Supporting UML-based Development of Embedded Systems by Formal Techniques*

Jozef Hooman
Radboud University Nijmegen, The Netherlands
Embedded Systems Institute, Eindhoven, The Netherlands
Hillel Kugler
New York University, New York, NY, USA
Iulian Ober
Toulouse-II University, Toulouse, France
Anjelika Votintseva
Siemens, Germany
Yuri Yushtein
CIMSOLUTIONS B.V., The Netherlands

To appear in the journal on Software and Systems Modeling

Abstract

We describe an approach to support UML-based development of embedded systems by formal techniques. A subset of UML is extended with timing annotations and given a formal semantics. UML models are translated, via XMI, to the input format of formal tools, to allow timed and non-timed model checking and interactive theorem proving. Moreover, the Play-Engine tool is used to execute and analyze requirements by means of live sequence charts. We apply the approach to a part of an industrial case study, the MARS system, and report about the experiences, results and conclusions.

1 Introduction

We report about our results and experiences on combining a number of formal techniques with UML-based development. This work has been carried out in the context of the EU project OMEGA (Correct Development of Real-Time Embedded systems in UML). A general aim of this project is to improve the quality of software for embedded systems by the use of formal techniques.

Embedded systems typically have an intensive real-time interaction with their environment. Although UML [3] has not been designed originally for such systems, one can observe an increasing use of object-oriented techniques and UML in this domain. There exists a number of specialized methods [43, 11], a UML profile for Schedulability, Performance and Time [35] and several dedicated CASE tools (e.g., Artisan's Real-time Studio, Rhapsody of I-Logix, Rational Rose RealTime, and Telelogic TAU).

Embedded applications typically have strong requirements on the correctness of the software. This, however, is not easy to achieve, since these applications are typically highly innovative, have intricate assumptions about the behavior of their environment, and are developed quite incrementally. Hence, it is important to detect errors as soon as possible during the development process. In this paper, we address the use of formal methods to improve the quality of UML-based development of embedded systems.

*This work has been supported by EU-project IST 33522 – OMEGA "Correct Development of Real-Time Embedded Systems in UML". For more information, see `http://www-omega.imag.fr/`. During this project, the second author was at the Weizmann Institute of Science, the third author at VERIMAG, the fourth author at OFFIS, and the fifth author at NLR.

There exists already a number of formal techniques that have been applied to UML [41, 30]. Early approaches to formal verification based on model checking UML models actually only consider single sub-languages of UML, like state-charts [29, 26], and they effectively verify only a single object in isolation. The vUML tool [38] provides a predefined set of checks of invariants, e.g., absence of deadlocks, queue overflows, and unreachability of invalid states. The specification language of [44] is the temporal logic of the underlying model checker, hence far from the level of UML. The work on model checking of xUML [46] is closest to the UVE approach described in this paper. A rich set of UML language concepts and features, like parallelism, inheritance, object creation/destruction, etc. are supported by the xUML approach. But it only deals with closed systems without taking a non-deterministic environment into account. Moreover, the used requirement specification language is restricted to a set of temporal patterns and has no graphical representation.

Related to the IFx tool considered in this paper for model checking real-time properties is the translation of timed UML state machines [24] to Uppaal [28]. Relevant is also the work in the context of the Fujaba real-time tool suite for UML-based development on the integration of Uppaal [7]. To support interactive verification of untimed UML models, a development environment has been developed [45], based on the theorem prover PVS [36]. A proposal for a general framework to integrate tools for UML and formal methods can be found in [32].

However, most techniques are not coupled to CASE tools and are based on a very small subset of UML. Often, it is difficult to express timing properties conveniently and at a sufficient level of abstraction, and assumptions about the environment can usually be expressed only by including the environment explicitly in the model, leading to a closed system. In the OMEGA project we investigated how this situation can be improved. We mention a few important points of the OMEGA approach: a tight integration of formal techniques in the development process, a sufficiently large subset of UML which allows convenient modeling of embedded systems, and the combination of techniques.

To be able to integrate formal techniques, we have established a coupling with commercial UML-based CASE tools by translating the standard XMI representation of a UML model into the format of formal tools, such as model checkers and theorem provers. In this way, the formal tools can be applied to a UML model that has been edited by means of any commercial UML-based CASE tool which is able to generate XMI. Although XMI is the XML standard for UML, unfortunately, most current UML tools use slightly different versions of XMI. In OMEGA we have concentrated on the XMI versions of Rational Rose and Rhapsody of I-Logix. The latter has been used for all experiments described in this paper.

To obtain a coherent set of tools, without having to deal immediately with the full UML language, we have defined a convenient subset of UML, called *the OMEGA kernel language*, which is close to the core UML language described in [13]. Basically, this consists of class diagrams for specifying the structure of the system, including structural relationships like generalization, association, and composition, and state machines to describe the behavior of classes. Objects may communicate by means of (asynchronous) signals and operations.

The language has been extended with suitable primitives to express real-time behavior. Timing extensions have been proposed, called *the OMEGA real-time profile for UML*, based on the profile for Schedulability, Time and Performance. Details can be found in [12]. The sequence diagrams of UML have been replaced by LSCs [8] which are more expressive [25] and also have been extended with primitives to express timing [19]. LSCs can be captured by the user by means of a separate tool, the Play-Engine. Note that our work on UML was mainly based on UML 1.4, since that was the standard during most of our project. Because we concentrate on the core modeling capability of UML, the differences with UML 2.0 [34] are not very relevant.

Clearly the coherence of this tool set also requires a common semantic model. Within OMEGA, this led to extensive discussions and decisions on semantic variation points and unclear issues in the definition of UML. Our starting point was an operational semantics [10, 9] which was especially inspired by the execution mechanism of Rhapsody. Whereas many formal methods require flat state machines, this semantics also includes hierarchy and orthogonality which is convenient for modeling. This semantic model, based on labeled transition systems, turned out to be convenient for the integration of the commercial tool and model checking.

We reformulated the semantics to make it more suitable for interactive theorem proving. For instance, we abstracted from the pending request table for operation calls in [9] and used explicit synchronization

between caller and callee. Moreover, the semantics has been defined in an incremental way, starting from a basic non-timed semantics. This has been extended with a continuous notion of time in an orthogonal way. Similarly, threads of control have been added in a modular way. More details can be found in [23], which clarifies a number of semantic questions and decisions, e.g., concerning the passing of control and the dispatching of signal events.

Another relevant aspect of the OMEGA approach is the combination of various formal techniques, to obtain flexible support with e.g., various specification styles, different visualizations, the possibility to deal with both closed and open systems - with assumptions about the environment, and both automated checks and user guided verification, depending on the properties to be verified. By experimenting with various tools on industrial examples, the aim is to derive guidelines about when and how to use the formal techniques. In particular, we consider the support of UML-based development by the following four formal techniques:

- Live Sequence Charts (LSCs), to capture specifications, using the Play-Engine tool
- Model checking of functional properties by means of the UVE tool
- Timed model checking, using the IFx tool
- Interactive verification supported by the PVS theorem prover

More details about these techniques will be given in Sect. 3.

The main aim of this paper is to describe the application of the formal techniques to UML models of an industrial case study, which has been provided by one of the industrial partners of the OMEGA project. We present the results of applying the first versions of the developed tools to the original model, leading to an intermediate conclusion about what had to be improved. Next the application has been remodeled and we put more emphasis on the combined application of the improved tools.

In general, the emphasis of this paper is on global results, experiences with the case studies, and general conclusions. Hence, we will neither expose all features of the techniques used, nor show the full functionality of all tools. During the case study we have often used preliminary versions of the tools, and the experiments illustrate tool development within the OMEGA project. Moreover, note that the aim is not to compare the tools, which would also be difficult because they have different goals and are rather complementary. Instead, the focus is on the synergy of the approaches and the possibilities to exploit the combination of the tools.

The rest of this paper is structured as follows. In Sect. 2 we introduce the industrial case study, the UML model of a part of it, and the properties to be verified for this part. Sect. 3 contains a brief introduction to the four formal techniques used, illustrated by their application to the UML model of the case study. Next, in Sect. 4 we redesign the considered part of the case study, to facilitate compositional techniques and abstraction, and apply the OMEGA techniques to this new model. An evaluation of the specification and verification experiments is presented in Sect. 5. Finally, concluding remarks can be found in Sect. 6.

2 The MARS system

We describe a selected part of the MARS system (Medium Altitude Reconnaissance System) from the NLR[1]. This system has been used as a common real-time embedded application within the OMEGA project.

The system controls the operation of a reconnaissance photo camera in an aircraft; ground survey photographs are taken by the camera during the flight. To obtain high-resolution images, the MARS system counteracts the image quality degradation caused by the forward motion of the aircraft by creating a compensation motion of the film during the film exposure, based on the current aircraft altitude, ground speed, etc. The system also performs health monitoring and alarm processing functions. The Reconnaissance Control Unit (RCU) is responsible for the three major categories of tasks: camera and film exposure control, film annotation, and system health monitoring and alarm processing.

[1] National Aerospace Laboratory, the Netherlands, http:// www.nlr.nl

In order to perform the camera and film-exposure control functions the RCU acquires the current altitude and velocity data from the avionics data bus of the aircraft. Based on these values it computes the film Frame Rate to be used and the value for the Forward Motion Compensation (FMC) signal. The computed values are sent to the trigger and exposure module via a serial link. To perform the film annotation functions, the RCU acquires the current navigation data (latitude, longitude and heading) as well as the time-of-day value from the avionics data bus of the aircraft. It formats the data and sends it to the annotation module via the serial link. Upon completion of each frame exposure, the camera halts the film and issues an annotation request to the annotation module. Upon reception of this request the annotation module provides the current annotation data to the camera to annotate the current frame. The annotation cycle must be completed before the next frame exposure begins.

The navigation and altitude data messages are provided by the corresponding subsystems of the aircraft. These data sources are independent and not synchronized; they provide data with a period P of 25 ms and a jitter J of +/- 5 ms (i.e. data may arrive up to 5 ms earlier or later). Data messages may occasionally be lost due to transmission errors.

The system has hard timing constraints, such as requirements on the age of data used for exposure control, on the data acquisition and on processing time in order to accommodate the data rate of the avionics data bus. The system is mission-critical as it is used for medium altitude reconnaissance missions over potentially hostile territories. Corrupted mission results will involve unnecessary additional risks to both the aircraft and the pilot in repeated attempts to execute the mission.

2.1 UML model of part of the MARS system

Here we present only a small part of the complete system, namely the data bus manager which is part of the data acquisition subsystem of the MARS system; the class diagram is given in Fig. 1. For simplicity, we

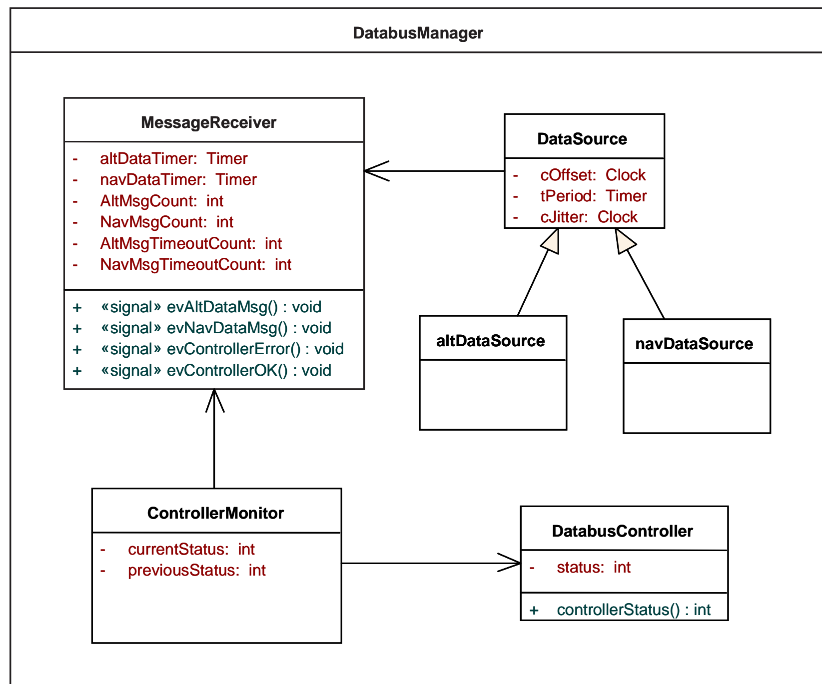

Figure 1: Class Diagram of the data bus manager

often refer to the *DatabusManager* as the MARS system. The focus is on the classes *ControllerMonitor* and *MessageReceiver* in an environment represented by the classes *DatabusController*, *altDataSource*, and *navDataSource*. The last two actors represent the data sources for altitude and navigation data. The main class is the *MessageReceiver* which processes the incoming data. The controller monitor periodically

calls an operation of the bus controller to obtain its status. In case of an error the monitor will send the *evControllerError* signal to the message receiver. The monitor sends the *evControllerOK* signal to the receiver if the bus controller indicates that the error situation is resolved.

For each class the behavior of its objects is defined by means of a state machine and methods (program text) for its so-called primitive operations. Non-primitive operations of a class are defined by means of its state machine (this is not used in the MARS case study).

The state machine of the class *DataSource*, is depicted in Fig. 2. It expresses that - non-deterministically - either data is sent, represented by primitive operation *sendData*, or no data is transmitted. The primitive operation *sendData* is overridden by the subclasses *altDataSource* and *navDataSource* to generate events *evAltDataMsg* and *evNavDataMsg*, respectively. This state machine uses interval conditions on clocks to model the non-determinism introduced by the starting time and by jitter. All transitions here are interpreted as delayable according to the terminology of timed automata with urgency [4], meaning that once they are enabled, they will be taken before their time guard becomes false (unless they are disabled by some other discrete transition). Together with usual non-Zenoness assumption, this guarantees in this example that the computation cannot get stalled in any state.

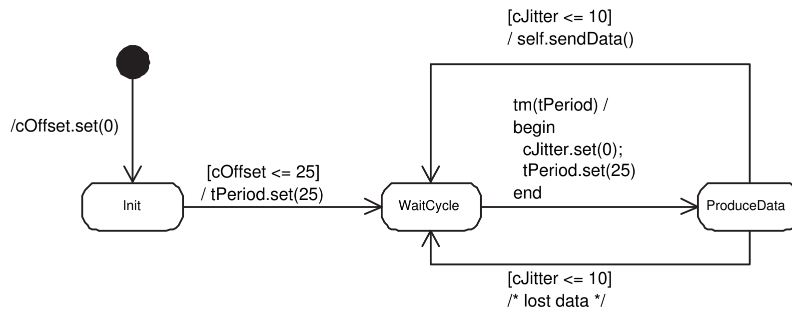

Figure 2: State machine diagram of a data source

The behavior of the *MessageReceiver* is modeled by the state machine diagram depicted in Fig. 3. The specification of the system expresses that a few failures from the data sources can be tolerated, but

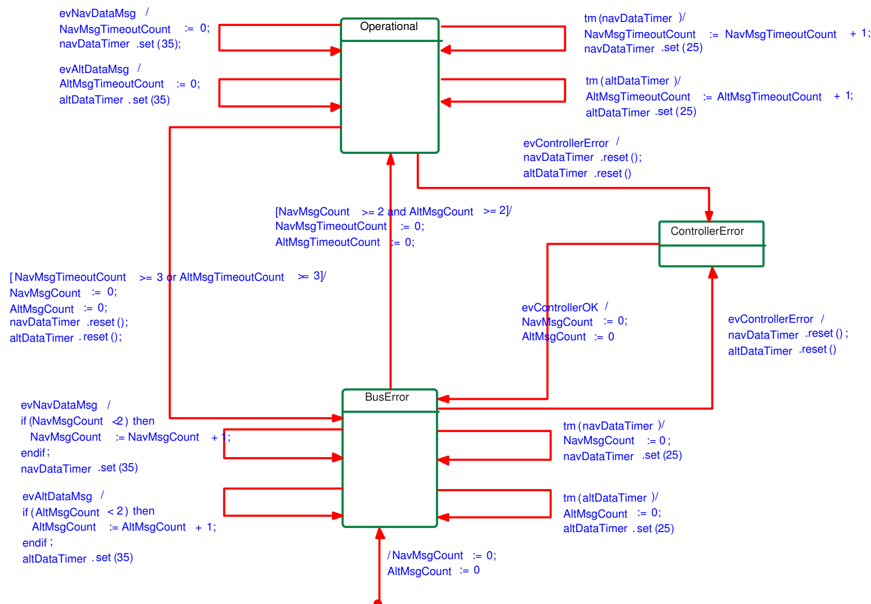

Figure 3: State machine diagram of the Message Receiver

if one of the sources fails to send data for three consecutive times, the receiver enters a *BusError* state. In state *Operational* failures are detected by the time-out of timers *navDataTimer* and *altDataTimer*. The number of consecutive failures of each data source is stored into failure counters *NavMsgTimeoutCount* and *AltMsgTimeoutCount*. If one of these counters has value 3, the receiver enters the bus error state (note that by the run to completion assumption adopted in UML, this will be done before accepting a new signal event).

The receiver recovers from the bus error state if it receives correct data for at least two consecutive times from both data sources. In the *BusError* state, the counters *NavMsgCount* and *AltMsgCount* are used to count consecutively accepted messages of each type. *BusError* is also the initial state, since the system should only be in state operational when sufficient messages have been received from both sources. State *ControllerError* is entered and exited based on the events received from the controller monitor.

Similarly, the behavior of the *ControllerMonitor* has been modeled by means of state machines. Details are not shown here, since the main focus is on the detection of failures of the data sources and the response to these failures.

2.2 Properties of the MARS system

During our verification experiments, we concentrated on the following two properties of the MARS system:

1. Timely detection of a Databus Controller error, leading to the *ControllerError* state of the Message Receiver, and proper recovery, i.e. returning to state *BusError* if the Controller is OK again.

2. Timely detection of an error in the databus, based on data message arrival monitoring, leading to state *BusError*, and proper recovery.

These properties include timing constraints to specify maximum response times. As shown in Fig. 4, for property 2 this can be split into two cases:

- Maximal response time to errors. This is defined as the upper bound on the time $R1$ between the moment the last message has been received from a faulty data source and the moment of the switch to state *BusError*.

- Maximal response time to recovery. This is defined as the upper bound on the time $R2$ between the receipt of the first message in a series of correct messages and the actual moment of the switch to state *Operational*.

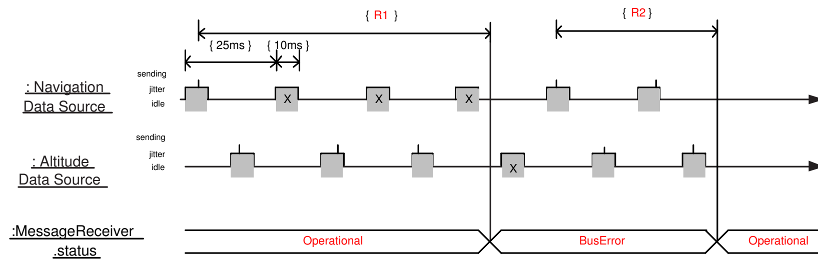

Figure 4: Response times of the Message Receiver

2.3 Abstracting from real-time

Often system specifications are split into functional and non-functional (including timing) requirements. This provides an opportunity to apply non-timed modeling and verification to the functional properties. To investigate this on the MARS system, we have extracted a non-timed version of the message receiver by abstracting from the setting of timers to particular time-out values. In order to make message loss observable in this non-timed model, the signals *evNavDataMsgTimeout* and *evAltDataMsgTimeout* have

been introduced. The data sources non-deterministically choose between sending a data message or sending a time-out event which models message loss. The state machine of the message receiver is depicted in Fig. 5.

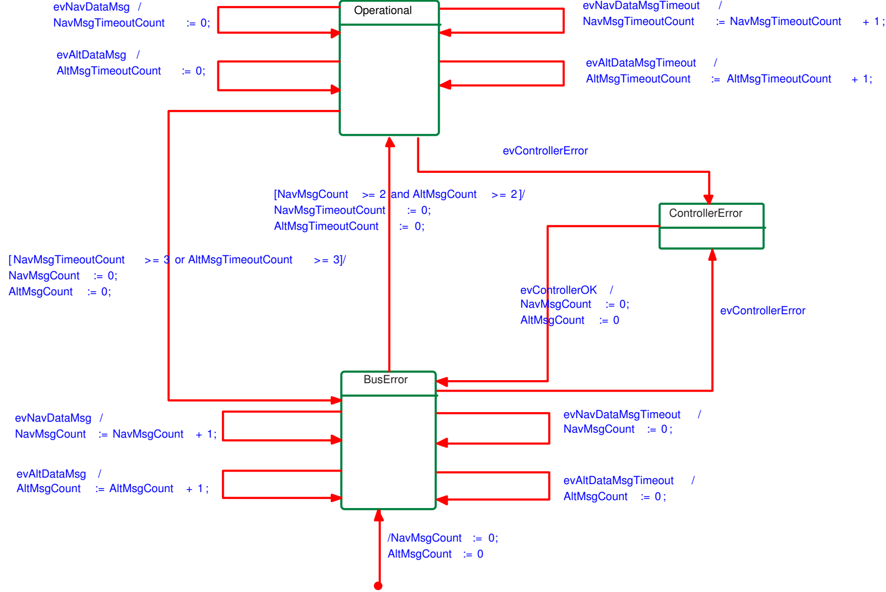

Figure 5: Non-Timed State Machine Diagram of the Message Receiver

3 Applying OMEGA techniques

We briefly describe the OMEGA techniques and mainly focus on the results of applying them to the MARS case study. We start with the LSCs in Sect. 3.1, present functional and timed model checking by UVE and IFx in Sections 3.2 and 3.3, respectively, and finally describe interactive theorem proving by means of PVS in Sect. 3.4.

3.1 LSC

The Play-Engine tool [20, 21] supports the specification and execution of scenario-based requirements. The underlying language for requirement specification is that of live sequence charts (LSCs) [8]. Live sequence charts are a powerful extension of the classical message sequence charts (MSCs) [48], that while retaining the intuitive spirit of MSCs, enhances their expressive power. LSCs distinguish between behaviors that may happen in the system (existential) from those that must happen (universal). Among other extensions they also allow specifying timing requirements [19] and generic properties using symbolic instances [31]. The Play-Engine assumes a discrete time model and adopts the synchrony hypothesis.

As mentioned above, one of the main extensions in LSCs – relative to classical MSCs – is the ability to distinguish between possible and mandatory behavior, using two types of charts. An existential chart describes a possible scenario in the system. Fig. 6 depicts an existential chart, as denoted by the dashed border, which represents a high level system behavior of the camera control. The vertical instances correspond to the participating objects — `AvionicsDatabus`, `RCU`, `ExposureModule`, `Camera` and the external object `ControlPanel`. Time progresses from top to bottom, thus the order of the events in this chart is `NavData`, `AltData`, `ComputeFramerate`, `ComputeFMC`, `FrameRate`, `FMC`, `StartFilming`, `ActivateExposures`, `triggerExposure` and `FMC`. In general, a scenario defines a partial order on the events appearing in the chart, where events on the same object line occur according to the visual order from top to bottom, and any given message can be received only after being sent. Since Fig. 6 is an

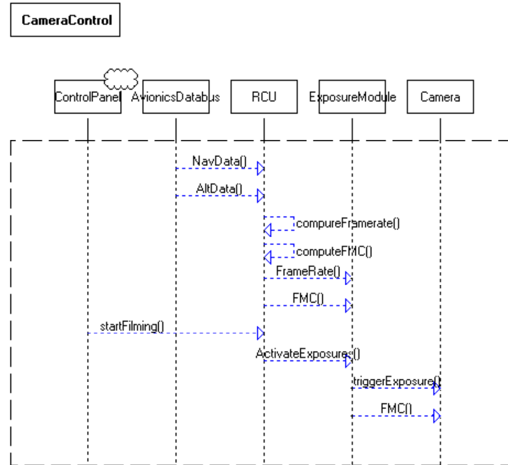

Figure 6: High level system behavior of camera control

existential chart, it specifies that there exists at least one run of the system which exhibits the sequence of events described above.

An example of a universal chart appears in Fig. 7. A universal chart contains a prechart (dashed hexagon), which specifies the scenario which, if successfully executed, forces the system to satisfy the actual chart body. In Fig. 7 the prechart starts with a time tick, followed by two assignments (first `ADS_Time` variable stores the global time, denoted by `Time`, and then `AltSend` variable stores the value of `ADS_Time` modulo the altitude data cycle `AD_Cycle`) and a condition which together check whether time has reached a multiple of the altitude data cycle `AD_Cycle`. If this is the case, the behavior specified in the main chart must follow. The main chart consists of an if-then-else construct; in this case the condition is a probabilistic choice, where with 95 % probability the altitude message is sent within 5 time units with no error and with 5 % probability there is an error with the altitude message. In a similar way the cyclic behavior for the navigation data is specified.

The expressive power of universal charts, based on the pattern "prechart implies main chart", forms the basis of an executable semantics for an LSC requirement model. As a response to an external event or time progress, a prechart may be satisfied, thus triggering the system events in the main chart to be executed according to the partial order, and these events may in turn activate additional universal charts. Play-out, the executable mechanism implemented in the Play-Engine tool, monitors progress along charts and performs system events in the main charts for universal charts that have been activated, trying to complete all universal charts successfully. As the events are being executed, their effects are visualized via a Graphical User Interface (GUI), thus providing an animation of the system behavior. The same GUI is used in an earlier stage called "play-in" to capture the LSCs by demonstrating the scenarios, while the Play-Engine records the behavior in the from of an LSC, thus providing an intuitive way for capturing the scenario-based requirements. The work described here addresses the execution and the analysis of the requirements rather then on the process of capturing them; the interested reader is referred to [20] for more information on play-in.

Play-out is actually an iterative process where after each step taken by the user, the Play-Engine computes a super-step, which is a sequence of events carried out by the system as its response to the event input by the user. One of the problems with play-out in its original form is related to the inherent non-determinism allowed by the LSC language. LSCs is a declarative, inter-object language, and as such it enables formulating high level behavior in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. This non-determinism, although very useful in early requirement stages, can cause undesired under-specification when one attempts to consider LSCs as the system's executable behavior. The play-out mechanism of [20] is rather naive when faced with non-determinism, and makes

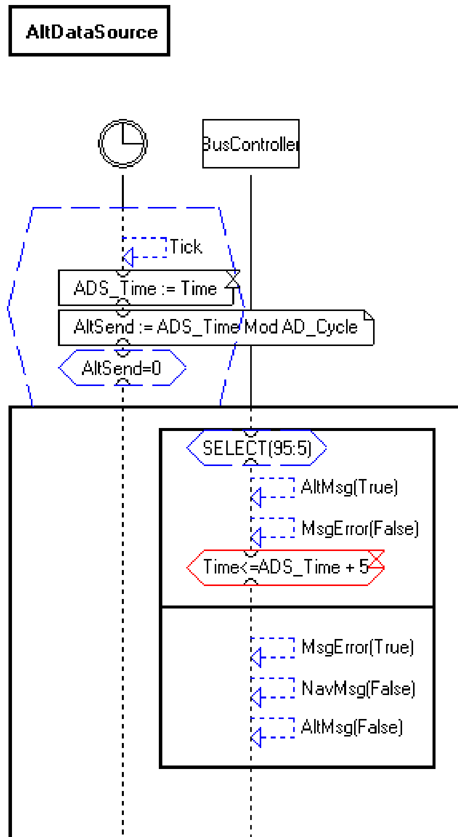

Figure 7: Universal chart concerning the altitude Data Source

essentially an arbitrary choice among the possible responses. This choice may later cause a violation of the requirements, whereas by making a different choice the requirements could have been satisfied.

To address this challenge, [15] introduces a more powerful technique for executing LSCs, called *smart play-out*. It takes a significant step towards removing the sources of non-determinism during execution, proceeding in a way that eliminates some of the dead-end executions that lead to violations. Smart play-out [15, 16] uses verification methods, mainly model-checking, to execute and analyze LSCs. There are various modes in which smart play-out can work. In one of the modes, smart play-out functions as an enhanced play-out mechanism, helping the execution to avoid deadlocks and violations. In this mode, smart play-out utilizes verification techniques to run programs, rather than to verify them. In another mode, smart play-out is given an existential chart and asked if it can be satisfied without violating any of the universal charts. If it manages to satisfy the existential chart, the satisfying run is played out, providing full information on the execution and reflecting the behavior via the GUI.

3.1.1 Results of LSC experiments

We have specified a high-level requirements model for the MARS application using universal LSCs. To explore the behavior of the system, the model has been simulated using the play-out capabilities of the Play-Engine. During play-out the active charts are displayed with a line which specifies how much progress has been made by each instance, as shown in Fig. 8.

Existential charts were used to test and verify system behavior; they do not drive the execution, but can be traced during play-out mode (depicted by a magnifying glass containing the letter “T” in the upper left corner) showing the progress along the scenario as shown in Fig. 9.

Another example of an existential chart is depicted in Fig. 10. In this scenario, after `AltMsg(True)`

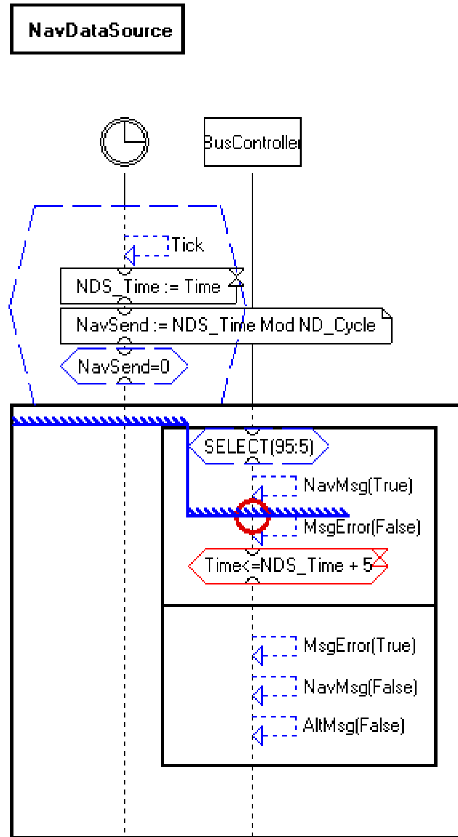

Figure 8: Example of play-out

occurs at the `BusController`, the `RCU` sends `receiveExpData` to the `ExposureModule`. This scenario also includes a timing constraint; time is stored immediately after `AltMsg(True)` occurs and the timed condition specifies that no more than 70 time units should have passed before the `receiveExpData` is received.

Being an existential chart, Fig. 10 implies that there is at least one run of the system satisfying this scenario. In early system design such charts serve as "sanity checks" showing that certain desirable behavior can be exhibited by the system. The user can then run play-out mode and attempt to drive the system behavior to satisfy the chart by providing the appropriate external events. A systematic way to check if a given existential chart can be satisfied by a set of universal charts is by invoking smart play-out with this query. If a satisfying run is found by smart play-out it will be displayed to the user, otherwise smart play-out proves that that it is impossible to satisfy the existential chart. The ability to prove that an existential chart cannot be satisfied is an advantage of smart play-out over the "naive" play-out mechanism. We have used this capability in our model by setting existential charts to designate a scenario that should never occur (anti-scenario); if this existential chart is traced to completion it indicates a problem in the design or in the specification of the universal charts. If smart play-out proves that this existential chart cannot be satisfied, we are guaranteed that this bad behavior is not allowed to occur.

An example of such an anti-scenario is shown in Fig. 11. We have modified the existential chart of Fig. 10 to capture the undesirable behavior in which the `AltMsg(True)` occurs at the `BusController`, and more than 70 time units pass without the `ExposureModule` receiving the `receiveExpData` from the `RCU`. The forbidden (cold) message `receiveExpData` at the bottom of Fig. 11 specifies that if this method occurs before 70 time units have passed, the chart is exited and thus not traced to completion (and no problem occurs). To formally express an anti scenario using the LSC language, the forbidden scenario can be placed in the prechart of a main chart which only contains the condition `FALSE`; this condition can

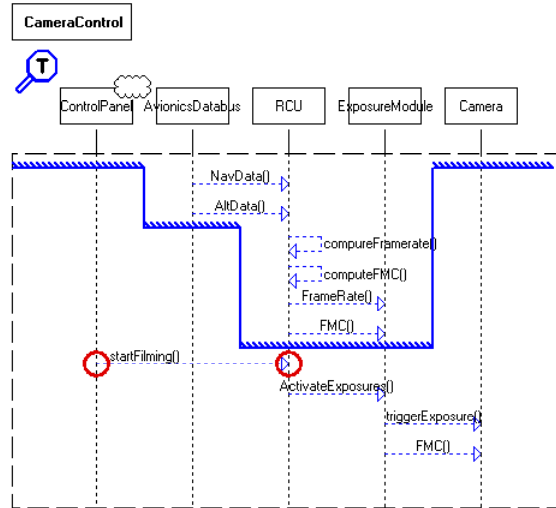

Figure 9: Tracing an existential chart

never be satisfied, thus implying that the scenario described in the prechart should never happen.

We have used play-out as a simulation tool, to explore the design, try out different parameter values and gain a better understanding of the application. To formally verify properties we used smart play-out. The model consists of around 10 scenarios and the running time of smart play-out was around 30 minutes per query, with some of the queries being answered in less than one minute. This running time includes the translation to the format of the model checker, and the feedback to the Play-Engine in case a satisfying run was found. Applying smart play-out to larger models remains a major challenge due to the performance of the underlying model checkers. The initial experience with smart play-out allowed the verification of some functional properties, but also pointed out several problems in the tool, that will be discussed in Sections 4.4 and 5.1.

3.2 UVE

The model-checking tool UVE [42] supports bounded and exhaustive verification of open systems with a non-deterministic environment. A discrete time semantics [9] has been implemented, where only the order between the observable entities is considered. A discrete time step corresponds to the execution of a transition in a state machine or the removal of an event from an event queue. Each action is considered instantaneous. Execution duration is counted in terms of the number of steps and this is used to bound the depth of the model exploration and to specify points of time during execution where conditions should be evaluated.

To formalize requirement specifications one can use the built-in temporal logic patterns, based on CTL (Computation Tree Logic) formulas, or the LSC formalism of the tool. Both notations use a discrete-time quantification in terms of steps which makes it possible to express bounded properties such as "if P then eventually Q within X steps". The UVE tool can be used to check whether a certain UML model satisfies its specified functionality. Functional requirements that can be checked with UVE can be distinguished into universal properties, which should hold for all system runs, and existential properties, which require at least one run as a witness of the property.

UVE supports a subset of the LSCs from Sect. 3.1. This subset does not contain real-time constraints/conditions, but instead has additional possibilities to put a bound on the number of steps for which a property should hold (and thus on the depth of model exploration). UVE also distinguishes between assumption and commitment LSCs, which are just different roles of the properties specifications within the verification process. This is important to restrict the non-deterministic environment of open systems.

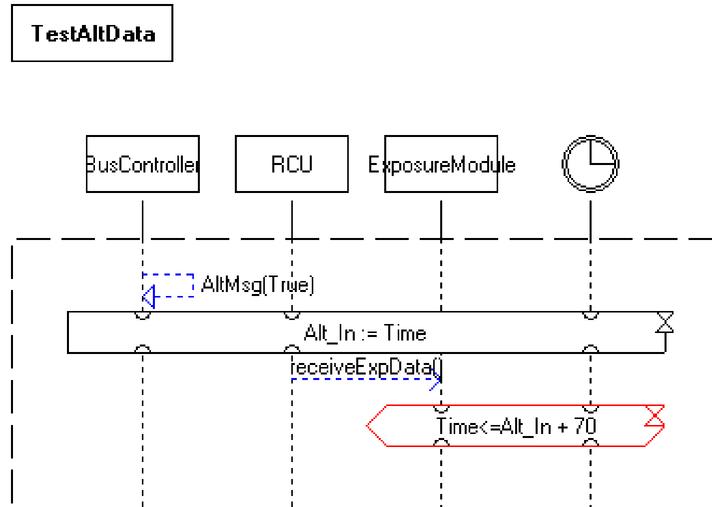

Figure 10: LSC representation of a timing property concerning altitude data

UVE is based on backward state space exploration over bounded values of natural numbers. The problem of scalability, typical for model-checking approaches, depends not only on the size of the model, but to a large extent also on the complexity of the property verified, the concurrency level in the model, and the amount of non-determinism of the environment.

An execution trace is represented in the form of an existential LSC, showing the order of communication events between objects, and a so-called Symbolic Timing Diagram (STD), which shows for observable execution steps the changes of variables, states, event queues, etc. These STDs are similar to the timing diagrams that have been added to UML 2.0. In UVE, STDs are only used to visualize verification results, making it easier to search for the source of an error. Whereas LSCs visualize the interaction between objects, STDs are convenient to show the internal computations of objects which is often the focus of UVE-based verifications.

3.2.1 UVE experiments on MARS

UVE has been used to verify a number of properties of a non-timed version of the MARS system, using the state diagram of the Message Receiver from Fig. 5. The properties mentioned in Sect. 2.2 have been expressed as bounded response properties where timing has been replaced by steps. The correct number of steps for which a property holds has been obtained by a series of verification experiments, involving counter-example analysis and fine-tuning the specified step counts by successive approximations. On the average between 5 and 10 experiments were needed per specified property depending on the complexity of the generated counter example.

When a universal property does not hold or an existential property holds, the UVE tool generates an error path or a witness path, respectively. The result of a verification consists of a summarizing text and a generated path (when available), represented as an STD or an LSC.

As a simple example, consider the property: "Each time when the Databus Controller encounters an error, the Message Receiver will detect this within 1 step." The property has been checked with the following assumptions: (a) if the Databus Controller encounters an error, it remains in its error state; (b) the Controller Monitor polls the current system state periodically at least once every 5 steps. Fig. 12 shows the textual representation of a verification result which detects that the property does not hold.

The STD view of the error path is depicted in Fig. 13. It shows the values of all elements in the model and from the property specification (objects, event queue, attributes, states of state machines, and parts of the specification, such as assumptions). In particular, at the third step of the system execution, the premise of the property became true (the Databus Controller is in the Error state), but the conclusion of the property

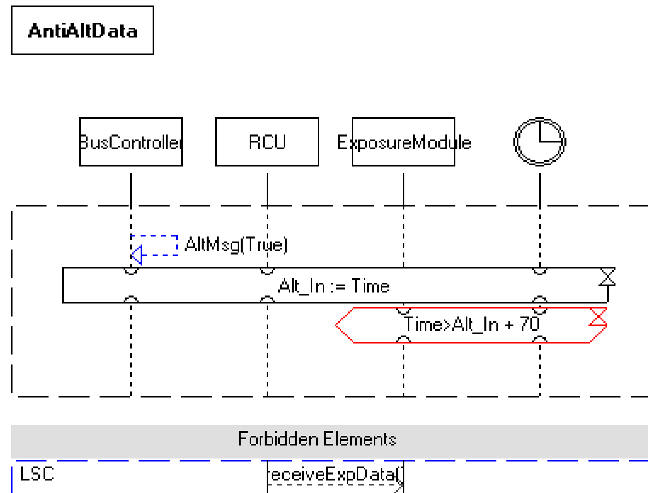

Figure 11: Using an existential chart as an anti-scenario

does not hold at the next step (the Message Receiver is still in state Operational). The LSC of Fig. 14 shows the message exchange between the objects leading to this erroneous situation.

Next the property has been corrected by increasing the number of steps necessary for the Message Receiver to detect an error from the Databus Controller. This leads to a positive verification result.

For the relatively small models of the MARS example, the internal model checker of UVE (called VIS) takes around 20 minutes of execution time. In addition, first the model is translated to the internal format of VIS and reduced with respect to the property to be verified. After running VIS, the error or witness trace is translated from the internal representation into an LSC and a STD visualization.

3.3 IFx

The IFx tool provides simulation and model checking capabilities for the OMEGA kernel language and the OMEGA real-time profile for UML. It is built on top of the IF toolbox [5, 6], and consists mainly of a translator for UML models to the IF language [33], a simulation front-end and extensions to the IF toolbox such as the ability to specify properties using observer automata.

The tool handles closed models, which contain a description of both the designed system and its environment. An explicit specification of complex environments is made possible by using timed or functional non-determinism to model different types of inputs coming from the environment at arbitrary or otherwise constrained time moments (e.g., within some periodic interval, or with some throughput constraints, etc.).

Besides rich interactive simulation capabilities, IFx allows to model-check a system against timed safety properties. They are specified in the form of UML observers, which are accepting automata (extended with variables and clocks) reacting to events, to conditions occurring in the system, as well as to time [33]. The tool supports both discrete and continuous time. In discrete time, time progress is represented by a *tick* transition common to all processes. Continuous time is represented symbolically, similar to the timed-automata based tools Kronos [47] and Uppaal [28]. Due to the different representations, in discrete time IFx allows more expressive time properties than in the continuous case. The symbolic representation leads in most examples to much smaller state spaces. The experiments on MARS were performed in symbolic continuous time.

The verification procedure is based on forward exploration of the state space, using various optimizations and abstractions to reduce combinatorial explosion. We mention here static optimizations (like dead variable factorization and dead code elimination) which reduce the state space of a model while fully pre-

```
vis> elapse: 12.1 seconds, total: 14.2 seconds
vis> +++ +++ +++

***
*** The Property 'inv_P_implies_finally_Q_B__immediate' with
***
***  P = root->p_DatabusManager->itsDatabusController->IS_IN(Error)
***  Q = root->p_DatabusManager->itsMessageReceiver->IS_IN(ControllerError)
***  max_X_Val = 1
***
*** under the assumptions 'first_P_implies_globally_Q__immediate' with
***
***  P = root->p_DatabusManager->itsDatabusController->IS_IN(Error)
***  Q = root->p_DatabusManager->itsDatabusController->IS_IN(Error)
***
*** and assumption 'inv_finally_P_B__immediate' with
***
***  P = ES_evPollController(ENV, root->p_DatabusManager->itsControllerMonitor)
***  max_X_Val = 5
***
*** with 'ndet'-mode external event list
***
***  root->p_DatabusManager->itsDatabusController, evControllerBIT_OK
***  root->p_DatabusManager->itsDatabusController, evControllerBIT_ERROR
***  root->p_DatabusManager->itsControllerMonitor, evPollController
***  root->p_DatabusManager->itsNavigationDataSource, evSendMsg(0)
***  root->p_DatabusManager->itsNavigationDataSource, evSendMsg(1)
***  root->p_DatabusManager->itsAltitudeDataSource, evSendMsg(0)
***  root->p_DatabusManager->itsAltitudeDataSource, evSendMsg(1)
***
*** does not hold.
***
*** A counterexample trace is generated.  Please stand by...
***
Build | Check Model | Configuration Management | Animation
```

Figure 12: Text message of UVE about error found

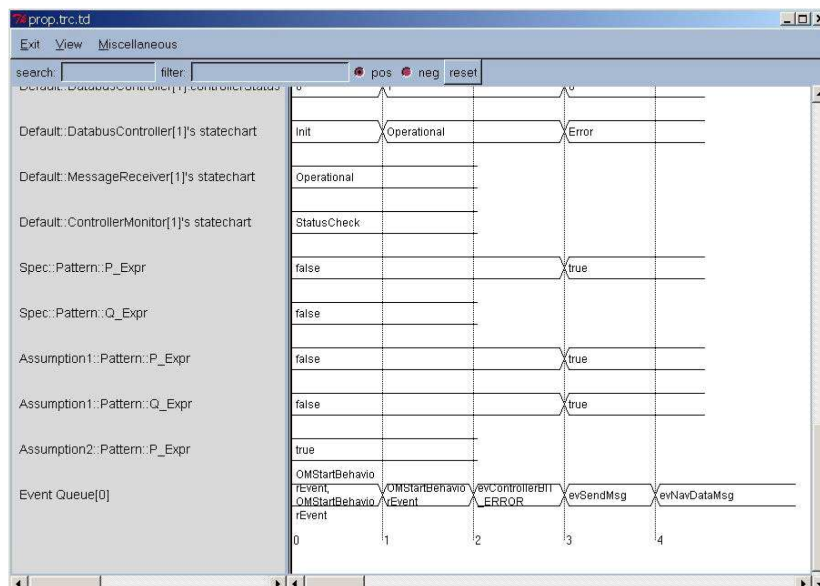

Figure 13: STD generated by UVE after detecting an error

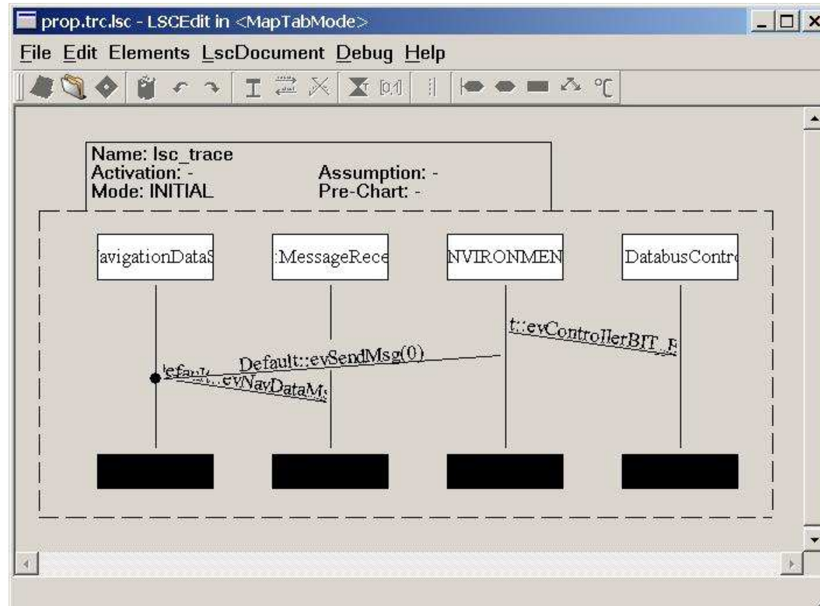

Figure 14: LSC generated by UVE after an error has been found

serving its behavior, or dynamic optimizations like partial-order reduction. The tool supports different kinds of data abstractions (slicing, queue abstractions, etc.).

Negative verification results are provided with a diagnostic trace which can be re-played and debugged in the simulator.

3.3.1 Timed modeling

The validation experiments were performed on the original timed version of the data acquisition subsystem presented in Sect. 2.1.

On the environment side, as mentioned before, an important need is the ability to naturally model non-deterministic behavior. This model should include, for example, the jitter of the data sender. This has been done in IFx using a clock and interval constraints, as shown in Fig. 2. After a few experiments with the model, we realized that a non-deterministic start-up time for the data sources had to be modeled (see state Init in Fig. 2), as otherwise the periods of the two sources are implicitly synchronized and this does not capture the real behavior of the MARS environment.

3.3.2 Results of IFx experiments

The functional and reactivity properties of the controller have been formalized using observers and verified against the timed model. Fig. 15 shows the observer checking the upper bound $BR1$ on the response time $R1$ between the last message correctly sent by a source, and the detection of the bus error if the following 3 messages are lost (as specified in Sect. 2.2 and illustrated in Fig. 4). We note that this observer monitors only the message loss from a single source (the altitude data source in this case). Due to symmetry, this can be done without loss of generality.

The first verification experiments led to state space explosion beyond the limits of the tool. The source of this explosion was established to be the existence of too many parallel, fully de-synchronized, timed behaviors. Even though the non-timed product of these components is of manageable size, the explosion is produced by the large number of possible configurations for the relations between time periods of these components (therefore producing a large number of symbolic time zones). Note that verification is very easy when we use only one source instead of two; this requires only 1084 states, 1420 transitions and less than one second of user time.

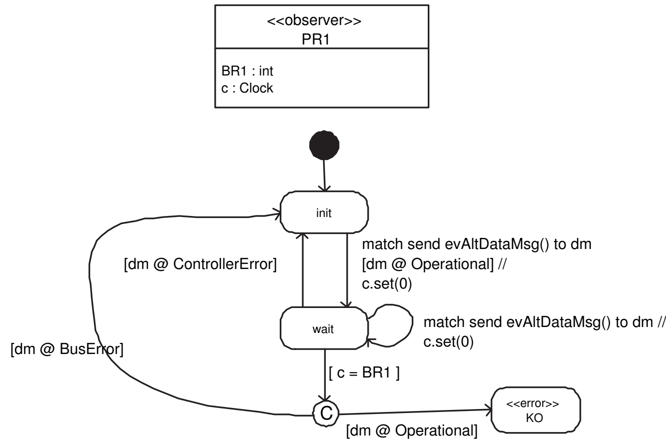

Figure 15: Timed safety property specified by means of an observer

To reduce the explosion we made the simplifying assumption that the two data sources are synchronized (i.e., their period of 25 ms always starts at the same time; the two types of messages can nevertheless be sent at different moments due to jitter). It is clear that this assumption is not conservative with respect to the verified properties as the reaction time due to message loss may (and is likely to) be longer when the two data sources are de-synchronized. But the specification could be verified, using 99355 states, 151926 transitions and 36 seconds of execution time. With the synchronization assumption on the environment, the initial MessageReceiver model presented in Fig. 3 satisfies the reactivity property above for $BR1 = 85ms$ $(= 3 \times P + 2 \times J)$. In order to fully verify the properties without this non-conservative assumption, we have used the redesigned model presented in Sect. 4.

In an attempt to simplify the model and to gain performance, a slightly different variant of the MessageReceiver was developed. This variant was believed to be an equivalent refactoring of the initial model. However, the IFx tool was able to detect that the variant had significantly lower reactivity to failures of the data links. The variant consisted in modifying the state machine of the Message Receiver (see Fig. 3), so that in state Operational, instead of counting the reception failures (one by one up to 3), a long timer of length 85ms was used to test, for each source, the absence of messages. The object switched to state BusError upon reception of any of these two timeouts, and the two timers were re-initialized when going back to state Operational.

The verification experiments showed that the above property was satisfied only for a duration $BR1 = 95ms$ (still under the simplifying assumption of synchronized sending windows described before). The cause of this decreased reactivity of the second variant of the Message Receiver could be analyzed on the error traces generated by the model checker. It comes from the moment when the timers are initialized when going from BusError to Operational: in the first model timers are kept running as they are in state BusError, while in the second one both timers are re-initialized when going from BusError to Operational because the timeout duration has to change from 35 to 85ms.

Although in the more realistic case of completely de-synchronized sources this property could not be checked (see also Sect. 4.3), a manual analysis shows that $BR1$ would have to be greater than $110ms$ $(= 4 \times P + 2 \times J)$ in this case.

3.4 PVS

To allow general verification of UML models with infinite state spaces, we have experimented with interactive theorem proving. We have used the PVS system, a general purpose theorem prover which is freely available [36, 37, 40]. PVS has a powerful specification language, based on higher-order typed logic. Specifications can be organized as hierarchies of parameterized theories, which may contain declarations, definitions, axioms, theorems, etc. Moreover, there is a large set of predefined theories, with e.g., real numbers, several data types, and many relations and functions.

The user-defined theories may contain theorems and the user can try to prove these theorems by means of the proof engine of PVS. To prove a particular goal, the user invokes proof commands - including powerful decision procedures and rewrite strategies - which should simplify the goal until it can be proved automatically by PVS.

To be able to use the PVS system for the verification of UML models, we have represented the semantics of the OMEGA kernel model in PVS [23]. The general idea is that an execution of the UML model is represented by a *run* (i.e., an execution trace) which is a sequence of the form $c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow \cdots$ where the c_i are *configurations*, representing a snapshot of the system during execution and each pair of successive configurations represents a step of a state machine of one of the objects or a time step. Time steps model the global progress of time, similar to timed automata (see, e.g., [28]), using the real numbers of PVS to obtain a continuous notion of time.

The semantics has been defined in a few PVS theories which are parameterized by the syntax of a UML model, including class names, transitions of state machines, methods, etc. These theories form the core of the techniques used to verify UML models by means of PVS. The user can model a system in the OMEGA kernel language using a commercial UML-based CASE tool. The XMI output of the tool, representing the model in XML, is first preprocessed to remove a large amount of XMI information and to get a concise representation. Next it is transformed into a representation of the syntax of the model in PVS [27].

Properties of the system under development might be specified in OCL and can then also be transformed into the PVS specification language. In the experiments described here, however, we have specified the properties directly in PVS because the industrial users were not acquainted with OCL and they did not have tool support for it.

To verify that the UML model satisfies a certain property, a PVS theory is created which imports the PVS representation of the UML semantics, parameterized by the syntax of the concrete model. It also defines the properties, expressed as predicates over runs of the model. Next the user can use the proof environment of PVS to try to prove the properties in an interactive way.

The focus of the PVS experiments with the MARS system was on the verification of general properties that are beyond the scope of model checkers. This concerns properties that express relations between parameters for system characteristics such as period, jitter, time-out values, the number of allowed retries, and the number of messages needed for recovery. Other work addresses systems where the number of objects is unbounded [1].

In the next two subsections we first present our verification experiments with the non-timed model of MARS, and next describe the attempts to verify the timed model.

3.4.1 Non-Timed model

As a starting point, to get some experience with the interactive verification of UML properties, we verified the non-timed model of Fig. 5 in Sect. 2.3. We proved a simple safety property, namely, that if the data sources never fail to send messages, then the message receiver never reaches the bus error location. To give an impression, we show how this property can be expressed in PVS as a predicate `SafetyPropert1`, using the abbreviations `NoSourceErrors` and `NoBusErrorsDetected`, where `r` is a run, `i` is an index in the run (hence `r(i)` is a configuration), and `event` and `obj` are fields of a configuration.

```
NoSourceErrors(r) : bool = FORALL i:
   r(i)`event /= evNavDataMsgTimeout  AND
   r(i)`event /=  evAltDataMsgTimeout

NoBusErrorsDetected(r) : bool = FORALL i:
   (r(i)`obj)(MessageReceiver)`state /= BusError

SafetyPropert1(r) : bool =
   NoSourceErrors(r) => NoBusErrorsDetected(r)
```

We also showed that the message receiver can only move from the bus error location to state operational if it receives two successive messages from each source.

The proofs proceed by global induction on the runs, proving a property for the initial configuration and showing that it is maintained by every step. As usual in deductive verification, a number of auxiliary invariants had to be invented and proved. Moreover, properties had to be strengthened to be able to prove them by induction on the runs. Altogether this required a lot of user interaction. For instance, a straightforward proof of property `SafetyPropert1`, without any attempt to shorten proofs or to exploit clever strategies, uses about 50 small lemmas. Most of them required between 10 and 20 user interactions (i.e., proof commands) to complete the proof. Rerunning the proofs, which is useful to check the correctness after small changes, requires only a few seconds.

To improve this, we also experimented with the use of the TLPVS package [1, 39] which has been developed to reduce user interaction for the verification of properties expressed in Linear Temporal Logic (LTL). TLPVS provides a set of theories, defining LTL and corresponding proof principles, together with powerful strategies that exploit the proof principles with minimal user interaction. Note that the correctness of the proof principles has been proved in PVS.

The use of the TLPVS package indeed reduced the amount of user interaction significantly; the strategies reduce a proof goal automatically into a number of small cases, which often can be discharged by a fixed sequence of commands. For property `SafetyPropert1`, the proof could be reduced to 3 lemmas which together required around 120 interactions (including calls to rather complex strategies). Note, however, that the insight obtained by the earlier proof has been used; for instance, all required auxiliary properties have been combined into a single lemma.

3.4.2 Adding time

Our main interest was a general verification of the timed case, using parameters for the timing values. We use *Tout* to represent the time-out value for message loss, and parameters *PN* and *PA* for the maximal processing time of navigation and altitude messages, respectively. Moreover, we use parameters N and K for the number of allowed data losses and the number of consecutive messages needed for recovery, respectively, to generalize the two properties mentioned in Sect. 2.2:

1. The receiver shall move to the bus error location if and only if one of data sources misses N consecutive messages

2. The receiver shall recover from a bus error if and only if both data sources send K consecutive messages

In the original MARS system, we have $N = 3$ and $K = 2$. We started the verification experiments with a simple case where $N = 1$, $K = 1$, and using a long time-out *Tout* (as shown in Sect. 3.3.2 this is less optimal than a sequence of small time-outs). We managed to prove a safety property, but this required about 50 auxiliary lemmas in PVS and a lot of user interaction and ingenuity. Positive was the identification of required relations between the parameters, such as $4 \times J < P$, $max(PN, PA) < P - 4 \times J$, and $N \times P + 2 \times J < Tout < (N+1) \times P - 2 \times J$. Observe that this implies, for instance, that jitter J should be relatively small compared to period P and it also gives an upper bound on the processing times *PN* and *PA*.

4 OMEGA techniques applied to a redesign of MARS

Evaluating the verification of the MARS system, it became clear that for complex examples compositionality and abstraction are essential to improve scalability. To be able to experiment with these techniques in the MARS system, we re-model this system. In addition to facilitating formal techniques, the new model is more modular and flexible, since it can be easily instantiated for an arbitrary number of data sources. Instead of using internal states to represent errors, we make them externally visible by sending *error* and *operational* signals. To show the essential part, we omit the controller error and focus on detecting and recovering from bus errors.

The basic idea is that we replace the original message receiver, which deals with two data sources, by the composition of three objects: two instances of a simpler message receiver *MR* which deals with a single data source, and an error logic object *EL*. Each *MR* object receives data from a particular source and sends

an *err* signal to *EL* if a number of consecutive data items is missing. When *MR* observes a recovery of the databus by receiving a number of successive data items, it sends an *ok* signal to *EL*.

The error logic object *EL* collects the *ok* and *err* signals; when the system is operating correctly then a single *err* signal leads to an *error* signal. After sending *error*, object *EL* sends signal *operational* if it is informed that both *MR* objects are correct, that is, if for both objects it has received an *ok* signal after every *err* signal.

Finally, we proposed a refinement of each *MR* object into two objects:

- A receiver *R* which receives data items and contains time-outs to detect the absence of data.
- A monitor *M* which gets information from receiver *R* about the absence or presence of data, counts these events, and generates the *err* and *ok* signals when needed.

Note that *M* is non-timed, containing only computations, whereas component *R* just detects the presence or absence of messages during specified time periods.

In the next subsections we apply the OMEGA techniques to this redesign of MARS. In Sect. 4.1 we prove the correctness of the high-level decomposition of the design using PVS. Sections 4.2 and 4.3 contain the application of functional and timed model checking, respectively. The use of LSCs is shown in Sect. 4.4.

4.1 PVS

For the redesign of MARS, the focus of interactive verification by means of PVS is on high-level verification on the level of components. The aim is to show the correctness of the design at this level, reasoning with the specifications of components without knowing their implementation. Hence we consider in this section only two constructs: parallel composition and hiding of internal events. To obtain a compositional framework for reasoning about components, we have changed the PVS framework in a number of aspects:

- The semantics has been reformulated to obtain a denotational (i.e. compositional) semantics for parallel composition and hiding. The semantics has been formulated in terms of traces, which are an abstraction of the runs in the basic OMEGA semantics. Since we are dealing with high-level decompositions, we abstract from all internal details such as internal objects and the values of their attributes. We only record the current time of the configurations and the external events which cause configuration changes.
- To express properties of (part of) a system, we use logical formulae that define a set of traces. To be able to formalize intermediate stages during the top-down design of a system, we have defined a framework where specifications and programming constructs can be mixed freely. Refinement corresponds to trace inclusion. This is inspired by similar work on non-timed systems and related to work on timed systems [22].
- Compositional proof rules for parallel composition and hiding have been formulated and proven to be sound in PVS. The rule for parallel composition expresses that the composition of two components satisfies the conjunction of the specifications of these components, provided their specifications only depend on the interface of the corresponding component.

This compositional framework has been applied to the high-level decomposition of MARS. First the required properties have been reformulated in terms of traces. As an example, we define below when a trace `tr` satisfies a property called `Prop1`. Here `d1` and `d2` are concrete data items and `d` a variable ranging over data items. `i` and `j` are variables over the natural numbers, used as indices in sequences. `T(tr)` is the sequence of time stamps in trace `tr` and `(e@@i)(tr)` denotes that event `e` occurs at position `i` in trace `tr` [2]. Constants `P` and `J`, represent period and jitter, respectively, and `N` is a parameter for the number of missing data items which should lead to an error.

First a long time-out is defined as a period of a certain time length in which no data item occurs (expressed by abbreviation `Never` - not shown here). Then `Prop1` expresses that if there is a time-out on one of the data sources, and no error has been generated yet since the end of this time-out period (using abbreviation `Error`), then we generate an `err` event within `DeltaErr`.

[2] To avoid clashes with the existing PVS syntax we use a double @ symbol.

```
LongTimeOut(d,i,j)(tr) : bool =
  Never(d,i,j)(tr) AND
  T(tr)(j) - T(tr)(i) >= N*P + 2*J

AfterWithin(err,j,delta)(tr) : bool =
  EXISTS i : i >= j AND (err@@i)(tr) AND
             T(tr)(i) - T(tr)(j) <= delta

Prop1(tr) : bool = FORALL i, j: i < j AND
     (LongTimeOut(d1,i,j)(tr) OR
         LongTimeOut(d2,i,j)(tr)) AND
     NOT Error(d1,j)(tr) AND NOT Error(d2,j)(tr)
       IMPLIES AfterWithin(err,j,DeltaErr)(tr)
```

We also specify that an error signal should only be sent if a time-out occurred. Similarly, the occurrence of an *ok* signal has been specified. This leads to an overall specification $TwoDataSources(d_1, d_2, err, ok)$ of a message receiver for two data sources d_1 and d_2. In a similar way, components are specified by properties of their traces: $MessageReceiver(d, err, ok)$ specifies a message receiver for a single source, and $ErrorLogic(err1, err2, ok1, ok2, err, ok)$ the error logic components.

Since it is rather easy to make errors in the declarative specifications, we increase the confidence by deriving some properties that are expected to hold. For instance, for the message receiver we consider various data sources $DataSource(d)$ (e.g., an always correct source, a source which fails to send any data after some point in time, etc.). Next we show that $DataSource(d) || MessageReceiver(d, err, ok)$ satisfies the expected properties, using the conjunction of the specifications.

To show the correctness of the redesign, we prove in PVS that the parallel composition $MessageReceiver(d_1, err_1, ok_1)$ $||$ $MessageReceiver(d_2, err_2, ok_2)$ $||$ $ErrorLogic(err1, err2, ok1, ok2, err, ok)$ satisfies the global top-level specification $TwoDataSources(d_1, d_2, err, ok)$, provided certain conditions on timing parameters hold. (We omit the hiding of internal events for simplicity.) This has been proved in PVS using the compositional rule for parallel composition mentioned above, which means that we had to show that the conjunction of the specifications of the components implies the desired top-level properties. The proof is highly non-trivial because suitable intermediate lemmas had to be found and a few hundred user interactions were needed. But having proved the correctness of the decomposition, next the components *MessageReceiver* and *ErrorLogic* can be implemented and checked in isolation, e.g., using model checking.

4.2 UVE

The aim of the UVE experiments with the redesigned MARS system was to evaluate modular verification, i.e., formal verification of individual components (at different levels of abstraction) within the whole system. In the UVE context, a component is an object of the root class together with all its children related to it via composition and aggregation associations. The behavior of a component must be fully specified with state machines and methods.

In the redesigned MARS system, the components *MR* (message receiver) and *EL* (error logic) have been considered separately, as well as the smaller subcomponent *M* and a larger subsystem containing a single data source, one *MR* object, and an *EL* object. The components were verified relative to assumptions about input from the environment. For instance, for the subsystem $R||M$, the non-timed component *M* has been verified with respect to a specification (with an over-approximation) of the timed part *R*.

Different kinds of property specifications were verified for the chosen system parts (propositional and temporal formulas, LSCs), making assumptions on the inputs (order of events, attribute values), and observing the responses of the component under verification.

Due to restrictions in the tool (e.g., there is no possibility to define or automatically detect component interfaces for outgoing events, and the targets of all events must be specified explicitly in the model), some further remodeling has been performed to enable compositional reasoning. This has been done using standard elements of UML 2.0 like ports and connections, but expressed in UML 1.4 since UVE was primary based on this version of UML. The whole remodeling in UVE took a few hours of an experienced

designer. It was more difficult (a few days of experimentation) to find suitable properties for the parts and the proper assumptions about the inputs to the ports.

The general observation concerning the redesign is that the decomposed model became larger due to additional communication mechanisms (handling communication abstraction, dealing with explicit targets of the output events). This increases the verification time of the entire system, but makes it possible to verify parts quickly and in a more general context. Analysis results for a larger model can then be obtained out of the properties of its parts and the connection structure between these parts. Without such decomposition, verification often took too long and sometimes had to be aborted, especially for models with a lot of concurrency, i.e. with multiple active objects. Observe, however, that the compositional approach requires more specialized modeling effort from designers.

4.3 IFx

The introduction of the compositional model has allowed to push forward the analysis with IFx. As mentioned before, the original model could not be verified in case of completely de-synchronized data sources.

The compositional model allows the construction of a simple abstraction for one of the Data Sources: its corresponding *MR* sends out non-deterministically either *ok* or *err*, at unspecified moments. The specification of properties has to be adapted accordingly. For example, the property "if there is no message from one of the Data Sources for more than T time units, then the MessageReceiver is in state BusError at the end of this interval" (introduced in Sect. 3.3) becomes "if there is no message from the *concrete* Data Source for more than T time units, then the MessageReceiver is in state BusError at the end of this interval". Using the symmetry of the two Data Sources, this property implies the original one.

The use of this abstraction brought the state space to a manageable size and allowed the verification of all properties, while effectively over-approximating the general case of de-synchronized Data Sources. The verification took 155166 states, 263368 transitions and 1 minute and 21 seconds of execution time.

The experiments pointed out a problem with the proposed redesign: the fact that an *MR* object does not send the *err* signal is not sufficient for recovery. For instance, if the *MR* misses a single data item, then this does not lead to *err*, but a global recovery does not allow any miss; it requires that all data must have been received. There are several solutions to this: either weaken the initial specification, or adapt the design by adding for example a *miss* signal, or use parameterized signals to represent the presence of the last N messages (instead of the simple *ok* and *err* messages).

4.4 LSC

The Play-Engine has been applied to the redesign of the MARS system in three phases: model construction, verification, and synthesis. The first phase involved mostly constructing an LSC model using play-in and simulation using play-out. We have modified our original LSC model and then used play-out to simulate the behavior for various types of data sources, including models where both sources never fail, where only one of the sources may fail, and where a source that fails never recovers and does not send any data. We have also changed the probability for failure in the original model and used play-out to see the effect on the behavior. In the second phase, we have applied smart play-out to formally verify properties of the LSC model. After extending smart play-out to support time and forbidden elements [16], we managed to verify timed properties of the model.

The problem of synthesis, i.e. constructing a program from a specification, is a long-known general and fundamental problem. In our case, the specification is represented by the LSC model and the aim is to synthesize a state machine. As a first step, we have applied synthesis algorithms from LSCs to statecharts, as described in detail in [14, 17]. We observed that the remodeled MARS application is more suitable for this synthesis than the original model, mainly because the requirements are more structured and there is a clear separation between the timed and non-timed aspects of the message receiver. A state machine synthesized by our tool for the non-timed part of the message receiver appears in Fig. 16. Transitions that have no trigger are called null transitions and are taken spontaneously as part of the run-to-completion step, once the source state is entered.

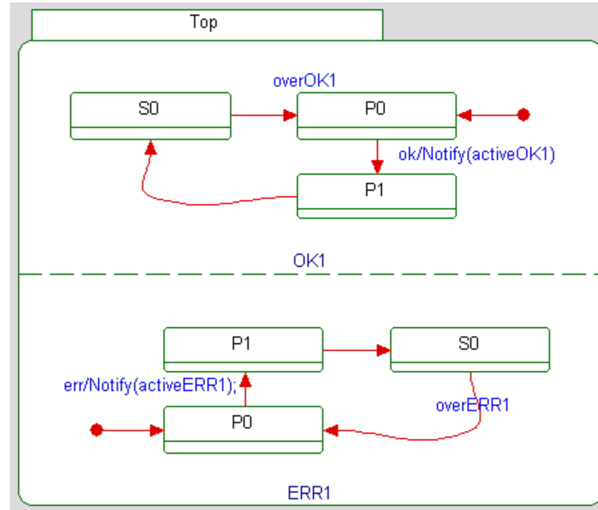

Figure 16: Synthesized state machine for the non-timed part of the message receiver

5 Evaluation

In the next subsections we evaluate the experiments on the MARS case study for each of the OMEGA techniques.

5.1 LSC

Although building an application-specific GUI is often convenient for capturing external user requirements and play-out, we have not used it for the MARS application because the data manager component that we consider only interacts with other objects of the system. Hence, we have used the internal object diagram supported by the Play-Engine, which is a variant of an object model diagram. Play-in is performed by pointing and clicking on the various objects, attributes and methods. This turned out to be a very convenient solution for our model, since it allows a quick specification and execution of LSCs without spending much time on the construction of a GUI. The positive experience of working with internal objects suggests that it will be very beneficial to improve the tool support in this respect, allowing the use of multiple diagrams and the application of layout algorithms. We still think that building a GUI is very worthwhile, especially for larger scale projects, because it makes it easier for additional stakeholders to participate in the requirements elucidation phase. To this goal, our experience showed that it would be helpful to improve the tool support for constructing GUI's for Play-Engine models, and indeed future work is planned in this direction.

When using smart play-out for verifying properties of the model we have faced several difficulties. Initially the subset of the LSC language supported by smart play-out did not include several of the important features used in the MARS model, such as time and forbidden elements. Smart play-out has then been extended to support these features as described in [16]. Smart play-out currently supports a discrete time model, where global time can be stored in time variables which can be used in conditions.

Another issue involves the type of queries supported by smart play-out. Given an existential chart and a set of universal charts (an execution configuration) smart play-out can be asked to try to satisfy the existential chart without violating any of the universal charts. Before our work on the MARS system, smart play-out allowed system events appearing in the existential chart to be taken in a spontaneous manner, even if the event does not appear in the main chart of an activated universal chart. As a result of our experience, we now support an additional mode that does not allow spontaneous system events to occur. Then an existential chart can be satisfied only if the system events in the existential chart indeed appear in the main chart of an activated universal chart. This mode is more useful for verifying if a design can exhibit a certain behavior, while the mode in which system events can occur spontaneously is useful during initial stages of building the requirement model, to check if a certain behavior is not contradicted by the existing universal

charts or to make sure that a certain ‘bad’ behavior is explicitly ruled out.

Our experience with the application of synthesis to the MARS model pointed out that additional work is required both in terms of improving the efficiency of the algorithms, extending the LSC subset supported by the synthesis tool, and improving the stability and usability of the tool. Another issue is the relationship between manually designed state machines and automatically synthesized ones. Although the synthesized model is guaranteed to be correct with respect to the given requirements, a manual design may have other advantages in terms of efficiency and the ability of the designer to understand and modify the model if necessary. Having said that, tool support for synthesizing even a relatively small model from LSCs into state machines and run it in a powerful UML tool like Rhapsody, opens up exciting development opportunities which we plan to continue exploring.

Additional information on using LSCs and the Play-Engine for other case studies in OMEGA appears in [18].

5.2 UVE

The UVE tool is intended to be used during the development of individual components, allowing the verification of executable models at different levels of abstraction. It has been found useful for estimating the relative discrete timing of the different entities in the model and for gaining insight into the step interleaving of model components as part of the total model behavior. The major problem of the application of UVE to a UML model, as observed during the experiments on MARS, is the significant decrease in performance while verifying more complex properties.

The complexity of the model-checking task depends on many parameters: the size of the model (the number of objects and links between them), the levels of non-determinism (e.g., due to multiple active objects, concurrency in state machines, and a non-deterministic environment), the complicated arithmetic (including loops, dynamic branching, etc.), the complexity of the temporal property under verification (e.g., long execution paths specified by different events and conditions, or many assumptions on the environment).

The experiments showed that it is difficult for the user to assess the verification complexity and to set the relevant parameters to reduce this complexity. For instance, it is difficult to find reasonable assumptions (both on the environment and on the system parts outside the verification scope) to reduce external non-determinism. The same holds for finding the correct step boundaries in properties.

To be able to verify components with UVE, a well-structured model is needed, e.g., communication between components must be asynchronous by means of signals. We strongly recommend to separate concurrent threads of control into different components, because verification of a component with multiple threads often meets complexity problems, not yet resolved in the tool. The design of each component should be incremental. A system containing a small number of components can only be verified at a quite high level of abstraction. After subcomponents have been refined with detailed implementations (and with specified connectors to their environments), they can be verified individually against the properties used as assumptions for the higher level of abstraction.

The most effective use of the tool is for verification of high-level models, or partial models of the critical parts only. The tool may provide effective feedback to early functional design decisions taken. Being already integrated with the industrially used Rhapsody tool, the UVE tool is potentially suitable for industrial acceptance. The learning curve for an effective use of the tool is quite steep; reasonable levels may be achieved within few weeks. On the other hand, the tool performance and the ability to deal with larger models and complex property specifications is highly dependable on the available hardware resources and can easily reach the current platform limits.

5.3 IFx

Like many explicit model-checking tools, IFx is quite effective for the verification of the functional aspects of reactive, concurrent systems. However, the experiments showed that the domain in which the tool becomes particularly interesting is the validation of timing conditions - where it can help in finding subtle errors like the reactivity degradation discussed in Sect. 3.3.2.

The tool is based on timed automata-theoretic methods and implements state of the art optimization and reduction techniques. However, timed verification is notoriously difficult and resource intensive, and thus the IFx tool is best used for solving isolated, hard timing problems in a UML design.

The ability to specify bounded or unbounded time non-determinism was found very useful for modeling realistic system environments, like the two Data Sources or the bus controller failures. However, bounded time non-determinism is the foremost cause of state space explosion. A rule of thumb is that the tool can handle up to about 8 non-deterministic cyclic behaviors in parallel (if the different time-bounds are in the same order of magnitude).

Relaxation of timing conditions was found to be a simple and efficient abstraction technique: removing the bounds on a non-deterministic element provides an easy to specify over-approximation of its behavior - meaning that the resulting specification defines a superset of the executions of the initial one. This transformation is conservative for the satisfaction of safety properties (including timed ones): if the resulting model satisfies a property, the initial one will too.

Applying this form of abstraction may drastically improve the analysis performance, as the state space of the resulting model is much smaller; due to the use of a symbolic representation of time, many, previously distinct, states are represented by the same symbolic state. The downside is that it can introduce false negative results. In the MARS example we have used this form of abstraction; when verifying properties concerning the detection of Data Sources failures, we have removed the bounds on the ControllerMonitor polling period, making the polling completely non-deterministic. This allowed a proof of the properties, while reducing the state space by at least two orders of magnitude (a precise number cannot be produced, since the analysis of the system without this abstraction did not terminate).

The experiments with IFx show that, generally, specification of properties using observers is intuitive and verification may be performed by a non-specialist after a relatively short learning period. This form of specification is limited to timed safety properties; it would be possible to interpret observers as Büchi automata for expressing liveness properties, but this leads to complex and error-prone specifications. For liveness properties we therefore recommend a temporal logic style of specification (e.g., using the μ-calculus evaluator of IF).

Some methodological guidelines for writing observers and for using the toolbox have been developed as a side result of the teamwork within OMEGA. The verification experiments also suggested several new useful features for the tool, like informal actions and inter-observer communication. Informal actions can be used as observation conditions, without affecting the behavior of the original model. Inter-observer communication allows a more intuitive and less complicated specification of complex properties, compared to the use of a single observer.

While currently exhibiting a more academic, non-industrial way of use, the IFx tool proves to be very effective in providing early feedback to the design decisions and their effect on timing matters. The tool has significant added value in cases of timing non-determinism of the system environment. The fact that the tool is, in principle, usable in combination with any commercial UML tool capable of exporting XMI, facilitates its potential introduction into the industrial software development process. The learning curve for the effective tool use can be characterized as medium, with a rough estimation of 4 to 5 weeks. This is mainly due to the involvement of internal tool information in user interaction, which obscures the UML-level overview. Moreover, the error messages are mostly at the level of the underlying IF tool and not at the level of the UML model. While acknowledging the named issues, the IFx tool has a potential of effective and beneficial use in the industrial environment.

5.4 PVS

In the PVS-based verification experiments, we first showed that it is possible to prove general properties of the non-timed version of MARS. In contrast to model-checking techniques, there are no limitations on finiteness of the UML model or the properties to be verified. The first attempts required quite some user interaction, which could be improved by the use of the TLPVS package. Still, expert knowledge and good skills in the use of the tool are required for the application of such interactive theorem provers.

The PVS-based verification experiments on MARS revealed that global, non-compositional verification of a timed version of the original model is difficult. In general, interactive verification of UML models

is very complex because we have to deal with a number of features simultaneously, such as timing, synchronous operation calls, asynchronous signals, threads of control, and hierarchical state machines. Hence, modularization of semantic definitions is important to allow the use of the minimal semantics for the features needed in a particular case study. For instance, if there are no asynchronous signals, then the complexity of manipulating event queues can be removed.

The compositional verification of the redesigned MARS example shows that PVS is more suitable for the correctness proof of high-level decompositions, to obtain relatively small components that are suitable for model checking. Observe that the compositional approach requires substantial additional effort to obtain appropriate specifications for the components. Moreover, finding suitable specifications is difficult and it is easy to make mistakes in declarative specifications. Hence, it is advisable to start with finite-state high-level models of the components and to simulate and to model-check them as much as possible. Interactive verification should be applied only when sufficient confidence has been obtained. Finally, it is good to realize that interactive verification is quite time consuming and requires detailed knowledge of the tool.

6 Concluding remarks

We have used the MARS case study to experiment with various forms of formal support for UML-based development. This includes requirements capturing by LSCs, timed and non-timed model checking, using IFx and UVE, and interactive verification by means of PVS.

Clearly, scalability is an important problem for all techniques. In the MARS example, small models lead to state explosion problems for model checkers, e.g., due to the asynchronous data sources. Moreover, the large number of features in UML (such as synchronous and asynchronous communication, threads of control, and hierarchical state machines) creates an additional layer of complexity, since the semantics of all these features has to be included in the tools. This makes, for instance, non-compositional verification of small examples in PVS already very complex and user intensive.

The work on the redesign of the MARS system shows that a clean modular design allows compositional PVS-based reasoning and improves the abstraction possibilities for model checkers. A high-level decomposition of the system into a set of components that communicate by asynchronous message passing (similar to the ROOM approach [43]) increases the size of the models that can be verified by a fruitful combination of tools, together with compositionality and abstraction.

To exploit the combination, we propose to use LSCs to capture high-level requirements and to specify the behavior of internal objects. Next, the correctness of high-level decompositions can be proved by theorem proving in PVS. Since it is usually difficult to find suitable specifications for components, it is important to precede the time consuming interactive verification task by experiments with the model-checking tools to simulate and to check finite instances of the decomposition. By modeling the behavior of components by abstract state machines, more insight can be obtained which will make it more easy to formulate suitable declarative component specifications. In this way, the efficiency of interactive theorem proving can be improved.

After proving the correctness of a decomposition by means of PVS, one may repeat the process to further decompose a component or, when relatively small components have been obtained, to prove the correctness of a detailed component design with respect to the high-level specification. This last step can again be done by means of a combination of formal techniques, using UVE for the non-timed, finite part, IFx for the finite timed part and PVS for the infinite aspects. An interesting topic, which requires further research, is the possibility to synthesize a state machine automatically from an LSC specification.

In general, we observed that the effectiveness of formal verification techniques, can be increased by designing well-structured models and applying verification already at the abstract levels, using assumptions about the non-implemented parts of the model, in addition to assumptions on the environment. Compositional verification can decrease the complexity of system analysis by wiring properties of small components together and deriving global requirements without analysis of the individual implementations. But this requires additional efforts concerning model (re)design and might introduce overhead to obtain a suitable abstraction of the communication between components. Hence, it creates an additional burden for industrial development process, and one has to investigate whether this is beneficial. In the MARS example, the redesign had positive side-effects on the re-usability, flexibility (e.g., for changing the error logic) and

extendability (e.g., to more data sources) of the system.

We have used many different specification formalisms, such as LSCs, temporal logic patterns, propositional logic, higher-order logic, and observer state machines. As expected, industrial users within the OMEGA project usually prefer the visual - more operational - descriptions, such as LSCs and observer state machines, to the declarative logic formulations. For instance, OCL has not been used because the limited amount of tool support for this notation in our project and the required learning time.

A disadvantage of the broad spectrum of specification languages is that specifications had to be reformulated manually when moving from one technique to the other. This was especially inconvenient during later stages of the OMEGA project, when there was more emphasis on the combinations of tools. Automatic translations between specification formalisms is very well conceivable, but out of the scope of the OMEGA project and a topic of future research. Besides an improved coupling at the syntactic level, e.g., using XML, also more research is needed to establish a coherence of the semantics. The current implementations of the semantics of the OMEGA kernel language have not been validated in a systematic way, but within the OMEGA project research has been started to include a description of this semantics in XML format, using the so-called Rule Markup Language (RML) [2].

Acknowledgements

Our thanks goes to all OMEGA partners for many extensive discussions about the formal techniques presented here and their application to UML models of industrial case studies. The reviewers are gratefully acknowledged for many useful comments and valuable suggestions.

References

[1] Arons, T., Hooman, J., Kugler, H., Pnueli, A., van der Zwaag, M.: Deductive verification of UML models in TLPVS. In: Proc. UML 2004, pp. 335–349. LNCS 3273, Springer-Verlag (2004)

[2] de Boer, F.S., Bonsangue, M.M., Jacob, J., Stam, A., van der Torre, L.W.N.: Enterprise architecture analysis with XML. In: HICSS (2005)

[3] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley (1999)

[4] Bornot, S., Sifakis, J.: Relating time progress and deadlines in hybrid systems. In: International Workshop HART, pp. 286–300. LNCS 1201, Springer-Verlag (1997)

[5] Bozga, M., Graf, S., Mounier, L.: IF-2.0: A validation environment for component-based real-time systems. In: Conf. on Computer Aided Verification (CAV), pp. 343–348. LNCS 2404, Springer-Verlag (2002)

[6] Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time, pp. 237–267. LNCS 3185, Springer-Verlag (2004)

[7] Burmester, S., Giese, H., Schäfer, W.: Model-driven architecture for hard real-time systems: From platform independent models to code. In: Model Driven Architecture - Foundations and Applications, First European Conference (ECMDA-FA), pp. 25–40. LNCS 3748, Springer-Verlag (2005)

[8] Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design **19**(1), 45–80 (2001)

[9] Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time UML semantics for concurrency and communication in safety-critical applications. Science of Computer Programming **55(1-3)**, 81–115 (2005)

[10] Damm, W., Josko, B., Votintseva, A., Pnueli, A.: A formal semantics for a UML kernel language. Available via http://www-omega.imag.fr/ Part I of IST/33522/WP1.1/D1.1.2, Omega Deliverable (2003)

[11] Douglass, B.: Real-time design patterns: robust scalable architecture for real-time systems. Object Technology. Addison-Wesley (2003)

[12] Graf, S., Ober, I., Ober, I.: A real-time profile in UML. Int. Journal on Software Tools for Technology Transfer (STTT) **8**(4), 113–127 (2006)

[13] Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Computer pp. 31–42 (1997)

[14] Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Int. J. of Foundations of Computer Science (IJFCS). **13**(1), 5–51 (2002)

[15] Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Conf. on Formal Methods in Computer-Aided Design (FMCAD), pp. 378–398. LNCS 2517, Springer-Verlag (2002)

[16] Harel, D., Kugler, H., Pnueli, A.: Smart play-out extended: Time and forbidden elements. In: Conf. on Quality Software (QSIC), pp. 2–10. IEEE Computer Society Press (2004)

[17] Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenarios-based requirements. In: Formal Methods in Software and System Modeling, pp. 309–324. LNCS 3393, Springer-Verlag (2005)

[18] Harel, D., Kugler, H., Weiss, G.: Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. In: Scenarios: Models, Algorithms and Tools, pp. 26–42. LNCS 3466, Springer-Verlag (2005)

[19] Harel, D., Marelly, R.: Playing with time: On the specification and execution of time-enriched LSCs. In: Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 193–202 (2002)

[20] Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag (2003)

[21] Harel, D., Marelly, R.: Specifying and executing behavioral requirements: The play in/play-out approach. Software and System Modeling (SoSyM) **2**(2), 82–107 (2003)

[22] Hooman, J.: Compositional verification of real-time applications. In: Compositionality – The Significant Difference (COMPOS), pp. 276–300. LNCS 1536, Springer-Verlag (1998)

[23] Hooman, J., van der Zwaag, M.: A semantics of communicating reactive objects with timing. Int. Journal on Software Tools for Technology Transfer (STTT) **8**(4), 97–112 (2006)

[24] Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), pp. 395–414. LNCS 2469, Springer-Verlag (2002)

[25] Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal logic for scenario-based specifications. In: Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 445–460. LNCS 3440, Springer-Verlag (2005)

[26] Kwon, G.: Rewrite rules and operational semantics for model checking UML statecharts. In: UML 2000, pp. 528–540. LNCS 1939, Springer-Verlag (2000)

[27] Kyas, M., Fecher, H., de Boer, F., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. In: Semantic Foundations of Engineering Design Languages (SFEDL), vol. 115, pp. 39–47. Electronic Notes in Theoretical Computer Science (ENTCS) (2005)

[28] Larsen, K., Petterson, P., Yi, W.: UPPAAL: Status & developments. In: Conf. on Computer Aided Verification (CAV), pp. 456–459. LNCS 1254, Springer-Verlag (1997)

[29] Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing **11**(6), 637–664 (1999)

[30] Lavazza, L., Quaroni, G., Venturelli, M.: Combining UML and formal notations for modelling real-time systems. In: European Software Engineering Conference, pp. 196–206. ACM SIGSOFT (2001)

[31] Marelly, R., Harel, D., Kugler, H.: Multiple instances and symbolic variables in executable sequence charts. In: Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 83–100 (2002)

[32] Mota, E., Clarke, E.M., Groce, A., Oliveira, W., Falcão, M., Kanda, J.: VeriAgent: an approach to integrating UML and formal verification tools. Electronic Notes in Theoretical Computer Science (ENTCS) **95**, 111–129 (2004)

[33] Ober, I., Graf, S., Ober, I.: Validating timed UML models by simulation and verification. Int. Journal on Software Tools for Technology Transfer (STTT) **8**(4), 128–145 (2006)

[34] OMG: UML 2.0 Superstructure. http://www.omg.org/cgi-bin/doc?formal/05-07-04 (2005)

[35] OMG: UML Profile for Schedulability, Performance, and Time, v1.1. Available from http://www.omg.org/cgi-bin/doc?formal/2005-01-02 (2005)

[36] Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Conference on Automated Deduction, pp. 748–752. Lecture Notes in Artificial Intelligence 607, Springer-Verlag (1992)

[37] Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Trans. Software Eng. **21**(2), 107–125 (1995)

[38] Paltor, I.P., Lilius, J.: Formalising UML state machines for model-checking. In: UML'99: The Unified Modeling Language - Beyond the Standard, pp. 430–445. LNCS 1723, Springer-Verlag (1999)

[39] Pnueli, A., Arons, T.: TLPVS: A PVS-based LTL verification system. In: Verification: Theory and Practice, pp. 598–625. LNCS 2772, Springer-Verlag (2003)

[40] PVS: http://pvs.csl.sri.com/

[41] Reggio, G., Astesiano, E., Choppy, C., Hußmann, H.: Analysing UML active classes and associated statecharts - a lightweight formal approach. In: Conf. on Fundamental Approaches to Software Engineering (FASE), pp. 127–146. LNCS 1783, Springer-Verlag (2000)

[42] Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML verification environment. In: Conf. on Software Engineering and Formal Methods (SEFM), pp. 174–183. IEEE Computer Society Press (2004)

[43] Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons (1994)

[44] Shen, W., Compton, K.J., Huggins, J.: A toolset for supporting UML static and dynamic model checking. In: Computer Software and Applications Conference (COMPSAC), pp. 147–152 (2002)

[45] Traoré, I., Aredo, D.B., Ye, H.: An integrated framework for formal development of open distributed systems. Information & Software Technology **46**(5), 281–286 (2004)

[46] Xie, F., Levin, V., Browne, J.C.: Objectcheck: A model checking tool for executable object-oriented software system designs. In: Conf. on Fundamental Approaches to Software Engineering (FASE), pp. 331–335. LNCS 2306, Springer-Verlag (2002)

[47] Yovine, S.: Kronos: A verification tool for real-time systems. Int. Journal on Software Tools for Technology Transfer **1**(1-2) (1997)

[48] Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva (1996)