A Logic-based Approach for the Verification of UML Timed Models

LUCIANO BARESI, DEIB - Politecnico di Milano
ANGELO MORZENTI, DEIB - Politecnico di Milano
ALFREDO MOTTA, DEIB - Politecnico di Milano
MOHAMMAD MEHDI POURHASHEM K., DEIB - Politecnico di Milano
MATTEO ROSSI, DEIB - Politecnico di Milano

This paper presents a novel technique to formally verify models of real-time systems captured through a set of heterogeneous UML diagrams. The technique is based on the following key elements: (i) a subset of UML diagrams, called C-UML, which allows designers to describe the components of the system and their behavior through several kinds of diagrams (e.g., state machine diagrams, sequence diagrams, activity diagrams, interaction overview diagrams), and stereotypes taken from the UML MARTE profile; (ii) a formal semantics of C-UML diagrams, defined through formulae of the metric temporal logic TRIO; (iii) a tool, called Corretto, which implements the aforementioned semantics and allows users to carry out formal verification tasks on modeled systems. We validate the feasibility of our approach through a set of different case studies, taken from both the academic and the industrial domain.

CCS Concepts: •**Software and its engineering** → **Model-driven software engineering; Model checking; Unified Modeling Language (UML);**

Additional Key Words and Phrases: Formal Verification, Metric Temporal Logic, Formal semantics, Timed systems

ACM Reference Format:
Luciano Baresi, Angelo Morzenti, Alfredo Motta, Mohammad Mehdi Pourhashem K., and Matteo Rossi, 20XX. A Logic-based Approach for the Verification of UML Timed Models. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 51 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

UML is a well-known and widely-used design notation. As such, it has been often criticized [Erickson and Siau 2007; Dori 2002] because it includes too many diagrams and symbols and because of the complexity of managing structural and behavioral views spread over many different diagrams. The semantics provided by the UML specification [OMG 2011] is given through textual descriptions and it is often ambiguous; a UML specification consists of a collection of loosely coupled models (classes, use cases, collaborations, activities, etc.) that are tied together by few and semantically weak rules [Glinz 2000]. The introduction of domain-specific extensions (profiles, according to the OMG jargon) further complicates the integration of standard and specific concepts.

These problems have often hampered the use of UML as a *rigorous* specification notation. There have been many different attempts to ascribe the notation with a precise, formally defined semantics, but the heterogeneity of the language has often led

Author's addresses: Luciano Baresi, Angelo Morzenti, Alfredo Motta, Mohammad Mehdi Pourhashem K., and Matteo Rossi, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano. (Current address) Politecnico di Milano, piazza L. da Vinci, 32 - 20133 Milano (Italy).

DOI: http://dx.doi.org/10.1145/0000000.0000000

the authors to only concentrate on some diagram types (e.g., state and collaboration diagrams), while neglecting the key characteristics of UML, that is, the rich set of diagram types and the freedom with which the designer can model a system.

While the formalization of single diagram types has been carried out in several ways [Störrle 2003; Eshuis 2006; Burmester et al. 2004], the integration into a single coherent framework of a significant number of diagram types is a distinguished characteristic of C-UML (Corretto UML), the extensible, formal, timed, subset of UML proposed in this article. Even fUML [OMG 2016] (Foundational UML), the official executable subset of UML, is not well-suited for the formal verification of defined models (as discussed in Section 2).

C-UML proposes a comprehensive framework that accommodates class, object, state machine, activity, sequence, and interaction overview diagrams. C-UML focuses on the interdependencies among the different diagrams by means of a suitable set of events shared among the different diagram types. In addition, it borrows a notion of *time* from MARTE (the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems), which makes it particularly suited for the specification of timed systems, whose correctness depends on their ability to ensure certain temporal relations concerning their activities and events. The Corretto Property Language (CPL) allows the user to state the properties to be verified for the system in an OCL-oriented way.

The formal semantics of C-UML is based on the TRIO [Ciapessoni et al. 1999] metric temporal logic, which ensures the flexibility and composability required to specify the semantics of a complex notation. The formalization presented in this paper aims at width instead of depth. We want to integrate different types of diagrams and reason on their interdependencies, instead of digging down and addressing any single element and semantic variation of a specific diagram type. The modularization and compositionality of the presented approach allow for the addition of many more details. For example, the interested reader can refer to [Baresi et al. 2014] for an in-depth treatment of the syntactic elements and semantic variations of Sequence Diagrams.

C-UML is supported by Corretto [Motta et al. 2017], a prototype verification tool that groups the above mentioned features inside a plugin for Papyrus,[1] a well-known open source UML-based engineering tool. *Corretto* translates C-UML models into the corresponding TRIO models, which are then verified through the $\mathbb{Z}$ot bounded model/satisfiability checker [Pradella et al. 2013]. A backward traceability mechanism allows the user to interpret the analysis results—provided as counterexamples generated by the model checker—in terms of elements of the analyzed UML models. This is similar to the features provided by tools for the controlled execution of programs, where lower-level execution actions are rendered in terms of a high-level programming language.

This article extends our previous work [Baresi et al. 2015; Baresi et al. 2012] in different ways. It provides: (i) a complete description of C-UML, which has been extended in several ways with respect to the aforementioned previous works, most notably by adding support for Activity Diagrams and by widening the range of available action types; (ii) a detailed presentation of the metric temporal logic semantics, which has been revised and extended to cover the new elements; (iii) a thorough experimental assessment of the capabilities of the tool; and (iv) an in-depth, exhaustive analysis of the state of the art. As for this last aspect, the survey categorizes the different approaches in terms of supported UML diagrams, type of verification performed, and availability of tool support. The comparative analysis highlights both the features and limitations of the various approaches, and frames C-UML as a significant step forward towards associating a formal semantics with the whole set of UML diagrams.

[1] www.eclipse.org/papyrus

The experimental assessment was conducted on five different models that come from heterogeneous sources, including case studies provided by industrial partners, instances from the literature on verification tools for timed UML, and classical academic examples commonly used to evaluate the scalability of different verification approaches.

The rest of the article is organized as follows. Section 2 frames the contribution with respect to the other UML-related specifications. Section 3 presents C-UML, our enriched subset of UML diagrams. Section 4 describes the metric temporal logic semantics we defined. Section 5 introduces Corretto. Section 6 summarizes the experiments we conducted to assess the validity of the approach. Section 7 surveys the state of the art and Section 8 concludes the paper. Appendix A gives a short introduction to TRIO, the formal language exploited in the paper. In addition, a number of resources are available from the Corretto repository [Motta et al. 2017], including the source code of the tool and of the examples presented in this paper, and the meta-model of the C-UML notation.

2. C-UML IN CONTEXT

The UML ecosystem has evolved significantly over the years and C-UML must be framed within it. Besides UML, C-UML must also be related to both fUML (Foundational UML [OMG 2016]) and Alf (Action Language for Foundational UML [OMG 2013]). fUML contributes a precise execution semantics for a subset of UML: composite structures, classes, and activities (behavior) [Tatibouët et al. 2014]. Alf defines a textual action language to specify complex execution behaviors; the resulting specifications can then be compiled to fUML. fUML borrows some of the metamodels defined in the UML specification [OMG 2011] as its abstract syntax, and augments them with well-formed OCL assertions for constraining some modeling concepts. fUML models can exploit both the usual UML graphical symbols and textual Alf statements. For example, one may mix "classic" UML shapes for classes and Alf specifications to define their behaviors.

The specification of the fUML's execution model remains *generic* on purpose. It leaves some key semantic elements unconstrained and defines explicit semantic variation points. Among them, the simulation of time is left open: both discrete and continuous time models can be adopted, but an extension to the execution model is required to support them [Benyahia et al. 2010].

The execution model is defined in an operational manner and specifies a virtual machine that interprets fUML models. The specification of the operations is written in Java, used as concrete (textual) representation of the UML activity models, which in turn define the behavioral modeling capabilities included in the Base UML (or bUML). bUML is the subset of fUML used to write the execution model.

The base semantics—that is, the semantics of bUML—is expressed as axioms of first order logic, rendered in Common Logic Interchange Format (CLIF), and in the Process Specification Language (PSL), that is a foundational axiomatization of processes. It only specifies when particular executions conform to a model defined in bUML, and does not define a virtual machine to execute models directly.

This is the context in which C-UML must be embedded. Figure 1 both summarizes the description above and identifies a role for C-UML. Everything starts from UML. While Alf complements it and fUML formalizes it—and then Alf itself is formalized through fUML—C-UML only supports a subset of UML, extends it with some key features, and its semantics is defined in TRIO. As already said, C-UML emphasizes the management and simulation of time, and thus we selected a proper temporal language for specifying its behavior. Similarly one may also think of a timed version of fUML, and use TRIO as underlying formal kernel, but this is out of the scope of the paper.

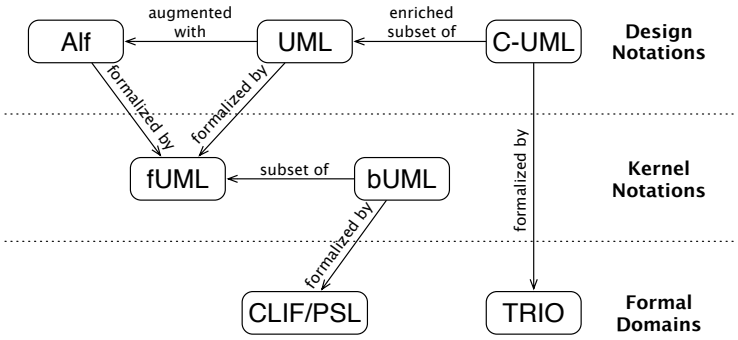


Fig. 1: Relationships among C-UML and the other UML specifications.

Moreover, while the "standard" UML languages are mainly design means, sometimes used for simulation and code generation [Ciccozzi 2016], C-UML was conceived mainly for the formal *verification* of designed models, which seems to be a key distinctive feature in the sketched UML ecosystem.

While other proposals aim at turning UML design models into an executable notation, the formalization approach proposed in this paper is functional to the formal *verification* of design models—and not of the corresponding code, even when this is automatically generated. We are interested in identifying the finite, complete execution domain of the modeled system and searching it against the properties of interest. We are not interested in defining a virtual machine that creates objects and calls methods on them. In our solution, objects (i.e., instances of classes) are only identified through a set of variables. Similarly, methods only assert on the possible evolutions of related variables. Features more related to the execution of code corresponding to UML models, like object creation and dynamic binding, can also be considered, but the domain must always be finite. This means that object creation can be mimicked by identifying a set of objects that are then switched on and off accordingly. Similarly, dynamic binding would be dealt with by considering all the possible alternatives while checking the model.

To conclude, MARTE also introduces a time model able to cope with dense, discrete, metric, and logical times [MARTE 2011], but the languages that it defines to express constraints, such as VSL (Value Specification Language) and CCSL (Clock Constraint Specification Language), either lack a formal semantics, or are limited in terms of the constraints they can express—for example, CCSL cannot express liveness properties [Gascon et al. 2011].

MARTE introduces mechanisms to capture Non-Functional Properties (NFPs) through VSL, which lets users express both qualitative and quantitative constraints on "event observations", whose nature however is not formally defined (an observation can be associated with any named element). C-UML essentially uses a subset of VSL to express constraints among elements of UML models, by precisely identifying—through a MARTE-inspired, but specialized, notation—the set of observed elements and by associating each of them with a predicate formalizing its occurrences and a suitable semantics. Also, CPL includes all TRIO temporal operators, through which one can express, e.g., liveness constraints; this, in turn, allows for the verification of properties under user-defined liveness assumptions concerning UML elements (e.g., signals). In addition, MARTE allows users to introduce clocks, to associate them with elements, and to express dependencies among them through CCSL. Formalizations of CCSL have been defined in literature [Mallet 2008]. [André 2009] formalizes a subset

of CCSL, called Kernel CCSL, and [Zhang et al. 2016] defines a technique to formally analyze stated CCSL constraints. CCSL expresses constraints only as dependencies between clocks, which can make expressing properties cumbersome—potentially, new clocks might have to be introduced for each constraint. Finally, we remark that, although this is out of the scope of the present work, the TRIO metric temporal logic which provides the formal foundation to our whole approach could also be used to provide a formalization of CCSL constraints.

3. C-UML

Corretto UML[2] (C-UML) was designed to be a simplified—and a special-purpose—version of UML for modeling complex time-dependent embedded systems. We conceived it as a verifiable subset of UML, to which we added some key elements of MARTE to address time-related aspects.

More specifically, C-UML comprises:[3] Class Diagrams, Object Diagrams, State Machine Diagrams, Sequence Diagrams, Activity Diagrams, and Interaction Overview Diagrams.[4] C-UML also offers a simple, OCL-like language, Corretto Property Language (CPL), that helps state the properties of interest during verification. A valid C-UML model must comprise at least a Class Diagram, while all the behavioral views are optional.

Class Diagrams provide the static definitions of the elements in a system. Every class should have at least one Instance Specification (object) instantiating the class in an Object Diagram. The number of Instance Specifications must be finite in order to run the formal verification. Class diagrams also allow for clock types (through stereotype «clockType») and timed events («timedEvent»), both borrowed from MARTE. The former can be used to constrain the timed behavior of components [MARTE 2011; André et al. 2007], while the latter can be used to associate clocks with events of behavioral diagrams—e.g., messages in Sequence Diagrams—to constrain them to be periodic.

State Machine Diagrams can be used to describe the behavior of significant classes in the system, that is, all their instances will behave as stated by these diagrams. Each class may be associated with one or more State Machine Diagrams, which would then run in parallel. A single object will then be into multiple states at a given time. Each of these states belongs to one of the state machines assigned to that object. Similarly, Activity Diagrams capture the dynamic aspects of objects by showing the flow of control among their activities, which emphasizes the sequences, conditions and parallelism among them. Each class can have a set of Activity Diagrams.

Sequence Diagrams describe partial behaviors of the system. They identify the messages exchanged between the instances of the classes defined in Class Diagrams. These messages in turn should be instances of the operations defined in the classes the objects belong to. Interaction Overview Diagrams constitute a high-level structuring mechanism that is used to compose Sequence Diagrams through standard operators such as sequence, iteration, concurrency, and choice [Baresi et al. 2011].

In addition to these standard UML diagrams, we add the notion of **time constraints** to predicate on the time dimension of the system. Time constraints can be introduced in Sequence Diagrams through special-purpose comments (stereotype «time-

[2]The notation was defined within the EU project MADES: www.mades-project.org

[3]In this paper we use the term "diagrams" (e.g., Class Diagrams) when referring to the various parts of a C-UML model, because this is the terminology most users are probably familiar with. However, what matters are the entities in the model (classes, instances, activities, interactions, and so on), rather than their actual graphical representation in diagrams. Diagrams are not taken into account during analysis.

[4]A document containing the C-UML meta-model is available on the Corretto repository [Motta et al. 2017].

Constraint»), in State Machine Diagrams as triggers or guards of a transition, and in Activity and Interaction Overview Diagrams as guards of control flows. They always have the format:

$$ev2 - ev1 \sim K$$

where `ev1` and `ev2` are events occurring in the model, $ev2 - ev1$ is the number of time instants between the occurrences of $ev1$ and $ev2$ (i.e., the difference of their timestamps), $\sim$ is a relation in the set $\{<, \leq, ==, !=, \geq, >\}$, and K is a numeric (integer) constant.

C-UML addresses the interdependencies among the different diagrams of a complete specification by means of **shared events**, such as invocations of operations and clock ticks. To refer to an event (i.e., a timed observation) we use the MARTE-inspired [Demathieu et al. 2008] notation $@id.event$, where id is the identifier of the element with which the event is associated. The events of interest are:

— `@id.start`, `@id.end`, `@id.stop`, where `id` is the name of a Sequence Diagram and the three events refer to the time instants at which the Sequence Diagram starts, ends, and is stopped. A Sequence Diagram ends when it successfully completes all the interactions shown in the diagram. If the interactions are not completed successfully (this can happen when they are part of an interruptible region—see Section 4.5—which is interrupted by a signal) we say that the Sequence Diagram has been stopped.
— `@id.send`, `@id.receive`, where `id` is the name of a message defined in a Sequence Diagram, and the two events refer to the time instants at which the message is sent and received.
— `@id.begin`, `@id.terminate`, where `id` is the name of an execution specification[5] in a Sequence Diagram, and the two events refer to the time instants at which the execution specification starts and ends.
— `@id.enter`, `@id.exit`, where `id` is the name of a state of a State Machine Diagram and the two events refer to the time instants at which an object enters and exits the state.
— `@id.call`, `@id.reply`, where `id` is the name of an operation defined in the Class Diagram and `call` (resp., `reply`) occurs when the operation is invoked (resp., returns). For example, given an operation defined in a Class Diagram, when a Sequence Diagram sends a message to an operation `op`, event `@op.call` occurs. If the user models an execution specification for the invoked operation and then a return message, `@op.reply` occurs when the reply message is received by the client. If no execution specifications, and reply messages, are provided the (implicit) assumption is that the execution is instantaneous. In addition, if the interaction is synchronous, no events can happen between `@op.call` and `@op.reply` on the caller's lifeline (if the sequence diagram has not been stopped in between).
— `@id.tick`, where `id` is the name of a clock instance defined in a Class Diagram and the tick event occurs every T time units, where T is the period associated with the clock.
— `@id.sig`, where `id` is the name of a signal defined in a Class Diagram and it may be triggered (`sig`) because (i) it is associated with an action of a transition in a State Machine Diagram; (ii) it is associated with a *send signal* action in an Activity or Interaction Overview Diagram (see below); or (iii) non-deterministically, when none of the previous cases holds, i.e., when the signal is left free.
— `@id.adstart`, `@id.adend`, where `id` is the name of an Activity Diagram and `adstart` (resp., `adend`) refers to the instants when the diagram starts (resp., ends) its execution.

[5] As defined in [OMG 2011], execution specifications are represented as thin rectangles on lifelines.

— `@id.iodstart`, `@id.iodend`, where `id` is the name of an Interaction Overview Diagram and the two events refer to the time instants at which it starts and ends its execution.
— `@now` is a special event that refers to the current time instant.

C-UML supports both local and static (i.e., global) **attributes**, and if they have default values, they are initialized accordingly. Attributes maintain their values except in case of assignments. However, it is sometimes desirable, for example with cyber-physical systems, to capture also non-deterministic behaviors to describe uncontrollable choices made by the application environment. Thus, when an attribute is tagged with stereotype «free», its value changes non-deterministically, without an explicit assignment (see Figure 4 and accompanying description for an example).

Actions are the fundamental units for specifying system behaviors in C-UML. They define elementary behaviors, comparable to atomic statements in traditional programming languages. Actions can be introduced through action nodes in Activity and Interaction Overview Diagrams, in the "action" part of transitions in State Machine Diagrams, and in messages of Sequence Diagrams. C-UML supports five kinds of actions: *assignment*, *call operation*, *send signal*, *accept event*, and *init sequence diagram*. Figure 2 shows which action types—from zero to many—are supported by the different diagrams.

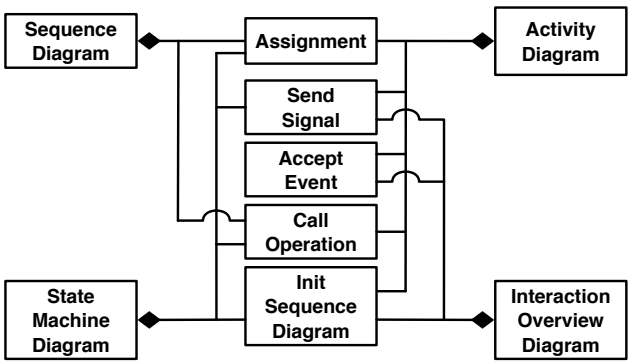


Fig. 2: Diagram and Action types.

Assignments can refer to and modify the values of attributes of classes, but they have a slightly different syntax than in usual programming languages to allow for greater flexibility in combining values taken at different instants of time. In fact, given an attribute `A`, simply `A` refers to its current value, `<Y>A` to its value at the previous time instant, and `<X>A` at then next one[6]. Then, to increment the value of attribute `A`, the assignment should be written as `A = <Y>A + 1`, to indicate that the current value of `A` is equal to the one at the previous time instant, plus one.

Actions *call operation*, *accept event*, *send signal* and *init sequence diagram* all exploit the notation introduced above for shared events. For example, the call operation `@OP.call` can appear in the action of a transition of a State Machine Diagram[7]. Similarly, the action of a transition can include an init sequence diagram `@SD.start`, where

[6]The use of `Y` and `X` is borrowed straightforwardly from LTL to refer to a value at the previous and next time instants, respectively.
[7]If an object can invoke the same operation on different objects, C-UML allows the user to disambiguate the target object of an operation call by means of *associationEnd.op.call*. This is exemplified in the State Machine Diagram of Figure 22 by the transition between states Main and Won.

SD is the id of a Sequence Diagram. This custom syntax has been introduced to ease the burden on the modeler who needs to call operations and initialize sequence diagrams within UML diagrams: this is supported by the standard, but realized in different ways in different tools.

In C-UML, the properties to be checked are expressed through the **Corretto Property Language** (CPL), which is inspired by OCL and has a straightforward translation into the TRIO metric temporal logic [Ciapessoni et al. 1999] (see Appendix A). The properties to be checked are introduced through a special-purpose constraint associated with the main C-UML model package, which is tagged with stereotype «property» and contains a sequence of declarations in CPL. The syntax of the CPL is defined by the following grammar:

$$
\begin{array}{lll}
CPLFm & ::= & \textbf{!}\,CPLFm \mid CPLFm\ Pop\ CPLFm \mid \\
 & & \textbf{Time.}Temp_1\textbf{(}CPLFm\textbf{)} \mid \textbf{Time.}Temp_2\textbf{(}CPLFm, CPLFm\textbf{)} \mid \\
 & & \textbf{Time.}TempInt\textbf{(}CPLFm, Integer\textbf{)} \mid \\
 & & signalID \mid InState \mid objID\hat{}opID\textbf{()} \mid ArithBool \\
InState & ::= & objID\textbf{.in(}State\textbf{)} \\
State & ::= & objID\textbf{.getState(}STDID, UMLStateID\textbf{)} \\
Pop & ::= & \textbf{\&\&} \mid \textbf{||} \mid \textbf{=>} \mid \textbf{<=>} \\
Temp_1 & ::= & \textbf{alw} \mid \textbf{som} \mid \textbf{alwF} \mid \textbf{somF} \mid \textbf{alwP} \mid \textbf{somP} \\
Temp_2 & ::= & \textbf{until} \mid \textbf{since} \mid \textbf{release} \mid \textbf{trigger} \\
TempInt & ::= & \textbf{futr} \mid \textbf{past} \mid \textbf{withinF} \mid \textbf{withinP} \mid \textbf{lasts} \mid \textbf{lasted} \\
ArithBool & ::= & ArithTerm(\textbf{>}|\textbf{>=}|\textbf{<}|\textbf{<=}|\textbf{==}|\textbf{!=})ArithTerm \\
ArithTerm & ::= & constant \mid ArithVar \mid ArithTerm(\textbf{+}|\textbf{-}|\textbf{*}|\textbf{/})ArithTerm \mid \\
 & & \textbf{Futr(}ArithTerm, Integer\textbf{)} \mid \textbf{Past(}ArithTerm, Integer\textbf{)}) \\
ArithVar & ::= & objID.attributeID \mid objID :: opID.parameterID \mid \\
 & & sdID\textbf{.getParameter(}parameterID\textbf{)}
\end{array}
$$

According to the grammar, a formula ($CPLFm$) of the language can refer to the occurrence of a signal ($signalID$), a state of an object being active ($InState$), or an operation invocation on an object ($objID\hat{}opID()$). For example, `obj1.in(obj1.getState(mainSTD, idle))` describes that object obj1 is in state idle of State Machine Diagram mainSTD, and `obj1^update()` refers to the invocation of operation update on obj1.

In addition to the usual propositional operators (Pop), CPL includes three types of temporal operators, characterized by prefix `Time` and identified as $Temp_1$, $Temp_2$ and $TempInt$ in the grammar. $Temp_1$ operators take a single formula as argument: for example, `Time.alw(! failureSignal)` states that signal failureSignal is never issued. $Temp_2$ operators have two formulae as arguments: for instance, `Time.until(! obj1.in(obj1.getState(mainSTD, shutdown)), failureSignal)` describes that obj1 will not be in state shutdown until failureSignal is issued. Finally, $TempInt$ operators take a formula and an integer as arguments: for example, `Time.lasts(obj1.in(obj1.getState(mainSTD, warning)), 5)` states that obj1 will be in state warning for at least the next 5 time units.

CPL also allows for arithmetic expressions through which users can express properties of attributes and parameters. Arithmetic formulae compare two terms, which can be the results of arithmetic operations, or single arithmetic variables. Arithmetic variables can be attributes and operation parameters of objects, or parameters of Sequence Diagrams. For example, `Time.alw(! (sensor.temperature - Past(sensor.temperature, 1) > 5))` states that there will never be an increase in the temperature measured by the sensor exceeding 5 degrees.

Figure 3 shows how CPL supports declarations to make formulae more readable. Lines 1 and 2 state that `inStateOn` and `isHeating` hold, respectively, when object heatingSys is in state on of State Machine Diagram mainSTD and when attribute temperature of object sensor is greater than its value at the previous time instant. Line 3 defines a surge, that is, an increase in temperature grater than 5 degrees, and Line 4 the property of interest, that is, that the temperature detected by the sensor always increases as long as the heating system is on, and a surge never occurs. Line 5 tells Corretto to check the property, which means verifying whether all executions satisfy the property. If one used `execute()` instead of `verify()`, Corretto would only return an execution trace of the modeled system.

```
1 inStateOn = heatingSys.in(heatingSys.getState(mainSTD, on))
2 isHeating = sensor.temperature > Past(sensor.temperature, 1)
3 surge = (sensor.temperature - Past(sensor.temperature, 1)) > 5
4 property = Time.alw(inStateOn => isHeating) && Time.alw(! surge)
5 Corretto.verify(property)
```

Fig. 3: An example property in CPL.

4. SEMANTICS

The semantics of C-UML is defined through the TRIO metric temporal logic [Ciapessoni et al. 1999], which is briefly introduced in Appendix A. Each C-UML diagram is translated into a set of predicates and variables, plus a set of TRIO formulae; the predicates and variables encode the diagram elements, while the axioms define their semantics by stating the constraints that rule their behavior.

The logic-based approach allows us to break down the semantics of UML diagrams into small pieces, captured by different groups of axioms, and thus to dominate its complexity.

More formally, let us define $D = \{CD \cup OD \cup AD \cup IOD \cup SM \cup SD\}$ as the set of diagrams included in a C-UML model (where CD is the set of Class Diagrams, OD is the set of Object Diagrams, AD is the set of Activity Diagrams, IOD is the set of Interaction Overview Diagrams, SM is the set of State Machine Diagrams, and SD is the set of Sequence Diagrams), $\mathcal{P}$ as the set of predicates and variables that encode their elements, and $\mathcal{A}_\mathcal{P}$ as the set of axioms that constrain the behavior of the elements in $\mathcal{P}$. The semantics is a triple $< \Gamma, \Delta, \Theta >$, where $\Gamma : D \rightarrow 2^\mathcal{P}$ is a function that given a diagram returns the set of predicates and variables assigned to it, $\Delta : D \rightarrow 2^{\mathcal{A}_\mathcal{P}}$ is a function that given a diagram returns the set of axioms that constrain its behavior, and $\Theta : D \times D \rightarrow 2^{\mathcal{A}_\mathcal{P}}$ is a function that, given two diagrams, returns the set of axioms that formalize their *combined behavior*, as determined by the items that they share, thus modeling features such as communication, coordination, and any kind of interaction among diagrams.

The semantics $\mathcal{S}em$ of a C-UML model is therefore built as follows.

$$\mathcal{S}em = \left(\bigcup_{d \in D} \Delta(d) \right) \cup \left(\bigcup_{d_1, d_2 \in D, d_1 \neq d_2} \Theta(d_1, d_2) \right) \quad (1)$$

This formalization/verification approach emphasizes and facilitates:

— *Decoupling.* The semantics is decoupled from the predicates that represent the elements of C-UML models. This means that one can change the semantics while keeping the translation from UML to the predicates unchanged. For example, if we

wanted to change the semantics of time associated with clocks, we do not need to alter the predicates, but only the axioms associated with them. This also gives the opportunity to experiment and evaluate different semantics for the same model.

— *Extensibility.* The logic-based formalization is easy to extend. Adding a new diagram type entails defining the predicates that represent its elements, and their associated axioms. If the new diagram type shares some predicates with the already existing ones, then the coordination between diagrams is obtained seamlessly through the formulae that predicate on shared elements.

— *Composability.* Introducing new details may result in specifications that become too big to be analyzed automatically. In such scenarios it is common practice to select an interesting subset of the model that focuses on a particular feature to simplify the analysis. To this end, our approach supports the analysis of partial models by simply avoiding the translation of the diagrams the user is not interested in.

The rest of the section uses a running example to illustrate the proposed semantics.

4.1. Car Collision Avoidance System

To explain the formal semantics of the various diagrams we use an example, Car Collision Avoidance System (CCAS for short). Let us remark that the CCAS model has been created with the purpose of showing a wide range of features of C-UML in an integrated, coherent way; a more economical use of C-UML diagrams to describe the same system is possible.

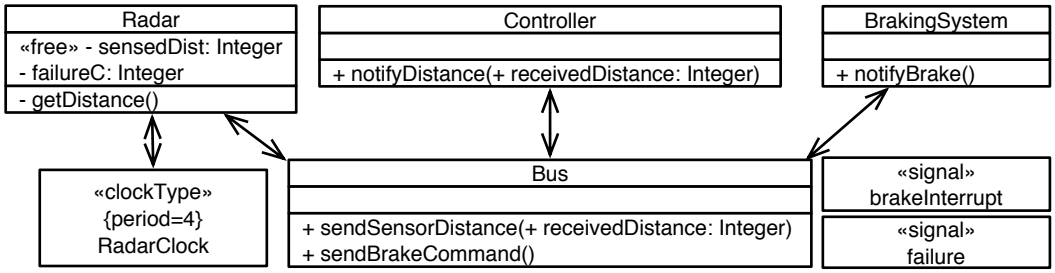


Fig. 4: CCAS: Class Diagram.

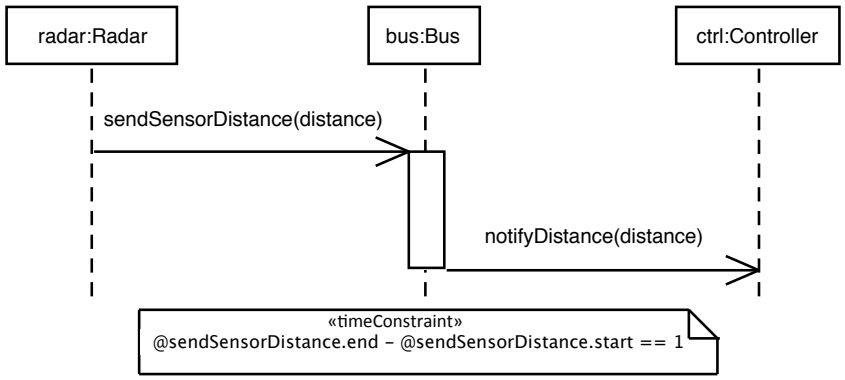


Fig. 5: CCAS: Sequence Diagram sendSensorDistance.

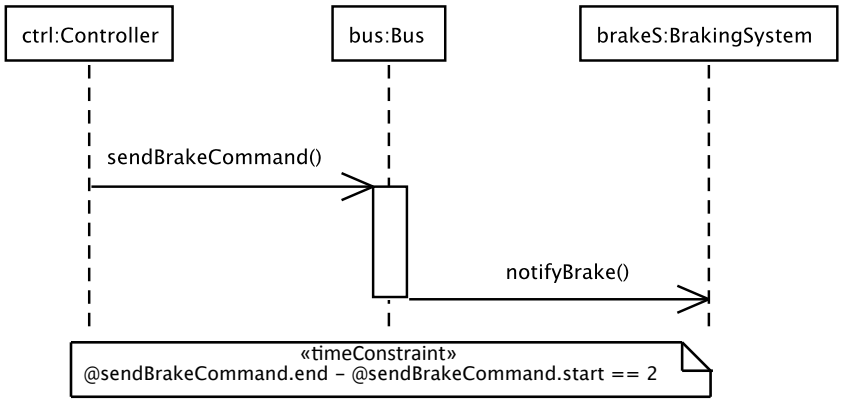


Fig. 6: CCAS: Sequence Diagram sendBrakeCommand.

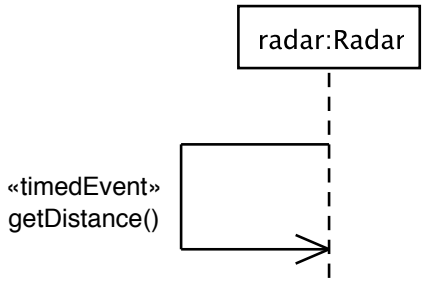


Fig. 7: CCAS: Sequence Diagram periodicReadDist.

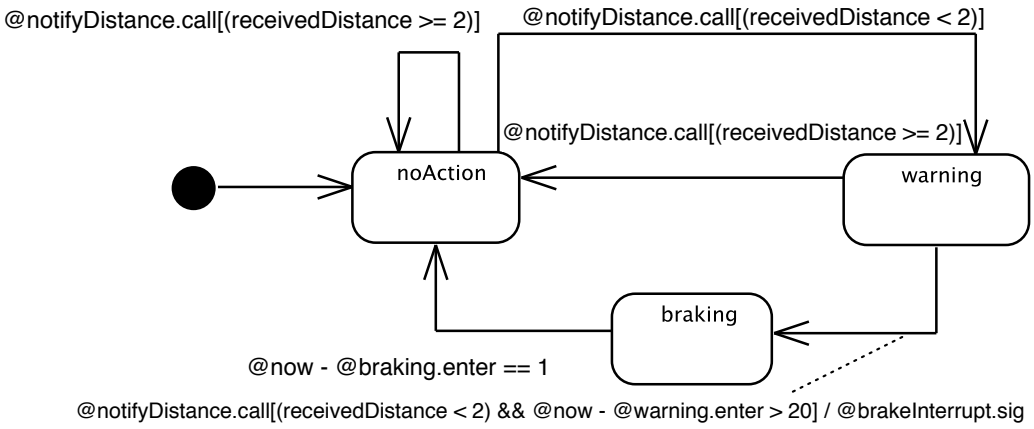


Fig. 8: CCAS: State Machine Diagram of class Controller.

The CCAS detects the position of the vehicle on which it is installed with respect to other objects. The distance between the car and the external objects is read through

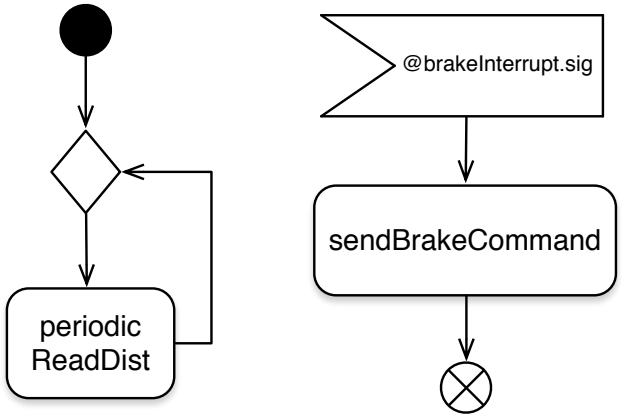


Fig. 9: CCAS: Interaction Overview Diagram.

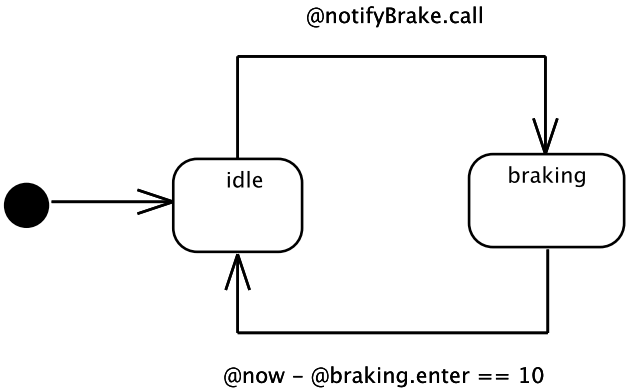


Fig. 10: CCAS: State Machine Diagram of class BrakingSystem.

an on-board Radar, which is in charge of validating and sending data to the Controller through the system Bus. When the distance between the car and the external objects is greater than or equal to 2 meters the CCAS should not perform any action. When the distance becomes strictly less than 2 meters the CCAS switches to a warning state. If the distance continues to remain less than 2 meters for 200 ms, the CCAS must brake. The brake command is sent to the BrakingSystem, which finally brakes the car. The Radar reads and validates the distance received from the sensor every 40 ms. If the sensor is broken, or if it produces an inconsistent value, the Radar invalidates the distance reading. A failure signal is issued by the Radar if the distance read has been invalidated more than two consecutive times. If the distance read is valid, it is sent to the controller via the Bus, which takes 10 ms.

The complete model comprises one Class Diagram, one Object Diagram, one Interaction Overview Diagram, one Activity Diagram, three Sequence Diagrams, and two State Machine Diagrams.

The Class Diagram of Figure 4 defines the different classes in the system. RadarClock is used to model the fact that Radar reads the distance periodically, and stereotype «clockType» defines the period of the clock, where one time unit corresponds to 10 ms. Attribute sensedDist of class Radar captures the value of the distance detected by

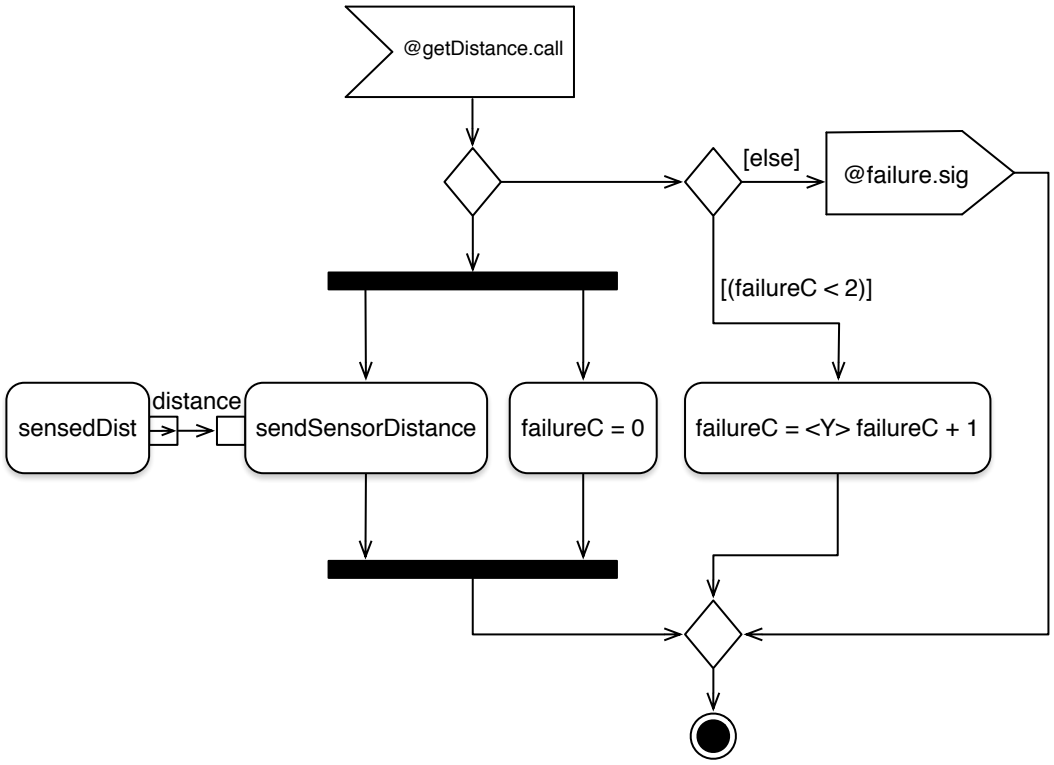


Fig. 11: CCAS: Activity Diagram of class Radar.

the radar. It is tagged as «free» because its value changes not because of assignments performed by the components of the system, but non-deterministically, depending on the environment in which the system operates. The Object Diagram is very simple, in that it instantiates exactly one object for each class of Figure 4, so it is not shown here for brevity.

The communication among the different objects in the system is captured by two Sequence Diagrams. The Sequence Diagram of Figure 5 says that the sending of the distance by the Radar through the Bus takes one time instant (10 ms). The Sequence Diagram of Figure 6 states how the Controller sends the brake command to the BrakingSystem via the Bus and that this takes exactly two time instants. In addition, the Sequence Diagram of Figure 7 describes that the Radar periodically reads its sensor value according to the period set by clock RadarClock.

The Controller monitors the value of the received distance according to the State Machine Diagram of Figure 8. If the Controller detects that the system has been in state warning for more than 20 time units (200ms), the transition from warning to braking takes place and consequently signal brakeInterrupt is issued. The Interaction Overview Diagram (Figure 9) captures the fact that Sequence Diagram periodicReadDist is executed repeatedly, whereas Sequence Diagram sendBrakeCommand is executed in response to signal brakeInterrupt[8]. The State Machine Diagram of Figure 10 captures the behavior of BrakingSystem leaving state braking after 10 time instants. Finally, the

[8]Note that this way only the current thread is terminated, but the diagram remains active and waits for the next occurrence of signal brakeInterrupt.

behavior of Radar is captured by the Activity Diagram of Figure 11. The diagram describes how a counter of the number of failures is updated with each new reading of the sensor value; for simplicity, the choice of whether to accept the reading or not is non-deterministic.

4.2. Class and Object Diagrams

A C-UML Class Diagram includes a finite set of classes, clocks, and signals. An Object Diagram defines the objects that instantiate the classes defined by a Class Diagram. The set of objects must be finite to enable the verification. Clocks define periodic events, whereas signals represent general-purpose events that are not bound to a specific class. For every clock c, a predicate $Clock_{id_c}Tick$ is declared, where id_c is the unique identifier of the clock. For every signal s we declare a predicate $Signal_{id_s}$. For every operation y that belongs to an object x we introduce the predicate $Obj_{id_x}Op_{id_y}$ to denote the invocation of the operation, and $Obj_{id_x}Op_{id_y}Reply$ to denote the corresponding reply. Note that prefix Obj_{id_x} is needed to distinguish the predicates generated for the same operation, but for a different object. Predicates for operation parameters, return values, and attributes are generated in a similar way. More precisely, the attributes of a class, the parameters and return values of operations are all translated into TRIO arithmetic variables. TRIO supports both integer and real variables.

From the point of view of the formalization of C-UML models, the role of Class Diagrams is twofold: on one hand, they declare part of the alphabet of the model (in particular, the elements that are shared across diagrams); on the other hand, they define the semantics of clocks. Concerning the latter issue, given a clock c with period T, the associated semantics is that its tick event must occur every T time units. In other words, its clock tick occurs if, and only if, it did not tick during the last $T-1$ time units, which implies that it ticks at times $T, 2T, 3T, \ldots$: [9]

$$\text{Lasted}(\neg Clock_{id_c}Tick, T-1) \Leftrightarrow Clock_{id_c}Tick \tag{2}$$

For instance, in the Class Diagram of Figure 4 the timing of Radar operations is specified by RadarClock, whose formal semantics is therefore captured by:

$$\text{Lasted}(\neg RadarClockTick, 3) \Leftrightarrow RadarClockTick \tag{3}$$

This clock is used in Sequence Diagram periodicReadDist (Figure 7) that forces the Radar object to invoke getDistance every 4 time instants, which in turn initiates its Activity Diagram (Figure 11). In fact, the message that represents the invocation of operation getDistance is tagged with stereotype «timedEvent», hence it is synchronized with the ticks of the RadarClock, as defined by Formula (4):

$$Obj_{radar}getDistanceSend \Leftrightarrow RadarClockTick \tag{4}$$

4.3. State Machine Diagrams

Each State Machine Diagram is owned by a class, and each class may have more than one State Machine Diagram. In this case, the different state machines run in parallel, that is, each object has a current state in each State Machine Diagram and they run independently. For each object o we generate the following predicates according to the State Machine Diagrams that describe its behavior. For every state s we declare three predicates $Obj_{id_o}State_{id_s}Enter$, $Obj_{id_o}State_{id_s}Exit$ and $Obj_{id_o}State_{id_s}$ that hold, respectively, when the object is entering or exiting the state, and when it is in the state.

[9] TRIO axioms are implicitly asserted for all time instants, hence Formula (2) is implicitly interpreted as $Always(\text{Lasted}(\neg Clock_{id_c}Tick, T-1) \Leftrightarrow Clock_{id_c}Tick)$.

If this state has also an invariant we declare predicate $Obj_{id_o}Invariant_{id_s}$. For every transition t of the same diagram we declare predicate $Obj_{id_o}Transition_{id_t}$. The same transition may have a trigger, a guard, and an action, which are mapped to predicates: $Obj_{id_o}Trigger_{id_t}$, $Obj_{id_o}Guard_{id_t}$, $Obj_{id_o}Action_{id_t}$ respectively.

The formalization of State Machine Diagrams is fairly standard, therefore we will only hint at some of its features. One of the key assumptions in our formalization is that the firing of a transition consumes a time instant, so we do not allow multiple transitions to occur at the same time instant, as some semantics of State Machine Diagrams/Statecharts do [Eshuis 2009]. The TRIO family of languages does indeed support mechanisms for modeling so-called zero-time transitions [Rossi et al. 2016], but they complicate the coordination among different components [Ferrucci et al. 2012], especially in a real-time setting.

In C-UML, diagrams types are ascribed with a common temporal semantics, and most of them are not naturally suited to accommodate zero-time transitions. Hence, to better allow for the synchronization of heterogeneous types of diagrams, the choice was made to avoid zero-time transitions also in State Machine Diagrams. This has no impact on the practical applicability of the approach: for example, multiple—non-conflicting—assignments (see Section 4.6) can still be performed in the same time instant.

Given a state s owned by object o, we define the set of its incoming and outgoing transitions as $Incoming_s$ and $Outgoing_s$, respectively. A necessary condition to enter the state is that one of the incoming transitions holds in the previous time instant:

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

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

Given a transition t that connects a source state s with a different destination state d, the sufficient condition to leave s and enter d is that t occurs:

$$Obj_{id_o}Transition_{id_t} \Rightarrow Obj_{id_o}State_{id_s}Exit \ \wedge \ \text{Futr}(Obj_{id_o}State_{id_d}Enter, 1) \tag{6}$$

This axiom is not generated if t is a self-loop. When object o is in state s, predicate $Obj_{id_o}State_{id_s}$ holds until $Obj_{id_o}State_{id_d}Exit$ holds. Similarly, if object o is not in state s, unless the corresponding $Enter$ predicate holds in the next time instant, it will remain in a different state. A necessary condition for taking the transition is that the object is in source state s and the corresponding guard and trigger hold, if present. The trigger is defined by the event associated with it. If e is the event associated with the trigger of transition t, the trigger predicate of t holds if, and only if, the object is in state s and event e occurs. Similarly, the guard predicate holds if, and only if, the associated Boolean condition holds and the object is in the source state. If the transition has actions associated with it, then the action predicates hold one time instant after the transition. Finally, the outgoing transitions of a state are mutually exclusive. If more than one transition can be fired at a given time instant, the one that will be actually taken is chosen non-deterministically. This is achieved in temporal logic formulae by stating that, if a transition is enabled, then there exists exactly one enabled transition that is taken.

As example, we introduce the formal semantics of the transition from state warning to state braking in Figure 8, along with its trigger, guard and action. The following formula states that when transition controllerTran1 occurs, it takes one time instant to

switch states:

$$controllerTran_1 \Rightarrow controllerWarningExit \ \wedge \ \mathrm{Futr}(controllerBrakingEnter, 1) \quad (7)$$

The transition is annotated with @notifyDistance.call[receivedDistance<2 && @now - @warning.enter>20] / @brakeInterrupt.sig, according to the syntax Trigger[Guard]/Action. Predicates $controllerTran_1Guard$, and $controllerTran_1Trigger$ are mapped to, respectively, the guard and trigger of the transition; they are defined by the following formulae, which state that the guard (trigger) holds when the corresponding Boolean condition (triggering event) holds:

$$controllerTran_1Guard \Leftrightarrow controllerWarning \wedge receivedDistance < 2 \wedge TC_1 \quad (8)$$

$$controllerTran_1Trigger \Leftrightarrow controllerWarning \wedge controllerNotifyDistance.call \quad (9)$$

In Formula 8, TC_1 is a placeholder for the formula capturing @now - @warning.enter>20; its precise definition will be given in Section 4.8. The following formula, instead, defines that the action (i.e., the emission of signal brakeInterrupt) is executed one time instant after the transition:

$$\mathrm{Past}(controllerTran_1 \wedge controllerWarning, 1) \Leftrightarrow brakeInterrupt \quad (10)$$

Finally, the next formula defines precisely when the transition is taken (i.e., when the State Machine Diagram is in the source state, and the guard and trigger both hold):

$$controllerTran_1 \Rightarrow \quad (11)$$
$$controllerWarning \wedge controllerTran_1Guard \wedge controllerTran_1Trigger$$

As mentioned above, a transition in a State Diagram can include actions, which can be of different kinds. The semantics of actions, which is common across different types of diagrams, is presented in Section 4.6.

4.4. Sequence Diagrams

For each Sequence Diagram s we generate predicates $SD_{id_s}Start$ and $SD_{id_s}End$ that are true at the beginning and at the end of the diagram execution, respectively. We also generate predicate $SD_{id_s}Stop$ that holds true when the diagram is interrupted, that is, when it is stopped without reaching its natural end. Finally, predicate SD_{id_s} holds true when the Sequence Diagram is active, that is, when the diagram has started, but it has not ended nor has it been stopped, yet. For every message m, we declare two predicates: $Msg_{id_m}Start$ and $Msg_{id_m}End$. The predicates hold at the beginning and at the end of the message, respectively. Note that these two time instants may—but need not—coincide. For every execution specification e, we define predicates $ExS_{id_e}Start$, $ExS_{id_e}End$, and ExS_{id_e}, which hold true, respectively, at the beginning, at the end, and during the execution specification. Parameters and assignments—within Sequence Diagrams—have received a number of different interpretations [Micskei and Waeselynck 2011]. C-UML allows each Sequence Diagram to define parameters whose values are chosen non-deterministically when the diagram starts and remain constant during the entire execution of the diagram, unless there is an assignment.

A parameter in a Sequence Diagram can be assigned in an Activity Diagram, where an object flow connects it to a variable. For example, in the Activity Diagram of Figure 11 Sequence Diagram sendSensorDistance is initiated while its parameter (*distance*) is set to the value of attribute *sensedDist* of Radar. With this feature the user can forward a value between the objects in a Sequence Diagram without any need for complex assignments. For example, by using parameter *distance* as the actual argument for operations sendSensorDistance and notifyDistance in the Sequence Diagram of Figure 5 we define that the value of *distance* is passed around the objects of the

diagram. For every variable y in a Sequence Diagram s, we declare a TRIO arithmetic variable $SD_{id_s}Par_{id_y}$. For every assignment a, we declare a predicate $Assignment_{id_a}$ that holds true when the assignment is performed in the diagram. Finally, for every time constraint c, we declare predicate $Constraint_{id_c}$ that holds true from the beginning until the end of the execution of the Sequence Diagram.

A Sequence Diagram is defined as a set of lifelines. Every lifeline is an ordered list of events. These events can be the start/end of messages, start/end of execution specifications, and assignments. Given a Sequence Diagram s, we define $Lifelines_s$ as the set of lifelines that belong to s. Also, given a lifeline $l \in Lifelines_s$, we define $LifelineEv_l$ as the set of events that belong to the lifeline. For every pair of events $Ev_i, Ev_j \in LifelineEv_l$, where Ev_i precedes Ev_j on the lifeline, if Ev_i holds at a given instant, then Ev_j will follow in the future if the diagram is not stopped. This is formalized by the following axiom:

$$Ev_i \;\Rightarrow\; \mathrm{Until}(\neg Ev_i \wedge \neg Ev_j, SD_x Stop) \;\vee\; \mathrm{Until}(\neg Ev_i \wedge \neg SD_x Stop, Ev_j) \tag{12}$$

Also, if Ev_j holds at a given instant, then Ev_i was true in the past and the execution has not been stopped in the meanwhile. This is formalized by the following axiom:

$$Ev_j \;\Rightarrow\; \mathrm{Since}(\neg Ev_j \wedge \neg SD_x Stop, Ev_i) \tag{13}$$

The first event in the Sequence Diagram is the start event that holds true when predicate $SD_{id_s}Start$ holds true. The first message of the Sequence Diagram may or may not start at the same time instant when the Sequence Diagram starts. The last event in the Sequence Diagram is represented by predicate $SD_{id_s}End$ that holds true when the last message of the Sequence Diagram occurs. The system is executing Sequence Diagram s if, and only if, $SD_{id_s}Start$ holds, or the diagram neither ended nor has been stopped since the last time the predicate held.

The different lifelines are connected through messages. To say that the send event of a message m is followed by the corresponding receive event, and that a receive event is preceded by the send event we use axioms similar to 12 and 13:

$$\begin{aligned} Msg_{id_m}Send \;\Rightarrow\; & \mathrm{Until}(\neg Msg_{id_m}Send \wedge \neg Msg_{id_m}Receive, SD_x Stop) \;\vee \\ & \mathrm{Until}(\neg Msg_{id_m}Send \wedge \neg SD_x Stop, Msg_{id_m}Receive) \end{aligned} \tag{14}$$

$$Msg_{id_m}Receive \;\Rightarrow\; \mathrm{Since}(\neg Msg_{id_m}Receive \wedge \neg SD_x Stop, Msg_{id_m}Send) \tag{15}$$

For every execution specification e, predicate ExS_{id_e} holds between $ExS_{id_e}Start$ and $ExS_{id_e}End$. The objects that are inside an execution specification must be considered busy. We define the set of execution specifications that belong to object o as $ExSSet_o$. For each pair of execution specification $i, j \in ExSSet_o$, the corresponding predicates—ExS_{id_i} and ExS_{id_j}—are mutually exclusive.

C-UML supports variable assignments in Sequence Diagrams represented as recursive messages. The name of the message contains the assignment expression. The semantics for assignment will be presented in Section 4.6. The same kind of semantics is used to set the actual values of the formal parameters of the methods invoked in Sequence Diagrams. For every parameter k that does not appear in the left-hand side of any assignment, we impose that the value of the parameter remains constant throughout the whole diagram. This means that the parameter keeps the value it had at the previous time instant—with the exception of the first instant the diagram is active. This is formalized by the following axiom, where $SD_{id_s}Par_{id_k}$ is the TRIO arithmetic variable that captures the value of parameter k:

$$SD \wedge \neg SD_s Start \;\Rightarrow\; SD_{id_s}Par_{id_k} = \mathrm{Past}(SD_{id_s}Par_{id_k}, 1) \tag{16}$$

Given a time constraint c associated with a Sequence Diagram s, the predicate $Constraint_{id_c}$ associated with the constraint holds while we are inside the Se-

quence Diagram (i.e., when predicate SD_{id_s} holds). This is formalized by the axiom $Constraint_{id_c} \Rightarrow SD_{id_s}$.

We can now exemplify all these definitions by considering the lifeline of object bus in the Sequence Diagram of Figure 5. Since the first and the last events of the lifeline are linked to the start and end points of the execution specification covering the lifeline (whose id ES_1 is not shown on the diagram, as customary for execution specifications), they must be synchronized with ES_1Start and ES_1End, respectively, as formalized by the following formulae:

$$ES_1Start \Leftrightarrow sendSensorDistanceEnd \tag{17}$$

$$ES_1End \Leftrightarrow notifyDistanceStart \tag{18}$$

Moreover, the following formulae force ES_1Start to be followed by ES_1End (unless SD_1Stop takes place), and ES_1End to be preceded by ES_1Start:

$$ES_1Start \Rightarrow \text{Until}(\neg ES_1Start \wedge \neg ES_1End,\ SD_1Stop) \vee \text{Until}(\neg ES_1Start \wedge \neg SD_1Stop,\ ES_1End) \tag{19}$$

$$ES_1End \Rightarrow \text{Since}(\neg ES_1End \wedge SD_1Stop,\ ES_1Start) \tag{20}$$

As mentioned above, an object cannot have more than one active execution specification. Accordingly, since object bus is engaged in the Sequence Diagrams of Figures 5 and 6, and in both cases it is covered by an execution specification, the following formula is added to prevent the object from having both execution specifications active at the same time:

$$ES_1 \Rightarrow \neg ES_2 \tag{21}$$

Sequence Diagrams also support the concept of *combined fragments*, but their semantics has been subject to different interpretations. For simplicity, in this paper we do not delve further in this issue. However, we have defined formal semantics for the various interpretations and our tool, Corretto, can handle each of them; interested readers can find further details in [Baresi et al. 2014].

4.5. Activity Diagrams and Interaction Overview Diagrams

C-UML supports both Activity Diagrams and Interaction Overview Diagrams. In C-UML the dynamic aspects of an object can be captured by State Machine Diagrams and Activity Diagrams, whereas Interaction Overview Diagrams are used to describe the coordination among the different (partial) interactions. For example, the State Machine Diagram of Figure 8 and the Activity Diagram of Figure 11 capture the behavior of Controller and Radar, respectively, whereas the Interaction Overview Diagram of Figure 9 specifies the interaction among Controller, Radar, Bus, and BrakingSystem.

Activity Diagrams contain *control*, *action*, and *object* nodes interconnected through *control flow* and *object flow* edges. Control flows define the chaining of activities, and object flows capture how data (object nodes) flow between the input/output pins of action nodes. For example, in the Activity Diagram of Figure 11 an object flow sets parameter distance of Sequence Diagram sendSensorDistance to the value of variable sensedDist.

Control nodes include *initial*, *decision*/*merge*, *fork*/*join*, *flow final*, and *activity final* nodes. Their behavior is formalized through a number of predicates that become true each time they become active. For every *decision* node d, we generate k predicates, where k is the number of control flows that leave the node: for every outgoing control flow p, predicate $Dec_{id_d}Path_p$ represents the fact that p has been taken. In addition, if a guard is associated with the control flow, a dedicated predicate $Dec_{id_d}Guard_p$ is added.

When a *decision* is activated, one of its control flows whose guard is true is selected non-deterministically, while a *merge* becomes active as soon as it receives a token from one of its incoming control flows. *Fork* nodes with guards are translated in a similar way (e.g., a predicate $Fork_{id_f}Path_p$ is introduced for each path leaving fork node f), but all of the outgoing control flows will be activated. For every *join* node j, predicate $Join_{id_j}End$ holds true when all its incoming control flows terminate their execution.

Action (executable) nodes are the fundamental elements of Activity Diagrams that can modify the state of the corresponding objects and initiate interactions with other objects. Among them, *accept event* nodes listen to signals and invocations of specific object operations. They are activated and receive a token as soon as the signal is issued or the operation is invoked. C-UML also supports nodes of type *assignment*, *call operation*, *send signal*, and *init sequence diagram*; the corresponding actions are formalized in Section 4.6.

A key part of the formalization of an action node defines the start and duration of the execution of the the node, which depend on the type of the node itself. Let us consider an action node n in an Activity Diagram i. Depending on the nature of n, a predicate is introduced to capture when the action associated with the node starts being executed; for example, if n is an *init sequence diagram* node, the predicate is $AD_{id_i}SD_{id_n}Start$,[10] whereas if n is a *call operation* node, it is $AD_{id_i}OP_{id_n}Call$. To generalize, in the following we refer to the predicate as $AD_{id_i}Action_{id_n}$. Let us remark that an action node can have at most one incoming control flow (*accept event* nodes need not have an incoming control flow, as Figure 11 shows). For an action node n to start its execution, an "activation condition" must be met, which depends on the nature of its input node u—i.e., the source of the incoming control flow of n. We indicate the activation condition as $AC_n(u)$, and we describe below how it is defined. Then, predicate $AD_{id_i}Action_{id_n}$ holds one time unit after the $AC_n(u)$ holds, unless n is an *accept event* node. In the latter case, predicate $AD_{id_i}AEAction_{id_n}$ holds if, and only if, the event associated with n holds in the current instant, the activation condition $AC_n(u)$ held sometimes in the past, and no other accept event action occurred since then.

Condition $AC_n(u)$ is defined recursively based on the nature of u. If u is an *initial* node, then $AC_n(u)$ simply corresponds to predicate $AD_{id_i}Start$ capturing the start of the Activity Diagram being true. If u is an *action* node that takes one instant to be executed (this is the case for *assignment*, *call operation*, *send signal* and *accept event* nodes), $AC_n(u)$ corresponds to $AD_{id_i}Action$ being true. If u is an *init sequence diagram* node, $AC_n(u)$ is $AD_{id_i}SD_{id_n}End$, which holds when the Sequence Diagram terminates.

We have also several inductive cases. If u is a *merge* node, and $h_1 \dots h_k$ are its input nodes, the activation condition of s holds when any of the incoming nodes terminates its execution. Formally, this is translated as:

$$AC_n(u) = AC_u(h_1) \dots \vee AC_u(h_k). \tag{22}$$

If u is a *fork* node, h is its input node, and p is the path connecting u with n, the activation condition of n holds when the predicate assigned to p holds and the activation condition of u holds:

$$AC_n(u) = AC_u(h) \wedge Fork_{id_u}Path_p. \tag{23}$$

If u is a *decision* node, h is its input node, and p is the path connecting u with n, the activation condition of n holds when the predicate assigned to p holds and the activation condition of u holds:

$$AC_n(u) = AC_u(h) \wedge Dec_{id_u}Path_p. \tag{24}$$

[10] For simplicity, in the rest of this section we omit prefix Obj_{id_o}, which identifies the object to which the Activity Diagram belongs, from diagram-related predicates such as $Obj_{id_o}AD_{id_i}SD_{id_n}Start$.

Unlike a fork node, the paths that leave a decision node are mutually exclusive, thus constraint $\forall p' \neq p(Dec_{id_u} Path_p \Rightarrow \neg Dec_{id_u} Path_{p'})$ holds.
If u is a ***join*** node with input nodes $h_1 \dots h_k$, $AC_n(u)$ corresponds to predicate $Join_{id_n} End$ being true, which in turn occurs at the time instant when the last activation condition among $AC_n(h_1) \dots AC_n(h_k)$ holds.

Finally, if n is an *activity final* node of the Activity Diagram i, with input node u, predicate $AD_{id_i} End$ holds at the same time instant as the activation condition $AC_n(u)$ holds (similarly for *flow final* nodes).

Interaction Overview Diagrams can be seen as special-purpose Activity Diagrams, where action nodes can only be of type *init sequence diagram* and *send/receive signal* (see Section 4.6). For example, the Interaction Overview Diagram of Figure 9 activates Sequence Diagram periodicReadDist infinitely often, since the decision node has only one output path with an implicit true guard, and Sequence Diagram periodicReadDist is both its input and output node. This is captured by the following formula:

$$periodicReadDistStart \Leftrightarrow \text{Past}(periodicReadDistEnd \vee IOD_1 Start, 1) \tag{25}$$

In addition, the Interaction Overview Diagram keeps listening to signal brakeInterrupt sent from State Machine Diagram Controller (Figure 8) and, upon receiving it, it activates Sequence Diagram sendBrakeCommand after one time instant; this is formalized as follows:

$$sendBrakeCommandStart \Leftrightarrow \text{Past}(brakeInterrupt, 1) \tag{26}$$

Interaction Overview Diagrams support the concept of Interruptible Regions. An Interruptible Region identifies a set of Sequence Diagrams, whose behavior can be interrupted when a given event e occurs. This means that for every Sequence Diagram s enclosed in at least one Interruptible Region, $SD_s Interrupts$ is the set of interrupts associated with the Interruptible Regions enclosing s. More precisely, if s belongs to an Interruptible Region associated with the interrupt event e, then e belongs to $SD_s Interrupts$. s is stopped if, and only if, the Interaction Overview Diagram has reached the end node, or when one of the Interruptible Regions is activated. More formally, we must constrain predicate $IOD_{id_i} SD_{id_s} Stop$ such that it holds if, and only if, predicate $IOD_{id_i} End$ holds, or one of the interrupt events e in $SD_s Interrupts$ holds.

4.6. Actions

To capture the semantics of *assignment* actions, a predicate $Obj_{id_x} Assignment Var_{id_v} UML_{id_u}$ is introduced for each assignment, where id_x is the name of the object in which the assignment takes place, id_v is the name of the variable (i.e., the attribute) that is assigned, and id_u is the identifier of the UML element to which the assignment is attached (e.g., an assignment node).

The assignment predicate holds and lasts for one time unit as a result of its action node receiving a token in an Activity Diagram, its transition being fired in a State Machine Diagram, or its corresponding message being received in a Sequence Diagram.

A set of formulae is produced to define the value of the variable at the time instant the assignment takes place. Let us consider the statement `x = <Y>y + z` in an assignment node of an Activity Diagram, and let us call Asg_1 the corresponding predicate, for simplicity. Predicate Asg_1 holds when the enclosing assignment node receives a token, and the following formula is produced:

$$Asg_1 \Rightarrow Var_x = \text{Past}(Var_y, 1) + Var_z \tag{27}$$

Note that, if a variable `v` appears in an assignment with the qualifier `<Y>` (resp., `<X>`), the corresponding TRIO term is $\text{Past}(Var_v, 1)$ (resp., $\text{Futr}(Var_v, 1)$). If variable `x` has n assignments $Asg_i \dots Asg_n$ in the various diagrams, the following formula is produced

to force the variable to maintain its value when there is no assignment.

$$\neg \bigvee_{i=1}^{n} Asg_i \Rightarrow Var_x = \mathrm{Past}(Var_x, 1) \tag{28}$$

For example, if we considered the Activity Diagram of Figure 11, where two assignment nodes modify the value of attribute failureC, the following set of formulae would be produced:

$$Obj_{radar}AssignmentVar_{failureC}UML_1 \Rightarrow failureC = 0 \tag{29}$$

$$Obj_{radar}AssignmentVar_{failureC}UML_2 \Rightarrow failureC = \mathrm{Past}(failureC, 1) + 1 \tag{30}$$

$$\neg(Obj_{radar}AssignmentVar_{failureC}UML_1 \vee Obj_{radar}AssignmentVar_{failureC}UML_2) \Rightarrow failureC = \mathrm{Past}(failureC, 1) \tag{31}$$

Objects can invoke operations that belong to themselves or to other objects if they are connected in the Object Diagram. To capture *call operation* actions, given an operation id_{op} invoked on object id_x, predicate $Obj_{id_x}Op_{id_{op}}$ holds when at least one of the triggers of the operation call occurs, as captured by the following formula:

$$Obj_{id_x}Op_{id_{op}} \Leftrightarrow \bigvee_{i=1}^{n} Trigger_i \tag{32}$$

where $Trigger_i$ can be the predicate associated with a *call operation* node in an Activity Diagram, a transition action in a State Machine Diagram, or a receiving message event in a Sequence Diagram. Additionally, if the operation to be invoked has input parameters, their value must be set at the invocation time, the node must have as many input pins as parameters, and they must be set by means of object flows.

The semantics of a *send signal* action is similar to the one of a *call operation*, that is, it is a disjunction over related triggers across diagrams. A sent signal is visible system-wide by all objects; it is instantly read by *accept event* actions in Activity and Interaction Overview Diagrams, such that the corresponding nodes receive a token and regulate the control flow thereafter. However, unlike in Interaction Overview Diagrams, *accept event* actions in Activity Diagrams can also read operation calls in addition to signals.

Finally, *init sequence diagram* is a kind of action that initializes a Sequence Diagram in Activity Diagrams and Interaction Overview Diagrams as soon as the corresponding node receives a token. Once the Sequence Diagram terminates, the token is transferred to the next node. A Sequence Diagram may be referenced by several *init sequence diagram* nodes or triggers in Activity and Interaction Overview Diagrams. Therefore, similarly to *send signal* and *call operation* actions, the predicate for the *init sequence diagram* action is constrained to coincide with at least one of its triggers.

4.7. Shared Events

Different diagrams may share a common set of events as enablers of the communication among them. Shared events include operation invocations, clock ticks, signals, Sequence Diagram starts, ends, and stops, Activity Diagrams starts and ends, and Interaction Overview Diagram starts and ends.

For example, Figure 12 shows the parts of the Interaction Overview Diagram of Figure 9, of State Machine Controller of Figure 8, and of Sequence Diagram sendSensorDistance of Figure 5 that communicate through the invocation of operation notifyDistance and signal brakeInterrupt. More precisely, let us consider the scenario where the CCAS activates the braking system. Object radar in Sequence Diagram sendSensorDistance of Figure 12(a) frequently updates the current distance in ctrl via bus, which

invokes operation notifyDistance. If the received distance is less than 2 meters, and the system has been in state warning for longer than 20 time instants, then it enters state braking and sends signal brakeInterrupt (Figure 12(c)). At the very same time the signal is received within the Interaction Overview Diagram, which entails that Sequence Diagram sendBrakeCommand is immediately activated (Figure 12(b)).

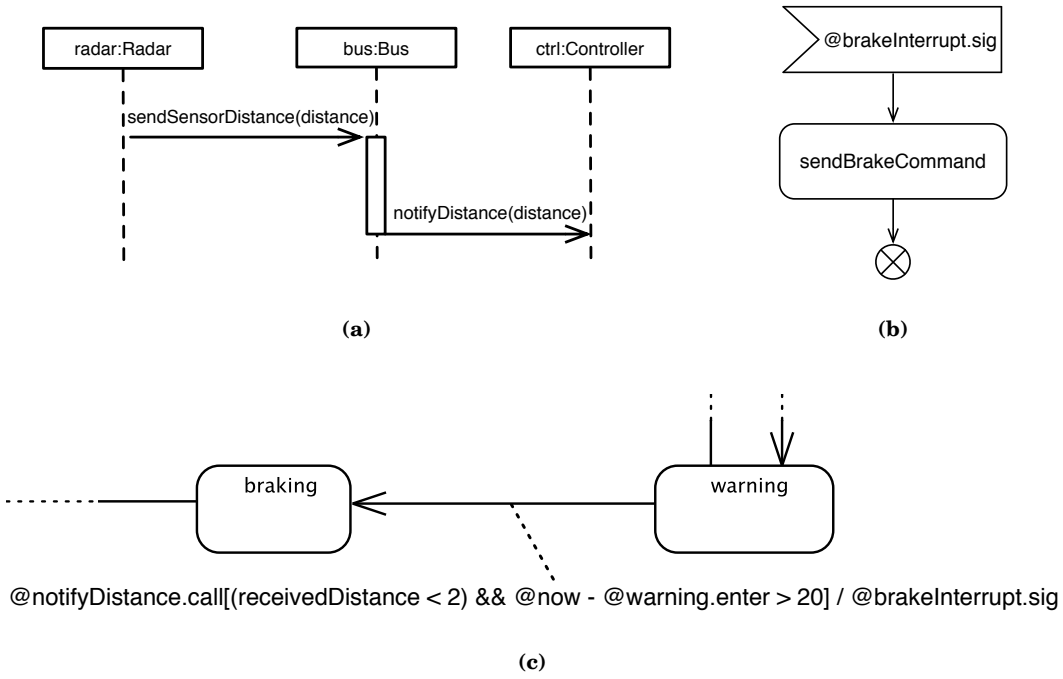


Fig. 12: Examples of intra-diagram interactions.

Given two diagrams d_x, d_y of a C-UML model, their shared events are formalized through their combined behavior $\Theta(d_x, d_y)$ as follows. Let us define $\chi : \mathcal{P} \times \mathcal{P} \rightarrow \{read, nil\}$ as the function that defines the type of communication between two predicates: $\chi(p_x, p_y) = read$ if the diagram that owns p_x *catches* an event p_y that occurs in another diagram; $\chi(p_x, p_y) = nil$ if we have no relation between p_x and p_y. The combined behavior $\Theta(d_x, d_y)$ is defined according to the values of function χ. For each predicate p_x that is associated with diagram d_x (i.e., $p_x \in \Gamma(d_x)$, where Γ is defined earlier in this section) let us define $Reasons_{p_x}$ as the set of predicates $p_y \in \Gamma(d_y)$, with $d_y \neq d_x$, such that $\chi(p_x, p_y) = read$. Intuitively, set $Reasons_{p_x}$ contains all the predicates that may cause predicate p_x to be true. This is formalized as $p_x \Leftrightarrow \bigvee_{r \in Reasons_{p_x}} r$.

In the example of Figure 12 we have that:[11]

— $Op_{notifyDistance}$ is the predicate associated with operation notifyDistance in the Class Diagram.
— $MessageEnd_{notifyDistance}$ is the predicate associated with the end of message notifyDistance in the Sequence Diagram.

[11] For the sake of readability, we omit prefix Obj_x from each predicate.

— $Trigger_{notifyDistance}$ is the predicate associated with the trigger of the transition of the State Machine Diagram associated with class Controller.
— $Action_{brakeInterrupt}$ is the predicate associated with the action of the transition of the State Machine Diagram associated with class Controller.
— $Signal_{brakeInterrupt}$ is the predicate associated with signal brakeInterrupt in the Interaction Overview Diagram.

The values of χ define the semantics of the various connections. The method predicate in the Class Diagram reads from the message end predicate in the Sequence Diagram and this is its only read relationship, therefore we produce formula $Op_{notifyDistance} \Leftrightarrow MessageEnd_{notifyDistance}$. The trigger predicate in the State Machine Diagram reads from the operation predicate in the Class Diagram and this is its only read relationship, therefore we produce formula $Trigger_{notifyDistance} \Leftrightarrow Op_{notifyDistance}$. Finally, the signal predicate in the Interaction Overview Diagram reads from the action predicate in the State Machine Diagram and this is its only read relationship, therefore we produce the formula $Signal_{brakeInterrupt} \Leftrightarrow Action_{brakeInterrupt}$. This is enough to connect the semantics of all these diagrams.

4.8. Time Constraints

As mentioned in Section 3, time constraints are of the form $@ev2 - @ev1 \sim K$, where $ev2$ can be now, and $\sim$ is one of $\{<, \leq, =, \neq, \geq, >\}$. In general, the meaning of a time constraint $@ev2 - @ev1 \sim K$ is, informally, "the distance in time between the last occurrence of $ev2$ and the last occurrence of $ev1$ is $\sim K$". If now is used instead of $ev2$, the constraint becomes $@now - @ev1 \sim K$, and its meaning is "the distance in time between the current instant and the last occurrence of $ev1$ is $\sim K$".

Time constraints can appear in guards and triggers of transitions in State Machine Diagrams (see for example Figure 8 and Figure 10), in guards of control flows in Activity and Interaction Overview Diagrams, and attached to whole diagrams (as exemplified in Figure 5). The formalization of time constraints changes depending on whether they are part of guards and triggers, or if they are attached to whole diagrams.

If a time constraint t of the form $@ev2 - @ev1 \leq K$ appears as a guard of a State Machine Diagram transition or of an Activity Diagram control flow, it is formalized by the following axiom:

$$\mathrm{Since}(\neg ev1 \wedge \neg ev2, ev2 \wedge \mathrm{WithinP_{ii}}(ev1, K)) \tag{33}$$

Essentially, the formula captures the condition "$ev2$ occurred, and $ev1$ also occurred no earlier than K time units before it", which can be used to enable transitions (or control flows), or to trigger them. Similar axioms can be derived if one of the other relations $\{=, <, \neq, >, \geq\}$ is used instead of $\leq$. A constraint of the form $@now - @ev1 \leq K$, instead, is translated into TRIO as $\mathrm{WithinP_{ii}}(ev1, K)$.

As further example, @now - @warning.enter > 20 appears in the annotation of the transition from state warning to state braking of Figure 8. We introduce predicate TC_1 to characterize the instants when the time constraint holds, by means of the following formula:

$$TC_1 \Leftrightarrow \mathrm{SomP}(warning.enter) \wedge \mathrm{Lasted_{ii}}(\neg warning.enter, 20)\,. \tag{34}$$

Then, predicate TC_1 can be referenced in other formulae, such as Formula (8).

If, instead, the time constraint is attached to a whole diagram, its meaning is to limit the behavior of that diagram, to guarantee that the constraint is met. For example, the constraint of Figure 5 states that in all executions of Sequence Diagram sendSensorDistance the end is reached one time instant after the execution is started. In this case,

an expression of the form $@ev2 - @ev1 \leq K$ is translated as:

$$ev2 \Rightarrow \text{WithinP}_{\text{ii}}(ev1, K) \wedge \neg\text{Since}(\neg ev1, SDStart \wedge \neg ev1) \tag{35}$$

that states that if $ev2$ occurs in the current instant, then $ev1$ occurred no more than K time instants in the past, within the same execution of the Sequence Diagram.

The semantics of the time constraints that use the other comparison operators can be easily derived from the above examples.

5. CORRETTO

Corretto is our prototype specification and verification tool for C-UML whose goal is to allow UML designers to carry out the formal verification of their models in a user-friendly manner. The goal is to have a verification tool that hides as many details about the creation of the formal representation as possible. The tool must provide the user with the means to understand how the model behaves according to the given semantics, specify the properties of interest without leaving the UML mindset, and understand the results produced by the underlying verification engine.

The formal verification of C-UML models is divided in four phases (Figure 13):

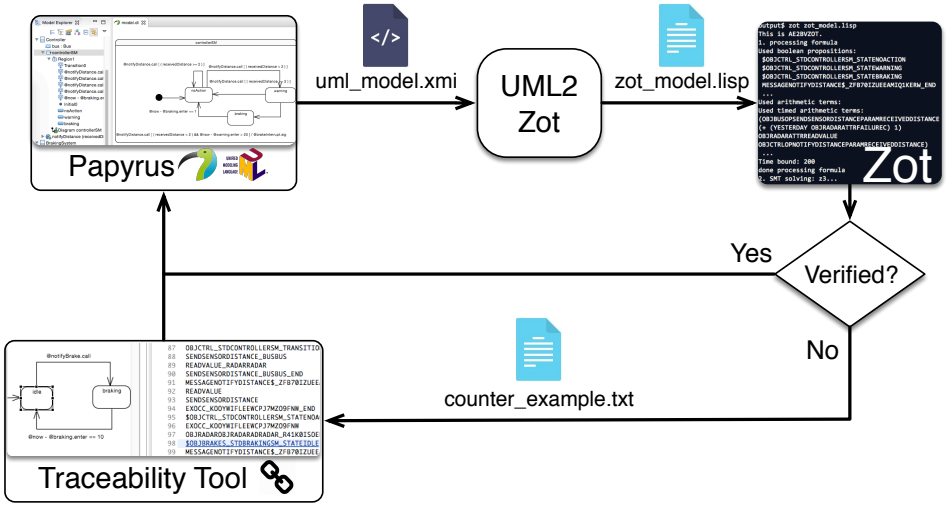


Fig. 13: The tools within Corretto.

Modeling. During this phase the user builds the model to be verified and specifies the properties of interest using C-UML. Corretto exploits *Papyrus*,[12] an Eclipse-based modeling tool that provides an integrated and user-oriented environment for editing models created with UML and related modeling languages such as SysML and MARTE. Papyrus is enriched with all the C-UML stereotypes to build models and specify the properties of interest.

Transformation. During this phase the C-UML model is translated—together with the properties to be verified—into its formal representation. The result, that is, the formal model, remains hidden to the user. To this end, we created **UML2ZOT**, a Java component that takes the XMI version of a C-UML model produced by Papyrus and automatically generates its corresponding TRIO representation. UML2ZOT is

[12] eclipse.org/papyrus

built on top of the Eclipse UML2 library, the Eclipse standard to read, create and modify UML-compliant models.

Verification. During this phase the required property is verified against the formal model. Corretto uses ℤot [Pradella et al. 2013; Baresi et al. 2015], a bounded satisfiability checker that encodes satisfiability (and validity) problems for discrete-time TRIO formulae as satisfiability problems for propositional logic or for decidable fragments of first-oder logic. Produced specifications are then checked with off-the-shelf SAT or SMT (Satisfiability Modulo Theories) solvers (e.g., Z3 [Moura and Bjørner 2008]). If the property specified by the user holds, then she is just notified of the result. If the property does not hold, ℤot returns a textual counterexample that violates the property.

Traceability. During this phase the results produced by the verification tool are inspected using a high-level representation. The aim is to bring verification results back to the modeling environment the user is familiar with. If the property does not hold, the **Traceability Tool** visualizes the textual counterexample produced by ℤot within Papyrus. The goal of this simple tool is to shorten the gap between the verification domain, to which the ℤot textual counterexample belongs, and the modeling domain, the one the user is familiar with. Currently this is achieved by simply *linking* the counterexample elements to their corresponding UML elements, but more sophisticated solutions could be built on top of this simple bridge. The tool enriches the textual elements of the trace with hyperlinks to the corresponding UML elements (see Figure 14) to provide a suitable means for the user to understand the results. In fact, by navigating the trace, the user can visualize elements of the UML model such as the states traversed by the various objects, the messages sent at each time instant, the activities being executed. For example, Figure 14 shows a trace such that at time instant 4 object brakes, which is of type BrakingSystem, is in state idle. The ℤot trace also shows, though not in the UML model, the values taken by each attribute at each time instant. For instance, in Figure 14, the right-most panel shows that at time 3 attribute failureC of object radar takes value 0.

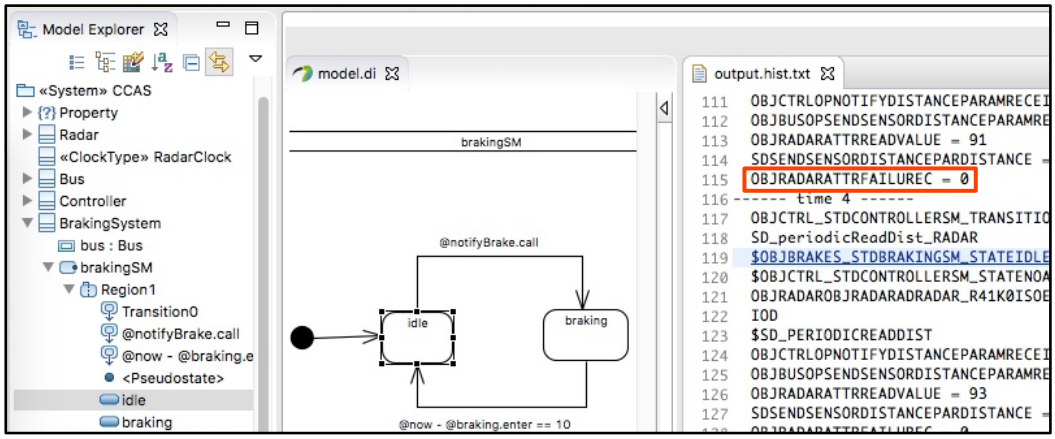


Fig. 14: Traceability navigation tool

In the end, she is therefore able to understand the verification results and modify the C-UML model accordingly.

6. EXPERIMENTAL EVALUATION

Corretto can be used to carry out various formal verification activities on C-UML models. These range from the automatic generation of traces compatible with the system model, e.g., for validation purposes, to the fully automated proof of relevant properties. In this section, we present six case studies. Each of them is meant to demonstrate some of the characteristics of our approach. In this section, we also perform some scalability experiments to study the performance of the Corretto tool. All experiments were carried out on a Linux desktop machine with a 3.4 GHz Intel ® Core ™ i7-4770 CPU and 8 GB RAM.[13] Note that our intent here is to show the feasibility of our UML-to-temporal logic approach for carrying out verification activities on UML diagrams, rather than studying optimizations that might make the tool more efficient, so we are not striving for maximum performance. Moreover, even if we use well-known examples in the domain of formal verification, our formal models are created automatically from higher-level specifications. As such they cannot be as efficient and optimized as those created on purpose and maybe by experts.

6.1. Car Collision Avoidance System

The Car Collision Avoidance System (CCAS) model has been developed in the MADES project[14] to demonstrate the features of C-UML in terms of supported UML elements and types of properties that can be proved. The model has already been used as running example in Section 4.1. Here we are interested in proving the following property: *if the distance remained less than* 2 *meters for* 45 *time units, then the system braked within those same* 45 *time units*. We expect this time frame to be wide enough for the system to react on time because the warning state lasts 20 time units and the bus delivery takes 2 time units. The property is specified by the CPL property shown in Figure 15.

```
1 smallDistance = sendSensorDistance.getParameter(distance) < 2
2 inBraking = brakeS.in(brakeS.getState(brakingSM, braking))
3 brakingInTime = Time.lasted(smallDistance && ! failure, 45)
4                 => Time.withinP(inBraking, 45)
5 property1 = Time.alw(brakingInTime)
6 Corretto.verify(property1)
```

Fig. 15: Property for the CCAS System.

Corretto transforms the diagrams together with the property to feed ℤot transparently. ℤot, in turn, determines in around 3 minutes that the property does not hold for the CCAS system, and it produces as counterexample a trace of the system that violates the property. The trace is shown in Figure 14. By navigating the trace, and looking at the corresponding UML elements, the user can understand that the CCAS does not enter the warning state immediately when the distance becomes less than 2 meters, but only when it receives message notifyDistance from the bus and the distance is less than 2. If we change the time constant in the property from 45 to 52 time units (i.e., 520ms), ℤot is able to show in around ten minutes that the new property holds.

This example shows how C-UML and Corretto help compose the behavior of four different types of UML behavioral diagrams (Sequence Diagrams, State Machine Diagrams, Activity Diagrams, and Interaction Overview Diagrams) and prove a non-trivial property.

[13] All the models and their verification results are available from the Corretto repository [Motta et al. 2017].
[14] www.mades-project.org

6.2. Automated Teller Machine

As a second example, we use the automated teller machine (ATM) introduced for the OMEGA2 project [Ober and Dragomir 2010]. Both Corretto and OMEGA2 propose approaches to model and verify UML models. This section aims to show how Corretto is flexible enough to model the same class of systems targeted by the OMEGA2 project and to overcome some of its limitations.

The ATM must allow the user to withdraw cash. The user is authenticated through her ATM card and personal identification number (PIN). If the ATM determines that the customer's PIN is invalid, the customer is required to re-enter the PIN. If the authentication succeeds, the ATM card is retained by the machine until the transaction is completed. The user can make a cash withdrawal from any suitable account linked to the card. Each withdrawal needs to be approved by the bank. The ATM must interact with the bank to obtain the approval. The transaction is considered complete by the bank once approved or disapproved.

OMEGA2 models the system by using a Component Diagram, while we use a Class Diagram to mimic the relevant information.[15] The Class Diagram models the main entities of the system at a high level of abstraction, like Bank, ATM, and Controller, which coordinates the interaction between ATM and Bank. The behavior of the objects of each class is described through a dedicated State Machine Diagram. The complete model comprises one Class Diagram, one Object Diagram, and seven State Machine Diagrams. Here we only show the State Machine Diagram of class Controller (Figure 16).

If we exclude the syntax used for the transitions, which is tool-specific, this diagram is equivalent to the one built by OMEGA2 [Ober and Dragomir 2010]. The meaning of the diagram is fairly intuitive. The user must insert a card into the CardReader, then she has to insert the PIN, and finally she must select the amount of money she wants to withdraw. When the PIN is wrong or the requested amount of money is not available, the Controller returns to the idle state.

Figure 17 shows the CPL property being checked against the model. It states that the amount of money provided by the ATM is always equal to the one requested by the user. This is equivalent to the following: if operation cd_releasemoney of CashDispenser is invoked, then parameter amount_released is equal to the amount of money requested by the user. This is specified by attribute amount of class User, initialized to 50 for this experiment. $\mathbb{Z}$ot takes 30 seconds to verify this property.

Comparing OMEGA2 and Corretto, OMEGA2 allows one to automatically generate the IF semantics [Bozga et al. 2004] of the UML model[16] and the IF command line tool can be used to inspect the IF model together with its property. While this process is definitely feasible for an expert of the IF language, it is not guaranteed that an average UML user would be able to perform the verification procedure on her own. Also, Corretto translates the UML model into the underlying formal language TRIO, but the verification procedure is performed in the background and the result is shown to the user by means of the traceability tool (see Figure 14). Finally, OMEGA2 requires that the property be expressed by means of a special type of State Machine Diagram, whereas Corretto uses CPL declarations introduced through a UML constraint.

To summarize, both Corretto and OMEGA2 can verify UML models and the properties can be expressed in a UML-like notation. OMEGA2, however, only supports a single UML behavioral diagram, State Machine Diagrams, and the verification proce-

[15]Since C-UML does not support Component Diagrams, we use a Class Diagram and we abstract components into objects. This is clearly not correct in a complete design model, but it is enough for verifying the system.

[16]IF is the target formal language of the OMEGA2 tool.

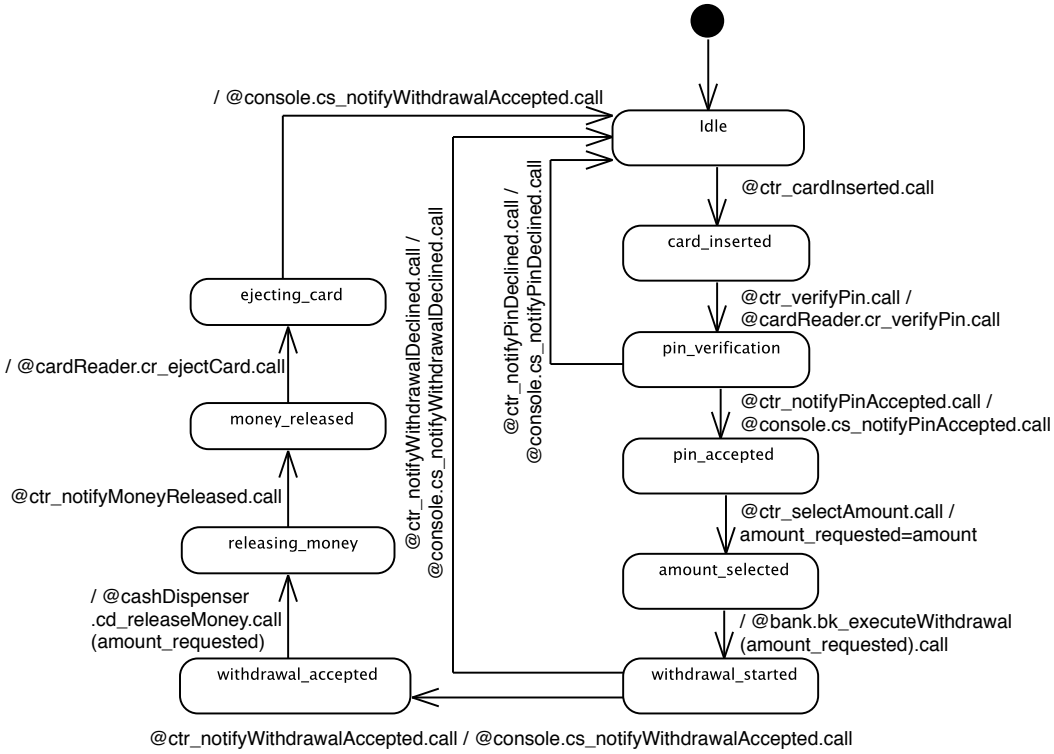


Fig. 16: State Machine Diagram for the Controller of the ATM System.

```
releaseMoney = cashdispenser^cd_releaseMoney()
Corretto.verify(Time.alw(releaseMoney
                         => cashdispenser::cd_releaseMoney.amount_released == 50))
```

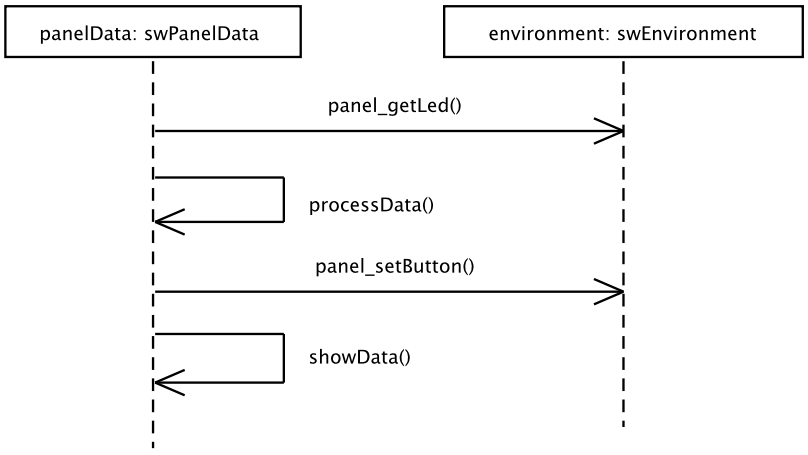
Fig. 17: Property for the ATM System.

dure is not transparent to the user. Corretto overcomes these limitations by supporting a wider set of UML diagrams and provides a traceability tool to let the user analyze the results provided by the verification engine without any knowledge of the formal language, and without changing mindset.

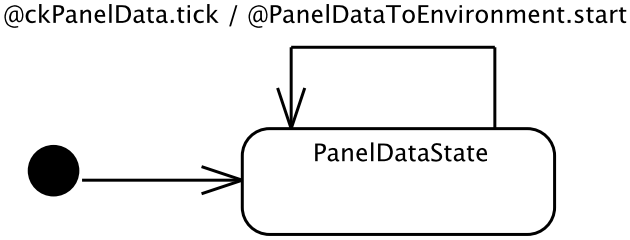
6.3. Radar System

Corretto has been used in the context of the MADES project also for the verification of two example Radar Systems, one on an airplane and one on the ground, provided by two industrial partners. In this article, we present the onboard system, and more precisely a component that carries out the delivery of the flight data from the environment to the User Interface (UI) of the pilot (the ground-based radar system is described in [Baresi et al. 2015]). Such a delivery is performed by a number of periodic tasks. There are four types of tasks: swPanelData, swRadarData, swNavData, and swWeatherData. The original model contains one object for each task type. Each task is in charge of reading one data type and of delivering it on time to the pilot UI, which is captured by class swMainMMI. To capture the communication between the tasks delivering the data and the MMI, a class swEnvironment is introduced, which acts as an intermediary

between senders and receivers. The C-UML model is made of one Class Diagram with five clocks, five Sequence Diagrams, and five State Machine Diagrams. The different Sequence Diagrams illustrate how the data are read and processed by the different periodic tasks. For example, the Sequence Diagram of Figure 18(a) shows how a pro-



(a)



(b)

Fig. 18: Radar System: (a) Sequence Diagram PanelDataToEnvironment that shows the radar's periodic behavior to retrieve and process the data. (b) State Machine Diagram of class swPanelData that regulates the period of Sequence Diagram PanelDataToEnvironment using the C-UML notation.

cess of type swPanelData interacts with an object of type swEnvironment to retrieve the data to be displayed on the pilot UI (i.e., swMainMMI). More precisely, swPanelData invokes operation panel_setButton on the swEnvironment object. A dual Sequence Diagram, not shown here for brevity, depicts that the swMainMMI object invokes operation main_panel_getButton on the same swEnvironment object to retrieve the data. The different State Machine Diagrams in the model are used to regulate the periods of the

```
setButtonCall = environment^panel_setButton()
getButtonCall = environment^main_panel_getButton()
Corretto.verify(Time.alw(setButtonCall => Time.withinF(getButtonCall, 15)))
```

Fig. 19: Property for the Radar System.

Sequence Diagrams according to the clocks defined in the Class Diagram. For example, the State Machine Diagram of Figure 18(b) has only one transition that is executed each time clock ckPanelData ticks. At that time Sequence Diagram PanelDataToEnvironment—shown in Figure 18(a)—is activated. All the tasks of reading and delivering a data type share the same resource, swEnvironment, and must wait if the resource is not available. Such waiting time could affect the responsiveness of the system. Corretto was used to prove that between messages panel_setButton in Figure 18(a) and main_panel_getButton, which are representative of the data retrieval cycle, there is always a time window smaller than 15 seconds. The corresponding property is captured by the statements of Figure 19.

This example was used to analyze how Corretto behaves when the number of objects that perform a periodic task increases. We increased the number of objects from one up to seven for each task type, and thus we obtained a maximum of 28 task objects. Figure 20 shows the results for two kinds of checks: verification that the model is indeed satisfiable (SAT), that is, that it is not inconsistent; and verification that the model satisfies the property presented in Figure 19, and the expected outcome is UNSAT, that is, it is not possible for an execution of the model to violate the property. As expected,

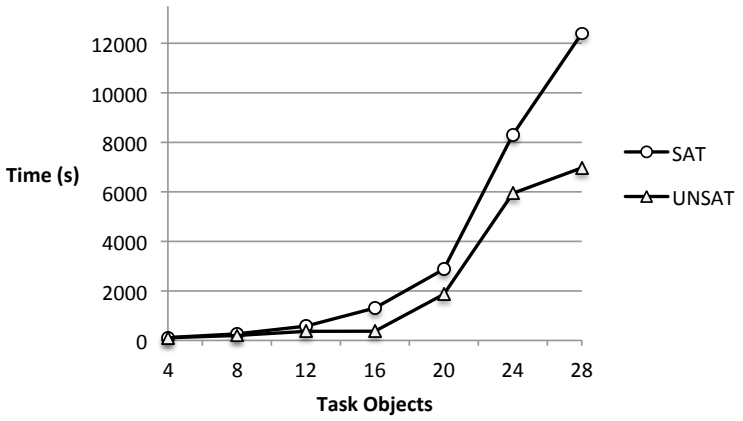


Fig. 20: Radar System: Verification times for the consistency check (SAT outcome) and property verification (UNSAT outcome).

the verification times for both the SAT and UNSAT cases increase significantly with the number of objects. For small models (4 and 8 task objects), the verification takes less than 4 minutes. On the other hand, the verification of the biggest radar model (28 task objects), which yields SAT as result, took some 4 hours.

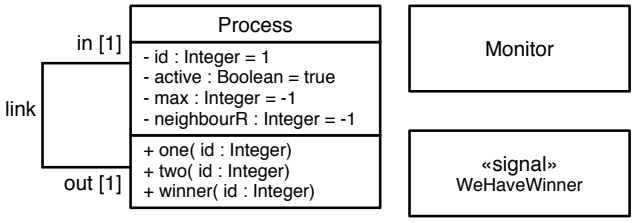
6.4. Leader Election

In this section we present the verification of a well-known algorithm for the election of a leader in a group of processes. More precisely, given a circular arrangement of n

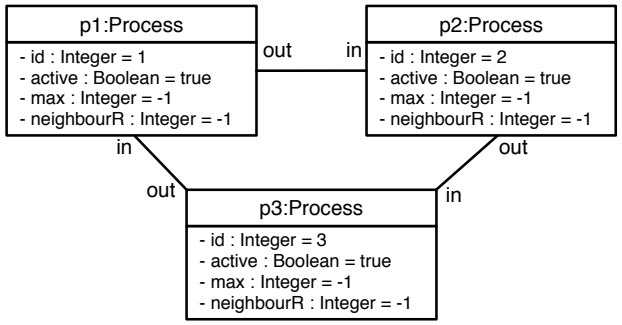
uniquely numbered processes, it determines the one with the highest identifier in a distributed manner.

The algorithm has been discussed in several research papers and it has been used to test the scalability of different verification tools. This article uses the version described in [Dolev et al. 1982] and implemented in Promela, the input language of the Spin model checker.[17] This algorithm has become a kind of benchmark to evaluate the performance and scalability of verification tools.

The algorithm can be rendered in UML with one Class Diagram and two State Machine Diagrams. The Class Diagram of Figure 21(a) comprises class Process, which models the processes that take part to the election. The ring is oriented and each



(a)



(b)

Fig. 21: Leader Election: Class Diagram (a) and Object Diagram (b).

process has two distinct association ends named in and out. Attribute id states the id assigned to the process. The process with the highest id wins the election. Attribute active is equal to *true* if the process is still participating in the election process, *false* otherwise. Attribute max contains the id of the current winner. In the initialization phase, each process assumes to be the leader in the ring by initializing max to its own id. Finally, attribute neighbourR contains the id of the process next to the current one that is still active. Each process offers three methods: method one is invoked by the neighbor process to communicate the id of the first active process next to the receiver.

[17]The Promela model of the leader election algorithm is available at http://spinroot.com/spin/whatispin.html.

Similarly, method two is invoked by the neighbor process to communicate the id of the second active process next to the receiver. Finally, method winner is invoked to communicate the winner of the election. The Object Diagram of Figure 21(b) shows an example ring with the state of three processes just after the initialization.

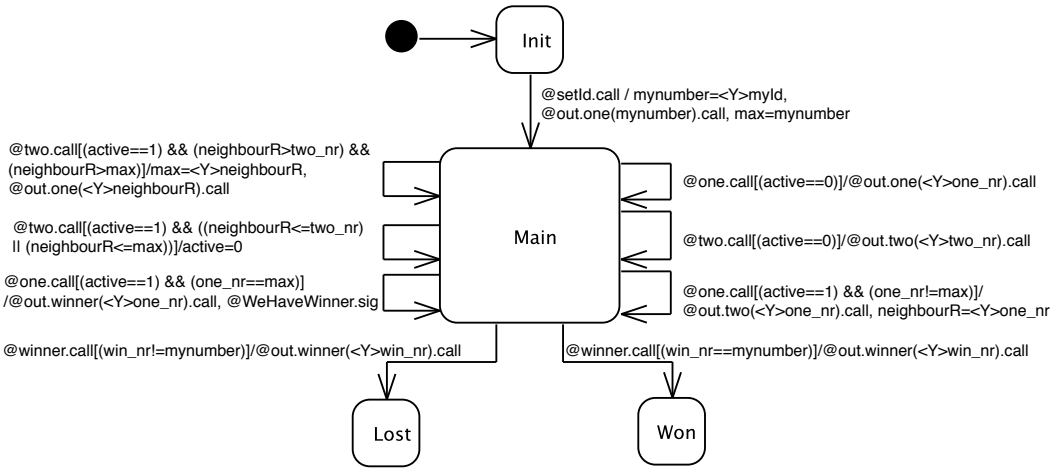


Fig. 22: Leader Election: State Machine Diagram of class Process.

As captured by the State Machine Diagram associated with class Process, shown in Figure 22, each process in the ring starts by invoking method one(id) on its neighbor following association out. The receiver saves the id of the neighbor and forwards it to the next process in the ring by invoking method two(id) on it. Each time an active process knows the ids of the two preceding neighbors, it compares them against max. If neighbourR is the highest of the three values, then max is assigned with the value of attribute neighbourR. If not, the process removes itself from the ring by setting attribute active to 0 and forwards the id of its first neighbor to the next process in the ring by calling one(neighbourR). Following these rules, the process that remains active and receives a message for one(neighbourR), where neighbourR is equal to max, knows the winner and can trigger signal WeHaveWinner shown in the Class Diagram. Class Monitor is just a helper class that switches to state Winner if a winner has been detected, and to state Error if the winner signal has been triggered more than once.

```
1 stateEnd = idGenerator.getState(IdGenerator_SM, end)
2 inStateEnd = idGenerator.in(stateEnd)
3 stateWinner = monitor.getState(Monitor_SM, state_Winner)
4 inStateWinner = monitor.in(stateWinner)
5 Corretto.verify(Time.alw(Time.somF(inStateEnd) => Time.som(inStateWinner)))
```

Fig. 23: Property $P1$ for the Leader Election.

We used Corretto to check that the model is satisfiable and to verify two properties, namely $P1$ and $P2$, whose CPL definitions are shown in Figure 23 and Figure 24, respectively. $P1$ checks whether the algorithm is always able to find a winner. $P2$ checks that the monitor never goes into state Error. These properties were verified with an increasing number of processes: 8, 10, 12, 14, and 15. For 15 processes, we were not able to finish the verification procedure in 11 hours, which was our threshold. Clearly

```
1 stateError= monitor.getState(monitor_SM, State_Error)
2 inStateError = monitor.in(stateError)
3 Corretto.verify(Time.alw(! inStateError))
```

Fig. 24: Property $P2$ for the Leader Election.

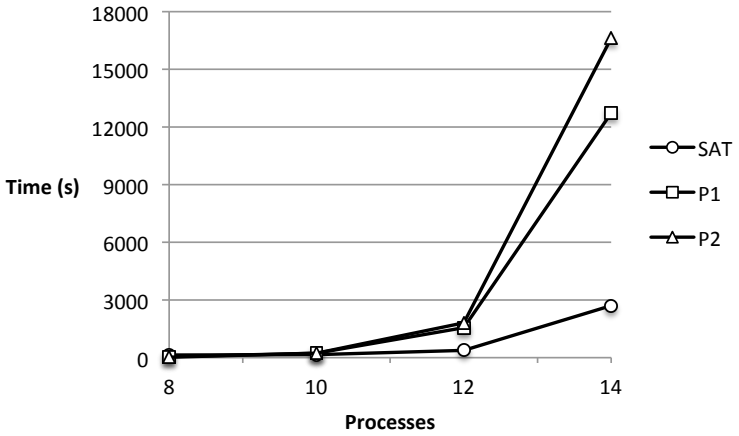


Fig. 25: Leader Election: Verification times for the consistency check (SAT outcome) and for the verification of properties $P1$ and $P2$.

the verification of the original Promela model in Spin is faster than the verification of our C-UML model with Corretto, and can consider more processes. However, the goals are completely different: the Promela model was created by hand by experts, and the goal was to conceive something optimized for verification. Our goal is simply to demonstrate the feasibility of the formal verification of high-level, verbose UML models. Solutions to shrink the size of generated models, and speed up the verification, are not part of this work and will be addressed in the future.

6.5. Fischer Protocol

The Fischer protocol is a well-known mutual exclusion protocol designed for n processes. It is a timed protocol where the concurrent processes check for both a delay and their turn to enter the critical section using a shared variable (*ID*). The protocol has been widely used to test the scalability of different verification approaches. Here we present the version described in the *Uppaal* tutorial [Behrmann et al. 2006]. When considering the reported figures the reader should keep in mind that Uppaal uses timed automata as the underlying formalism, whereas in this section we describe our C-UML representation.

The C-UML model of the protocol comprises one Class Diagram and one State Machine Diagram. The Class Diagram contains a single class Process with one (local) attribute pid and two static (i.e., global) attributes ID and COUNTER. The State Machine Diagram (Figure 26) comprises five states. Each process in the system starts competing for the critical section non-deterministically by moving from state FischerP to state Req. Each process is allowed to remain in state Req at maximum $a = 3$ time instants before moving to state Updated. After this move, the process assigns attribute ID to its own pid. After exactly $d = 4$ time instants the process moves to state Wait. At this point, if the value of attribute *ID* is still equal to the process pid, then the process

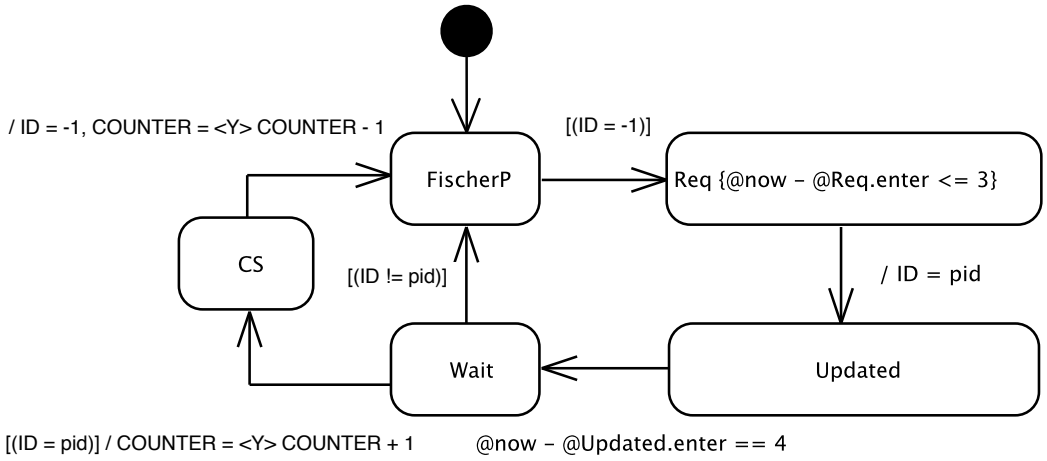


Fig. 26: State Machine Diagram of the Fischer Protocol.

is allowed to enter the critical section, represented by state CS. If not, it must restart the above process by moving to state FischerP. The key feature of the protocol is the relation between parameters a and d. In this simplified version, the protocol works as long as $a < d$. Each process that enters the critical section increments attribute COUNTER, whereas each process exiting the critical section decrements it.

Firstly we used Corretto to check that the system is satisfiable (i.e., consistent). Figure 27(a) shows the performance for this check with an increasing number of processes competing for the critical section. Secondly, we used Corretto to prove that at maximum one process is allowed in the critical section at any given time instant. This is equivalent to checking that attribute COUNTER is always less than or equal to one, which in turn corresponds to verifying the property `Corretto.verify(Time.alw(process.counter` $\leq$ `1))`. Figure 27(b) shows the performance for this check. In this case, the verification terminated for systems made of up

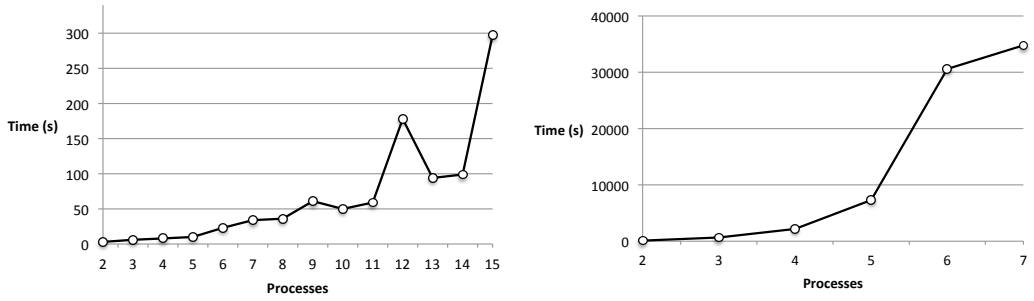


Fig. 27: Fischer Protocol: Verification times for (a) consistency check (outcome SAT) and (b) property verification (outcome UNSAT).

to 7 processes, with a maximum verification time of 10 hours. Again, since the aim of Corretto is not to study smarter and faster verification algorithms, but to demonstrate the feasibility of the formal verification of UML—or domain-specific—models, we believe these results are interesting and pave the ground to a wider adoption of formal methodologies in practice.

6.6. Summary of the Experimental Evaluation

The C-UML models introduced in the previous sections come from different sources, with different perspectives and modeling approaches. As a consequence, overall they use very heterogeneous C-UML elements and diagrams to capture the structure and behavior of the target systems. On the one hand this shows the flexibility and adaptability of our modeling and verification approach, which can handle very different modeling styles. On the other hand, it hampers the comparison of the various models size-wise, since different models use different diagrams in varying mixtures (some rely more on Sequence Diagrams, other more on State Machine Diagrams, etc.).

This section suggests a comparison of the verified models based on a rather fine-grained view of the C-UML elements involved in them, in order to give a rough idea of the size of the models that can be analyzed through Corretto within a reasonable amount of time.

Table I shows the number of modeling elements used in the biggest verified model in each of the previous sections. The element count was automatically produced by

Table I: Number of C-UML elements involved in the models of case studies.

	CCAS	ATM	Radar System	Leader Election	Fischer
SM_States	5	24	29	61	90
SM_Transitions	9	51	58	145	120
SD_Lifelines	7	0	58	0	0
SD_Messages	5	0	137	0	0
SD_Parameters	1	0	0	0	0
SD_Time Constraints	2	0	0	0	0
OD_Objects	5	7	35	16	15
OD_Clocks	1	0	5	0	0
IOD_Elements	6	0	0	0	0
AD_Elements	10	0	0	0	0
Arithmetic Variables	5	7	0	126	17
Total	56	89	322	348	242

Corretto, and it includes all the key elements for each type of diagram. More precisely, it keeps track of the number of states and transitions in State Machine Diagrams, the number of lifelines, messages, parameters and time constraints in Sequence Diagrams, the number of objects and clocks in Object Diagrams, the number of elements in Interaction Overview Diagrams and Activity Diagrams, and finally the number of arithmetic variables (integer- or real-valued). The last row of Table I shows the total number of elements for each model, and can be used as a rough measure of the size of models. We always count the number of states and transitions in the *instances* of State Machine Diagrams and Activity Diagrams in order to keep this measure more realistic. For example, the C-UML model of the Fischer protocol contains only one State Machine Diagram that has 6 states, but in the biggest analyzed instance of this model there are 15 instances of this State Machine Diagram in the corresponding formal model (one for each object), hence accordingly we count 90 states.

Note that in comparing two models, it is not always the case that the model with fewer C-UML elements is easier to verify. A single element, or a combination of few elements, can put a significant burden—or has no impact—on the verification process depending on the type of element and the resulting TRIO formula. Estimating the complexity of the verification process from the sheer number of C-UML elements is not straightforward. For example, adding a time constraint with a time window of K time units can be much heavier than adding a transition that imposes an ordering between two events with a time difference of one instant. It can be reasonably expected that metric time constraints, which appear in Sequence Diagrams as stand-alone elements, and also as part of transitions in State Machine Diagrams, are the most cumbersome to deal with. For example, the model of the Fischer protocol includes many transitions with metric time constraints and many states with invariants that are also temporal constraints, and it is in fact the heaviest to check. Given the indirect nature of the relationship between the number of C-UML elements and the strain of the verification process, the maximum *Total* in Table I is important to give the reader and idea of the maximum size of the models we verified and also of their complexity in terms of UML elements. These numbers must always be associated with the UML element type they refer to; they cannot be interpreted as simple syntactical upper-bounds on the size of verifiable models: bigger models, but with verification-wise lighter elements could be managed properly.

7. RELATED WORK

Many researchers have addressed the problem of providing a precise semantics to part of UML. Most of the works only target a subset of the UML diagrams that are important for the specific domain of interest and no absolute definition of correct/incorrect semantics exist: The UML specification itself is imprecise [Harel 2004]; and many works propose their own semantic variations that are aimed at a particular domain. Even if one focused on a single diagram type, the number of supported operators varies, together with their interpretation.

The different proposals in the literature can be divided in two groups: (a) those that ascribe a semantics to UML as a purely theoretical exercise to reveal inconsistencies of the UML specification (e.g., [Störrle 2003]), and (b) those that do not only ascribe a semantics to UML, but also propose tools to formally verify the models (e.g., [Diethers and Huhn 2004]). Focusing on the types of diagrams taken into account provides a further dimension to our analysis: (a) the first set contains the solutions that only deal with structural views of the system; (b) the second group comprises the works that specifically address the behavior of UML models; and (c) the third group deals with the works that try to ascribe a comprehensive interpretation to UML by means of a common-ground semantics.

The works that deal with the structural view of the system usually concentrate on the satisfiability of OCL constraints over structural diagrams. Some well-known examples are the approaches presented in [Cabot et al. 2014; Gogolla et al. 2014; Ahrendt et al. 2005]. Our work does not take into account this aspect of the verification, as we concentrate on the verification of the behavioral models, which is orthogonal to the verification of OCL constraints. As soon as the OCL constraints on the structure of the UML model are satisfied, the user can analyze its behavior.

While there seems to be a general consensus on (a large portion of) the static semantics of UML, the dynamic semantics of its behavioral diagrams, such as activity diagrams, interaction overview diagrams, and state machines, is still an open problem [Broy and Cengarle 2011]. The works that address the behavior of UML models only focus on a limited number of diagram types (oftentimes, they only consider a single type) and neglect the interdependencies with the other diagrams of a system. Nota

that while Section 7.1 presents these approaches in detail, our approach moves a step further by allowing users to verify models composed of multiple heterogeneous behavioral diagrams.

Finally, a number of works tried to build a common semantic ground to integrate the different UML structural and behavioral views. For example, Kuske et al. in [Kuske et al. 2009] show how a central part of UML can be integrated into a single visual semantic model (graphs) to support the visual simulation of integrated specifications. Posse et al. [Posse and Dingel 2016] translate UML-RT (a UML profile for real-time and embedded systems) models into the *kiltera* language, that is a real-time extension of the π-calculus. Their translation tool produces *kiltera* code, and covers Capsule Diagrams, one kind of structural diagrams, and State Machine Diagrams, one kind of behavioral diagrams. However, not only the work focuses on a very limited subset of diagrams, but it does not provide mechanisms for the formal verification of the considered diagrams.

Broy et al. [Broy et al. 2014] present a ground mathematical semantics, called *system model*, to integrate the different views. The system model defines a universe of interacting state machines that describe the behavior of the objects and their interrelationships. In a sense, this is akin to the approach of fUML [OMG 2016], which has been presented in Section 2. Compared to Kuske [Kuske et al. 2009] and Broy [Broy et al. 2014], we do not focus on the structure of the UML models (we only consider Class and Object Diagrams), but on their behavior, and we aim at a complete verification tool, developed incrementally to support a growing number of UML elements, with the underlying metric temporal logic as enabler.

In a similar vein, the GEMOC initiative [Combemale et al. 2014] has released GEMOC Studio,[18] which supports mechanisms for defining a common semantics for heterogeneous languages, with a focus on their executability. The C-UML approach focuses, instead, on formal verification purposes. Indeed, our TRIO-based approach could be the basis for a common semantics of heterogeneous notations that enables the verification of combined models, much like Corretto currently allows for the formal verification of a combination of UML diagrams.

7.1. UML Behavioral Models

Table II and Table III summarize the works that focus on behavioral diagrams. Most of the approaches surveyed in this section focus on the analysis of models. Those that also consider advanced object-oriented features are explicitly identified in Table III.

Table II presents the diagrams each work supports: Class Diagrams (CDs), Component Diagrams (CpDs), Communication Diagrams (CommDs), Sequence Diagrams (SDs), Activity Diagrams (ADs), State Machine Diagrams (SMs) and OCL. One star (*) says the work only supports the basic operators of the diagram type; two stars (**) mean that the work supports most of the operators; and three stars (***) that the work supports all the operators. The various works have been divided in different sets according to the main supported diagram types. For the works that support multiple diagrams a separate *multiple diagrams* group is created. Note that many works use State Machine Diagrams to describe the behavior of the system, and Sequence Diagrams to specify the requirements that must be satisfied by the State Machine Diagrams. In these cases, there is only one diagram type that contributes to the specification of the behavior (State Machine Diagrams), while the other is used to check the desired properties (Sequence Diagrams). This is why some approaches are not included in the *multiple diagrams* group even if in the table they are associated with more than one diagram type.

[18] gemoc.org/studio

Table III presents additional details about the technology underlying each solution. Column *Domain* specifies the semantic domain assigned to the diagrams. The remaining columns are reserved to the works that provide the user with an analysis tool, that is, everything that can help the user understand the behavior of the UML model. We identified three kinds of analysis: *simulation (S)*, *automated (A)*, and *manual (M)*. Simulation tools animate UML models according to the specified semantics. Automated tools provide the means to automatically analyze properties of the system. Manual tools provide the instruments to conduct a formal proof on top of the semantics associated with the UML models. The *CASE*-labeled column indicates whether the tool is integrated into a Computer-Aided Software Engineering environment (for example, Eclipse) or if an XMI import is available therefore leaving the choice of the UML tool to the user. For the model checking tools, the column *Properties* specifies whether it is possible to express the property to be checked using some UML-like or equivalent high-level formalism, while column *Results* specifies if it is possible to map the results of the model checking analysis onto the UML model. In other words, the user must be able to understand the results of the analysis without being an expert of the underlying notation and tools. Column *AdvOO* specifies whether the approach also supports advanced object-oriented features such as object creation/destruction (x marker), and/or code generation (* marker).

	CDs	CPDs	CMDs	SDs	ADs	SMs	OCL
Sequence Diagrams							
[Störrle 2003]				***			
[Lund and Stolen 2006]				***			
[Cengarle and Knapp 2004]				***			
[Hammal 2006]				***			
[Eichner et al. 2005]				***			
Activity Diagrams							
[Störrle 2004]					***		
[Eshuis 2006; Eshuis and Wieringa 2004]					***		
[Bouabana-Tebibel 2009]				**	**		
[Apvrille et al. 2004]	**				**		
[Daw et al. 2015]					**		
State Machine Diagrams							
[Knapp and Wuttke 2006]	**			***		***	
[Paltor and Lilius 1999]	**		***			***	
[Damm et al. 2005]	**					***	
[Diethers and Huhn 2004]	**			**		**	
[Hammal 2005]						***	
[Ober and Dragomir 2010]	**	**				***	
[Hansen et al. 2010b]	**					***	
[Bouabana-Tebibel 2007]	*			*		**	**
[ter Beek et al. 2011]	**					***	
[Burmester et al. 2004]		**				***	
[Pap et al. 2005]						***	**
[Kyas et al. 2005]	**					**	**
Multiple Diagrams							
[Lano 2009]	**			**		**	**
[Graw and Herrmann 2004]	**			**	**	**	
Corretto	**			**	**	**	

Table II: Different semantics (part I).

	Domain	Tool	Type	CASE	Properties	Results	AdvOO
Sequence Diagrams							
[Störrle 2003]	Math						
[Lund and Stolen 2006]	Math						
[Cengarle and Knapp 2004]	Math						
[Hammal 2006]	Graphs						
[Eichner et al. 2005]	Petri net	P-UMLaut	S				
Activity Diagrams							
[Störrle 2004]	Petri net	NuSMV	A	x		x	x
[Eshuis 2006; Eshuis and Wieringa 2004]	LTS						
[Bouabana-Tebibel 2009]	Petri net	PROD	A		x		
[Apvrille et al. 2004]	Process Alg.	RT-LOTOS	A	x			x
[Daw et al. 2015]	PROMELA	UPPAL, SPIN, NuSMV, PES	A	x	x		
State Machine Diagrams							
[Knapp and Wuttke 2006]	PROMELA	SPIN	A		x		
[Paltor and Lilius 1999]	PROMELA	SPIN	A	x	x		
[Damm et al. 2005]	STS	dSPIN	A	x	x		x
[Diethers and Huhn 2004]	T. Automata	UPPAAL	A	x		x	
[Hammal 2005]	Petri net						
[Ober and Dragomir 2010]	T. Automata	IF-Toolkit	A	x	x		x
[Hansen et al. 2010b]	mCRL2	LTSmin	A	x	x		x
[Bouabana-Tebibel 2007]	Petri net	PROD	A		x		
[ter Beek et al. 2011]	LTS	UMC	A				
[Burmester et al. 2004]	T. Automata	UPPAAL	A	x		x	
[Pap et al. 2005]	Kripke Str.	SPIN	A	x			
[Kyas et al. 2005]	HOL	PVS	M				x
Multiple Diagrams							
[Lano 2009]	Action Logic	B-Toolkit	A				x*
[Graw and Herrmann 2004]	cTLA	TLC	A				x
Corretto	T. Logic	ZOT	A	x	x	x	

Table III: Different semantics (part II).

The following subsections group the different works with respect to the main diagram types they refer to. The description is not exhaustive with respect to the elements listed in Table II, as we focus on the types of behavioral—rather than structural—diagrams each approach relies on.

7.1.1. Sequence Diagrams. The work by Störrle [Störrle 2003] defines a partial order semantics for plain interactions, and also for combined fragments compliant with the OMG specification. Stolen et al. [Lund and Stolen 2006; Haugen et al. 2005b; Haugen et al. 2005a] present a similar work on Sequence Diagrams and propose an operational semantics for this diagram type. The semantics is defined by means of functions over an abstract syntax. This semantics is used to implement a tool for the translation of Sequence Diagrams into the Maude language, which in turn is the basis for a test generation tool for Sequence Diagrams. Cengarle and Knapp [Cengarle and Knapp 2004] propose a denotational semantics for Sequence Diagrams. Compared to the previous ones, they provide a detailed analysis of the *neg* operator. The work presented by Hammal [Hammal 2006] proposes a semantics based on a branching time structure for Sequence Diagrams. The branching time structure is represented by means of graphs annotated with time information. This is the enabler for timeliness and performance analysis. Eichner et al. [Eichner et al. 2005] present a translation from Sequence Diagrams to Petri nets. The advantage of using such a semantics is that it is defined compositionally, founded on basic Petri net composition operations. On the other hand, it cannot deal with the time dimension of the system, and it cannot deal with object-oriented features. The authors also present a tool called P-UMLaut that simulates the behavior of Sequence Diagrams according to the provided Petri net semantics.

7.1.2. Activity Diagrams. The work by Störrle [Störrle 2004] defines the semantics of Activity Diagrams by means of Petri nets. The work is characterized by strict conformance with the OMG specification, but no automated tool is provided and also nothing is said about other diagram types (e.g., the integration with the semantics for Sequence Diagrams by the same author [Störrle 2003]). The approaches presented by Eshuis and Wieringa [Eshuis 2006; Eshuis and Wieringa 2004; Eshuis and Wieringa 2001] ascribe semantics to UML Activity Diagrams at two different levels of abstraction—requirements level and implementation level—through suitable translations into Labeled Transition Systems (LTS), using NuSMV as verification engine. The path that corresponds to the counterexample returned by NuSMV is highlighted in the Activity Diagram. Bouabana-Tebibel [Bouabana-Tebibel 2009] proposes a semantics for Interaction Overview Diagrams—which can be considered a particular variation of Activity Diagrams where nodes are Sequence Diagrams—based on hierarchical Petri Nets (HPN). A tool is provided to translate the Interaction Overview Diagrams into the input language of PROD, a model checker for HPN. The user must then write an LTL formula in PROD to enact the verification phase. Apvrille et al. [Apvrille et al. 2004] present the TURTLE framework that formalizes an extension of Class Diagrams and Activity Diagrams specified by the TURTLE UML profile. The semantics is given in terms of a particular process algebra, called RT-LOTOS. The framework is supported by a CASE tool and the properties must be expressed using formulae over the reachability graph produced by the formal specification. Daw et al. [Daw et al. 2015] present UML Verification Tool (UML-VT), that is implemented as an Eclipse-plugin. UML-VT automatically translates Activity Diagrams to input the model checkers UPPAAL, SPIN, NuSMV, or PES. It allows the user to select the target model checker to be used in the verification.

7.1.3. State Machine Diagrams. Most of the works that formalize State Machine Diagrams [Schfer et al. 2001; Knapp and Wuttke 2006; Paltor and Lilius 1999; Lilius

and Paltor 1999; Damm et al. 2005; Damm and Westphal 2003; Diethers and Huhn 2004; Hammal 2005] also support Sequence Diagrams to check whether the interaction shown in the Sequence Diagrams can be satisfied by the behaviors described in the State Machine Diagrams. Therefore Sequence Diagrams (or Communication Diagrams) are used as a property-specification mechanism and do not contribute to the possible behaviors of the system. The approach presented by Knapp et al. [Schfer et al. 2001; Knapp and Wuttke 2006] translates State Machine Diagrams and Sequence Diagrams into the input language of UPPAAL, into PROMELA for SPIN, the language of the theorem prover KIV, and into Java and SystemC code.

Paltor and Lilius [Paltor and Lilius 1999; Lilius and Paltor 1999] present the vUML tool to verify Class Diagrams, State Machine Diagrams and Communication Diagrams using SPIN [Gerard J. Holzmann 2004]. Damm et al. [Damm et al. 2005; Damm and Westphal 2003] give semantics to rtUML, a subset of UML that includes Class Diagrams and State Machine Diagrams. The semantics is given by using symbolic transitions systems (STS). Properties are expressed using Live Sequence Charts, and this is similar to the idea of using Sequence Diagrams, but with greater expressiveness. A prototype of a discrete-time verification environment, integrated in the UML modeling tool Rhapsody, automates the verification. Diethers and Huhn [Diethers and Huhn 2004] propose the VOODUU framework, which supports Class Diagrams, State Machine Diagrams and Sequence Diagrams, formalized through timed automata, then analyzed by UPPAAL. A plugin for the Poseidon CASE tool eases the specification/verification process and displays on the Sequence Diagrams the counterexamples generated by UPPAAL (even if no details are provided). Hammal [Hammal 2005] proposes a semantics for State Machine Diagrams based on Petri nets. He suggests that the timed Petri net that corresponds to the State Machine Diagram can then be checked against the constraints imposed by a Sequence Diagram. It would be interesting to understand how this could be done by exploiting the semantics for Sequence Diagrams proposed by the same author [Hammal 2006], but unfortunately no further details are provided. Choppy et al. [Choppy et al. 2011] propose a translation of UML state diagrams into Colored Petri nets, and the verification of desired properties can be carried out automatically. The OMEGA framework by Ober and Dragomir [Ober and Dragomir 2010] addresses Class Diagrams, Component Diagrams, and State Machine Diagrams in terms of communicating extended timed automata, then translated into the IF language—supported by a number of verification tools. Hansen et al. [Hansen et al. 2010b; Hansen et al. 2010a] introduce xUML, which includes Class Diagrams and State Machine Diagrams. The semantics is given through the mCRL2 formal specification language. The system can then be model checked by using the LTSmin model checker.

Both Ober and Dragomir [Ober and Dragomir 2010] and Hansen et al. [Hansen et al. 2010b; Hansen et al. 2010a] express properties using a variation of State Machine Diagrams where the user can tag the error states. In [Hansen et al. 2010b; Hansen et al. 2010a], the result of the verification is also mapped back onto the Sequence Diagrams. Other approaches like those proposed by Bouabana-Tebibel [Bouabana-Tebibel 2007], ter Beek et al. [ter Beek et al. 2011], Burmester et al. [Burmester et al. 2004], and Pap et al. [Pap et al. 2005] map State Machine Diagrams onto Petri nets, LTS, timed automata, and kripke structures, respectively. Only Burmester et at. [Burmester et al. 2004] address the time dimension of the system, and none of them propose a UML-like notation for properties. Burmester et al. also support the traceability of the verification results, but it is only possible to verify properties belonging to a subset of the model. The authors define them as *local properties*.

Finally, Kias et al. [Kyas et al. 2005] present a higher-order logic semantics for Class Diagrams with OCL constraints and flat State Machine Diagrams. The obtained for-

mal model can be analyzed by means of the PVS theorem prover. This is complementary to our approach and the previous ones as it can be a viable solution when there is the need for navigating the details of the proof that demonstrates that a given property holds for the system.

7.1.4. Multiple Diagrams. The approach presented by Graw and Herrmann [Graw and Herrmann 2004] proposes a semantics based on a compositional Temporal Logic of Actions (cTLA). The work integrates Class Diagrams, Sequence Diagrams, Activity Diagrams and State Machine Diagrams, and the authors provide a transformation tool from XMI to cTLA. The work presented by Lano [Lano 2009] gives semantics to Class Diagrams, State Machine Diagrams, and Sequence Diagrams by using the Real Time Action Logic formalism (RAL). A transformation tool from UML to B is provided as a standalone Java program. The analysis must be performed using B Tools. Note that Lano [Lano 2009] and Graw and Herrmann [Graw and Herrmann 2004] are the only solutions that are able to consider heterogeneous models like us and their semantics is also based on temporal logic.

None of the works provide a tool for the analysis of Sequence Diagrams, while we support a broader range of diagram types and provide a tool to automatically analyze them and also to deal with the real-time dimension of systems. Following the guidelines of the UML/MARTE specification [MARTE 2011] we are able to predicate on the time dimension of the system using clocks and time constraints. We do not provide a simulation tool, but we can ask for execution traces of the system. We support the back translation of verification results on heterogeneous models, while Eshuis and Wieringa [Eshuis 2006] only support this for Activity Diagrams. Finally, our tool is integrated with the Papyrus Eclipse modeling environment, where the user can design the model and also express the properties of interest using an OCL-like notation. The traceability mechanism allows the user to understand verification results on designed UML models.

8. CONCLUSIONS

This paper introduces a UML-based technique for the modeling and formal verification of embedded, real-time systems. The technique is based on two main pillars: (i) a modeling notation, called C-UML, which uses UML diagrams and borrows from the UML MARTE profile to describe the features of the systems under design; and (ii) a verification tool, called Corretto, which translates C-UML models into an internal representation, expressed in the metric temporal logic TRIO, and then verifies them against user-defined properties. Our logic-based approach exhibits several valuable features, such as flexibility, customizability, and compositionality. The current version of C-UML comprises a significant set of widely-used UML diagram types; the addition of new diagrams (e.g., Component Diagrams) would require the use of the same conceptual framework and the proper identification of the links and inter-dependencies with the other diagram types. We have illustrated our approach through several case studies consisting of non-trivial examples from industrial applications and classic examples from the literature on formal verification.

The C-UML approach is currently being considered to provide formal verification capabilities to the design of data-intensive applications within the DICE project[19]. In the future, we will explore mechanisms to optimize the verification phase in the Corretto tool.

[19]http://www.dice-h2020.eu/.

In addition, we will investigate the possibility of founding the semantics of C-UML on a more general model of time. In fact, as mentioned in Section 4, C-UML models currently employ a discrete notion of time where state changes, operation invocations, activations of Sequence Diagrams, etc., all consume a time unit. It is sometimes useful, however, to describe system changes that occur at speeds that are much faster than others, by separating between micro- and macro-steps, where the former capture logical, instantaneous evolutions in the system and the latter the advancement of real time (see for example [Lee and Seshia 2011], Chapter 6). The TRIO language, which provides the formal foundations to C-UML, can indeed support such a finer notion of time, even in a metric setting [Rossi et al. 2016], although at the price of making models more costly to analyze.

Another way to introduce a finer-grained model of time is through the adoption of a continuous—rather than a discrete—notion of time. The Zot bounded satisfiability checker which is used to carry out the formal verification of C-UML models supports continuous-time models in several ways: through the notion of sampling [Furia and Rossi 2010; Furia et al. 2008], which can be used to integrate in the same framework discrete and continuous time models—as done in the predecessor to the Corretto tool for purposes of simulation of system designs [Baresi et al. 2015]; and through the CLTLoc temporal logic [Bersani et al. 2015], which employs a metric, continuous notion of time, equivalent to the one adopted in timed automata [Bersani et al. 2017]. In fact, in future works we will explore the possibility of providing new, complementary semantics for C-UML diagrams, based on the CLTLoc temporal logic.

Acknowledgments. We would like to thank the anonymous reviewers for their useful suggestions, which have helped us improve the paper.

REFERENCES

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H Schmitt. 2005. The KeY tool Integrating object oriented design and formal verification. *Software and Systems Modeling* (2005), 32–54.

Charles André. 2009. *Syntax and semantics of the clock constraint specification language (CCSL)*. Ph.D. Dissertation. INRIA.

Charles André, Frédéric Mallet, and Robert de Simone. 2007. Modeling Time(s). In *Model Driven Engineering Languages & Systems (MoDELS) (LNCS)*, Vol. 4735. 559–573.

Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. 2004. TURTLE: a real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering* 30, 7 (2004), 473–487.

Luciano Baresi, Gundula Blohm, Dimitrios S. Kolovos, Nicholas Matragkas, Alfredo Motta, Richard F. Paige, Alek Radjenovic, and Matteo Rossi. 2015. Formal verification and validation of embedded systems: the UML-based MADES approach. *Software and Systems Modeling* 14, 1 (2015), 343–363. DOI:http://dx.doi.org/10.1007/s10270-013-0330-z

Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. 2011. From interaction overview diagrams to temporal logic. *Models in Software Engineering* 6627, i (2011), 90–104. http://www.springerlink.com/index/3V4348H45N0M2125.pdf

Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. 2012. A Logic-based Semantics for the Verification of Multi-diagram UML Models. *SIGSOFT Softw. Eng. Notes* 37, 4 (July 2012), 1–8. DOI:http://dx.doi.org/10.1145/2237796.2237811

Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. 2014. Flexible modular formalization of UML sequence diagrams. In *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering - FormaliSE 2014*. ACM Press, New York, New York, USA, 10–16. DOI:http://dx.doi.org/10.1145/2593489.2593492

Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. 2015. Efficient Scalable Verification of LTL Specifications. In *International Conference on Software Engineering (ICSE)*.

Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. 2016. How Bit-Vector Logic Can Help Improve the Verification of LTL Specifications over Infinite Domains. In *ACM Symposium on Applied Computing*.

Gerd Behrmann, Alexandre David, and Kim G Larsen. 2006. *A Tutorial on Uppaal 4.0*. Technical Report.

Abderraouf Benyahia, Arnaud Cuccuru, Safouan Taha, François Terrier, Frédéric Boulanger, and Sébastien Gérard. 2010. Extending the standard execution model of UML for real-time systems. In *Distributed, Parallel and Biologically Inspired Systems*. Springer, 43–54.

Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2017. A logical characterization of timed regular languages. *Theoretical Computer Science* 658, Part A (2017), 46–59. DOI:http://dx.doi.org/10.1016/j.tcs.2016.07.020

Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2015. A Tool for Deciding the Satisfiability of Continuous-time Metric Temporal Logic. *Acta Informatica* (2015), 1–36. DOI:http://dx.doi.org/10.1007/s00236-015-0229-y

Thouraya Bouabana-Tebibel. 2007. Roles at the basis of UML validation. *Journal of Computing and Information Technology* (2007), 171–183. DOI:http://dx.doi.org/10.2498/cit.1000882

Thouraya Bouabana-Tebibel. 2009. Semantics of the interaction overview diagram. In *IEEE International Conference on Information Reuse Integration (IRI)*. 278–283. DOI:http://dx.doi.org/10.1109/IRI.2009.5211565

Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. 2004. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems (SFM-RT)*. 237–267.

Manfred Broy and María Victoria Cengarle. 2011. UML formal semantics: lessons learned. *Software and Systems Modeling* 10, 4 (June 2011), 441–446. DOI:http://dx.doi.org/10.1007/s10270-011-0207-y

Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. 2014. Towards a System Model for UML. The Structural Data Model. *CoRR* abs/1409.6613 (2014). http://arxiv.org/abs/1409.6613

Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. 2004. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems (SVERTS)*.

Jordi Cabot, Robert Clarisó, and Daniel Riera. 2014. On the Verification of UML/OCL Class Diagrams using Constraint Programming. *Journal of Systems and Software* 93, 0 (2014), 1 – 23. DOI:http://dx.doi.org/10.1016/j.jss.2014.03.023

Victoria Cengarle and Alexander Knapp. 2004. UML 2.0 Interactions : Semantics and Refinement. In *Critical System Development with UML (CSDUML)*.

Christine Choppy, Kais Klai, and Hacene Zidani. 2011. Formal Verification of UML State Diagrams: A Petri Net Based Approach. *SIGSOFT Softw. Eng. Notes* 36, 1 (Jan. 2011), 1–8. DOI:http://dx.doi.org/10.1145/1921532.1921561

Emanuele Ciapessoni, Alberto Coen-Porisini, Ernani Crivelli, Dino Mandrioli, Angelo Morzenti, and Piergiorgio Mirandola. 1999. From formal models to formally-based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 8, 1 (1999), 79—-113.

Federico Ciccozzi. 2016. On the automated translational execution of the action language for foundational UML. *Software & Systems Modeling* (2016), 1–27. DOI:http://dx.doi.org/10.1007/s10270-016-0556-7

Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert France, Jean-Marc Jézéquel, and Jeff Gray. 2014. Globalizing Modeling Languages. *IEEE Computer* (June 2014), 68–71. DOI:http://dx.doi.org/10.1109/MC.2014.147

Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. 2005. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming* 55, 1-3 (2005), 81–115.

Werner Damm and Bernd Westphal. 2003. Live and Let Die : LSC-Based Verification of UML-Models. In *Formal Methods for Components and Objects (FMCO)*. 99–135.

Zamira Daw, John Mangino, and Rance Cleaveland. 2015. UML-VT: A Formal Verification Environment for UML Activity Diagrams. (2015).

Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. 2008. First Experiments Using the UML Profile for MARTE. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)* 1 (May 2008), 50–57. DOI:http://dx.doi.org/10.1109/ISORC.2008.36

Karsten Diethers and Michaela Huhn. 2004. Vooduu: Verification of Object-Oriented Designs Using UPPAAL. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, Vol. 2988. 139–143.

Danny Dolev, Maria Klawe, and Michael Rodeh. 1982. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* 3, 3 (Sept. 1982), 245–260. DOI:http://dx.doi.org/10.1016/0196-6774(82)90023-2

Dov Dori. 2002. Why significant UML change is unlikely. *Communications of the ACM (CACM)* 45, 11 (2002), 82–85.

Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. 2005. Compositional semantics for UML 2.0 sequence diagrams using Petri nets. In *International SDL Forum*. 133–148. http://www.springerlink.com/index/XYAMC31QHVLVCMCM.pdf

John Erickson and Keng Siau. 2007. Theoretical and practical complexity of modeling methods. *Communications of the ACM (CACM)* (2007).

Rik Eshuis. 2006. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology* 15, 1 (2006), 1–38.

Rik Eshuis. 2009. Reconciling statechart semantics. *Science of Computer Programming* 74, 3 (2009), 65–99. DOI:http://dx.doi.org/10.1016/j.scico.2008.09.001

Rik Eshuis and Roel Wieringa. 2001. A Real-Time Execution Semantics for UML Activity Diagrams. In *Fundamental Approaches to Software Engineering (FASE)*, Vol. 02. 1–15.

Rik Eshuis and Roel Wieringa. 2004. Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering* 30, 7 (2004), 437–447. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1318605

Luca Ferrucci, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. 2012. Modular Automated Verification of Flexible Manufacturing Systems with Metric Temporal Logic and Non-Standard Analysis. In *Formal Methods for Industrial Critical Systems*. Lecture Notes in Computer Science, Vol. 7437. 162–176.

Carlo A. Furia, Matteo Pradella, and Matteo Rossi. 2008. Automated Verification of Dense-Time MTL Specifications via Discrete-Time Approximation. In *Proceedings of the 15th International Symposium on Formal Methods (FM'08) (Lecture Notes in Computer Science)*, Jorge Cuéllar and Tom Maibaum (Eds.), Vol. 5014. Springer-Verlag, 132–147.

Carlo A. Furia and Matteo Rossi. 2010. A Theory of Sampling for Continuous-Time Metric Temporal Logic. *ACM TOCL* 12, 1 (2010), 8:1–8:40.

Régis Gascon, Frédéric Mallet, and Julien Deantoni. 2011. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In *Temporal Representation and Reasoning (TIME), 2011 Eighteenth International Symposium on*. IEEE, 141–148.

Gerard J. Holzmann. 2004. *The Spin Model Checker: Primer and Reference Model*.

Martin Glinz. 2000. Problems and deficiencies of UML as a requirements specification language. In *10th International Workshop on Software Specification*. http://dl.acm.org/citation.cfm?id=857171.857222

Martin Gogolla, Lars Hamann, and Frank Hilken. 2014. Checking Transformation Model Properties with a UML and OCL Model Validator. In *Proc. 3rd Int. STAF'2014 Workshop Verification of Model Transformations (VOLT'2014)*.

Günter Graw and Peter Herrmann. 2004. Transformation and Verification of Executable UML Models. *Electronic Notes in Theoretical Computer Science* 101 (Nov. 2004), 3–24. DOI:http://dx.doi.org/10.1016/j.entcs.2004.09.006

Youcef Hammal. 2005. A Formal Semantics of UML StateCharts by Means of Timed Petri Nets. In *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*. LNCS, Vol. 3731. 38–52.

Youcef Hammal. 2006. Branching Time Semantics for UML 2.0 Sequence. In *Formal Techniques for Networked and Distributed Systems*. 259–274.

Helle Hansen, Jeroen Ketema, Bas Luttik, MohammadReza Mousavi, and Jaco van de Pol. 2010b. Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering* 6, 1 (2010), 83–90.

Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, and Mohammadreza Mousavi. 2010a. Automated Verification of Executable UML Models. In *Formal Methods for Components and Objects (FMCO)*. 1–26.

David Harel. 2004. What's the Semantics of "Semantics"? *IEEE Computer* (2004).

Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. 2005a. Why Timed Sequence Diagrams Require Three-Event Semantics. In *Scenarios: Models, Transformations and Tools*. 1–25.

Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. 2005b. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling (SOSYM)* 4, 4 (Oct. 2005), 355–357. DOI:http://dx.doi.org/10.1007/s10270-005-0087-0

Alexander Knapp and Jochen Wuttke. 2006. Model checking of UML 2.0 interactions. In *Workshops and Symposia at MoDELS 2006 (Lecture Notes in Computer Science)*, Vol. 4634. 42–51.

Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. 2009. Towards an Integrated Graph-based Semantics for UML. *Software and Systems Modeling* 8, 3 (2009), 403–422. DOI: http://dx.doi.org/10.1007/s10270-008-0101-4

Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. 2005. Formalizing UML Models and OCL Constraints in PVS. *Electronic Notes in Theoretical Computer Science* 115 (Jan. 2005), 39–47. DOI: http://dx.doi.org/10.1016/j.entcs.2004.09.027

Kevin Lano. 2009. A compositional semantics of UML-RSDS. *Software & Systems Modeling* 8, 1 (Aug. 2009), 85–116. DOI: http://dx.doi.org/10.1007/s10270-007-0064-x

Edward A. Lee and Sanjit A. Seshia. 2011. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. http://LeeSeshia.org.

Johan Lilius and Ivan Porres Paltor. 1999. vUML: A tool for verifying UML models. *Automated Software Engineering (ASE)* (1999). http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=802301

Mass Soldal Lund and Ketil Stolen. 2006. A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In *Formal Methods (FM)*. 380–395.

Frédéric Mallet. 2008. Clock Constraint Specification Language: Specifying Clock Constraints with UML/-MARTE. *Innovations in Systems and Software Engineering* 4, 3 (2008), 309–314.

OMG MARTE. 2011. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). (2011).

Zoltán Micskei and Hélène Waeselynck. 2011. The many meanings of UML 2 Sequence Diagrams: a survey. *Software and Systems Modeling (SOSYM)* 10, 4 (2011), 489–514. http://dx.doi.org/10.1007/s10270-010-0157-9

Alfredo Motta, Mohammad Mehdi Pourhashem Kallehbasti, Luciano Baresi, Angelo Morzenti, and Matteo Rossi. 2017. Corretto UML. github.com/deib-polimi/Corretto. (2017).

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3 : An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

Iulian Ober and Iulia Dragomir. 2010. OMEGA2: A new version of the profile and the tools. *International Conference on Engineering of Complex Computer Systems (ICECCS)* (2010). DOI: http://dx.doi.org/10.1109/ICECCS.2010.59

OMG. 2011. *OMG Unified Modeling Language Superstructure*. Technical Report August. http://www.omg.org/spec/UML/2.4.1/

OMG. 2013. *Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), v1.0.1*. http://www.omg.org/spec/ALF/1.0.1/

OMG. 2016. *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.2.1*. http://www.omg.org/spec/FUML/1.2.1/

Ivan Paltor and Johan Lilius. 1999. Formalising UML State Machines for Model Checking. In *International Conference on the Unified Modeling Language (UML) (LNCS)*, Vol. 1723. 430–445.

Zsigmond Pap, István Majzik, András Pataricza, and András Szegi. 2005. Methods of checking general safety criteria in UML statechart specifications. *Reliability Engineering & System Safety* 87, 1 (Jan. 2005), 89–107. DOI: http://dx.doi.org/10.1016/j.ress.2004.04.011

Ernesto Posse and Juergen Dingel. 2016. An executable formal semantics for UML-RT. *Software & Systems Modeling* 15, 1 (2016), 179–217.

Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. 2008. Refining Real-Time System Specifications through Bounded Model- and Satisfiability-Checking. *IEEE/ACM International Conference on Automated Software Engineering* (Sept. 2008), 119–127. DOI: http://dx.doi.org/10.1109/ASE.2008.22

Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. 2013. Bounded Satisfiability Checking of Metric Temporal Logic Specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 3 (2013), 20:1–20:54.

Matteo Rossi, Dino Mandrioli, Angelo Morzenti, and Luca Ferrucci. 2016. A temporal logic for micro- and macro-step-based real-time systems: Foundations and applications. *Theoretical Computer Science* 643 (2016), 38–64.

Timm Schfer, Alexander Knapp, and Stephan Merz. 2001. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science* 55, 3 (2001), 357–369. DOI: http://dx.doi.org/10.1016/S1571-0661(04)00262-2

Harald Störrle. 2003. Semantics of Interactions in UML 2.0. In *IEEE Symposium on Human Centric Computing Languages and Environments (HCC)*. 129–136.

Harald Störrle. 2004. Semantics of UML 2 . 0 Activities. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*.

Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. 2014. Formalizing Execution Semantics of UML Profiles with fUML Models. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 133–148.

Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2011. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming* 76, 2 (Feb. 2011), 119–135. DOI: http://dx.doi.org/10.1016/j.scico.2010.07.002

Min Zhang, Frédéric Mallet, and Huibiao Zhu. 2016. An SMT-Based Approach to the Formal Analysis of MARTE/CCSL. In *International Conference on Formal Engineering Methods*. Springer, 433–449.

A. A SHORT INTRODUCTION TO TRIO AND $\mathbb{Z}$OT

TRIO [Ciapessoni et al. 1999] 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 $\mathrm{Dist}(F, t)$ specifies that F holds at a time instant whose distance is exactly t time units from the current instant. $\mathrm{Dist}(F, t)$ 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 the C-UML semantics 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 IV defines some of the most significant ones, including those used in defining the semantics of C-UML.

Operator	**Definition**
$\mathrm{Futr}(\phi, t)$	$t \geq 0 \wedge \mathrm{Dist}(\phi, t)$
$\mathrm{Past}(\phi, t)$	$t \geq 0 \wedge \mathrm{Dist}(\phi, -t)$
$\mathrm{Alw}(\phi)$	$\forall d : \mathrm{Dist}(\phi, d)$
$\mathrm{AlwF}(\phi)$	$\forall d \geq 0 : \mathrm{Futr}(\phi, d)$
$\mathrm{AlwP}(\phi)$	$\forall d \geq 0 : \mathrm{Past}(\phi, d)$
$\mathrm{Som}\phi$	$\exists d : \mathrm{Dist}(\phi, d)$
$\mathrm{SomF}(\phi)$	$\exists d \geq 0 : \mathrm{Futr}(\phi, d)$
$\mathrm{SomP}(\phi)$	$\exists d \geq 0 : \mathrm{Past}(\phi, d)$
$\mathrm{Lasts}(\phi, t)$	$\forall d \in (0, t] : \mathrm{Futr}(\phi, d)$
$\mathrm{Lasted}(\phi, t)$	$\forall d \in (0, t] : \mathrm{Past}(\phi, d)$
$\mathrm{WithinF}(\phi, t)$	$\exists d \in (0, t] : \mathrm{Futr}(\phi, d)$
$\mathrm{WithinP}(\phi, t)$	$\exists d \in (0, t] : \mathrm{Past}(\phi, d)$
$\mathrm{Until}(\psi, \phi)$	$\exists d \geq 0 : \mathrm{Futr}(\phi, d)$ *and* $\forall i, 0 \leq i < d\ \mathrm{Futr}(\psi, i)$
$\mathrm{Since}(\psi, \phi)$	$\exists d \geq 0 : \mathrm{Past}(\phi, d)$ *and* $\forall i, 0 \leq i < d\ \mathrm{Past}(\psi, i)$

Table IV: TRIO derived temporal operators.

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 [Pradella et al. 2013], a bounded satisfiability checker

that supports verification of discrete-time TRIO models. $\mathbb{Z}$ot[20] encodes satisfiability (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 more efficient encodings that exploit the features of Satisfiability Modulo Theories (SMT) solvers [Baresi et al. 2016; Baresi et al. 2015]. 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.

As a simplest example on which to discuss the introduced concepts we consider a so-called timer-reset-lamp (TRL, [Pradella et al. 2008]), i.e., a lamp with two buttons, called ON and OFF. When the ON button is pressed the lamp is lighted and it remains so for Δ time units (t.u.) and then it goes off, unless the OFF button is pushed before the Δ time-out expires (in which case the light goes off immediately after the push of the OFF button , even if this occurs before the end of the time-out period), or unless the ON button is pressed again, before the time-out, in which case the lamp will remain lighted for Δ more t.u. (unless the OFF button is pressed before the time-out expires, etc.). To ensure that the pressure of a button is always meaningful, it is assumed that ON and OFF cannot be pressed simultaneously.

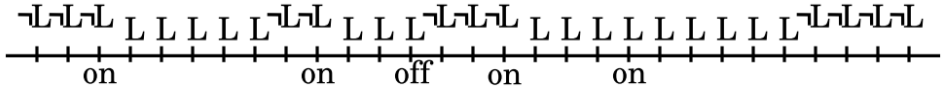


Fig. 28: A history for the timed lamp example.

An example of a trace of execution of the TRL system (a so-called history) is represented in Figure 28 , for the case $\Delta = 5$. The history shows typical behaviors of the modeled system: the lamp being off is turned on by pushing button ON and then it turns off “spontaneously" after Δ t.u.; then the lamp is lighted again and then turned off within Δ t.u. by pressing button OFF; the lamp is kept on by pushing again button ON before the Δ time-out, and then it finally goes off spontaneously. The descriptive model of the TRL is based on the following three propositional letters, with the indicated meaning:

— L: the light is on.
— ON: the button to turn it on is pressed.
— OFF: the button to turn it off is pressed.

The model is made of two simple formulae. the first one states that the lamp is on (at the current time) if and only if the ON button was pressed no more than Δ time units ago and since then the OFF button has not been pressed. In TRIO this is formalized by the following formula:

$$(D1) \quad L \Leftrightarrow \mathrm{WithinP}(ON, \Delta) \wedge \mathrm{Since}(\neg OFF, ON)$$

The second formula expresses the mutual exclusion between the pressing of the ON and OFF buttons, namely:

$$(D2) \quad \neg(ON \wedge OFF)$$

The descriptive model of the formula simply consists of the conjunction of these two formulae, enclosed in a universal temporal quantification (an Alw operator) asserting

[20] github.com/fm-polimi/zot

that they hold for all instants of the temporal domain.

$$(DM) \quad \mathrm{Alw}(D1 \wedge D2)$$

The descriptive model, despite its simplicity and succinctness, characterizes completely the TRL system: starting from it the history depicted in Figure 28 can be generated using the $\mathbb{Z}$ot tool, or one can prove that the following (conjectured) property

$$(P1) \quad \mathrm{Alw}(\neg \mathrm{Lasts}(L, \Delta + 1))$$

(i.e., the lamp will never remain on for more than Δ time units) does not hold, by generating, through the $\mathbb{Z}$ot tool, a counter-example consisting of a history similar to the one shown in Figure 28, including two push actions of the ON button at distance less than Δ; the $\mathbb{Z}$ot tool can instead prove, from the descriptive model, the following property

$$(P2) \quad \mathrm{Alw}(\mathrm{Lasts}(L, \Delta + 1) \Rightarrow \mathrm{WithinF}(ON, \Delta + 1))$$

(i.e., the lamp remains lighted for more than Δ time units only if we have another press action of the ON button at a distance of less than Δ t.u.).