# From Interaction Overview Diagrams to Temporal Logic*

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

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

**Abstract.** In this paper, we use UML Interaction Overview Diagrams as the basis for a user-friendly, intuitive, modeling notation that is well-suited for the design of complex, heterogeneous, embedded systems developed by domain experts with little background on modeling software-based systems. To allow designers to precisely analyze models written with this notation, we provide (part of) it with a formal semantics based on temporal logic, upon which a fully automated, tool supported, verification technique is built. The modeling and verification technique is presented and discussed through the aid of an example system.

**Keywords:** Metric temporal logic, bounded model checking, Unified Modeling Language.

## 1 Introduction

Complex embedded systems such as those found in the Aerospace and Defense domains are typically built of several, heterogeneous, components that are often designed by teams of engineers with different backgrounds (e.g., telecommunication, control systems, software engineering, etc.). Careful modeling starting from the early stages of system development can greatly help increase the quality of the designed system when it is accompanied and followed by verification and code generation activities. Modeling-verification-code generation are three pillars in the model driven development of complex embedded systems; they are most effective when (i) modeling is based on user-friendly, intuitive, yet precise notations that can be used with ease by experts of domains other than computer science; (ii) rigorous, possibly formal, verification can be carried out on the aforementioned models, though in a way that is hidden from the system developer as much as possible; (iii) executable code can be seamlessly produced from verified models, to generate implementations that are correct by construction.

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

J. Dingel and A. Solberg (Eds.): MODELS 2010 Workshops, LNCS 6627, pp. 90–104, 2011.

This work, which is part of a larger research effort carried out in the MADES European project[1] [1], focuses on aspects (i) and (ii) mentioned above. In particular, it is the first step towards a complete proposal for modeling and validating embedded systems. The plan is to exploit both "conventional" UML diagrams [15] and a subset of the MARTE (Modeling and Analysis of Real-Time and Embedded systems) UML profile [14]. We want to use Class Diagrams to define the key components of the system. State Diagrams to model their internal behaviors, and Sequence and Interaction Overview Diagrams to model the interactions and cooperations among the different elements. These diagrams will be augmented with clocks and resources taken from MARTE. The result is a multi-faceted model of the system, automatically translated into temporal logic to verify it. Temporal Logic helps glue the different views, create a single, consistent representation of the system, discover inconsistencies among the different aspects, and formally verify some global properties.

This paper starts from Interaction Overview Diagrams (IODs) since they are often neglected, but they provide an interesting means to integrate Sequence Diagrams (SDs) and define coherent and complex evolutions of the system of interest. IODs are ascribed a formal semantics, based on temporal logic, upon which a fully automated, tool supported, verification technique is built.

The choice of IODs as the starting point for a modeling notation that is accessible to experts of different domains, especially those other than software engineering, is borne from the observation that, in the industrial practice, SDs are often the preferred notation of system engineers to describe components' behaviors [3]. However, SDs taken in isolation are not enough to provide a complete picture of the interactions among the various components of a complex system; hence, system designers must be given mechanisms to combine different SDs into richer descriptions, which is precisely what IODs offer.

In this article we provide a preliminary formal semantics of IODs based on metric temporal logic. While this semantics is not yet complete, as it does not cover all possible mechanisms through which SDs can be combined into IODs, it is nonetheless a significant first step in this direction. The provided semantics has been implemented into the Zot bounded satisfiability/model checker [16][2], and has been used to prove some properties of an example system.

This paper is structured as follows. Section 2 briefly presents IODs; Section 3 gives an overview of the metric temporal logic used to define the formal semantics of IODs, and of the Zot tool supporting it; Section 4 introduces the formal semantics of IODs through an example system, and discusses how it has been used to prove properties of the latter; Section 5 discusses some relevant related works; finally, Section 6 draws some conclusions and outlines future works.

## 2 Interaction Overview Diagrams

Most UML behavioral diagrams have undergone a significant revision from version 1.x to version 2.x. To model interactions, UML2 offers four kinds of

[1] `http://www.mades-project.org`

[2] Zot is available at `http://home.dei.polimi.it/pradella/Zot`

diagrams: communication diagrams, sequence diagrams, timing diagrams and interaction overview diagrams. In this work we focus on Sequence Diagrams (SDs) and Interaction Overview Diagrams (IODs).

SDs have been considerably revised and extended in UML2 to improve their expressiveness and their structure. IODs are new in UML2. They allow a designer to provide a high-level view of the possible interactions in a system. IODs constitute a high-level structuring mechanism that is used to compose scenarios through mechanisms such as sequence, iteration, concurrency or choice. IODs are a special and restricted kind of UML Activity Diagrams (ADs) where nodes are interactions or interaction uses, and edges indicate the flow or order in which these interactions occur. Semantically, however, IODs are more complex compared to ADs and they may have different interpretations. In the following the fundamental operators of IODs are presented. Figure 2 shows an example of IOD for the application analyzed in Section 4, which will be used throughout this section to provide graphical examples of IOD constructs. IODs include also other operators whose study is left to future works.

IODs include the **initial node**, **final node** and **flows final node** operators, which have exactly the same meaning of the corresponding operators found in ADs. For example, The IOD of Figure 2 has an initial node at the top, but no final or flow final nodes.

A **control flow** is a directed connection (flow) between two SDs (e.g., between diagrams *delegateSMS* and *downloadSMS* in Figure 2). As soon as the SD at the source of the flow is finished, it presents a token to the SD at the end of the flow.

A **fork node** is a control node that has a single incoming flow and two or more outgoing flows. Incoming tokens are offered to all outgoing flows (edges). The outgoing flows can be guarded, which gives them a mechanism to accept or reject a token. In the IOD of Figure 2, there is one fork node at the top of the diagram (between the initial node and SDs *waitingCall* and *checkingSMS*) modeling two concurrent execution of the system. The dual operator is the join node, which synchronizes a number of incoming flows into a single outgoing flow. Each (and every) incoming control flow must present a control token to the join node before the node can offer a single token to the outgoing flow.

A **decision node** is a control node that has one incoming flow and two or more outgoing flows. In the IOD of Figure 2 there are four decision operators (e.g., the one between SDs *waitingCall* and *delegateCall*) with their corresponding Boolean conditions. Conversely, a **merge node** is a type of control node that has two or more incoming flows and a single outgoing flow.

## 3 TRIO and $\mathbb{Z}$ot

TRIO [7] is a general-purpose formal specification language suitable for describing complex real-time systems, including distributed ones. TRIO 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 formula $F$ and

**Table 1.** TRIO derived temporal operators

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

a term $t$ indicating a time distance (either positive or negative), the formula $\text{Dist}(F,t)$ specifies that $F$ holds at a time instant whose distance is exactly $t$ time units from the current one. While TRIO can exploit both discrete and dense sets as time domains, in this paper we assume the nonnegative integers $\mathbb{N}$ as discrete time domain. For convenience in the writing of specification formulae, TRIO defines a number of *derived* temporal operators from the basic Dist, through propositional composition and first-order logic quantification. Table 1 defines some of the most significant ones, including those used in this paper.

The TRIO specification of a system includes a set of basic *items*, such as predicates, representing the elementary modeled phenomena. The system behavior over time is formally specified by a set of TRIO formulae, which state how the items are constrained and how they vary in time, 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 $\Sigma$ and $\rho$, 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 refer to ℤot [16], a bounded model checker which supports verification of discrete-time TRIO models. ℤot 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 a more efficient encoding that exploits the features of Satisfiability Modulo Theories (SMT) solvers [2]. Through ℤot one can verify whether stated properties hold for the system being analyzed (or parts thereof) or not; if a property does not hold, ℤot produces a counterexample.

# 4 Formal Semantics of Interaction Overview Diagrams

This section introduces the formal semantics of IODs defined in terms of the TRIO temporal logic. The semantics is presented by way of an example system, whose behavior modeled through a IOD is described in Section 4.1. Then, Section 4.2 discusses the TRIO formalization of different constructs of IODs, and illustrates how this is used to create a formal model for the example system. Section 4.3 briefly discusses some properties that were checked for the modeled system by feeding its TRIO representation to the ℤot verification tool. Finally, Section 4.4 provides a measure of the complexity of the translation of IODs into metric temporal logic.

## 4.1 Example Telephone System

The example system used throughout this section is a telephone system composed of three units, a *TransmissionUnit*, a *ConnectionUnit* and a *Server*, depicted in the class diagram of Figure 1. The *ConnectionUnit* is in charge of checking for the arrival of new SMSs on the *Server* (operation *checkSMS* of class *Server*) and to handle new calls coming from the *Server* (operation *IncomingCall* of class *ConnectionUnit*). The *TransmissionUnit* is used by the *ConnectionUnit* to download the SMSs (operation *downloadSMS*) and to handle the call's data (operation *beginCall*). The *TransmissionUnit* receives the data concerning SMSs and calls from the *Server* (operations *receiveSMSToken* and *receiveCallData*).

The behavior of the telephone system is modeled by the IOD of Figure 2. The fork operator specifies that the two main paths executed by the system are in parallel; for example the *checkingSMS* and *receiveCall* sequence diagrams run in parallel. Branch conditions are used in order to distinguish between different possible executions; for example after checking for a new SMS on the *Server* the system will continue with downloading the SMSs if one is present, otherwise it will loop back to the same diagram. It can be assumed that the *Server* allocates a dedicated thread to each connected telephone; this is why the sequence diagrams of Figure 2 report the interaction between only one *ConnectionUnit*, one *TransmissionUnit* and one *Server*.

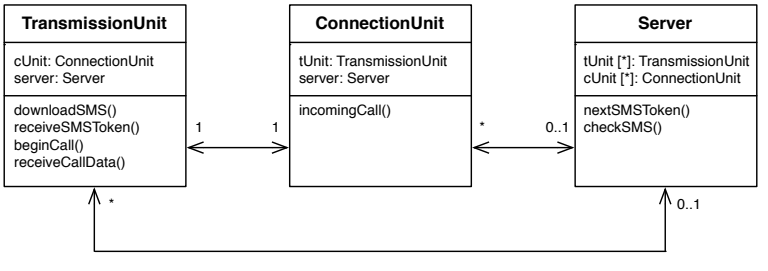


**Fig. 1.** Class diagram for the telephone system

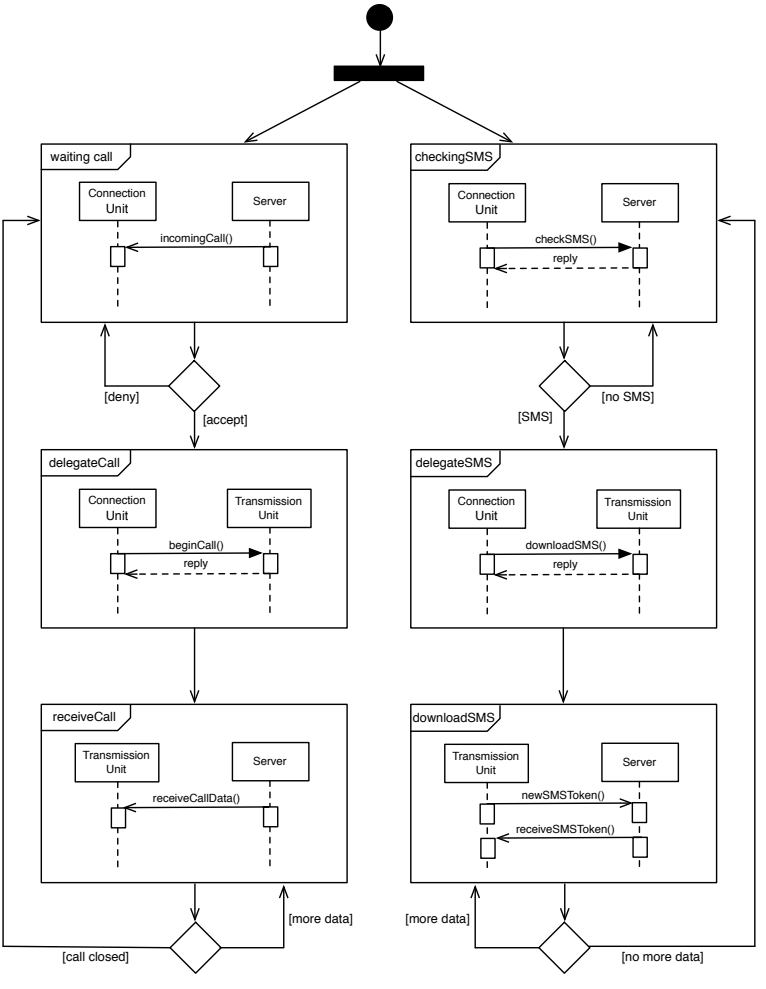


**Fig. 2.** Interaction Overview diagram for the telephone system

### 4.2 TRIO Formalization

The formalization presented here was derived from the diagram of Figure 2 by hand. The availability of a tool, which we are building, will allow us to analyze more complex models and assess the actual scalability of the proposed technique. The formalization is organized into sets of formulae, each of them corresponding to one of the SDs appearing in the IOD. Every set can be further decomposed into three subsets modeling different aspects of the SDs:

– **diagram-related formulae**, which concern the beginning and the end of the execution of each SD, and the transition between a SD and the next one(s);

- **message-related formulae**, which concern the ordering of the events within a single SD;
- **component-related formulae**, which describe constraints on the execution of operations within single components.

These subsets are presented in the rest of this section.

**Diagram-related Formulae.** In this presentation of the semantics of IODs we assume, for the sake of simplicity, that, within each SD of an IOD, messages are totally ordered, hence we can clearly identify a begin message and an ending message. This assumption is not restrictive because, given any IOD that does not satisfy it, we can use the fork/join operators to obtain an equivalent IOD that satisfies the assumption, simply by splitting diagrams where messages may occur in parallel, into diagrams where messages are totally ordered. Then, for each SD $D_x$, it is possible to identify two messages, $m_s$ and $m_e$, which correspond to the start and the end of the diagram. For each SD $D_x$, we introduce predicates $D_xSTART$ and $D_xEND$ that are true, respectively, at the beginning and at the end of the diagram. We also introduce, for each message $m$ appearing in diagram $D_x$, a predicate $m$ that holds in all instants in which the message occurs in the system (this entails that components synchronize on messages: send and receive of a message occur at the same time). Then, the correspondence between $D_xSTART$ (resp. $D_xEND$) and the starting (resp. ending) message $m_s$ (resp. $m_e$) is formalized by formulae (1-2)[3]. In addition, we introduce a predicate $D_x$ that holds in all instants in which diagram $D_x$ is executing; hence, predicate $D_x$ holds between $D_xSTART$ and $D_xEND$, as stated by formula (3).

$$D_xSTART \Leftrightarrow m_s \tag{1}$$

$$D_xEND \Leftrightarrow m_e \tag{2}$$

$$D_x \Leftrightarrow D_xSTART \vee \text{Since}(\neg D_xEND, D_xSTART) \tag{3}$$

For example, the instances of formulae (1-3) for diagram *delegateSMS* correspond to formulae (4-6).

$$delegateSMSSTART \Leftrightarrow downloadSMS \tag{4}$$

$$delegateSMSEND \Leftrightarrow reply3 \tag{5}$$

$$delegateSMS \Leftrightarrow delegateSMSSTART \vee \text{Since}(\neg delegateSMSEND, delegateSMSSTART) \tag{6}$$

Notice that if the IOD contains $k$ different occurrences of the same message $m$, $k$ different predicates $m_0 ... m_k$ are introduced. For this reason in formula (5) $reply3$ appears instead of $reply$.

[3] Note that TRIO formulae are implicitly temporally closed with the Alw operator; hence, $D_xSTART \Leftrightarrow m_s$ is actually an abbreviation for $\text{Alw}(D_xSTART \Leftrightarrow m_s)$.

A diagram $D_x$ is followed by a diagram $D_y$ for either of two reasons: (1) $D_x$ is directly connected to $D_y$, in this case the end of $D_x$ is a sufficient condition to start $D_y$; (2) $D_x$ is connected to $D_y$ through some *decision* operator, in this case a sufficient condition for $D_y$ to start is given by the end of $D_x$, provided (i.e. conjoined with the requirement that) the condition associated with the decision operator is true. If a diagram $D_x$ is preceded by $p$ sequence diagrams, we introduce $p$ predicates $D_xACTC_i$ ($i \in \{1...p\}$), where $D_xACTC_i$ holds if the $i$-th sufficient condition to start diagram $D_x$ holds. We also introduce predicate $D_xACT$, which holds if any of the $p$ necessary conditions holds, as defined by formula (7). After the necessary condition to start a diagram is met, the diagram will start at some point in the future, as stated by formula (8). Finally, after a diagram starts, it cannot start again until the necessary condition to start it is met anew, as defined by formula (9).

$$D_xACT \Leftrightarrow D_xACTC_0 \vee ... \vee D_xACTC_m \tag{7}$$

$$D_xACT \Rightarrow \mathrm{SomF}(D_xSTART) \tag{8}$$

$$D_xSTART \Rightarrow \neg\mathrm{SomF}(D_xSTART) \vee \mathrm{Until}(\neg D_xSTART, D_xACT) \tag{9}$$

In the case of SD *downloadSMS* of Figure 2, the instances of formulae (7-9) are given by (12-14). In addition, formulae (10-11) define the necessary conditions to start diagram *downloadSMS*: either diagram *delegateSMS* ends, or diagram *downloadSMS* ends and condition *moredata* holds. Currently, we can only deal with atomic Boolean conditions. The representation of more complex data, and conditions upon them, is already in our research agenda.

$$downloadSMSACTC_1 \Leftrightarrow delegateSMSEND \tag{10}$$

$$downloadSMSACTC_2 \Leftrightarrow downloadSMSEND \wedge moredata \tag{11}$$

$$downloadSMSACT \Leftrightarrow \begin{pmatrix} downloadSMSACTC_1 \\ \vee\, downloadSMSACTC_2 \end{pmatrix} \tag{12}$$

$$downloadSMSACT \Rightarrow \mathrm{SomF}(downloadSMSSTART) \tag{13}$$

$$\begin{aligned} downloadSMSSTART \Rightarrow \\ \neg\mathrm{SomF}(downloadSMSSTART) \vee \\ \mathrm{Until}(\neg downloadSMSSTART, downloadSMSACT) \end{aligned} \tag{14}$$

**Message-related Formulae.** Suppose that, in a SD, a message $m_i$ is followed by another message $m_j$. Then the occurrence of $m_i$ entails that $m_j$ will also occur in the future; conversely, the occurrence of $m_j$ entails that $m_i$ must have occurred in the past. This is formalized by formulae (15-16). In addition, after an instance of $m_j$, there can be a new instance of the same message only after a new occurrence of $m_i$; this is stated by formula (17), which defines that, after $m_j$, there will not be a new occurrence of $m_j$ until there is an occurrence of $m_i$.

$$m_i \Rightarrow \text{SomF}(m_j) \wedge \neg m_j \tag{15}$$
$$m_j \Rightarrow \text{SomP}(m_i) \wedge \neg m_i \tag{16}$$
$$m_j \Rightarrow \neg\text{SomF}(m_j) \vee \text{Until}(\neg m_j, m_i) \tag{17}$$

If, for example, formulae (15-17) are instantiated for SD *checkingSMS* of Figure 2, one obtains formulae (18-20).

$$checkSMS \Rightarrow \text{SomF}(reply1) \wedge \neg reply1 \tag{18}$$
$$reply1 \Rightarrow \text{SomP}(checkSMS) \wedge \neg checkSMS \tag{19}$$
$$checkSMS \Rightarrow \neg\text{SomF}(checkSMS) \vee \text{Until}(\neg checkSMS, reply1) \tag{20}$$

**Component-related Formulae.** This set of formulae describes the conditions under which the entities of the system are busy, hence cannot perform further operations until they become free again. For example, in the telephone system of Figure 2, when the execution is inside the *checkingSMS* diagram, the *ConnectionUnit* cannot perform any other operations during the time interval between the invocation of operation *ckechSMS* and its corresponding *reply* message, since the invocation is synchronous (as highlighted by the full arrow).

In general, a synchronous invocation between objects $A$ and $B$ that starts with message $m_i$ and ends with message $m_j$ blocks both components from the moment of the invocation until its end; this is formalized by formulae (21-22), in which $h$ and $k$ are indexes identifying the occurrences of invocations (either received of issued) related to objects $A$ and $B$ in the IOD. In case of an asynchronous message $m$ between $A$ and $B$ (such as, for example, *incomingCall* in SD *waitingCall*, as denoted by the wire-like arrow), the semantics is the one defined by formulae (23-24), which state that the objects are blocked only in the instant in which the message occurs.

$$m_i \vee \text{Since}(\neg m_j, m_i) \Leftrightarrow ABLOCKED_h \tag{21}$$
$$m_i \vee \text{Since}(\neg m_j, m_i) \Leftrightarrow BBLOCKED_k \tag{22}$$
$$m \Leftrightarrow ABLOCKED_h \tag{23}$$
$$m \Leftrightarrow BBLOCKED_k \tag{24}$$

Finally, if $n$ is the number of occurrences of invocations involving object $A$ in the IOD, formula (25) states that all executions involving $A$ are mutually exclusive.

$$\forall 1 \leq i, j \leq n(i \neq j \wedge ABLOCKED_i \Rightarrow \neg ABLOCKED_j) \tag{25}$$

The following formulae are instances of (21-25) for object *ConnectionUnit*, which is involved in four separate invocations in the IOD of Figure 2:

$$
\begin{aligned}
ConnectionUnitBLOCKED1 &\Leftrightarrow checkSMS \vee \\
&\quad \text{Since}(\neg reply1, checkSMS) \\
ConnectionUnitBLOCKED2 &\Leftrightarrow incomingCall \\
ConnectionUnitBLOCKED3 &\Leftrightarrow downloadSMS \vee \\
&\quad \text{Since}(\neg reply2, donwloadSMS) \\
ConnectionUnitBLOCKED4 &\Leftrightarrow beginCall \vee \\
&\quad \text{Since}(\neg reply3, beginCall) \\
\forall 1 \leq i, j \leq 4 (i \neq j \wedge ConnectionUnitBLOCKED_i &\Rightarrow \\
&\quad \neg ConnectionUnitBLOCKED_j)
\end{aligned}
$$

### 4.3 Properties

Using the formalization presented above, we can check whether the modeled system satisfies some user-defined properties or not, by feeding it as input to the Zot verification tool.[4]

We start by asking whether it is true that, if no SMS is received in the future, then nothing will ever be downloaded. This property is formalized by the following formula:

$$\neg \text{SomF}(SMS) \Rightarrow \neg \text{SomF}(downloadSMS) \tag{26}$$

After feeding it the system and the property to be verified, the Zot tool determines that the latter *does not* hold for the telephone system of Figure 2. In fact, between the check for a new SMS and its download there can be an arbitrary delay; hence, the situation in which the last SMS has been received, but it has not yet been downloaded, violates the property. Zot returns this counterexample in around 8.5 seconds.[5]

The following variation of the property above, instead, holds for the system:

$$\neg(\text{SomP}(SMS) \vee SMS) \Rightarrow \neg \text{WithinF}(downloadSMS, 3) \tag{27}$$

Formula (27) states that, if no SMS has yet been received, for the next 3 instants there will not be an SMS download. Zot takes about 7 seconds to determine that formula (27) holds.

The following formula states that after a *nextSMSToken* request from *TransmissionUnit* to *Server*, no data concerning an incoming call can be received by the *TransmissionUnit* until a new SMS is received.

$$nextSMSToken \Rightarrow \text{Until}(\neg receiveCallData, receiveSMSToken) \tag{28}$$

[4] The complete Zot model can be downloaded from `http://home.dei.polimi.it/rossi/telephone.lisp`

[5] All tests have been performed with a time bound of 50 time units (see [16] for the role of time bounds in Bounded Model/Satisfiabliity Checking), using the Common Lisp compiler SBCL 1.0.29.11 on a 2.80GHz Core2 Duo laptop with Linux and 4 GB RAM. The verification engine used was the SMT-based Zot plugin introduced in [2], with Microsoft Z3 2.8 (`http://research.microsoft.com/en-us/um/redmond/projects/z3/`) as the SMT solver.

ℤot verifies that property (28) does not hold in around 8 seconds. As witnessed by the counterexample produced by ℤot, the reason why (28) does not hold is that the *downloadSMS* diagram and the *receiveCall* diagram can run in parallel, and after sending a *nextSMSToken* message the *TransmissionUnit* and the *Server* are free to exchange a *receiveCallData* message.

### 4.4 Complexity

In this section we estimate the complexity of the translation from the IOD of the system into a set of temporal logic formulas. The purpose of this analysis is to provide an *a priori* estimation of the feasibility of the approach, i.e., to ensure that the approach is scalable and effectively implementable by means of an automatic software tool. It is to be noted that the estimation of the number of predicates and formulas produced by the translation procedure does not allow us to draw conclusions about the complexity of the algorithms for model verification (e.g., through simulation or property proof), because this depends on several features of the verification engine that will be employed by the verification tool (which could be, for instance, a SAT-based or SMT-based solver). Such an analysis is therefore left for future work.

We measure the complexity of the translation in terms of the number of predicates and the size of the TRIO formulas that are produced, and we consider, as parameters of such evaluation, the number $n_d$ of SDs in the IOD and the number $n_o$ of objects composing the system. The worst case occurs when every SD is connected to all the others (including itself) in an IOD, and thus every SD has $n_d$ incoming flows. Moreover, still in a worst case scenario, every object in every SD sends one synchronous message to all the other objects in the system. According to these hypotheses, an estimation of the number of predicates and of the order of magnitude of the number of logic formulae generated by the translation can be carried out as follows.

For every SD the translation generates $3n_d$ predicates ($D_x$, $D_xSTART$, $D_xEND$) and $3n_d$ formulae according to axioms (1–3). Further, since every SD has $n_d$ incoming flows the translation generates $n_d(n_d+1)$ predicates ($D_xACTC_0$, ..., $D_xACTC_{n_d}$, $D_xACT$) and $n_d(n_d+3)$ formulae (7–9).

If we assume that every object in every SD sends one synchronous message to every other object in the same SD, we have $2n_dn_o(n_o-1)$ messages. Every message instance has its own predicate and this results in $2n_dn_o(n_o-1)$ generated predicates. Moreover we have $5 \cdot 2n_dn_o(n_o-1)$ generated formulae, according to axioms (15–17,21,22).

Finally, since every object is blocked while sending or receiving a message, in every SD the number of operation executions for a single object is $2(n_o-1)$ (the object sends $n_o-1$ messages and receives $n_o-1$ messages). This generates $n_dn_o2(n_o-1)$ predicates overall. The mutual exclusion of these predicates is stated in axiom (25); because every object has $n_d2(n_o-1)$ operation executions, we have $n_d2(n_o-1)(n_d2(n_o-1)-1)$ instances of axiom (25) for each object, and $n_on_d2(n_o-1)(n_d2(n_o-1)-1)$ formulae overall.

If $n_{dec}$ is the number of decision operators in the IOD, the overall number of predicates is:

$$3n_d + n_d(n_d + 1) + 4n_d n_o(n_o - 1) + n_{dec}$$

which is in the order of $O(n_d^2 + n_d n_o^2)$, since the number $n_{dec}$ of decision operators can be safely assumed to be $O(n_d)$; also, the overall number of formulae is:

$$3n_d + n_d(n_d + 3) + 5 \cdot 2n_d n_o(n_o - 1) + n_o n_d 2(n_o - 1)(n_d 2(n_o - 1) - 1)$$

which is in the order of $O(n_d^2 * n_o^3)$.

Note, however, that real-world models do not follow this kind of worst-case topology. For a more realistic analysis, we can assume that each diagram is connected to a constant number of diagrams ($n_d^c$, number of diagrams connected, which also entails that the number of decision operators is a constant $n_{dec}^c$), and that each object sends a constant number of messages (e.g., $m$ synchronous messages) to the other objects. In this case, the number of predicates becomes:

$$3n_d + n_d(n_d^c + 1) + 4n_d n_o m + n_{dec}^c$$

which is in the order of $O(n_d n_o)$, and the number of formulae becomes:

$$3n_d + n_d(n_d^c + 3) + 5 \cdot 2n_d n_o m + n_o n_d 2m(n_d 2m - 1)$$

which is in the order of $O(n_d^2 n_o)$.

Looking at the number of formulae, the term that weighs the most is the last one, which originates from the fact that every execution occurrence cannot be true at the same time instant as another execution occurrence. This can only happen if the execution occurrences are inside diagrams that can be executed in parallel (because of fork operators). If we assume, like in our example, that the system only has $p$ parallel paths, then the generated number of formulae becomes: $3n_d + n_d(n_d^c + 3) + 5 \cdot 2n_d n_o m + n_o n_d 2m(\frac{n_d}{p} 2m - 1)$, but its complexity remains in the order of $O(n_d^2 n_o)$. Also, the number of predicates is not affected.

If the system is implemented in a modular fashion, we can also assume that each object does not appear in every SD, but that each diagram comprises a maximum number $n_o^d$ of objects. This means that an object is used in $\frac{n_d n_o^d}{n_o}$ diagrams, and it has $2m\frac{n_d n_o^d}{n_o}$ execution occurrences. These hypotheses transform the number of generated predicates into:

$$3n_d + n_d(n_d^c + 1) + 4n_d n_o^d m + n_{dec}^c$$

which is in the order of $O(n_d)$ and the number of generated formulae into:

$$3n_d + n_d(n_d^c + 3) + 5 \cdot 2n_d n_o^d m + n_o(2m\frac{n_d n_o^d}{n_o})((2m\frac{n_d n_o^d}{n_o}) - 1)$$

with a further reduction to the order of $O(\frac{n_d^2}{n_o})$.

The above complexity figures agree with the intuition that the verification can be carried out more efficiently if the model of the system under analysis is adequately modularized.

Finally, note that the current formal model has not yet been optimized to minimize the number of generated formulae: improvements can surely be obtained, at least as far as the constant factors in the above complexity measures are concerned.

## 5 Related Work

The research community has devoted a significant effort to studying ways to give a formal semantics to scenario-based specifications such as UML sequence diagrams, UML interaction diagrams, and Message Sequence Charts (MSCs).

Many works focus on the separate formalization of sequence diagrams and activity diagrams. Störrle analyzes the semantics of these diagrams and proposes an approach to their formalization [18]. More recently, Staines formalizes UML2 activity diagrams using Petri nets and proposes a technique to achieve this transformation [17]. Also, Lam formalizes the execution of activity diagrams using the $\pi - Calculus$, thus providing them with a sound theoretical foundation [13]. Finally, Eshuis focuses on activity diagrams, and defines a technique to translate them into finite state machines that can be automatically verified [9][8].

Other works investigate UML2 interaction diagrams. Cengarle and Knapp in [6] provide an operational semantics to UML 2 interactions, and in [5] they address the lack of UML interactions to explicitly describe variability and propose extensions equipped with a denotational semantics. Knapp and Wuttke translate UML2 interactions into automata and then verify that the proposed design meets the requirements stated in the scenarios by using model checking [12].

When multiple scenarios come into play, like in IODs, there is the problem of finding a common semantics. Uchitel and Kramer in [19] propose an MSC-based language with a semantics defined in terms of labeled transition systems and parallel composition, which is translated into Finite Sequential Processes that can be model-checked and animated. Harel and Kugler in [10] use Live Sequence Charts (LCSs) to model multiple scenarios, and to analyze satisfiability and synthesis issues.

To the best of our knowledge very little attention has been paid to IODs. Kloul and Küster-Filipe [11] show how to model mobility using IODs and propose a formal semantics to the latter by translating them into the stochastic process algebra PEPA nets. Tebibel uses hierarchical colored Petri nets to define a formal semantics for IODs [4]. Our work is quite different, because it uses metric temporal logic to define the semantics of IODs; as briefly discussed in Sections 1 and 6, this opens many possibilities as far as the range of properties that can be expressed and analyzed for the system is concerned.

## 6 Conclusions and Future Works

In this paper we presented the first steps towards a technique to precisely model and analyze complex, heterogeneous, embedded systems using an intuitive UML-based notation. To this end, we started by focusing our attention on Interaction

Overview Diagrams, which allow users to describe rich behaviors by combining together simple Sequence Diagrams. To allow designers to rigorously analyze modeled systems, the basic constructs of IODs have been given a formal semantics based on metric temporal logic, which has been used to prove some properties of an example system.

The work presented in this paper is part of a longer-term research, and it will be extended in several ways.

First, the metric features of TRIO will be used to extend the formalization of SDs and IODs to real-time features that will be introduced in the modeling language by providing support for the MARTE UML profile.

Furthermore, we will provide semantics to constructs of IODs that are not yet covered. This semantics will be used to create tools to automatically translate IODs into the input language of the ℤot tool, and to show designers the feedback from the verification tool (e.g., counterexamples) in a user-friendly way. In particular, we will define mechanisms to render graphically the counterexamples provided by ℤot as SDs. These tools will allow domain experts who have little or no background in formal verification techniques to take advantage of these techniques in the analysis of complex systems.

## References

1. Bagnato, A., Sadovykh, A., Paige, R.F., Kolovos, D.S., Baresi, L., Morzenti, A., Rossi, M.: MADES: Embedded systems engineering approach in the avionics domain. In: Proceedings of the First Workshop on Hands-on Platforms and Tools for Model-Based Engineering of Embedded Systems, HoPES (2010)
2. Bersani, M.M., Frigeri, A., Pradella, M., Rossi, M., Morzenti, A., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: Proc. of the Int. Symp. on Temporal Representation and Reasoning (TIME), pp. 43–50 (2010)
3. Blohm, G., Bagnato, A.: D1.1 requirements specification. Technical report, MADES Consortium, Draft (2010)
4. Bouabana-Tebibel, T.: Semantics of the interaction overview diagram. In: Proc. of the IEEE Int. Conf. on Information Reuse Integration (IRI), pp. 278–283 (2009)
5. Cengarle, M.V., Graubmann, P., Wagner, S.: Semantics of UML 2.0 interactions with variabilities. Elec. Notes in Theor. Comp. Sci. 160, 141–155 (2006)
6. Cengarle, M.V., Knapp, A.: Operational semantics of UML 2.0 interactions. Technical Report TUM-I0505, Technische Universität Mnchen (2005)
7. Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., Morzenti, A.: From formal models to formally-based methods: an industrial experience. ACM TOSEM 8(1), 79–113 (1999)
8. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
9. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. IEEE Trans. Software Eng. 30(7), 437–447 (2004)
10. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 1–33. Springer, Heidelberg (2001)
11. Kloul, L., Küster-Filipe, J.: From intraction overview diagrams to PEPA nets. In: Proc. of the Work. on Process Algebra and Stochastically Timed Activities (2005)

12. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Models in Software Engineering. LNCS, vol. 4634, pp. 42–51 (2007)
13. Lam, V.S.W.: On $\pi$-calculus semantics as a formal basis for uml activity diagrams. International Journal of Software Engineering and Knowledge Engineering (2008)
14. Object Management Group. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. Technical report, OMG (2009) formal/2009-11-02
15. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Technical report, OMG (2010) formal/2010-05-05.
16. Pradella, M., Morzenti, A., San Pietro, P.: The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In: Proceedings of ESEC/SIGSOFT FSE, pp. 312–320 (2007)
17. Staines, T.S.: Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In: Proc. of the IEEE Int. Conf. on the Engineering of Computer-Based Systems, pp. 191–200 (2008)
18. Störrle, H., Hausmann, J.H.: Towards a formal semantics of UML 2.0 activities. In: Software Engineering. Lect. Notes in Inf., vol. 64, pp. 117–128 (2005)
19. Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: Proc. of the Int. Conf. on Software Engineering, pp. 188–197 (2001)