Timing analysis and validation of the embedded MARS bus manager*

Iulian Ober[1], Susanne Graf[2], and Yuri Yushtein[3]

[1] GRIMM-ISYCOM Laboratory
IUT de Blagnac
1, place Georges Brassens, BP 73
31703 Blagnac Cedex, France
ober@iut-blagnac.fr

[2] VERIMAG
2, av. de Vignate
38610 Gières, France
susanne.graf@imag.fr

[3] yushtein@xs4all.nl

Abstract. This paper presents a case study in UML-based modelling and validation of the intricate timing aspects arising in a small but complex component of the airborne Medium Altitude Reconaissance System produced by NLR[1].
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 a UML profile with intuitive features for modeling timing and related properties.

1 Introduction

The analysis and design of real-time systems often raises very intricate problems as system development aims at preserving certain timing conditions and at guaranteeing 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 conception of systems in terms of local hypotheses and local solutions is nevertheless a mandatory requirement for being able to design non-trivial 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 (some relevant examples of unexpected timing conditions resulting from this aggregation will be shown on the case study presented in this paper).

One solution to this problem lies in using automated tools to analyze the timeliness of a system. There are basically two types of frameworks for reasoning with time: model-based and axiomatic ones. While axiomatic frameworks (for example the duration calculus [CHR92]) allow to reason about time independently of a behavioral model, model-based frameworks allow to mix in a same *model* both functional and timing aspects. Having a functional model of a system (which is a design artifact normally available for any system), it is in principle easier to obtain a faithful and complete mixed model. This is done by annotating the initial model with timing information.

* This work has been partially funded by the European OMEGA project (IST-2001-33522).
[1] National Aerospace Laboratory, The Netherlands.

On the level of semantics and of techniques underlying model-based timing analysis, one can find timed extensions of Petri-nets, process algebras or automata (for a bibliographic survey see [Obe01]). From the point of view of tooling, the most elaborate verification features can be found in tools based on (various flavors of) timed automata, like [BGO$^+$04, LPY97, Yov97].

In order for automated timing analysis to be put effectively to work, two educational barriers have to be stepped over. The first concerns understanding the limitations of current technology, which cannot work for systems presenting both a very complex functional behavior, and complex timing constraints. From our experience, interesting insights in the timing aspects of a system are usually gained only when the (unrelated) details of the functional part are abstracted away.

The second barrier concerns the complexity of the formalism capturing a timing model and its properties. In this case, the timing analysis tool can help the common user overcome the barrier, for example by embedding the timed formalism in a familiar language, and by reusing as much as possible from the user's knowledge. A relevant example in this sense is that of the languages used for expressing timing properties. In the literature there are various extensions of temporal logics extended with means to express quantitative time constraints, which prove to be very flexible. However, from our experience, property formalisms based on familiar concepts (like state machines accepting or rejecting a set of observations) are more easily accepted by the users.

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: §2 presents the case study, with focus on the timing aspects. §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 §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 being 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 sub-system of MARS which presents interesting timing problems. This sub-system, called Databus Manager (*DM* in the following) monitors the health of the data bus controller and, in general, the health of the communication going on trough the data bus.

As mentioned before, the MARS receives input 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 the status when 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 $P = 25ms$ and a jitter of $\pm J = 5ms$, and may occasionally get 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 the requirements above do not define quantitatively the moment when the status change takes place. It is obvious that a maximal reactivity is desirable. Two reactivity measures (at least) can be defined for the DM:

– 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 $85ms$ ($3P + 2J$). The actual reactivity depends on the implementation chosen for the DM, as we will see in the next section. This value of $85ms$ 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 $60ms$ ($2P + 2J$).

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 above mentioned functional properties, and
(2) to determine the reactivity bounds offered by the different proposed implementations (and point out the optimal solution).

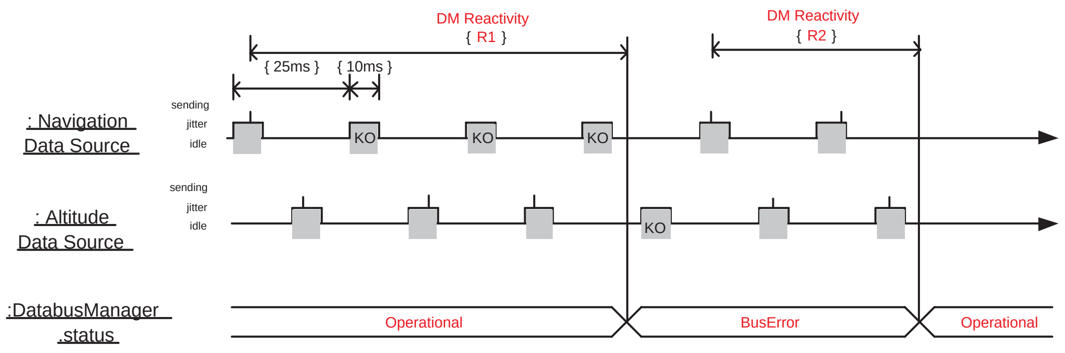

Fig. 1. Timing diagram showing the message emission windows, DM status and reactivity measures.

3 UML modeling and validation experiments

3.1 Background on the OMEGA UML profile and the IFx toolset

The MARS sub-system was modeled using the OMEGA UML profile and timing and functional validation was performed using the IFx toolset. Here we briefly introduce these technologies.

The OMEGA profile defines an operational semantics for a large subset of UML, designed to suit the needs of designers of real-time embedded systems. Concerning *functionality*, the semantics defines aspects pertaining to control (like the rules governing concurrency and the execution of active and passive objects) and communication primitives. These aspects of the profile are detailed in [DJPV03, DJPV05].

Concerning *timing*, which is described in detail in [GOO03, GOO05], the profile defines a series of lightweight extensions to UML for naming *events* associated with syntactic constructs and for describing time-driven behavior by using *timers*, *clocks*, *timed guards* or *transition urgency* attributes. A different part of the profile, not used here, deals with the declaration of timing constraints independently of the functional model. A third part concerns the formalization of timing and functional requirements, which in OMEGA UML are expressed by *observer* objects. Observers represent set of accepted observations which is defined by a state machine reacting to *events* and *conditions* occurring at the execution of the system. Observers define safety properties by use of states stereotyped with <<`error`>> as final states.

IFx [OGO05] is a toolset providing simulation and verification functionalities for OMEGA UML models. At the core of the tool there is a state space exploration engine for an extended communicating timed automata model (IF [BGM02, BGO$^+$04]). In order to scale to complex models, IF supports optimization and abstraction in several ways. The tool implements static and dynamic optimizations like dead variable factorization, dead code elimination, partial-order reduction and abstract interpretation of clocks. Each such optimization is conservative with respect to certain types of properties, and in particular with respect to timed safety properties which are of interest in the MARS system.

3.2 Overview of the MARS model

The architecture of the MARS model is shown in the UML context diagram in Figure 2. The main component is the *DatabusManager* (DM) object which maintains the global status and monitors message loss. For simplicity, the user has separated the polling of the bus controller in a different object, the *ControllerMonitor*.

In order to verify the DM under the assumptions on message arrival and controller errors mentioned in § 2.2, what the OMEGA profile prescribes is a modeling of the environment using the same concepts as for modeling the system. In Figure 2 we see therefore three more objects corresponding to the altitude data source, the navigation data source and the bus controller.

As we will see in the following sections, the requirements concerning the DM can be achieved in several ways. The model in Figure 3 shows a possible realisation of the DM by a single state machine. Transitions are triggered either by signals coming from the data sources ($evAltDataMsg$, $evNavDataMsg$), or from the *ControllerMonitor* ($evControllerError$, $evControllerOK$), or finally by internal timeouts used to detect message loss ($altDataTimer$, $navDataTimer$).

Concerning the environment, that needs to be modeled as well, an important need is the ability to naturally model nondeterministic behavior. For example, Figure 4 shows the state machine of data sources, using interval conditions on clocks to model the nondeterminism due to the uncertainty on the starting time of the first period and due to possible jitter.

3.3 Expressing properties

Both, the functional and the reactivity properties to be verified on this model described in § 2.2 can be expressed by observers. Figure 5 shows the observer checking the upper bound $BR1$ guaranteed

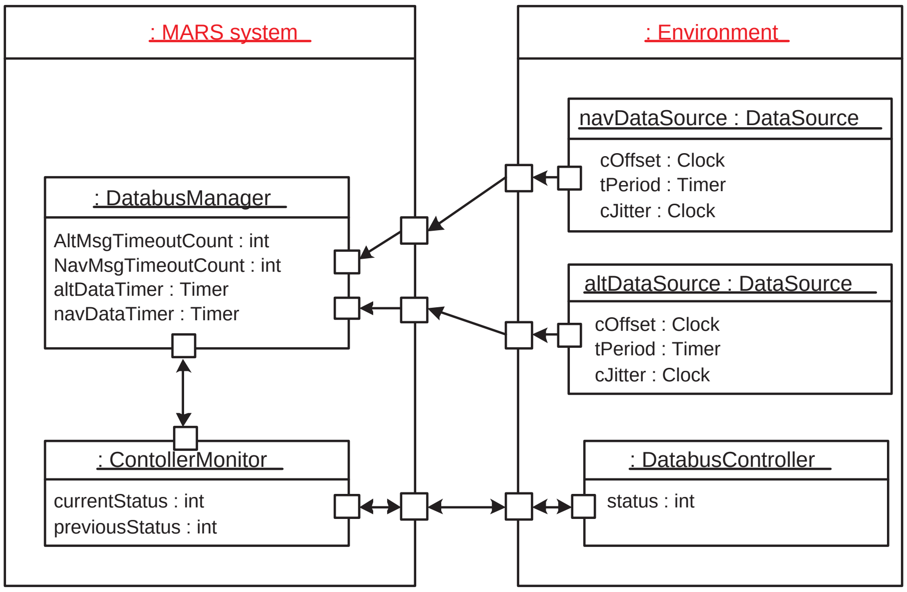

Fig. 2. Composite structure of the MARS model.

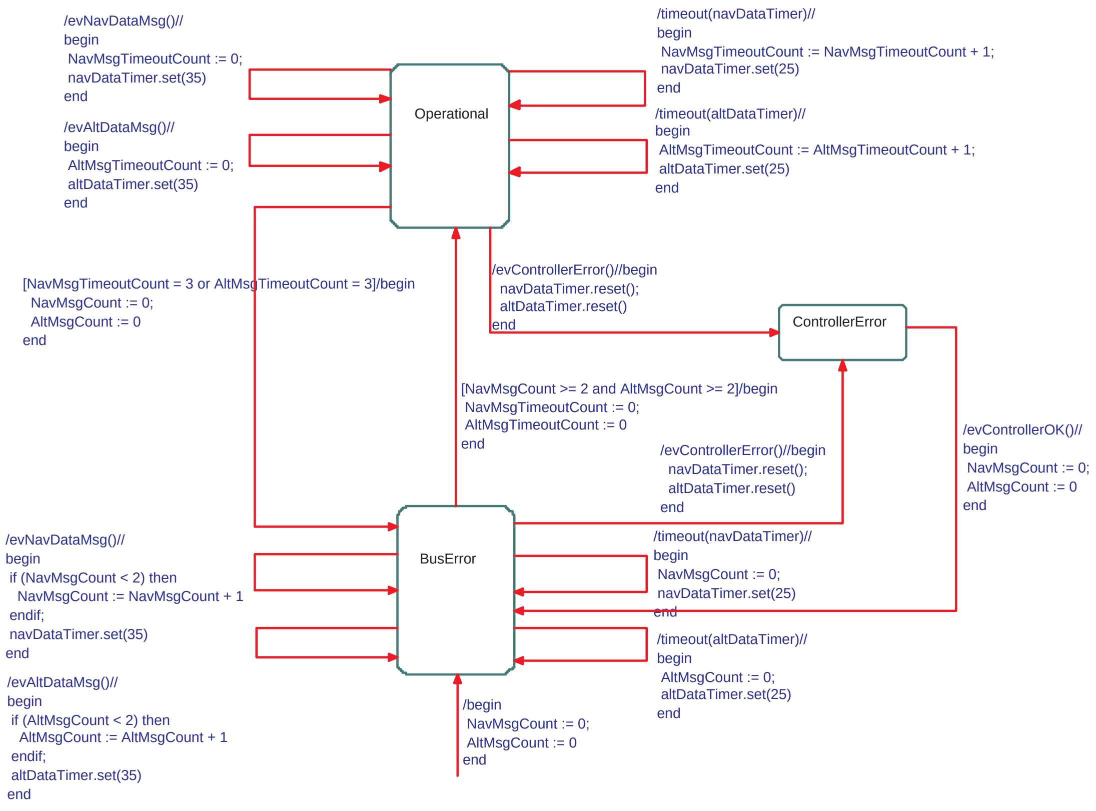

Fig. 3. State machine of the DatabusManager.

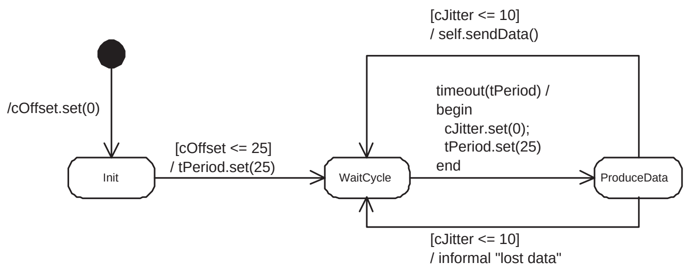

Fig. 4. Environment model: state machine of the data sources.

for the $R1$ value. 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.

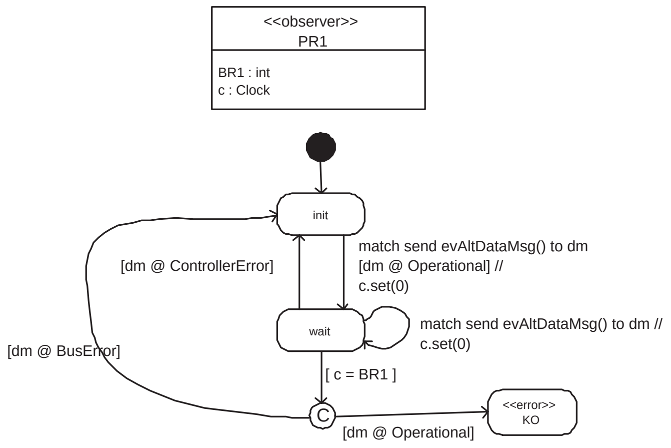

Fig. 5. Observer for verifying the reactivity bound for $R1$.

3.4 Initial results

The functional and reactivity properties have been verified against the model presented before. 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 de-synchronized. In order to fully verify the properties without this non-conservative assumption, a different model for the *DM* had to be designed, which is presented in the next section.

The functional properties have been proved to hold on the different *DM* models that we have considered. However, comparison has shown that very similar models may present different reactivity bounds. For example, consider the model for the *DM* in Figure 6 (the *ControllerError* state has been omitted).

The internal transitions of the *BusError* state are identical to those from Figure 3. The difference between the two models is that for each data source, in Figure 3 we count three short

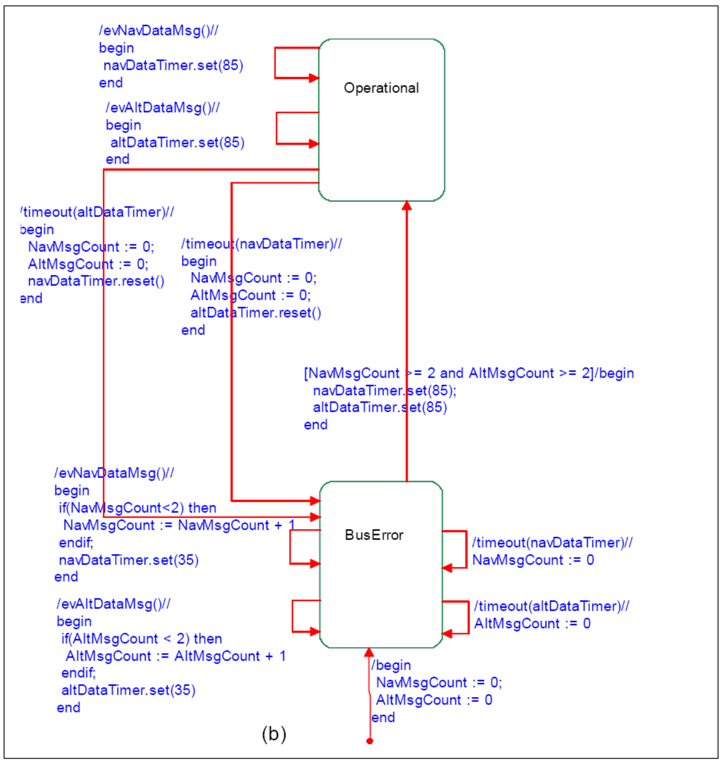

Fig. 6. An alternative model of the Databus Manager.

timeouts of 35, 25 et 25ms in state *Operational* in order to go in *BusError*, while in Figure 6 we use a long timer of 85ms which has to expire in order to go in *BusError*. Apparently, the latter is more efficient, as it makes less use of timers and no counting in state *Operational*.

By experiments, we have determined that the reactivity of the first model, as measured by the observer in Figure 5, is of $85ms$ while the reactivity of the second one is of only $95ms$. Although in the more realistic case of completely *de-synchronized* sources this property could not be checked (see also §3.5), analyzing the cause has shown that BR would have to be greater than $110ms$ in this case.

The diagnostic traces provided by the model-checker have shown that the difference 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 $85ms$. This is a simple mistake that a designer may do when refactoring the model in order to gain performance.

Using the model-checker on this abstract model easily exposed the loss of reactivity due to this “optimization”.

3.5 Compositionality, abstractions and further results

In order to fully verify 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 (see Figure 7):

– 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 3 reception windows. It sends out a signal *evCnt* with a numeric parameter of 3 bits, which represents the status of the last 3 reception windows (1 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 *Receivers*, maintains the global status. It goes from *BusError* state to *Operational* state when all *Receivers* have received correct messages during the last 2 reception windows. It goes from *Operational* state to *BusError* state when at least a *Receiver* has received no correct messages during the last 3 reception windows.

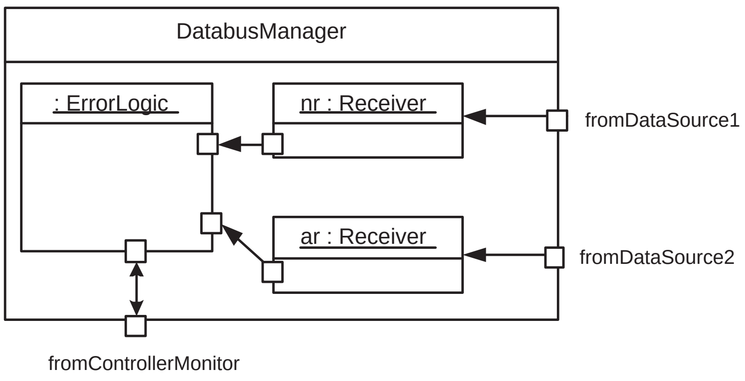

Fig. 7. Decomposition of the *DM*.

This model enables us to use an abstraction for verifying safety and reactivity properties in the general case of de-synchronized sources, using the following abstraction: we replace one data source and one *Receiver* with a chaotic abstraction "*ReceiverAbs*" which may send *evCnt* with any parameter value between 0 and 7 at any time. This is a very rough over-approximation of the source–receiver pair, but it proves to be sufficient for preserving the desired properties. The model is also interesting as it allows a generalization to a system with more than two data sources.

In order to further reduce the state space explosion, we also over-approximate the *Controller-Monitor* polling cycle (10ms in the initial model) by using a completely non-deterministic polling which may take place at any time. While this introduces new execution traces that are impossible in the initial model, the resulting state space is much smaller as, due to the use of a symbolic representation of time, many, previously distinct, states are represented by the same symbolic state.

To assess the efficiency of these abstractions, the table in Figure 8 below shows the size of the state space and the processing time for several configurations of the MARS system. For example, the last three lines show that the use of a (conservative) lazy abstraction for the *CM* polling cycle yields a model which is only 1.5 times bigger than the model with no polling at all (non-conservative abstraction), and more than 10 times smaller than the initial model which uses a 10ms polling cycle.

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	$1m21s$

Fig. 8. Verification times and state space sizes for different verification configurations.

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.

The use of the OMEGA UML profile in order to capture timing models and properties has facilitate a 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 like 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 over-approximation of the system's behavior, it is always conservative for the satisfaction of safety properties, including timed ones. The downside is that it can introduce false negative results. However, in the MARS example this has never occurred, showing that, by exercise, a designer can learn to use abstractions which do not break the verified properties.

We have also found out during the experiments that some methodological guidelines for writing observers and for using the IFx toolbox are necessary during the learning process. A set of guidelines has been developed as a side result of this teamwork (see also [OGL05]).

References

[BGM02] Marius Bozga, Susanne Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen*, number 2404 in LNCS. Springer Verlag, June 2002.

[BGO$^+$04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM-04:RT 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, LNCS, June 2004.

[CHR92] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[DJPV03] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

[DJPV05] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time uml semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 2005. (to appear).

[GOO03] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations in UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS 2003), a satellite event of UML 2003, San Francisco, October 2003*, October 2003. downloadable through http://www-verimag.imag.fr/EVENTS/SVERTS/.

[GOO05] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations in UML. *submitted to STTT, Int. Journal on Software Tools for Technology Transfer*, 2005.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[Obe01] Iulian Ober. *Specification and Validation of Timed Systems with Formal Description Languages*. PhD thesis, Institut National Polytechnique de Toulouse, September 2001.

[OGL05] Iulian Ober, Susanne Graf, and David Lessens. A case study in uml model-based dynamic validation: the ariane-5 launcher software. submitted, 2005.

[OGO05] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Accepted for publication in STTT, Int. Journal on Software Tools for Technology Transfer, 2004*, 2005.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.