individual diagrams. Only few approaches tried to give a semantics that addresses different diagrams. For example, Hansen et al. [13] describe a translation of a subset of executable UML (xUML) into the process algebraic specification language mCRL2. The subset consists of class diagrams, state machines, and an action language that complies with the UML action semantics. This approach does not take into account sequence diagrams and it only concentrates on well-defined fairness and safety properties.

Diethers and Huhn [9] present Vooduu, a tool to automatically verify whether a set of statechart diagrams that model a system satisfies communication and timing constraints given as sequence diagrams. Both types of diagrams are translated into timed automata for the verification. Also Damm et al. [8] define the semantical foundation of a sublanguage of UML that is mostly based on statechart diagrams. The semantics is given in terms of symbolic transition systems, and it mostly addresses the concurrency and communication between objects.

Burmester et al. [5] exploit real-time component diagrams and real-time statechart diagrams to model the static and dynamic parts of a system. These diagrams are formalized in terms of hierarchical timed automata, which allow the authors to run compositional verifications of partial models. Saldhana and Shatz [19] describe a methodology to develop a Petri net of the system. They derive an Object Petri Net Model (OPM) from statechart diagrams connected through collaboration diagrams. The analysis is carried out by exploiting the usual techniques for Petri nets.

As mentioned above there are many more works that focus on the separate formalization of single diagrams. Hammal [12] defines a method for translating statechart diagrams into Interval Timed Petri Nets (ITPN) to run consistency analyses. The ITPN enables the analysis of performance and time properties of complex systems. Störrle [20] investigates the alignment activity diagrams to Petri nets. It provides a mapping of the basic elements of activity diagrams onto Petri nets and discusses the problems that arise from this translation. Eshuis [10] proposes two translations from activity diagrams to the input language of NuSMV, a well-known symbolic model checker. Both translations map activity diagrams into finite state machines and are inspired by existing semantics for statechart diagrams. Finally, Cengarle and Knapp [6] investigate interaction diagrams and provide an operational semantics for them, while Tebibel [4] uses hierarchical colored Petri nets to define a formal semantics for interaction overview diagrams.

7 Conclusions

This paper builds on the need for formal UML dialects for the design and validation of timed systems. It starts from the MADES modeling notation, which is a particular extension to UML that borrows concepts from SysML and MARTE, and it proposes a formal semantics for a verification-oriented version of the language. The verification notation can be seen as an "abstract" notation for the complete design notation. It filters out irrelevant elements and ascribes a formal semantics to the other ones. The formal semantics covers a wide range of UML diagrams and concentrates on time-related aspects. The formal semantics and the verification tools will be used in the MADES project for the early verification of embedded systems.

The paper outlines the formal semantics for the verification notation based on the TRIO temporal logic. The definition is provided in a modular and methodological way to let the reader understand how the different pieces fit together. Finer-grained improvements, optimizations, and tailoring are part of our ongoing work. The complete verification tool suite implementing the approach described in this paper will be tested for usability by the industrial partners of the MADES project.

Acknowledgments. This research was supported by the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement n. 248864 (MADES), and by the Programme IDEAS-ERC, Project 227977-SMScom.

References

1. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: D3.1 domain-specific and user-centred verification. Technical report, MADES Consortium (2010)
2. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: D3.3 formal dynamic semantics of the modelling notation. Technical report, MADES Consortium (2011)

3. Bersani, M.M., Frigeri, A., Pradella, M., Rossi, M., Morzenti, A., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: Proc. of the Int. Symp. on Temporal Representation and Reasoning (TIME), pp. 43–50 (2010)
4. Bouabana-Tebibel, T.: Semantics of the interaction overview diagram. In: Proc. of the IEEE Int. Conf. on Information Reuse Integration (IRI), pp. 278–283 (2009)
5. 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: Proc. of the 27th Int. Conf. on Soft. Eng., ICSE 2005, pp. 670–671 (2005)
6. Cengarle, M.V., Knapp, A.: Operational semantics of UML 2.0 interactions. Technical Report TUM-I0505, Technische Universität Mnchen (2005)
7. Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., Morzenti, A.: From formal models to formally-based methods: an industrial experience. ACM TOSEM 8(1), 79–113 (1999)
8. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time uml semantics for concurrency and communication in safety-critical applications. Sci. Comput. Program. 55, 81–115 (2005)
9. Diethers, K., Huhn, M.: Vooduu: Verification of object-oriented designs using uppaal. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 139–143. Springer, Heidelberg (2004)
10. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
11. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. IEEE Trans. Software Eng. 30(7), 437–447 (2004)
12. Hammal, Y.: A formal semantics of uml statecharts by means of timed petri nets. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 38–52. Springer, Heidelberg (2005)
13. Hansen, H., Ketema, J., Luttik, B., Mousavi, M., van de Pol, J.: Towards model checking executable uml specifications in mcrl2. Innovations in Systems and Software Engineering 6, 83–90 (2010)
14. Object Management Group. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. Technical report, OMG, formal/2009-11-02 (2009)
15. Object Management Group. OMG Systems Modeling Language (OMG SysML). Technical report, OMG, formal/2010-06-01 (2010)
16. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Technical report, OMG, formal/2010-05-05 (2010)
17. Pradella, M., Morzenti, A., San Pietro, P.: The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In: Proceedings of ESEC/SIGSOFT FSE, pp. 312–320 (2007)
18. Pradella, M., Rossi, M., Mandrioli, D.: ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 381–395. Springer, Heidelberg (2005)
19. Saldhana, J.A., Shatz, S.M.: Uml diagrams to object petri net models: An approach for modeling and analysis. In: Proc. of SEKE 2000, pp. 103–110 (2000)
20. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Software Engineering. Lec. Not. in Inf, vol. 64, pp. 117–128 (2005)