

Model Based Architecting and Construction of Embedded Systems (ACES-MB 2010)

Stefan Van Baelen¹, Iulian Ober², Huascar Espinoza³,
Thomas Weigert⁴, Ileana Ober⁵, and Sébastien Gérard⁶

¹ K.U.Leuven - DistriNet, Belgium, Stefan.VanBaelen@cs.kuleuven.be

² University of Toulouse - IRIT, France, Iulian.Ober@irit.fr

³ Tecnalia, Spain, Huascar.Espinoza@tecnalia.com

⁴ Missouri University of Science and Technology, USA, weigert@mst.edu

⁵ University of Toulouse - IRIT, France, Ileana.Ober@irit.fr

⁶ CEA - LIST, France, Sebastien.Gerard@cea.fr

Abstract. The third ACES-MB workshop brought together researchers and practitioners interested in model-based software engineering for real-time embedded systems, with a particular focus on the use of models for architecture description and domain-specific design, and for capturing non-functional constraints. Twelve presenters proposed contributions on metaheuristic search techniques for UML, modelling languages and mappings, model based verification and validation, software synthesis, and embedded systems product lines. In addition, a lively group discussion tackled these issues in further detail. This report presents an overview of the presentations and fruitful discussions that took place during the ACES-MB 2010 workshop.

1 Introduction

The development of embedded systems with real-time and other critical constraints raises distinctive problems. In particular, development teams have to make very specific architectural choices and handle key non-functional constraints related to, for example, real-time deadlines and to platform parameters like energy consumption or memory footprint. The last few years have seen an increased interest in using model-based engineering (MBE) techniques to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models in a layered construction of systems.

MBE techniques are interesting and promising because they allow to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models, and they support a layered construction of systems, in which the (platform independent) functional aspects are kept separate from architectural and non-functional (platform specific) aspects, where the final system is obtained by combining these aspects later using model transformations.

The Third Workshop on *Model Based Architecting and Construction of Embedded Systems* (ACES-MB 2010) brought together researchers and practitioners

interested in all aspects of model-based software engineering for real-time embedded systems. The participants discussed this subject at different levels, from requirements specifications, model specification languages and analysis techniques, embedded systems product lines, model synthesis, to model based verification and validation.

2 Workshop Contributions

The keynote [2] was given by Prof. Lionel C. Briand from the University of Oslo and the Simula Research Laboratory, Norway, who discussed the use of metaheuristic search for the analysis and verification of UML models.

There is a growing research activity around the use of metaheuristic search techniques (e.g., genetic algorithms) in software engineering, for example to support test case generation, often referred to as search-based software engineering (SBSE). Several years of research have focused on using metaheuristic search to support the analysis and verification of UML models and its extensions such as MARTE and OCL. Examples include the analysis of real-time deadlines (schedulability analysis), concurrency problems, and constraint solving, for example for supporting model-based test case generation. Results suggest that applying metaheuristic approaches to these problems lead to practical and scalable solutions that rely solely on UML and extensions, and does not require translations into other languages and formalisms.

6 full papers and 5 short papers had been accepted for the workshop, see [1]. A synopsis of each presentation is given below. Extended versions of articles [5] and [6] are included in this workshop reader.

[3] presents a MARTE to AADL mapping that is valuable for MARTE users in order to enable the use AADL analysis tools on MARTE models. For example, CAT, the Consumption Analysis Toolbox, allows for system-level power and energy consumption estimation for AADL models.

[4] addresses the problem that real-time embedded software today is commonly built using programming abstractions with little or no temporal semantics. The paper discusses the use of an extension to the Ptolemy II framework as a coordination language for the design of distributed real-time embedded systems. Specifically, the paper shows how to use modal models in the context of the PTIDES extension of Ptolemy II.

[5] uses UML Interaction Overview Diagrams as the basis for a user-friendly, intuitive, modelling notation that is well-suited for the design of complex, heterogeneous, embedded systems developed by domain experts with little background on modelling software based systems. To allow designers to precisely analyse models written with this notation, a part of it is provided with a formal semantics based on temporal logic, upon which a fully automated, tool supported verification technique is built.

[6] argues that system development and integration with a sufficient maturity at entry into service is a competitive challenge in the aerospace sector, and can only be achieved using efficient model-based techniques for system design

as well as for system testing. Building on the general idea of model-based systems engineering, an integrated virtual verification environment for modelling systems, requirements, and test cases is proposed, so that system designs can be simulated and verified against the requirements in the early stages of system development. The paper exemplifies its application in a ModelicaML modelling environment.

[7] addresses the early validation of automobile electronic systems by providing a transformation of EAST-ADL models to SystemC at different layers of abstraction. This allows specific analysis with hardware-software co-simulation iteratively in the development process. The proposed approach is realized in a tool chain and demonstrated by an automotive use case, showing the potential of an early validation of system and software designs based on architecture models.

[8] argues that modelling tools should become development environments and support a methodologically guided development in which milestones are indicated and warnings are generated to inform the user about issues that are to be solved to reach these milestones. The paper indicates model maturity levels that correspond to an underlying development method and shows in the model maturity view which elements or parts of the model do not yet reach a certain level and why.

[9] proposes to abstract away from architectural platforms and their induced architectural styles to more abstract representation of applications. Architecture-independent application models, developed using modern model-based development techniques, can be mapped to application architectures in a variety of architectural styles. Architectural mappings therefore play an important role in synthesis of software implementations from abstract application models.

[10] proposes to integrate multiple partially overlapping models from different tools, since each tool and associated modelling language have different strengths and weaknesses. It is crucial that relevant dependencies between models and related timing properties are explicitly captured, allowing the analysis of the impact of changes on the timing properties and timing requirements. The paper proposes to use the concept of megamodels as a solution for the support of those dependencies relevant for timing properties, so that no violation may remain undetected.

[11] presents a model of an evolutionary product line process based on architecture transformations. The model attempts to give an accurate description of how real architects actually work. Key elements of the approach are how the transformations interact with consistency constraints and with feasibility in terms of resource limitations.

[12] proposes an approach for the identification of features supported by class models annotated with stereotypes. The models are automatically reverse engineered by a tool called Rejasp/Dmasp where attributes and methods are stereotyped if they have some relation with candidate features. The approach consists of four guidelines and focuses on identifying features in embedded systems for ground vehicles.

[13] states that much meaning can be given to a model using a domain specific language (DSL), and the code generation rate can be increased. Model-based product line development is possible using code generation to realize variability. The paper presents a case study where a high rate of code generation was achieved by using two DSLs, the characteristics of which supplement each other. Structure is described by a highly general DSL and behaviour by a specialised DSL. Various kinds of products have been developed from a product line efficiently by using code generation from DSLs to realize variability.

3 Summary of the Workshop Discussions

The workshop was divided into 3 sessions: modelling languages and mappings, verification and validation, and a position statement session. After each session, a group discussion was held on issues raised during the session presentations. The following integrates and summarizes the conclusions of the discussions.

Mappings between modelling languages

An important issue for mappings between modelling formalisms concerns the level of detail that can be expressed in each formalism and the ability to transform information from one modelling formalism into another. One viewpoint is to consider a mapping between modelling formalisms as a dedicated transformation for a specific purpose, e.g., model analysis in the case of the MARTE to AADL mapping. This means that a mapping can abstract away certain details from the source model that are useless for the transformation purpose. In addition, specific information can be refined or added to the target model by enlarging the target model or by using an in-between Platform-Specific Model (PSM) profile on top of the source model, e.g., an AADL profile for MARTE. OCL constraints can be added to the PSM profile in order to validate the correctness of the added information. In order to bridge the semantic gap between the two modelling levels, abstraction patterns can be introduced, e.g., by creating user-defined extensions for AADL.

Constructing architectural models

The construction of architectural models raises a number of issues, such as dealing with a multitude of inter-model dependencies and coping with model changes and the ripple effects they can cause. In addition, a product line approach can be beneficial but even further complicates the architectural modelling phase, since one should focus on an architecture for the whole product line instead of for a single product. When generative techniques are used, there was a common agreement on never touching the generated models or code. If changes seem necessary, either the source model or the generator itself should be changed.

Using a megamodel approach

Although the use of megamodels was recognised as a need in order to support different modelling formalisms, it was not clear if one should stay in the

same technological space or not. It can be difficult to arrive at a combined megamodel. Therefore, it is sometimes better to use transformations or filters to combine different models rather than try to squeeze them into a single megamodel.

Acknowledgements

This workshop was supported by the IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design (<http://www.artist-embedded.org>), the research project EUREKA-ITEA EVOLVE (<http://www.evolve-itea.org>), the research project EUREKA-ITEA VERDE (<http://www.itea-verde.org>), the research project EUREKA-ITEA OPEES (<http://www.opees.org>), and the research project ICT FP7-INTERESTED (<http://www.interested-ip.eu>).

References

1. Van Baelen, S., Ober, I., Espinoza, H., Weigert, T., Ober, I., Gérard, S., eds.: Third International Workshop on Model Based Architecting and Construction of Embedded Systems. CEUR Workshop Proceedings Vol.644, CEUR, Aachen, Germany (2010)
2. Briand, L.C.: Using metaheuristic search for the analysis and verification of UML models. In: [1]. pp. 9–9
3. Turki, S., Senn, E., Blouin, D.: Mapping the MARTE UML profile to AADL. In: [1]. pp. 11–20
4. Eidson, J.C., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: A time-centric model for cyber-physical applications. In: [1]. pp. 21–35
5. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: From interaction overview diagrams to temporal logic. In: [1]. pp. 37–51
6. Schamai, W., Helle, P., Fritzson, P., Paredis, C.J.J.: Virtual verification of system designs against system requirements. In: [1]. pp. 53–67
7. Weiss, G., Zeller, M., Eilers, D., Knorr, R.: Approach for iterative validation of automotive embedded systems. In: [1]. pp. 65–83
8. Grosse-Rhode, M.: Model maturity levels for embedded systems development, or working with warnings. In: [1]. pp. 85–99
9. Bagheri, H., Sullivan, K.: Towards a systematic approach for software synthesis. In: [1]. pp. 101–105
10. Neumann, S., Seibel, A.: Toward mega models for maintaining timing properties of automotive systems. In: [1]. pp. 107–111
11. Axelsson, J.: A transformation-based model of evolutionary architecting for embedded system product lines. In: [1]. pp. 113–117
12. Durelli, R.S., Conrado, D.B.F., Ramos, R.A., Pastor, O.L., de Camargo, V.V., Penteadó, R.A.D.: Identifying features for ground vehicles software product lines by means of annotated models. In: [1]. pp. 119–123
13. Tokumoto, S.: Product line development using multiple domain specific languages in embedded systems. In: [1]. pp. 125–129

Virtual Verification of System Designs against System Requirements

Wladimir Schamai, Philipp Helle, Peter Fritzson, and Christiaan J.J. Paredis

¹ EADS Innovation Works, Germany wladimir.schamai@eads.net

² EADS Innovation Works, UK philipp.helle@airbus.com

³ Department of Computer and Information Science, Linköping University, Sweden
petfr@ida.liu.se

⁴ Georgia Institute of Technology, Atlanta, USA chris.paredis@me.gatech.edu

Abstract. System development and integration with a sufficient maturity at entry into service is a competitive challenge in the aerospace sector. With the ever-increasing complexity of products, this can only be achieved using efficient model-based techniques for system design as well as for system testing. However, natural language requirements engineering is an established technique that cannot be completely replaced for a number of reasons. This is a fact that has to be considered by any new approach. Building on the general idea of model-based systems engineering, we aim at building an integrated virtual verification environment for modeling systems, requirements, and test cases, so that system designs can be simulated and verified against the requirements in the early stages of system development. This paper provides a description of the virtual verification of system designs against system requirements methodology and exemplifies its application in a ModelicaML modeling environment.

Keywords: Requirements, Verification, ModelicaML, Modelica, MBSE, Model-based testing

1 Introduction

The ever-increasing complexity of products has had a strong impact on time to market, cost and quality. Products are becoming increasingly complex due to rapid technological innovations, especially with the increase in electronics and software even inside traditionally mechanical products. This is especially true for complex, high value-added systems such as aircraft and automobile that are characterized by a heterogeneous combination of mechanical and electronic components. System development and integration with sufficient maturity at entry into service is a competitive challenge in the aerospace sector. Major achievements can be realized through efficient system specification and testing processes. Limitations of traditional approaches relying on textual descriptions are progressively addressed by the development of model-based systems engineering¹ (MBSE) approaches. Building on this general idea of MBSE, we aim at

¹ The International Council on Systems Engineering (INCOSE) defines MBSE as follows: "Model-based systems engineering (MBSE) is the formalized application of

building a virtual verification environment for modeling systems, requirements and test cases, so that a system design can be simulated and verified against the requirements in the early system development stages.

1.1 Scope

For our methodology we assume that the requirements from the customer have been elicited² as requirement statements according to common standards in terms of quality, e.g. according to Hull et al.[4] stating that the individual requirements should be unique, atomic, feasible, clear, precise, verifiable, legal, and abstract, and the overall set of requirements should be complete, non-redundant, consistent, modular, structured, satisfied and qualified. The methods to achieve this have been well defined and can be considered to be established. Furthermore, the overall MBSE approach to system design, that is the development of a system design model from textual requirements, is not within the scope of this paper³.

Paper structure: First we establish and describe the idea of virtual verification of system designs against system requirements (Section 2). Then we present background information on ModelicaML and the running example (Section 3) before we will explain the methodology in detail with the help of said running example (Section 4). Tool support and automation will be discussed in section 5. Finally, we close with a summary of the current status and propose a number of ideas for future research (Sections 6 and 7).

2 Virtual Verification of System Designs Against System Requirements

This chapter provides the motivation behind our work, a general description thereof and the benefits of using the virtual verification of system design against system requirements (vVDR) approach. Furthermore, related work is discussed.

2.1 Objectives

A number of studies have demonstrated that the cost of fixing problems increases as the lifecycle of the system under development progresses, e.g. Davis[7].

Thus, the business case for detecting defects early in the life cycle is a strong one. Testing thus needs to be applied as early as possible in the lifecycle to keep the relative cost of repair for fixing a discovered problem to a minimum. This

modelling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases” [1].

² A description of the various requirement elicitation, i.e. capturing, techniques can be found in [2] and [3].

³ The interested reader can find a detailed overview of existing solutions for that in [5] and [6].

means that testing should be integrated into the system design phase so that the system design can be verified against the requirements early on. To enable an automatic verification of a design model against a given set of requirements, the requirements have to be understood and processed by a computer. MBSE typically relies on building models that substitute or complement the textual requirements. Links between the model elements and the textual requirements are usually kept at the requirements' granularity level, meaning that one or more model elements are linked to one requirement. This granularity is good enough for basic traceability and coverage analysis but fails when an interpretation of a requirement's content by a computer is necessary. There is research concerning the automatic translation of natural language requirements into behavioral models to support the automation of system and acceptance testing (see e.g. [8]) but it is not widely adopted in industrial practice[9]. Formal mathematical methods may be used to express requirements, but their application requires high expertise and, hence, they are not very common in industrial practice. A recent survey came to the conclusion that "in spite of their successes, verification technology and formal methods have not seen widespread adoption as a routine part of systems development practice, except, arguably, in the development of critical systems in certain domains." [10]. The bottom line is that natural language is still the most common approach to express requirements in practice[9]. We want to provide a solution to the question of how to formalize requirements so that they can be processed and evaluated during system simulations in order to detect errors or inconsistencies in a way that is easy to understand and to apply.

2.2 vVDR Concept

Figure 1 depicts the relationship of the various engineering artifacts in the frame of vVDR.

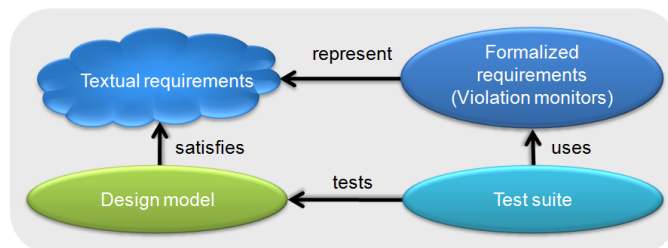


Fig. 1. Engineering data relations overview

A subset of a given set of textual requirements is selected and formalized into so-called requirement violation monitors by identifying measurable properties addressed in the requirement statement. A requirement violation monitor is

basically an executable model for monitoring if the constraints expressed by the requirement statement are adhered to. To test a given design model, the requirement violation monitors are linked to the design model using explicit assignment statements. Furthermore, a test suite consisting of a test context and a number of test cases has to be built manually. The test suite uses the formalized requirements as test oracles for the test cases, i.e., if a requirement is violated during a test, the test case is deemed failed. The separation of requirement and system design modeling provides a degree of independence that ensures a high fidelity in the testing results. The test cases, requirement violation monitors and the design model can be instantiated and run automatically. Visual graphs (e.g. plots) allow the monitoring of the requirement violation monitors during run-time to see if the design model fails to implement a requirement.

2.3 Benefits

Our approach contributes to three main steps in the system development lifecycle: requirements analysis, system design and system testing. Experience shows that the main benefit of modeling in general is a contribution to the identification of ambiguities and incompleteness in the input material. Even though we assume that the textual requirements that are provided as an input to the process adhere to a high quality standard, vVDR enables the requirements analyst to further improve the quality by modeling the requirements in a formal representation as this forces a detailed analysis of the requirements. The main contribution of vVDR is to the quality of the system design. The automatic verification of a design model based on the formalized requirements allows the detection of errors in the system design. The separation of requirements modeling and design modeling allow a reuse of the requirements for the verification of several alternative system designs. Furthermore, even for one design model the same requirements violation monitors can be instantiated several times. As described in [11], the benefits of using a model-based testing approach during the system design phase facilitates error tracing and impact assessment in the later integration and testing stages by providing a seamless traceability from the initial requirements to test cases and test results. Furthermore, it allows reusing the artifacts from the engineering stage at the testing stage of the development cycle which results in a significant decrease in overall testing effort. By integrating the requirements model in a test bench the test models can also be reused for hardware-in-the-loop test setups.

2.4 Related work

In [12] an approach to the incremental consistency checking of dynamically definable and modifiable design constraints is presented. Apart from focusing on design constraints instead of design requirements which can be argued as being a marginal issue, the main difference to vVDR is that the constraints are expressed using the design model variables whereas our approach is based on a separation of the requirements and the design model. Only for a specific test

context are they connected using explicit assignment statements. Additionally, the monitoring of model changes and the evaluation of the defined constraints is done by a separate "Model Analyzer Tool" whereas our approach relies on out-of-the-box modeling capabilities. The Behavior Modeling Language (BML) or more specifically the Requirement Behavior Tree technique that is a vital part of the BML is another method for formalizing requirements into a form that can be processed by computers[13][14]. But whereas vVDR relies on a separation between the set of independent requirements that are used to verify a design model and the building of a design model by the system designer, the BML methodology merges the behavior trees that each represent single requirements into an overall design behavior tree (DBT). In other words, the transition from the requirements space to the solution space is based on the formalized requirements.

3 Background

This chapter provides background information on the graphical modeling notation ModelicaML [15] and its underlying language Modelica [16] which was used to implement our approach, and introduces the running example that will be used to illustrate the vVDR methodology in Section 4.

3.1 Technical Background

Modelica is an object-oriented equation-based modeling language primarily aimed at physical systems. The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with difference equations/discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy, signals, or other continuous-time or discrete-time interactions between system components.

The Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering and the Systems Modeling Language (SysML) is an adaptation of the UML aimed at systems engineering applications. Both are open standards, managed and created by the Object Management Group (OMG), a consortium focused on modeling and model-based standards.

The Modelica Graphical Modeling Language is a UML profile, a language extension, for Modelica. The main purpose of ModelicaML is to enable an efficient and effective way to create, visualize and maintain combined UML and Modelica models. ModelicaML is defined as a graphical notation that facilitates different views (e.g., composition, inheritance, behavior) on system models. It is based on a subset of UML and reuses some concepts from SysML. ModelicaML is designed to generate Modelica code from graphical models. Since the ModelicaML profile is an extension of the UML meta-model it can be used as an extension for both UML and SysML⁴.

⁴ SysML itself is also a UML Profile. All ModelicaML stereotypes that extend UML meta-classes are also applicable to the corresponding SysML elements.

3.2 Running Example: Automated Train Protection System

In this section we introduce an example, which will be used in the remainder of this paper to demonstrate the vVDR approach. It is based on the example from [13]. Most railway systems have some form of train protection system that uses track-side signals to indicate potentially dangerous situations to the driver. Accidents still occur despite a train protection system when a driver fails to notice or respond correctly to a signal. To reduce the risk of these accidents, Automated Train Protection (ATP) systems are used that automate the train's response to the track-side signals. The ATP system in our example design model has three track-side signals: proceed, caution and danger. When the ATP system receives a caution signal, it monitors the driver's behavior to ensure the train's speed is being reduced. If the driver fails to decrease the train's speed after a caution signal or the ATP system receives a danger signal then the train's brakes are applied. The textual requirements for the ATP can be found in Appendix A.

4 Methodology Description

The following subsections contain a description of the method steps and illustrate the methodology using our running example.

4.1 Role: Requirements Analyst

Generally speaking, a requirements analyst acts as the liaison between the business professionals and the customer on the one hand and the system design team on the other hand. The requirements analyst is responsible for ensuring the correctness and completeness of the input requirements that are handed down from a business analyst. The objective is to create a set of requirements, which ensure that the product fulfils its original intent. The analyst needs analytical skills to critically evaluate the information gathered from multiple sources, reconcile conflicts and distinguish solution ideas from requirements [17]. In our scope, the requirements analyst is responsible for translating the textual requirements into a set of requirement violation monitors. This task includes selecting suitable requirements, creating the requirements in a formal model, identifying measurable properties and defining the requirements violation monitors.

4.2 Role: System Designer

A system designer develops the system design based on the system requirements received from the requirements analyst. An engineer in this role creates the static architecture and defines the dynamic behavior of the system. The system designer needs technical and creative skills in his work [18]. As the building of the system model is not in the scope of this paper, the system designer only has a supporting role here. He/she supports the requirements analyst in selecting the requirements for formalization and the tester in linking the formalized requirements' properties to the system design model.

4.3 Role: System Tester

[4] states that "in its broadest sense, testing is any activity that allows defects in the system to be detected or prevented, where a defect is a departure from requirements". The system tester therefore has to confirm that a particular system design meets the system requirements. The system tester builds a test model that defines the test context for a given system. He creates test cases, links requirements to design models, executes test cases, and reports test results.

4.4 Method Step: Select Requirements to Be Verified

From the set of agreed input requirements the requirements analyst selects requirements that are to be verified by means of simulation. The selection criteria depend on the requirement types as well as on the system design models that are planned to be created. Generally speaking, the requirements analyst needs to decide if the vVDR approach is suitable to test a given requirement. This step requires a close collaboration between the requirements analyst and the system designer. The output of this activity is a selected subset of the input requirements. This activity contributes to the system design modeling by clarifying the level of detail that is required of the model for an automatic evaluation of the selected requirements. For example, the requirements 001, 001-2 and 002 would not be selected because appropriate models will be missing or simulation is not best suited⁵ for their verification. In contrast, the requirements 003-009 are good candidates for the verification using simulations. The selected subset of requirements will then be transferred into the modeling tool and used in the subsequent steps.

4.5 Method Step: Formalize Textual Requirements

The second step is to formalize each requirement in order to enable its automatic evaluation during simulations. Note, that in an ideal world, i.e. in a company where the vVDR method is used throughout the development process, the requirements were already formalised at a higher engineering level by the person that formulated the requirements and allocated them to our system under development in the first place.

Consider requirement 006-1: "If at any time the controller calculates a "caution" signal, it shall, within 0.5 seconds, enable the alarm in the driver cabin." Based on this statement we can:

- Identify measurable properties included in the requirement statement, i.e., the reception of a caution signal, the activation of the alarm and the time frame constant,
- Formalize properties as shown in Fig. 2 and define a requirement violation monitor as illustrated in Fig. 3.

⁵ For example, design inspection could be sufficient.

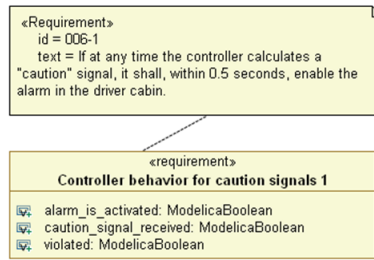


Fig. 2. Formalized requirement properties in ModelicaML

In order to determine if a requirement is fulfilled the following assumption is made: A requirement is implemented in and met by a design model as long as its requirement violation monitor is evaluated but not violated. Now the violation relations can be defined. This example uses a state machine⁶ (as shown in Fig. 3) to specify when the requirement is violated. In general, it is recommended to

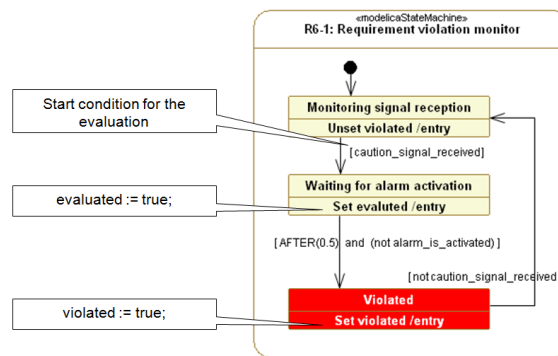


Fig. 3. Requirement violation monitor example

create the following attributes for each requirement:

- evaluated: Indicates if the requirement was evaluated at least once,
- violated: Indicates if this requirement was violated at least once.

The evaluated attribute is necessary, because, while a violation during a simulation provides sufficient indication that a requirement is not met, a non-violation is not enough to ensure the satisfaction of a requirement. For example, if the value of "caution_signal_received" is never true during a particular test case simulation this can mean that either this requirement is not addressed by the design

⁶ A ModelicaML state machine is one possible means to express the violation of a requirement. It is also possible to use other formalisms, equations or statements for it.

(i.e., the caution signals are not received by the controller at all), or that this requirement is not verified by this test case because the test case does not provide appropriate stimuli for the design model.

This method step supports the requirements analyst in improving the quality of the selected requirements by identifying ambiguities or incompleteness issues. Any issues that are identified in this step have to be resolved with the stakeholders and all affected textual requirements have to be updated accordingly.

4.6 Method Step: Select or Create Design Model to Be Verified against Requirements

The actual system design is not in the scope of this paper. The system designer builds a design model for each design alternative that he comes up with⁷. Since the requirements are kept separate from the design alternatives, the same requirements can be reused to verify several designs, and the same requirement violation monitors can be reused in multiple test cases.

4.7 Method Step: Create Test Models, Instantiate Models, Link Requirement Properties to Design Model Properties

After the formalization of the requirements and the selection of one design model for verification, the system tester starts creating test models, defining test cases and linking requirement properties to values inside the design model. The recommended procedure is as follows:

- Define a test model that will contain test cases, a design model, the requirements and their requirement violation monitors.
- Define test cases for evaluating requirements. One test case can be used for evaluating one or more requirements.
- Create additional models if necessary, for example, models that simulate the environment, stimulate the system or monitor specific values.
- Bind the requirements to the design model by setting the properties of a requirement to values inside the design model using explicit assignments.

Particularly the last step will require the involvement of the system designer in order to ensure that the requirement properties are linked properly, i.e. to the correct properties values inside the design model. For example, the assignment for the requirement property *caution_signal_received* is as follows:

```
caution_signal_received =  
design_model.train1.pc1.tcs.controller.tracks_signals_status == 1
```

⁷ For ease of use, the design will normally be modelled in the same notation and the same tool as the requirements. However, it can be imagined to build interfaces to executable models that were built using different modelling notations in different tools and then subsequently use vVDR to test these models.

This means that the requirement property *caution_signal_received* will become true when the controller property *tracks_signals_status* is equal to one⁸.

Another example is the assignment of the requirement property *alarm_is_activated*. Here the system tester will have to decide which design property it should be linked to. It could be accessed from the ATP controller or from the HMI system, that is between the controller and the driver, or from the driver HMI port directly. The answer will probably be: It should be accessed from the driver HMI port because failures in HMI system may also affect the evaluation result. Furthermore, it is recommended to create the following attributes and statements⁹ for each test model:

- `test_passed := evaluated and not violated;`
Indicates if the test is passed or failed.
- `evaluated := if req1.evaluated and ... and reqN.evaluated then true ...;`
Indicates if the test case has evaluated all requirements.
- `violated := when {req1.violated, ... ,reqN.violated} then true ...;`
Indicates if any of requirements was violated.

These definitions enable an automated test case results evaluation by using the requirement violation monitors of the involved requirements as a test oracle for the test case. Figure 4 presents an example of a test case that drives the simulation.

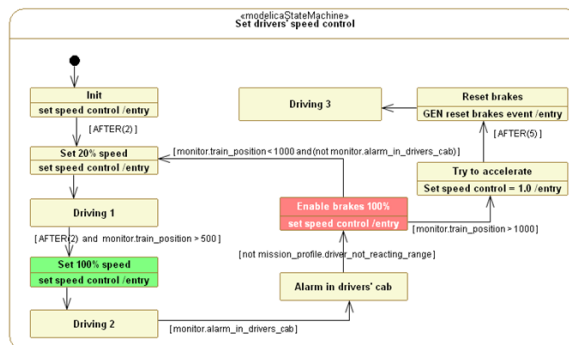


Fig. 4. Test case example

4.8 Method Step: Test and Observe Requirement Violations

After having created the test models, the system tester can run simulations and observe the results. Hereby, the system tester will be interested in knowing if

⁸ "1" denotes a caution signal in the design model

⁹ These statements are written in Modelica.

test cases have passed or failed. A test case is deemed to have failed when not all requirements were evaluated or some requirements were violated during the execution of the test case.

4.9 Method Step: Report and Analyze Test Results

After the execution of all test cases, the system tester creates a simulation report. This information is the basis for discussions among the involved parties and may lead to an iterative repetition of the system design and testing process described here. Furthermore, it allows the precise reproduction of test results at a later state. Additionally, these reports can be reused as a reference for later product verification activities, i.e., the physical system testing at a test bench.

5 Tool support and automation

Past experience shows that the acceptance of new methods by systems engineers highly correlates with the ease of application of said new method. The field of formal methods, although it is proven that its application can reduce errors drastically, is just one example where a good method was not adapted widely in the daily work of the engineers mainly due to its steep learning curve. To ease the adaptation of vVDR by systems engineers, a number of assistant tools has been developed that is constantly growing. These tools that are integrated into the vVDR development environment aim at simplifying complex and/or repetitive tasks within the vVDR method by automation.

For the linking of requirement properties to design properties a wizard for preparing the binding statements has been developed as shown by Fig. 5. The wizard collects all requirements that have been instantiated within a selected simulation model and prepares a Modelica modification in that simulation model with assignment statements that have the input variables of the requirements on the left hand side of the equation and "TBD" on the right hand side of the equation. Therefore, the tester who prepares the simulation now only has to go through these statements and can replace the "TBD" on the right hand side with a pointer to the appropriate variable in the design model. Without this helper, the tester would need to look manually through each instantiated requirement and manually write the complete assignment statement for all the requirement's input variables.

A second improvement in the vVDR development environment aims at simplifying the same task for the system tester. As shown in Fig. 6 a dedicated view supports the task of binding requirements properties to design model properties and the task of discovering the right stimuli for the design model in a test case. This view filters the model instance tree for inputs and outputs. Inputs are potential stimuli. Outputs are potential observation points. This view limits the potential possibilities that can be placed on the right hand side of the properties binding assignment and gives the system tester an easy tool for identifying the right ones.

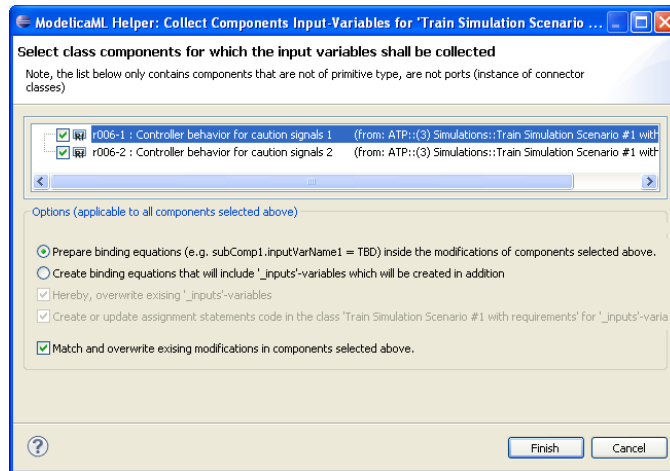


Fig. 5. Wizard for variables binding support

Another helper was built for automatically writing the overall test evaluation code (see 4.7) where the evaluation and violation of the individual requirements is combined to provide an overall test verdict. The helper is able to create the code and update it after a model change.

6 Current Status and Future Directions

The methodology presented in this paper has been successfully applied in several case studies. However, the case studies included only a small number of requirements. In the future, a real-sized case study is planned, i.e., one that contains more than a hundred requirements to be verified using the vVDR method to determine the applicability and scalability of this approach.

The traceability between requirements and design artefacts is a critical issue in the daily engineering work, particularly with regards to change impact analysis. vVDR already support this but we aim at improving its capabilities. We see the need to improve the level of granularity at which requirements and model elements are linked with each other to support change impact analysis more efficiently. For example, parts of a requirement statement, i.e., single words, can be linked to the model elements that they are referring to. Moreover, an effective visualization and dependencies exploration is necessary.

A model-based development approach enables an effective and efficient reporting on and monitoring of the requirements implementation. For example, a bidirectional traceability between requirement and design allows the determination of the system development status and supports project risk management and planning. Template-based reporting engines support automatic generation of various kinds of reports on demand.

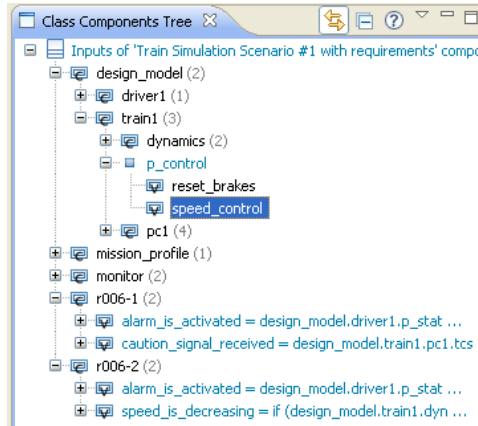


Fig. 6. View to support input and output variables connection

While the test cases for our running example can be easily derived directly from the input requirements, manual test case generation becomes an increasingly tedious task for real-life specifications with hundreds of requirements. Model-based testing provides methods for automated test case generation some of which already work on UML models[19] and look promising to be adapted to vVDR.

Requirements traceability, a higher test automation through adaptation of model-based testing techniques as well as reporting topics are subject to our future work.

7 Conclusion

This paper presents a method for the virtual verification of system designs against system requirements by means of simulation. It provides a detailed description of all method steps and illustrates them using an example case study that was implemented using ModelicaML. It points out that this method strongly depends on the design models that are planned to be created and that not all type of requirements can be evaluated using this method. In the vVDR approach, formalized requirements, system design and test cases are defined in separate models and can be reused and combined into test setups in an efficient manner. In doing so, a continuous evaluation of requirements along the system design evolution can be done starting in the early system design stages. This approach enables an early detection of errors or inconsistencies in system design, as well as of inconsistent, not feasible or conflicting requirements. Moreover, the created artifacts can be reused for later product verification (i.e., physical testing) activities.

References

1. C. Haskins, Ed., *Systems Engineering Handbook: A guide for system life cycle processes and activities*. INCOSE, 2006.
2. D. Gause and G. Weinberg, *Exploring requirements: quality before design*. Dorset House Pub, 1989.
3. P. Loucopoulos and V. Karakostas, *System requirements engineering*. McGraw-Hill, Inc. New York, NY, USA, 1995.
4. E. Hull, K. Jackson, and J. Dick, *Requirements engineering*. Springer Verlag, 2005.
5. J. Estefan, "Survey of model-based systems engineering (MBSE) methodologies," *IncoSE MBSE Focus Group*, vol. 25, 2007.
6. P. Helle, A. Mitschke, C. Strobel, W. Schamai, A. Rivière, and L. Vincent, "Improving Systems Specifications - A Method Proposal," in *Proceedings of CSER 2008 Conference, April 4-5 2008, Los Angeles, CA*, 2010.
7. A. Davis, *Software requirements: objects, functions, and states*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.
8. V. A. d. Santiago Júnior, *Natural language requirements: automating model-based testing and analysis of defects*. São José dos Campos: Instituto Nacional de Pesquisas Espaciais, 2010.
9. L. Mich, M. Franch, and P. Novi Inverardi, "Market research for requirements analysis using linguistic tools," *Requirements Engineering*, vol. 9, no. 2, pp. 151–151, 2004.
10. J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009.
11. P. Helle and W. Schamai, "Specification model-based testing in the avionic domain - Current status and future directions," in *Proceedings of the Sixth Workshop on Model-Based Testing 2010, Paphos, Cyprus*, 2010.
12. I. Groher, A. Reder, and A. Egyed, "Incremental Consistency Checking of Dynamic Constraints," *Fundamental Approaches to Software Engineering*, pp. 203–217, 2010.
13. T. Myers, P. Fritzson, and R. Dromey, "Seamlessly Integrating Software & Hardware Modelling for Large-Scale Systems," in *2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, Paphos, Cyprus*, 2008.
14. D. Powell, "Requirements evaluation using behavior trees-findings from industry," in *Australian Software Engineering Conference (ASWEC07)*, 2007.
15. W. Schamai, P. Fritzson, C. Paredis, and A. Pop, "Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations," in *Proc. of the 7th International Modelica Conference, Como, Italy*, 2009.
16. P. Fritzson, *Principles of object-oriented modeling and simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.
17. S. Sheard, "Twelve systems engineering roles," in *Proceedings of INCOSE*, 1996.
18. J. Morrell, C. Mawhinney, G. Morris, W. Haga, and A. Smolkina, "The systems analyst: a post mortem?" in *Proceedings of the Fourth International Conference of the Academy of Business and Administrative Sciences, Quebec City, Canada*, 2001.
19. M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan, "A survey on automatic test case generation," *Academic Open Internet Journal*, vol. 15, 2005.

A ATP requirements

ID	Requirement Text (based on [13])
001	The ATP system shall be located on board the train.
001-2	The ATP system shall consist of a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.
002	The sensors shall be attached to the side of the train and read information from approaching track-side signals, i.e. they detect what the signal is signaling to the train driver.
002-2	Within the driver cabin, the train control display system shall display the last track-side signal values calculated by the controller.
003	Three sensors shall generate values in the range of 0 to 3, where 0, 1 and 2 denote the danger, caution, and proceed track-side signals respectively. Each sensor shall generate the value 3 if a track-side signal that is out of the range 0..2 is detected.
004	The controller shall calculate the majority of the three sensor readings. If no majority exists then the value shall be set to "undefined" (i.e. 3).
005	If the calculated majority is "proceed" (i.e. 0) then the controller shall not take any action with respect to the activation of the braking system.
006-1	If at any time the controller calculates a "caution" signal, it shall, within 0.5 seconds, enable the alarm in the driver cabin.
006-2	If the alarm in the driver cabin has been activated due to a "caution" signal and the train speed is not decreasing by at least $0.5m/s^2$ within two seconds of the activation, then the controller shall within 0.5 seconds activate the automatic braking.
007-1	If at any time the controller calculates a "danger" signal it shall within 0.5 seconds activate the braking system and enable the alarm in the driver cabin.
007-2	If the alarm in the driver cabin has been activated due to a "caution" signal, it shall be deactivated by the controller within 0.5 seconds if a "proceed" signal is calculated and the automatic braking has not been activated yet.
008	If at any time the automatic braking has been activated, the controller shall ignore all further sensor input until the system has been reset.
009	If the controller receives a reset command from the driver, then it shall within 1 second, deactivate the train brakes and disable the alarm within the driver cabin.

From Interaction Overview Diagrams to Temporal Logic^{*}

Luciano Baresi, Angelo Morzenti, Alfredo Motta, 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 work, which is part of a larger research effort carried out in the MADES European project¹ [1], focuses on aspects (i) and (ii) mentioned above. In particular, it is the first step towards a complete proposal for modeling and validating

^{*} 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.

¹ <http://www.mades-project.org>

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 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).

² Zot is available at <http://home.dei.polimi.it/pradella/Zot>.

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 Zot

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 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.

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

Table 1. TRIO derived temporal operators

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 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 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

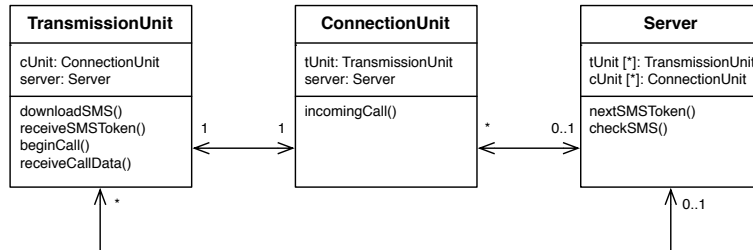


Fig. 1. Class diagram for the telephone system.

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*.

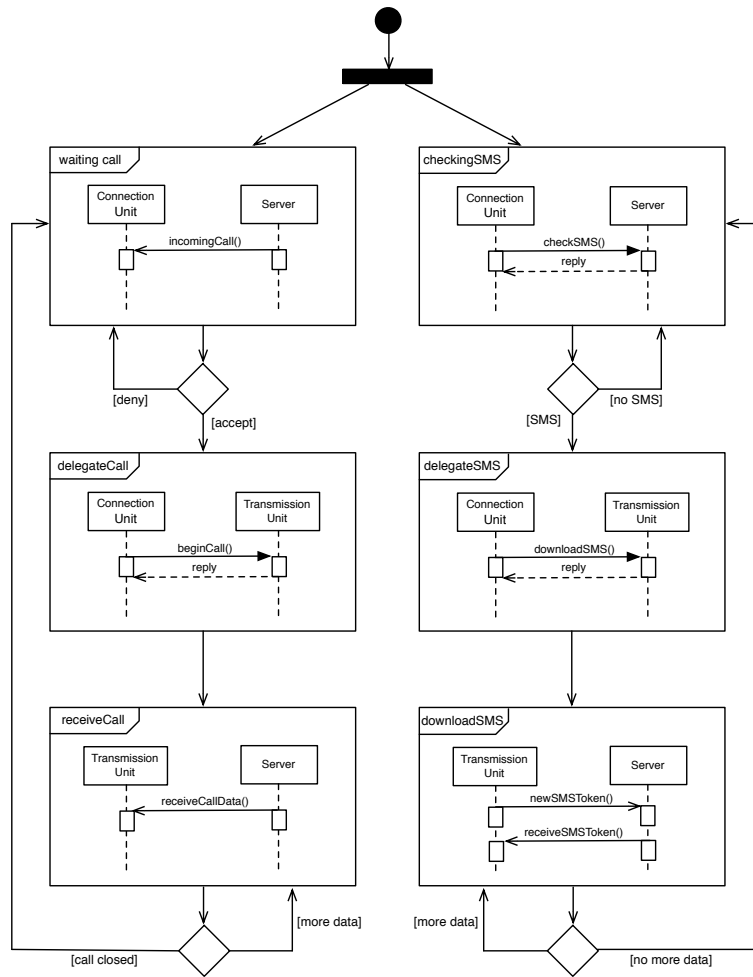


Fig. 2. Interaction Overview diagram for the telephone system.

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).

$$D_xSTART \Leftrightarrow m_s \quad (1)$$

$$D_xEND \Leftrightarrow m_e \quad (2)$$

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

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

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

$$delegateSMSSTART \Leftrightarrow downloadSMS \quad (4)$$

$$delegateSMSSEND \Leftrightarrow reply3 \quad (5)$$

$$delegateSMS \Leftrightarrow delegateSMSSTART \vee \quad (6)$$

$$Since(\neg delegateSMSSEND, delegateSMSSTART)$$

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

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 ($i \in \{1 \dots p\}$), 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).

$$D_xACT \Leftrightarrow D_xACTC_0 \vee \dots \vee D_xACTC_m \quad (7)$$

$$D_xACT \Rightarrow \text{SomF}(D_xSTART) \quad (8)$$

$$D_xSTART \Rightarrow \neg \text{SomF}(D_xSTART) \vee \text{Until}(\neg D_xSTART, D_xACT) \quad (9)$$

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.

$$downloadSMSACTC_1 \Leftrightarrow delegateSMSSEND \quad (10)$$

$$downloadSMSACTC_2 \Leftrightarrow downloadSMSSEND \wedge moredata \quad (11)$$

$$downloadSMSACT \Leftrightarrow \left(\begin{array}{l} downloadSMSACTC_1 \\ \vee downloadSMSACTC_2 \end{array} \right) \quad (12)$$

$$\text{downloadSMSACT} \Rightarrow \text{SomF}(\text{downloadSMSSTART}) \quad (13)$$

$$\begin{aligned} \text{downloadSMSSTART} &\Rightarrow \\ &\neg \text{SomF}(\text{downloadSMSSTART}) \vee \\ &\text{Until}(\neg \text{downloadSMSSTART}, \text{downloadSMSACT}) \end{aligned} \quad (14)$$

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 .

$$m_i \Rightarrow \text{SomF}(m_j) \wedge \neg m_j \quad (15)$$

$$m_j \Rightarrow \text{SomP}(m_i) \wedge \neg m_i \quad (16)$$

$$m_j \Rightarrow \neg \text{SomF}(m_j) \vee \text{Until}(\neg m_j, m_i) \quad (17)$$

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

$$\text{checkSMS} \Rightarrow \text{SomF}(\text{reply1}) \wedge \neg \text{reply1} \quad (18)$$

$$\text{reply1} \Rightarrow \text{SomP}(\text{checkSMS}) \wedge \neg \text{checkSMS} \quad (19)$$

$$\text{checkSMS} \Rightarrow \neg \text{SomF}(\text{checkSMS}) \vee \text{Until}(\neg \text{checkSMS}, \text{reply1}) \quad (20)$$

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 *checkSMS* 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 or 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.

$$m_i \vee \text{Since}(\neg m_j, m_i) \Leftrightarrow \text{ABLOCKED}_h \quad (21)$$

$$m_i \vee \text{Since}(\neg m_j, m_i) \Leftrightarrow \text{BBLOCKED}_k \quad (22)$$

$$m \Leftrightarrow \text{ABLOCKED}_h \quad (23)$$

$$m \Leftrightarrow \text{BBLOCKED}_k \quad (24)$$

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.

$$\forall 1 \leq i, j \leq n (i \neq j \wedge \text{ABLOCKED}_i \Rightarrow \neg \text{ABLOCKED}_j) \quad (25)$$

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

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

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:

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

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

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

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:

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

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.

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

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

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/Satisfiability 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.

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 $2n_d n_o(n_o-1)$ messages. Every message instance has its own predicate and this results in $2n_d n_o(n_o-1)$ generated predicates. Moreover we have $5 \cdot 2n_d n_o(n_o-1)$ 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 $n_d n_o 2(n_o-1)$ predicates overall. The mutual exclusion of these predicates is stated in axiom (25); because every object has $n_d 2(n_o-1)$ operation executions, we have $n_d 2(n_o-1)(n_d 2(n_o-1)-1)$ instances of axiom (25) for each object, and $n_o n_d 2(n_o-1)(n_d 2(n_o-1)-1)$ 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 π -*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. A. Bagnato, A. Sadovykh, R. F. Paige, D. S. Kolovos, L. Baresi, A. Morzenti, and M. Rossi. 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. M. M. Bersani, A. Frigeri, M. Pradella, M. Rossi, A. Morzenti, and P. San Pietro. Bounded reachability for temporal logic over constraint systems. In *Proc. of the Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 43–50, 2010.
3. G. Blohm and A. Bagnato. D1.1 requirements specification. Technical report, MADES Consortium, 2010. Draft.
4. T. Bouabana-Tebibel. Semantics of the interaction overview diagram. In *Proc. of the IEEE Int. Conf. on Information Reuse Integration (IRI)*, pages 278–283, 2009.
5. M. V. Cengarle, P. Graubmann, and S. Wagner. Semantics of UML 2.0 interactions with variabilities. *Elec. Notes in Theor. Comp. Sci.*, 160:141–155, 2006.
6. M. V. Cengarle and A. Knapp. Operational semantics of UML 2.0 interactions. Technical Report TUM-I0505, Technische Universität Mnchen, 2005.
7. E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM TOSEM*, 8(1):79–113, 1999.
8. R. Eshuis. Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.
9. R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Trans. Software Eng.*, 30(7):437–447, 2004.
10. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. In *Proceedings of the International Conference on the Implementation and Application of Automata*, volume 2088 of *LNCS*, pages 1–33, 2000.
11. L. Kloul and J. Küster-Filipe. From intraction overview diagrams to PEPA nets. In *Proc. of the Work. on Process Algebra and Stochastically Timed Activities*, 2005.
12. A. Knapp and J. Wuttke. Model checking of UML 2.0 interactions. In *Models in Software Engineering*, volume 4634 of *LNCS*, pages 42–51, 2007.
13. V. S. W. Lam. On -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. M. Pradella, A. Morzenti, and P. San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *Proceedings of ESEC/SIGSOFT FSE*, pages 312–320, 2007.
17. T. S. Staines. Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. *Proc. of the IEEE Int. Conf. on the Engineering of Computer-Based Systems*, pages 191–200, 2008.
18. H. Störrle and J. H. Hausmann. Towards a formal semantics of UML 2.0 activities. In *Software Engineering*, volume 64 of *Lect. Notes in Inf.*, pages 117–128, 2005.
19. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proc. of the Int. Conf. on Software Engineering*, pages 188–197, 2001.