Innovations Syst Softw Eng (2008) 4:301–308
DOI 10.1007/s11334-008-0066-z

ORIGINAL PAPER

# Timing analysis and validation with UML: the case of the embedded MARS bus manager

**Iulian Ober · Susanne Graf · Yuri Yushtein · Ileana Ober**

Received: 1 July 2008 / Accepted: 26 July 2008 / Published online: 23 August 2008

**Abstract** This paper presents a case study in UML-based modeling and validation of the intricate timing aspects arising in a small but complex component of the airborne Medium-Altitude Reconnaissance System produced by the Netherlands National Aerospace Laboratory. The purpose is to show how automata-based timing analysis and verification tools can be used by field engineers for solving isolated hard points in a complex real-time design, even if the press-button verification of entire systems remains a remote goal. We claim that the accessibility of such tools is largely improved by the use of an UML profile with intuitive features for modeling timing and related properties.

This work has been partially funded by the European OMEGA project (IST-2001-33522). Iulian Ober and Ileana Ober: performed work while at VERIMAG, Grenoble, France. Y. Yushtein: at the moment of writing in National Aerospace Laboratory NLR.

I. Ober (✉) · I. Ober
Université de Toulouse, IRIT,
118 Route de Narbonne, 31062 Toulouse, France
e-mail: iulian.ober@irit.fr

I. Ober
e-mail: ileana.ober@irit.fr

S. Graf
CNRS, VERIMAG, 2, Av. de Vignate, 38610 Gières, France
e-mail: Susanne.Graf@imag.fr

Y. Yushtein
National Aerospace Laboratory NLR,
Embedded System Department, Amsterdam, The Netherlands

*Present Address:*
Y. Yushtein
Systems, Software and Technology Department,
European Space Agency/ESTEC, Noordwijk, The Netherlands
e-mail: Yuri.Yushtein@esa.int

## 1 Introduction

The analysis and design of real-time systems often raises highly intricate problems as system development aims to preserve certain timing conditions and guarantee that the system responds appropriately and in a timely fashion to a complex environment. The cause of this intricacy is the very nature of time, which (at the level of human perception and of presently designed systems) appears as an *absolute*, *global* notion, thus implicitly aggregating the *relative* and *local* timing conditions appearing in system design.

The construction of systems based on local hypotheses and local solutions is nevertheless a mandatory requirement for being able to design nontrivial systems by functional decomposition. Consequently, designers seem to be obliged to build systems by component aggregation, without knowing a priori what effect this aggregation will have on the timeliness of each component and of the system as a whole. Examples of unexpected timing conditions resulting from this aggregation will be shown by using the case study presented in this paper.

One solution to this problem lies in using automated tools to analyze the timeliness of a system. When timing aspects are limited to task execution times, task activation time constraints, and task deadlines, it is often possible to use scheduling theory (e.g., rate monotonic analysis or response time analysis [6]) and tools such as MAST [12]. However, when the functionality of the system itself depends on timing aspects (time-adaptive systems), as in the case of the Medium-Altitude Reconnaissance System (MARS) system presented in this paper, one has to use general modeling and property verification frameworks to analyze timeliness. There are two large classes of methods: *model checking*, which analyzes a semantic model algorithmically, and *theorem proving*. This paper is about using a model-checking approach, which is more automatic. Very high-level parametric models are sometimes better tackled by proof-based techniques; however, in general these models are elaborated by verification experts rather than the engineers that develop the system, and the distance between these high-level models and the developed system may be important; e.g., when using duration calculus [7] as the verification framework, almost all of the functional model has to be abstracted.

Nevertheless, automated verification tools have well-known limitations, and a first obstacle to putting these tools to work effectively is that designers have to understand them and build models with these limitations in mind. From our experience, interesting insights into the timing aspects of a system are usually gained only when the (unrelated) details of the functional part are abstracted away. Not many software engineers do this naturally, but they can learn to do so when they see the benefit.

The second obstacle is the complexity of the formalism for capturing a timing model and its properties. Although from a theoretical point of view the verification approach we propose in this paper is very close to what is done in "traditional" timed model checking tools such as Kronos [22] or UPPAAL [2], the main difference lies in how designers model their system and its properties. We argue that a tool is more easily adopted if the formalism it uses is intuitive for the designers and based on concepts that they already know. In the literature there are various extensions of temporal logics with quantitative time operators, which have the required expressiveness. However, from our experience, property formalisms based on familiar concepts (such as state machines) are more easily accepted by users and are more expressive.

In this paper, we present the results of a case study conducted jointly by experts and industrial users, in which meaningful results about timing properties of the studied system were obtained by analyzing a model tailored for this purpose using a user-friendly UML-based tool. The rest of the paper is structured as follows: Sect. 2 presents the case study, with focus on the timing aspects. Section 3 presents the modeling of this case study using a specific formalism (the OMEGA UML profile), the main results of timing validation, and the techniques employed during the experience. In Sect. 4, we discuss some conclusions that can be drawn from this study.

## 2 The MARS system

### 2.1 Overall presentation

The acronym MARS stands for Medium-Altitude Reconnaissance System. The system controls a high-resolution photo camera embedded in a military aircraft, taking pictures of the ground from medium altitude. The system counteracts the image quality degradation caused by the forward motion of the aircraft by creating a compensating motion of the film during the film exposure. The system is also responsible for annotating the frames with the current time and position. The system also performs health monitoring and alarm processing functions.

Exposure control [forward motion compensation (FMC) and frame rate] as well as annotations are computed in real time based on the current aircraft altitude, ground speed, navigation data (latitude, longitude, heading), time-of-day, etc. These parameters are acquired from the avionics data bus of the aircraft.

### 2.2 The Databus Manager

For the purpose of this case study we concentrated on a subsystem of MARS that presents interesting timing problems. This subsystem, called the Databus Manager ($DM$ in the following) monitors the health of the data bus controller and, in general, the health of the communication going on through the data bus.

The MARS system receives data concerning altitude and navigation from other components of the avionic system. The $DM$ component supervises the (non-)reception of data messages, and provides a *status*, which is used by the system's alarm logic. In addition, the $DM$ periodically polls the databus controller and changes its status when the controller fails/recovers. Thus, the status computed by the $DM$ has three values: *Operational*, *BusError*, and *ControllerError*. The precise requirements on the $DM$ status computation are described below.

The two types of data inputs of the $DM$ are received periodically, with a period of $P = 25$ ms and a jitter of $\pm J = 5$ ms, and may occasionally be lost. The periods are not synchronized and may have an arbitrary offset smaller than the length of one period. Figure 1 shows a possible configuration of the reception windows along the time axis (windows in which no message reaches the $DM$ are marked with $KO$).

The basic functional requirements on the $DM$ status are:

- Failure of the controller leads to a change of status to *ControllerError*. Recovery leads to *BusError*.
- Status changes from *BusError* to *Operational* when two correct consecutive messages are received from both sources (assuming no controller error).
- Status changes from *Operational* to *BusError* when three consecutive messages from a source are lost (assuming no controller error).

We note that these requirements do not define quantitatively the moment when the status change takes place. In fact, maximal reactivity is desirable. Two reactivity measures (at least) can be defined for the $DM$:

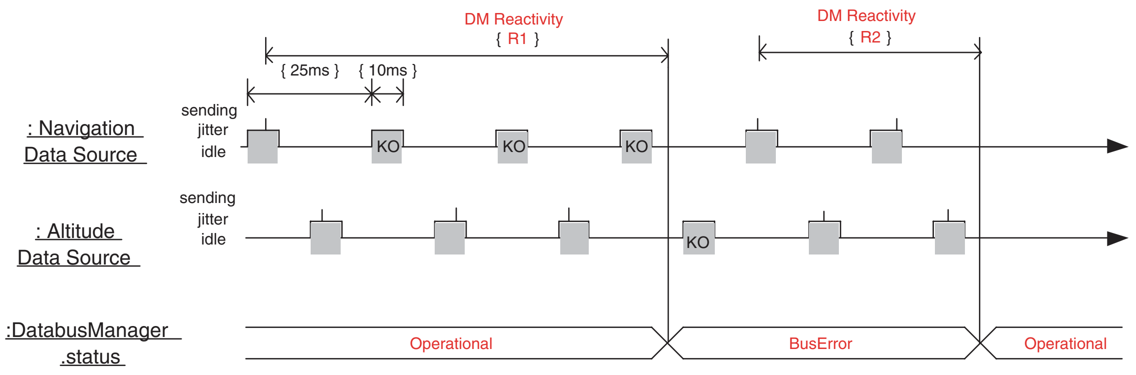

**Fig. 1** Timing diagram showing the message emission windows, $DM$ status, and reactivity measures

- Reactivity to errors, defined as the upper bound that the $DM$ guarantees for the time ($R1$) between the last correctly received message from the source causing a switch to *BusError*, and the actual moment of the switch.
  Analysis shows that $R1$ is necessarily greater than 85 ms ($3P + 2J$). The actual reactivity depends on the implementation chosen for the $DM$, as we will see in the next section. This value of 85 ms gives us an ideal reactivity that should be approached.
- Reactivity to recovery, defined as the upper bound that the $DM$ guarantees for the time ($R2$) between the first message in a series of correct messages leading to a switch to *Operational* and the actual moment of the switch.
  In this case, $R2$ depends on the offset between the periods of the two data sources. However, even in the worst case $R2$ is less than 60 ms ($2P + 2J$).

The experiments we conducted are described in the next section. They had two goals:

(1) to check that the proposed implementations for the $DM$ verify the aforementioned functional properties
(2) to determine the reactivity bounds offered by the different proposed implementations (and determine the optimal solution).

## 3 UML modeling and validation experiments

### 3.1 Background on OMEGA UML and the IFx toolset

The MARS subsystem was modeled using the OMEGA UML profile and timing and functional validation was performed using the IFx toolset. Here we briefly introduce these technologies; for further details we refer the reader to [11, 19]

OMEGA UML is an *executable* UML profile designed to suit the needs of designers of real-time embedded systems. It is comprised of: (1) a *complete syntax and operational semantics* which makes choices with respect to the UML semantic variation points left open in the standard (e.g., concurrency model, object interaction primitives, concrete syntax of actions) and (2) *extensions for describing timing aspects*, which can be compared to those defined in the MARTE profile [20] (but much simpler and more limited in scope).

The semantic choices (1) largely correspond to the computation model of the Rhapsody UML tool (see [14] for details); for the complete semantics of OMEGA UML we refer the reader to [8].

The syntax of actions, used for specifying method bodies and transition effects (see, for example, Figs. 2 and 3) is fixed by the OMEGA action language (OMAL). This imperative language covers object creation and destruction, operation calls, expression evaluation, variable assignment, signal output, return action, and control flow structuring statements (conditionals and loops). The concrete syntax relies on commonly used conventions in object-oriented languages and is not further detailed.

The *timing aspects* (2) are described in detail in [11]. The profile is compatible with the basic time-related notions of UML 2.0 by defining a series of lightweight extensions

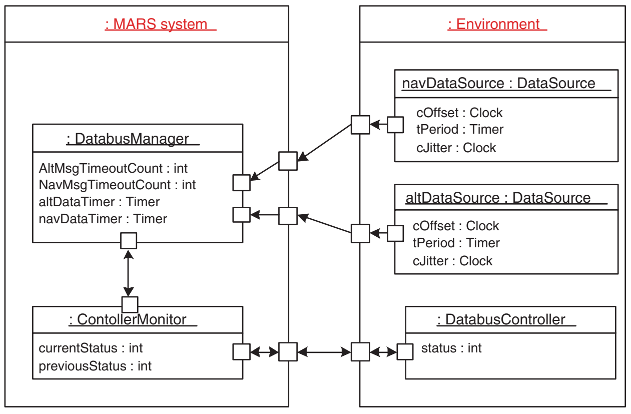

**Fig. 2** State machine of the DatabusManager

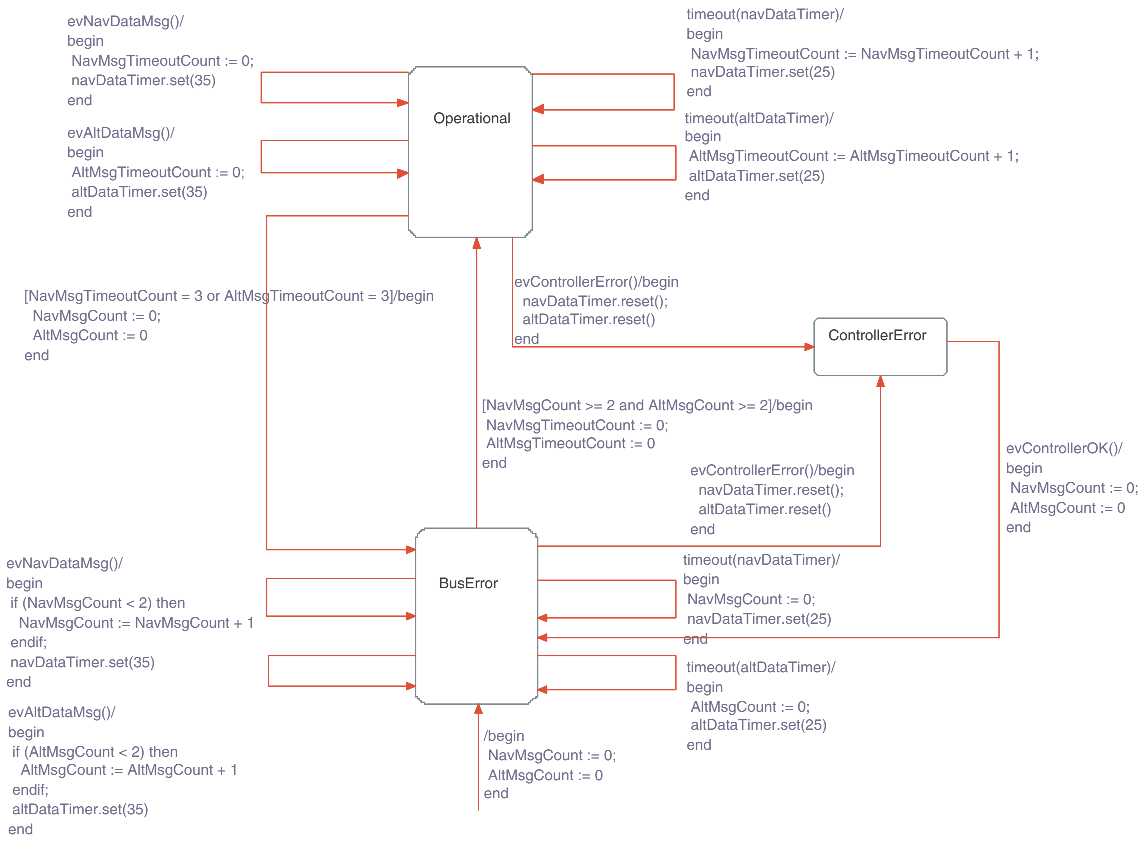

**Fig. 3** Observer for verifying the reactivity bound for $R1$

to UML for describing time-driven behavior using timers, clocks, and timed guards. In addition it allows the definition of transition *urgency*, a concept taken from timed automata with urgency [5]. For the expression of timing and functional requirements, the OMEGA UML profile proposes notions of *events* and *observer objects*. Events are any semantic-level state changes; the profile defines a concrete notation for referring to them in the model. Observers are characterized by a state machine which reacts to the aforementioned events and to conditions occurring in the system, and acts as an acceptor of system executions by using states stereotyped with <<`error`>> as final states. An example of a property expressed by observers appears later in the paper (Fig. 3).

The IFx toolset offers simulation and verification functionality for OMEGA UML models. It works by translating the models (which are given as XMI files exported from UML editors such as Rational Rose or Rhapsody) into the input language of the IF model checker [3], a language based on extended communicating timed automata. The details of the translation and tool integration can be found in [19].

The IF model checker contains a state-space exploration engine and is designed to scale to complex models by providing several optimizations and support for abstraction. The tool implements static and dynamic optimizations such as dead variable factorization, dead code elimination, partial-order reduction, and abstract interpretation of clocks. All optimizations strongly preserve timed safety properties which are of interest in the MARS system. In addition, the tool supports simple abstractions which preserve satisfaction of safety properties, but may show spurious counterexamples.

## 3.2 Overview of the UML model for MARS

The architecture of the MARS model as proposed by the designer of the system is shown in the UML composite structure diagram in Fig. 4. The main component is the *DatabusManager* object which maintains the global status and monitors message loss. For simplicity, the designer has separated

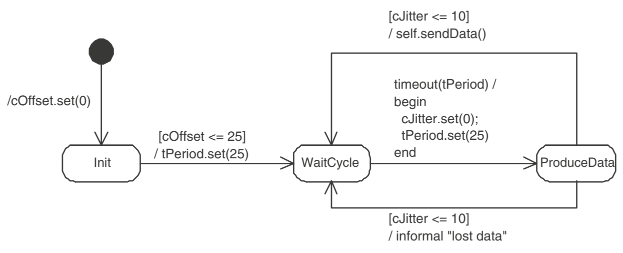

**Fig. 4** Composite structure of the MARS model

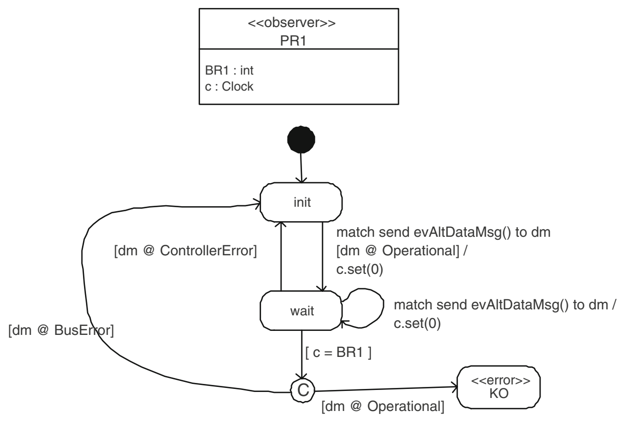

**Fig. 5** Environment model: state machine of the data sources

the polling of the bus controller into a different object, the *ControllerMonitor*.

In order to verify the $DM$ under the assumptions on message arrival and controller errors mentioned in Sect. 2.2, the OMEGA profile allows the environment to be modeled using the same concepts as used for modeling the system, in particular an explicit object with the behavior expressed by the assumption. In Fig. 4, we see therefore three environment objects, corresponding to the altitude data source, the navigation data source, and the bus controller.

In particular, for modeling the environment, the possibility to express nondeterministic behavior is important. This is allowed in the OMEGA profile; for example, Fig. 5 shows the state machine of data sources, using interval conditions on clocks to model the nondeterminism introduced by the starting time and by jitter. This state machine indeed describes a data source with the required period and jitter as all transitions of environment objects are interpreted as *delayable*[1], that is, once they are enabled, they will be taken before their time guard becomes false or they may be disabled by some discrete transition. Moreover, Zeno computations[2] are not valid computations, which guarantees in this example that the computation cannot become stuck in any state (since in every state there is an upper bound on the time that may pass).

As we will see later, the requirements concerning the $DM$ can be achieved in several ways. The first design provided by the engineer consisted of a single state machine with transitions triggered by events from both data sources ($evAltDataMsg$, $evNavDataMsg$) and from the *ControllerMonitor* ($evControllerError$, $evControllerOK$), or by timeouts corresponding to message loss detection ($altDataTimer$, $navDataTimer$). The principle is to use a timer to measure the duration of the period for each data source—where the end of the period is defined by the reception of a message, where the timer is re-armed, or a timeout—and to always keep track of the number of consecutive received or lost messages.

## 3.3 Expressing properties and first evaluation results

Both the functional and the reactivity properties described in Sect. 2.2 can be expressed as observers to be verified on this model. Figure 3 shows the observer checking a bound guaranteed for $R1$, that is, the maximal delay needed for transmission problem detection (Sect. 2.1). Note that observer transitions *synchronize* with observed events, and thus take place at the same time point as the observed event. Note also that this observer monitors only one data source (the altitude data source); due to the symmetry of the model, the failure of the other source will result in the $DM$ switching to the $BusError$ state in the same time interval.

All functional and reactivity properties were verified against the initial design presented above. Due to the important state explosion encountered, a simplifying assumption was made on the environment: we consider that the two data sources are synchronized (i.e., their period of 25 ms begins at the same time in every cycle; the two data can nevertheless be sent at different moments due to jitter). It is clear that this assumption is not conservative as the reaction time due to message loss may (and is likely to) be longer when the two data sources are desynchronized. In order to verify the properties fully without this nonconservative assumption, a different model for the $DM$ had to be designed, which is presented later on.

Nevertheless, this initial model was useful for understanding the potential problems, debugging the model, and ruling out some variants that had been proposed to increase efficiency. Under the simplifying assumption, all functional properties have finally been proved to hold on debugged versions of the $DM$ design.

An interesting outcome is that very similar designs may present different reactivity bounds. For example, consider a slight variant of the design model presented above, in which in the *operational* state a *long timeout* is used, detecting the absence of messages during three consecutive periods, instead of detecting the absence of individual messages and counting them. At first sight, this version looks more efficient as it does not need a counter in the *operational* state.

Using different variants of the reactivity constraints defined by the observer in Fig. 3, we have determined with the help of our verification tool that the initial design has a better reactivity (85 ms) than the new one (110 ms). As the motivation for the entire case study was to gain reactivity with respect to the existing synchronous design, which observes events only at fixed time points, this is not an acceptable solution.

[1] According to the terminology defined in timed automata with urgency [5].

[2] Computations with an infinity of steps but with finite time progress.

The diagnostic traces provided by the model checker show that the difference stems from the status of the timers at the transition from the *BusError* to the *Operational* state: in the initial version, timers are not affected by a status change, they just count periods of the data sources, while in the new version, the long timer is not needed in the error state and initialized when entering the operational mode. This might delay the detection of a bus error occurring just at this moment by an entire period. In order to correct the problem, the timer must be set depending on the actual "age" of the "period" timer used in the error state.

This kind of errors is quite common when trying to optimize a design, and the tools were very helpful for checking all the properties after any modification. This allows feedback to be obtained immediately, as running the verification is generally fast (see also the table). Another group has used in parallel tool-based automatic or interactive theorem proving. They could prove a parametric version of the property which is impossible to prove with a model-checker such as ours, but they were grateful for the quick feedback provided by our tool: once we had a (simplified) working model on which the properties could be shown to hold, we could analyze small modifications of the UML model, just by "pushing the button" to translate the model and then re-running the verification of all properties to find bugs.

## 3.4 Use of a compositional model and abstractions

In order to verify fully the desired properties without imposing unrealistic assumptions on the environment, we need to use conservative abstractions in the model of the *DM*. In order to do so, a more compositional model has been designed. The *DM* is decomposed into several parts (Fig. 6):

- A *Receiver* component for each data source. This component supervises the messages sent by one source and keeps track of the correct and erroneous messages during the last three reception windows. It sends out a signal *evCnt* with a numeric parameter of three bits, which represents the status of the last three reception windows (one for a correctly received message, 0 for a missed message, where the least significant bit corresponds to the most recent message).
- An *ErrorLogic* component, which receives *evCnt* messages from the *Receivers*, and which maintains the global status. It goes from the *BusError* state to the *Operational* state when all *Receivers* have received correct messages during the last two reception windows. It goes from the *Operational* state to the *BusError* state when at least one *Receiver* has received no correct messages during the last three reception windows.

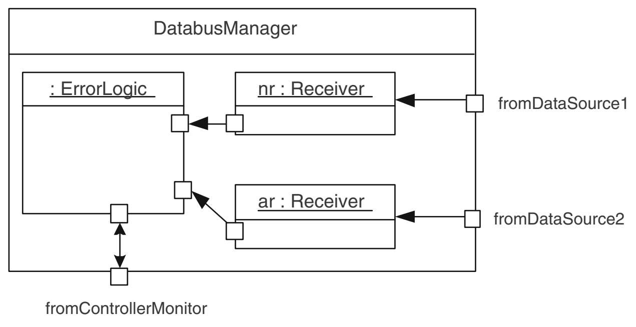

**Fig. 6** Decomposition of the *DM*

Using this model, we could verify all properties for the general case of desynchronized sources, using a compositional and conservative abstraction.

The abstraction used consists of replacing one *Receiver* with a chaotic abstraction *ReceiverAbs* which may send *evCnt* with any parameter at any time. This is a very rough approximation of the source–receiver pair, but it proved to be sufficient for preserving the desired properties. The abstraction is particularly interesting as it allows a generalization to a system with more than two data sources.

A second conservative abstraction used consisted of replacing the deterministic polling cycle of the *ControllerMonitor* (10 ms in the initial model) by a completely nondeterministic polling policy. While this introduces new executions, impossible in the initial model, the resulting state space is smaller as many previously disjoint states are grouped together (when a symbolic representation of time is used in the verification tool).

To assess the efficiency of these abstractions, the table in Fig. 7 shows the size of the state space and the processing time for several configurations of the MARS system. In particular, desynchronizing two resources has a tremendous effect on the size of the state space, which is in fact due to the simultaneous presence of jitter and desynchronization. Notice that the effect is much more important than it appears, as we stopped the exploration when, after reaching 1 million states and 10 min, the state space was still growing rapidly—experience told us that it was likely not to be worth to wait any further: even if the exploration were to converge (which we did not really anticipate) the result was not very useful to us, as we wanted to be able to rerun experiences on variants in a short time. As we have not run our experiences on particularly well-equipped machines (especially in terms of memory), this means that we still have the potential to extend the approach by a few orders of magnitude and handle slightly more complex systems.

The use of both types of conservative abstractions leads to a state space of about the same size as the much simpler system with synchronized sources, and which is still precise enough to satisfy all properties.

**Fig. 7** Verification times and state-space sizes for different verification configurations

| Configuration | Number of states | Number of transitions | User time |
|---|---|---|---|
| Initial model with only one source (no *CM* polling) (*non-conservative*) | 1084 | 1420 | $< 1s$ |
| Initial model with two synchronized sources (no *CM* polling) (*non-conservative*) | 99355 | 151926 | $36s$ |
| Initial model with two de-synchronized sources (no *CM* polling) (*conservative* – does not finish) | $> 1136768$ | $> 1676126$ | $> 9m30s$ |
| Abstract model, 10ms *CM* polling (*conservative* – does not finish) | $> 1494864$ | $> 701120$ | $> 8m12$ |
| Abstract model with no *CM* polling (*non-conservative*) | 118690 | 174871 | $45s$ |
| Abstract model with lazy *CM* polling (*conservative*) | 155166 | 263368 | $1m21s$ |

## 4 Conclusion

The experiment presented here has shown that timing analysis tools may be used efficiently for solving isolated, hard timing problems in a UML design, even if fully automated verification for large designs remains a remote goal. Also we believe that more systematic use of functional decomposition, as used in the example, can definitely enable the verification of much larger designs in a compositional fashion, as there is no need for the verification of a model in which all parts are described in full detail.

The use of the OMEGA UML profile in order to capture timing models and properties facilitated very quick learning and adoption of our tools by experienced UML designers. Without the knowledge of a verification expert, the designers were able to use even advanced techniques such as abstractions.

A very efficient abstraction technique in such models is the relaxation of timing constraints, which is usually very simple to model (in many cases it involves only the change of the urgency attribute of some transitions in the system model). Since this kind of abstraction is always an overapproximation of the system's behavior, it is always conservative for the satisfaction of safety properties, including timed ones.

On the other hand, functional abstractions may be less obvious and may imply a refactoring of the system, as was shown in Sect. 3.4. The experience and intuition of the analyst plays a big role in identifying and modeling such abstractions. The MARS example showed that, by practice, a designer can acquire these skills and design abstractions that do not break the verified properties (for example, the ones presented in this paper did not introduce any spurious error trace).

### *Related and future work*

A number of tools have been proposed for the validation of UML models by translating a subset of UML into the input language of some existing validation tools (see [1,4,9,10,16–18,21] to mention only some of the relevant work in the context of real-time and embedded systems). Like IFx, most of these tools are based on existing model checkers such as SPIN [15] (in [17,18]) or COSPAN [13] (in [21] for untimed systems, and Kronos [22] (in [4]) or UPPAAL [2] (in [10,16]) for the verification of systems with timing constraints. Also the translation into proof-based frameworks has been proposed.

With respect to the coverage of UML concepts, IFx goes beyond many existing tools, as it handles a rich subset of UML including inheritance, dynamic object creation, and powerful timing features. Most of the cited UML validation tools are restricted to static systems, fitting exactly the formalism of the underlying model checker. Often they handle properties written in the property language of the model checker, without defining an appropriate formalism, whereas IFx proposes to use OMEGA *observers* for this purpose.

While our tool does not push forward the *theoretical* boundaries of verification technology, it presents a unique combination of features which prove to be very efficient in fighting scalability problems encountered in practice. It includes and combines the on-the-fly exploration of SPIN, the symbolic representation of time constraints of Kronos and UPPAAL, bisimulation-based reduction techniques, and adds verification-targeted optimizations based on static analysis, as well as support for industry standards such as SDL and UML.

At the request of the European Space Agency, the authors are currently planning to update the tool and to integrate it with the latest version of the Rhapsody environment.

## References

1. Arons T, Hooman J, Kugler H, Pnueli A, van der Zwaag M (2004) Deductive verification of UML models in TLPVS. In: Proceedings UML 2004, pp 335–349. LNCS 3273, Springer, Heidelberg
2. Behrmann G, David A, Larsen KG, Håkansson J, Pettersson P, Yi W, Hendriks M (2006) Uppaal 4.0. In: QEST, pp 125–126. IEEE Computer Society, New York
3. Bozga M, Graf S, Mounier L (2002) If-2.0: a validation environment for component-based real-time systems. In: Brinksma ED, Larsen KG (eds) CAV, vol 2404. Lecture notes in computer science, pp 343–348. Springer, Heidelberg
4. Del Bianco V, Lavazza L, Mauri M (2002) Model checking UML specifications of real time software. In: Proceedings of 8th international conference on engineering of complex computer systems. IEEE, New York
5. Bornot S, Sifakis J (2000) An algebraic framework for urgency. Inf Comput 163(1):172–202
6. Burns A, Wellings A (2001) Real-time systems and programming languages, 3rd edn. Addison Wesley, Reading
7. Chaochen Z, Hoare CAR, Ravn AP (1992) A calculus of durations. Informa Process Lett 40(5):269–276
8. Damm W, Josko B, Pnueli A, Votintseva A (2005) A discrete-time uml semantics for concurrency and communication in safety-critical applications. Sci Comput Program 55(1–3):81–115
9. del Mar Gallardo M, Merino P, Pimentel E (2002) Debugging UML designs with model checking. J Object Technol 1(2):101–117
10. David A, Möller O, Yi W (2002) Formal verification UML statecharts with real time extensions. In: Proceedings of FASE 2002 (ETAPS 2002), vol 2306, LNCS. Springer, Heidelberg
11. Graf S, Ober I, Ober I (2006) A real-time profile for uml. STTT 8(2):113–127
12. Harbour MG, Gutiérrez García JJ, Palencia Gutiérrez JC, Drake Moyano JM (2001) MAST: modeling and analysis suite for real time applications. In: ECRTS, pp 125–134. IEEE Computer Society, New York
13. Har'El Z, Kurshan RP (1988) Software for analysis of coordination. In: Conference on system science engineering. Pergamon Press, New York
14. Harel D, Kugler H (2004) The Rhapsody semantics of statecharts (or, on the executable core of the UML)—preliminary version. In: Ehrig H, Damm W, Desel J, Große-Rhode M, Reif W, Schnieder E, Westkämper E (eds) SoftSpez final report, vol 3147, LNCS, pp 325–354. Springer, Heidelberg
15. Holzmann GJ (1997) The model-checker SPIN. IEEE Trans Softw Eng 23(5):279–295
16. Knapp A, Merz S, Rauh C (2002) Model checking timed UML state machines and collaborations. In: Damm W, Olderog ER (eds) FTRTFT 2002, vol 2469 of LNCS, pp 395–414. Springer, Heidelberg
17. Latella D, Majzik I, Massink M (1999) Automatic verification of a behavioral subset of UML statechart diagrams using the SPiN model-checker. Formal Aspects of Computing (11)
18. Lilius J, Paltor IP (1999) Formalizing UML state machines for model checking. In: France R (ed) Proceedings of UML'1999, vol 1723, LNCS. Springer, Heidelberg
19. Ober I, Graf S, Ober I (2006) Validating timed UML models by simulation and verification. STTT 8(2):128–145
20. The Object Management Group. UML profile for modeling and analysis of real-time and embedded systems. OMG document ptc/07-08-04, 2007
21. Xie F, Levin V, Browne JC (2001) Model checking for an executable subset of UML. In: Proceedings of 16th IEEE international conference on automated software engineering (ASE'01). IEEE
22. Yovine S (1997) Kronos: a verification tool for real-time systems. STTT 1(1–2):123–133