Modeling and validation of a software architecture for the Ariane-5 launcher*

Iulian Ober[1], Susanne Graf[2], and David Lesens[3]

[1] Toulouse University, GRIMM/ISYCOM laboratory†
IUT-B 1 pl. Brassens BP 73, 31703 Blagnac, France
iulian.ober@imag.fr
[2] VERIMAG
2, av. de Vignate, 38610 Gières, France
susanne.graf@imag.fr
[3] EADS SPACE Transportation
66, route de Verneuil - BP 3002, 78133 Les Mureaux Cedex - France
david.lesens@space.eads.net

Abstract. We present the modeling and validation experiments performed with the IFx validation toolset and with the UML profile developed within the IST Omega project, on a representative space vehicle control system: a model of the Ariane-5 flight software obtained by manual reverse engineering. The goal of the study is to verify functional and scheduling-related requirements under different task architecture assumptions. The study is also a proof of concept for the UML-based validation technique proposed in IFx.

1 Introduction

Model-driven engineering is making its way through the habits of software designers and developers, pushed forward by the increasing maturity of modeling languages and tools. This paradigm promotes a complete re-foundation of software engineering activities on the basis of *models*, as well as the use of automatic tools for most post-design activities. In this context, the *software model* is the central artifact which gathers different aspects ranging from the requirements to software architecture, to component behavior, etc.

More recently, the trend is extending beyond software development activities, to *system* design. For this activity, which traditionally employed rather ad-hoc models, the community is currently seeking new formalisms, like SysML [19] or architecture description languages (ADLs). In the end, this adds new aspects (environment, hardware architecture, process and thread mappings, etc.) to the central artifact which becomes the *system model*.

The use of such heterogeneous models is justified by the complexity of current systems which have to satisfy tightly interwoven functional and non-functional requirements.

† Work performed while at VERIMAG.
* This work has been partially financed by the OMEGA IST project

In this paper we discuss the case of such a complex system, the control software of the Ariane-5 launcher, which is typical for the space vehicle control domain. Applications in this field typically involve a time driven part which implements the attitude and orbit control loop, and an asynchronous, event driven part, which performs mission management tasks. The different sub-systems share resources like busses and other spacecraft equipment.

The current practice, which consists in using cyclic sampling of asynchronous events and a Rate Monotonic Scheduling (RMS) policy [17], offers static criteria for deciding schedulability, and can offer correctness by construction (under some additional hypotheses) for other properties like exclusive access to shared resources. However, this policy proves to be very inflexible under demanding reactivity constraints or under processor overloads.

Consequently, more dynamic solutions are sought by system designers, like using fixed priority preemptive scheduling outside the sufficient (but not necessary) schedulability conditions of RMS. Such solutions rely on automatic verification methods, which have to take into account some functional aspects of the system.

In this paper, we describe a study in which the Omega UML profile [5, 10] (defined within the IST Omega project[1]) is used for modeling both functional and architectural aspects of a representative subset of the Ariane-5 system. We discuss the verification of both functional and scheduling-related requirements with the IFx toolset [18] which implements the profile.

1.1 A short introduction to Omega UML and the IFx tool

Omega UML is a profile targeting the design of real-time systems. The profile supports a large subset of the operational concepts of UML: classes, with most of their relationships (associations, composition, generalization), features (attributes, operations) and behavior descriptions (state machines). Actions, which are used to describe the effect of operations and of state machine transitions, are written in a syntax compliant with the UML action semantics. The language contains imperative constructs like assignments, operation calls, object creation, signal exchange, etc.

The description of concurrent systems is supported by means of *active* classes. Instances of active classes define a partition of the object space into *activity groups*, each group having its own thread of control, and functioning in run-to-completion steps. Communication is possible inside or between groups, by exchanging asynchronous signals and by calling operations. The execution model is an extension of the semantics implemented by the Rhapsody UML tool.

Detailed descriptions of Omega UML execution model can be found in [5]. On top of the concepts mentioned above, the Omega profile also defines a set of time-related constructs [10].

IFx[2] [18] is a toolset providing simulation and model-checking functionalities for Omega UML models. It is built on top of the IF environment [4], and provides

[1] `http://www-omega.imag.fr`
[2] `http://www-if.imag.fr/IFx`

a compiler of UML models to IF specifications. Models may be edited with any XMI-compatible editor[3].

Model checking is based on efficient forward state-space exploration methods for timed automata. Timed safety properties may be expressed as observers, which are described in the sequel. Generated diagnostic traces can be analyzed by simulation. In order to scale to complex models, IF supports optimization and abstraction in several ways: by "exact" static optimizations (like dead variable factorization and dead code elimination), by partial-order reduction, by data abstraction (static slicing). More details can be found in [18].

2 The Ariane-5 software

The Ariane-5 flight software controls the launcher's mission from lift-off to payload release. It operates in a completely autonomous mode and has to handle both external disturbances (e.g. wind) and different hardware failures that may occur during the flight.

This case study takes into account the most relevant points required for such an embedded application and focuses on the real time critical behavior by abstracting from complex functionality (like control algorithms) and implementation details, such as specific hardware and operating system dependencies. Nevertheless, it is fully representative of an operational space system. The typical characteristic of such systems is that they implement two kinds of behavior:

- *Cyclic synchronous algorithms.* These are principally the control/command algorithms (in the sequel they are called GNC for Guidance, Navigation and Control). The algorithms and their reactivity constraints are defined by the control engineers based on discretization of continuous physical laws.
- *Aperiodic, event driven algorithms.* These algorithms manage the mission phases and perform particular tasks when the spacecraft changes from one permanent mode to another (engine ignition and stop, stage release, etc.), or when hardware failures occur (alternative or abortion manoeuvres).

The software components implementing this functionality are physically deployed on a single processor[4] and share a common bus for acquiring sensor data and sending commands to the equipment.

The proof of correctness of the mission management components can be made by (almost completely) abstracting from the control algorithms. In an earlier experiment, we have used an SDL model of the mission management in order to verify this kind of properties [3].

The correctness of control algorithms concerns two issues: their numerical computation, and their concurrency behavior. The numerical correctness is not considered here. For the concurrency, the proof of correctness is usually done using the synchrony hypothesis. The synchronous approach makes verification using a non-timed semantics much simpler. The non-timed semantics just assumes that all entries are available when the computation cycle starts and the

[3] Rational Rose and I-Logix Rhapsody have been tested used in the OMEGA project
[4] In fact, a set of replicated processors, but this is out of the scope of our case study

results of the computation are made available at the end of each cycle. There exist results stating sufficient conditions under which such a synchronous design can be implemented in a distributed and/or multi-threaded environment [20]. However, in this case study the sufficient conditions do not hold in all cases, and the satisfaction of the synchrony hypothesis must be verified.

Therefore, in this case study we have considered more particularly the problem of verifying that a low-level software architecture (a task model), together with a set of other non-functional assumptions (worst case execution times, arrival model), satisfy the reactivity constraints imposed on the software and ensure the synchrony hypothesis for the cyclic algorithms.

The current practice for ensuring such non-functional constraints consists in using an RMS-based scheduler. Asynchronous events are sampled with the frequency of the smallest cycle. The schedulability of this architecture can be decided statically, under the assumption that relevant values for WCET of tasks can be provided. Moreover, in the current solution, the exchange of data between the different tasks or between a functional task and the bus are allowed only in predefined time slots at the beginning and at the end of each task's cycle (even if the computation finishes earlier in the cycle). Consequently, mutual exclusion between writes in the exchange memory is satisfied by construction, and with no possibility for priority inversion.

On the other hand, this architecture is very inflexible in the following circumstances:

– when some acyclic events need shorter reaction time than the basic cycle (which is in fact the case for the Ariane-5 system),
– when some cyclic algorithm needs more recent measurement data, that has to be acquired during the cycle (also the case in the Ariane-5 system),
– when an algorithm needs a longer time than the pre-assigned slot, for instance in case of high CPU load (this feature, not required today, will become mandatory for future highly autonomous space systems).

In the case of Ariane-5, the software designers have used a more flexible architecture, which is still based on fixed priority preemptive scheduling, but which violates some of the RMS assumptions mentioned above in order to ensure better reactivity (reads and writes during the cycle, triggering of asynchronous code during a cycle). Nevertheless, such an architecture has to be formally validated with respect to the non-functional requirements concerning timing, scheduling and mutual exclusion. This is the main objective of our case study.

3 The verification model

The UML model used for verification has been built manually from the existing software code, by a team from EADS Space Transportation. Its functional decomposition is independent of the task architecture; it is structured around 6 objects implementing the main categories of functionality, each defined by a singleton active class. They are:

- *Acyclic*: the main mission management object, which handles the start of the flight sequence and the switching from one phase to another. Its behavior is described by a state machine reacting to event receptions from the GNC algorithms (e.g., end of thrust detection) or from the environment, and to time conditions (e.g., time window protections ensuring that the treatment associated to an external event is performed within a predefined time window even in case of failure of the event detection mechanism).
- A set of specific objects which handle the acyclic management activities related to a particular launcher stage. They react to events received from *Acyclic* or to internal time constraints. In the study, we considered only two stages: *EAP* (lateral booster) and *EPC* (main stage of the Ariane 5 launcher).
- *Cyclics*: This object manages the activation of the cyclic control/command algorithms (GNC). The algorithms are executed in a predefined order, depending on the current state of the launcher, which is tracked by the *Acyclic* class. Its state machine appears in an example later on in Fig. 3. We consider in more detail two of the algorithms activated by *Cyclics*, each implemented by a separate object: *Thrust_Monitor*, responsible for the monitoring of the *EAP* thrust, and *Guidance_Task*, which has the particularity that its activation frequency is lower than that of the other GNC algorithms.

In order to validate the software, a part of the environment needs to be modeled. In our case, it includes two kinds of spacecraft equipment – *Valves* and *Pyrotechnic commands* (the model includes possible hardware failures), the external environment – namely the ground control centre, as well as abstractions of parts of the software which are not described in the model (such as: a numerical algorithm or the 1553MIL bus allowing the communication between the main software and the equipment).

3.1 Capturing functional and timing requirements

Using Omega UML, requirements can be formalized by means of *observers*, and verified against the design model. In this section, we discuss briefly the concepts and we give an example of how they are put to work in the Ariane-5 model. More detail on observers can be found in [18].

Observers are special objects which monitor the execution of the model and give verdicts when a requirement is satisfied or violated. Observers may have their own local memory (attributes), and their behavior is described by a state machine, in which some states are labeled with the stereotypes <<`success`>> or <<`error`>> providing verdicts. The monitoring of model execution is done by observing events like signal outputs, operation calls or returns, state changes, etc., or by observing the state of the system, like attribute values, contents of queues, states of the state machines, etc. [5]

[5] A formal discussion of the expressivity of observers is out of scope. We note that: (1) observers are used to express linear timed *safety* properties, which may combine state or event-based atomic propositions, and (2) observers embed general algorithms,

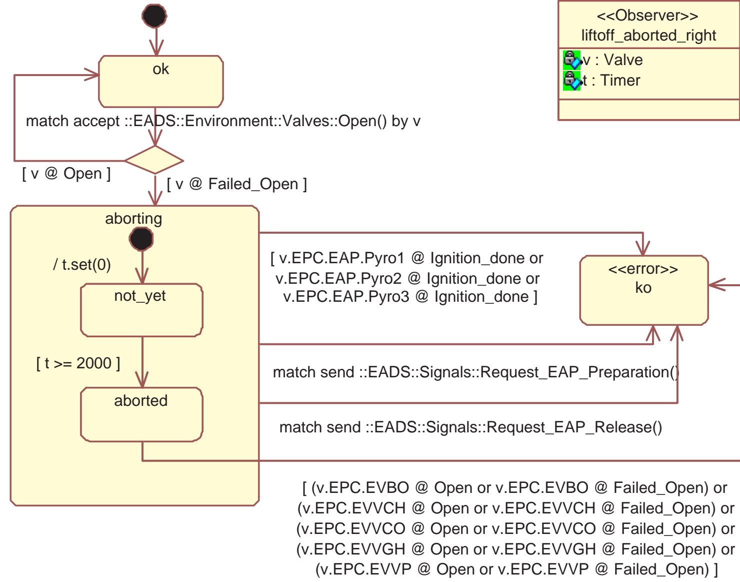

Fig. 1. Property p1.

We take for example the following property:

Property P1. *The launcher shall not lift-off if an anomaly is detected during the Vulcain engine ignition. In case of lift-off abort, the valves shall all be closed within 2 seconds and the pyrotechnic commands shall not be ignited.*

An anomaly on the Vulcain ignition corresponds, in our modeling of the environment, to a *Valve* object entering the *Failed_Open* state. This failure shall be detected by the software, which shall then abort the lift-off and secure the launcher. Thus, this property is expressed more precisely as follows:

If any instance of the *Valve* class enters one of the states *Failed_Open* or *Failed_Close*, then:

- All the instances of the *Pyro* class shall never enter the state *Ignition_done*.
- 2 seconds after the valve failure, all instances of the *Valve* class shall be in state *Close* or *Failed_Close*, and then remain in this state forever.

This property is based on a pure black-box view of the software. Nevertheless, since several components are involved in aborting the lift-off, the designers have completed the property with the requirement that the internal events *Request_EAP_Preparation* and *Request_EAP_Release* are never emitted.

therefore their termination (hence also their satisfaction) is undecidable in general. For practical applications, we do not view this as a limitation.

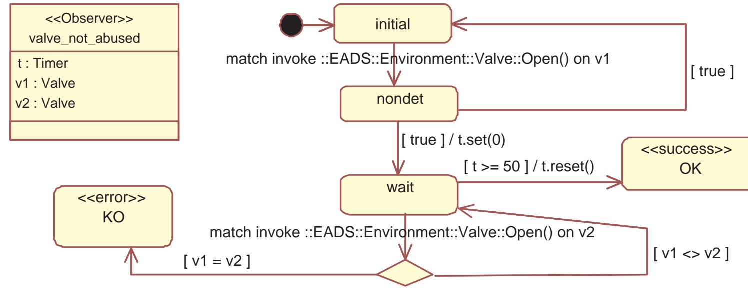

Fig. 2. Property p2.

Fig. 1 shows how this property can be expressed using a timed observer: each time an *Open* command is received by some valve v, the observer tests whether v reaches the state *Failed_Open*.

If this premise holds, the observer enters state *aborted*, in which *Pyro* objects entering state *Ignition_done*, as well as emissions of the signals *Request_EAP_Preparation* and *Request_EAP_Release* are prohibited. After 2 seconds from entering state *aborting*, the observer goes to the inner state *aborted* in which, additionally, *Valves* entering the state *Open* or *Failed_Open* are also prohibited.

Note that the testing of the premise done by the observer corresponds to the universal quantification appearing textually in the premise of the property $P1$ ("any instance of the *Valve* class"). Such a universal quantification would also have to be used in a state logic formula, had we used a temporal logic for formalizing the properties. However, to the best of our knowledge, no major model checking tool based on LTL or CTL supports directly this kind of first order logic in the specification of state formulas. For example, in UPPAAL [14], the same property could be expressed either by using a helper (observer) automaton which synchronizes with any *Valve* entering *Ignition_done*, or by using quantifier elimination (that only works under the restriction that the set of objects is known in advance – which is generally not true in Omega UML or IF specifications). This supports our claim that observers are a flexible formalism for expressing common event or state-related timed safety properties.

Consider another property, required by electrical constraints on the hardware:
Property P2. *The software shall not send two Open commands to the same valve at less than 50ms of interval.*

$P2$ constrains the distance between pairs of events concerning a same instance of class *Valve*. The particularity of this property is that it concerns any consecutive pair in the series of commands sent by the software to a (any) *Valve*. This kind of property is typically easier to specify with a temporal logic.

Nevertheless, we show in Fig. 2 an observer which uses non-determinism to pick each particular occurrence of an event pair at a time and either verify the time distance, or skip to the next one (as the model checker explores all alterna-

tives, all pairs are eventually verified)[6]. In state *initial*, the observer waits for a command to be sent to any *Valve*, stores the reference of the concerned *Valve* in $v1$ and proceeds to state *nondet*. (This behavior ensures universal quantification over the set of valves, in the same way as in $P1$). In state *nondet*, it chooses non-deterministically whether to proceed by verifying the timing of the next command sent to the same valve, or to return to *initial* and wait for another command to (any) *Valve*.

The rest of the observer tests a simple timed safety condition: the next command sent to the *Valve* $v1$ does not come before 50ms. The clock t is used to measure 50ms. In state *wait*, other commands may come, but they cause an *error* only if they concern the valve $v1$. If more than 50ms elapse without error, the observer reaches a success state and considers the property verified for this particular events occurrence pair.

Stereotyping the *OK* state with <<`success`>> allows also to make model checking more efficient: after the observer has reached *OK*, the execution of the (system, observer) pair cannot lead to *KO* anymore, and may safely be ignored.

3.2 Scheduling constraints and objectives

As mentioned before, the main focus in this study is on validating a particular task architecture based on fixed priority preemptive scheduling, in order to check that is satisfies several conditions concerning schedulability and mutually exclusive access to the bus.

In the architecture that we consider, the hypotheses of RMS [17] are not satisfied, as asynchronous events need to be handled as soon as they arrive. Another difficulty in using RMS-like schedulability decision criteria is that the execution time of the cyclic tasks varies a lot depending on the current flight phase. Fig. 3 shows the state machine of the control cycle, on which we can see that the worst case execution time of this cycle is around 64ms, while the best case is 37ms and the average measured by simulation is around 42ms. One cannot simply consider at each cycle the worst case execution time (of sporadic and cyclic tasks), as this would lead to a huge over-approximation of resource occupation and to the conclusion that the system is non schedulable. We also relax in some cases the requirement that reads and writes are done only in the beginning and in the end of each task's cycle. Therefore, the access to the bus is not mutually exclusive by construction.

The technique we adopt for proving these constraints is to take into account the functional behavior of the system and its impact on resource consumption.

Assigning priority levels to activities. The priorities were assigned just as in the RMS solution from which we started, according to the relative responsiveness required from an activity. Three levels of priority are used:

– Functions of the Regulation components have highest priority. They are sporadic and take about 2 to 5 ms each time a command is executed (open a valve, ignite a pyrotechnic command, etc.)

[6] The observer presented here is not optimal in the required verification time/memory.

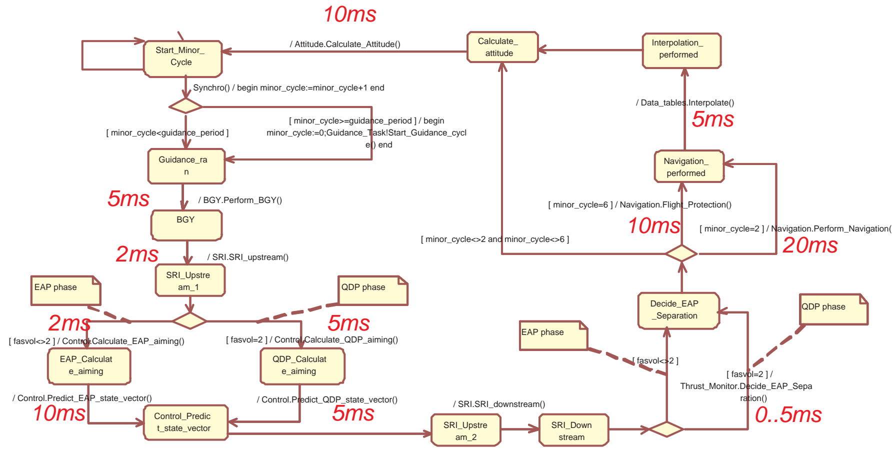

Fig. 3. Statechart of the Control cycle with unitary execution times.

– Functions of the Navigation-Control components have medium priority. They are periodic, with a period of 72ms and take 37 to 64ms to execute depending on the current phase of the flight and other parameters.
– Functions of the Guidance components have lowest priority. They execute every 576ms. One of the goals of scheduling analysis was to determine how much processor time they can take in each cycle in order for the system to remain schedulable.

Activities that are on the same priority level are handled by the same runtime task, that is without overlapping.

Modeling the task architecture in Omega UML. Scheduling policies and resource consumption can be modeled using the lower level constructs of the Omega profile: objects which manipulate clocks and do the resource bookkeeping. In parallel with the Ariane-5 study, we developed a reusable model library for the IFx tool, which provides support for different types of schedulers.

The *Scheduling* library contains basically two kinds of classes organized in two hierarchies:

– *Task* classes used to annotate the user model with requests for execution time. Requests are parameterized with a *duration*, and depending on the scheduling policy, with information like *priority*, *deadline*, etc. Instances of *Task* classes can be shared by several objects.
– *Scheduler* classes are used to model the different scheduling policies. Each created *Task* has to be associated with a *Scheduler*. Subsequently, every time a *Task* requires processing time, it will communicate with its *Scheduler* in order to determine the actual time of finish, based on the task duration and on the state of the *Scheduler* (i.e. the scheduling policy and the charge at that moment).

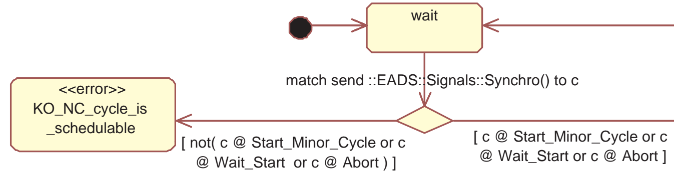

Fig. 4. Scheduling objective: the control cycle finishes in time.

For modeling the behavior of the fixed priority preemptive scheduler in timed automata constructs, we use the scheme proposed in [9].

Scheduling objectives are modeled by observers. They are:

- The Navigation-Control (NC) functions must terminate within the 72ms cycle and the Guidance functions within the 576ms cycle.
  For the NC functions, this property is formalized in the observer in Fig. 4, by the fact that the *Cyclics* component receives the signal *Synchro*, which signifies the beginning of a cycle, only in the states *Start_Minor_Cycle*, *Wait_Start* or *Abort*. If a cycle does not finish in time, the *Cyclics* component is in an intermediate computation state when the next *Synchro* is received and this property is violated.
  The observer expressing the analogous property for the Guidance function is similar.
- The application uses a 1553 MIL bus. In this protocol, all data transfers are performed under the supervision of a bus controller (the main on-board computer in the case of the Ariane 5 case study). The software components read and write data in an exchange memory which is transferred via the bus to the equipment (also called remote terminal) at specific time frames (this process is called low-level transfer). The consistency condition for bus reads and writes is that the software components do not read or write the bus during the low-level transfer time frames. (calls to *read* and *write* operations do not occur while the *Bus* is in *Transfer* state).

4 Ariane-5 verification results

4.1 Validation methodology

In the context of the IFx toolset, the validation of UML models means performing several activities, which range from simple syntactic and static semantic checking to dynamic property verification, with the goal of improving the model and its conformance to its requirements. These activities are supported by different tools. The standard workflow used also in this case study is summarized below[7].

[7] A more detailed description can be found in [4]

1. The *translation phase* consists in invoking the *uml2if* compiler. Standard static checks are performed (name and type checks, checking of well formedness constraints).
2. The *simplification phase* consists in the application of static analysis and abstraction methods implemented in IF:
   - in the early validation phases we use mainly methods fully preserving verification results (such as dead variable / dead code analysis and clock reduction)
   - in the later phases (verification), we use in addition methods leading to over approximations such as abstractions of variables or clocks, or relaxation of *urgency* constraints.
3. The *simulation phase* consists in exploring the model by a mixture of interactive, guided and random simulation which allows usual debugging tasks like saving and reloading a played scenario, stepping back and forward through it, inspecting the system state, inserting conditional breakpoints, etc.
4. The *model-checking phase* is the main validation phase, in which the product space of the relevant part of the model and of a set of observers is searched for absence of *error* states, while avoiding the parts of the graph reachable only via *success* states.

   In this phase, there are 2 possibilities for handling time: discrete or symbolic. With discrete time, time progress is represented by a *tick* transition common to all processes, and this representation allows the use of more expressive time constraints. In case of the symbolic representation of time, a DBM is associated with each system state, like in the timed-automata based tools Kronos [22] and Uppaal [14]. The symbolic representation leads in most examples to much smaller state spaces.
5. the IF toolset implements a number of other verification techniques. The most interesting ones are comparison of models and minimization of models with respect to simulations and bisimulations. Minimization has been used in our case study to extract most general properties with respect to an observation criterion, given by a set of observable events (see in [3, 4]).

4.2 State explosion and use of abstractions in Ariane-5

The duration of a basic cycle of the cyclic behavior of the Ariane-5 flight software is 72 ms. Each basic cycle contains several hundreds of steps. As the acyclic behavior uses some timers also to measure long durations, when composed with the cyclic behavior, every state reached through these steps is a new global system state. This quickly leads to an explosion, especially in the case of Ariane-5, where the footprint of a system state is quite large (see also §4.3).

In order to cope with the complexity of the model, we had to apply more evolved abstraction and reduction techniques which need a good understanding of both the functioning of the system and the verification and abstraction technology.

Compositional Abstraction. We have applied this well known technique which consists in the verification of properties of a subsystem, by replacing the other parts of the system — which play here the role of an environment — by a simpler descriptions representing an abstraction. The variable abstractions implemented in IF were not sufficient for the Ariane-5 model and we have built manual abstractions, which were still relatively simple, by using the existing decomposition of the system into a cyclic and an acyclic part and the clear interface between them.

To illustrate this, we take the example of the safety properties related only to the acyclic part (flight program and error handling). To prove their correctness, the cyclic GNC part has been abstracted by eliminating all the internal behavior and by sending messages (flight phase change commands) at arbitrary moments rather than at the precise time points computed by the concrete GNC. This represents clearly an abstraction and it was sufficient to show the satisfaction of all the properties of the asynchronous part (see [18, 3] for an older experience concerning this part). Note also that such an abstraction can in principle be constructed automatically.

Reduction of the duration of the flight phases. In order to validate the properties related to schedulability and concurrent bus access, we have used an alternative reduction without behavioral abstraction. As mentioned before, a huge source of state explosion is the difference of the time scale between the asynchronous and the cyclic behavior.

Asynchronous events are *rare*, and the system is working without occurrence of any asynchronous events during a large number of basic cycles (called *stable phases*). Moreover, most of the output of the cyclic part is irrelevant for the timing properties to be verified. Thus, it is sufficient to perform the proof on a functional abstraction of the cyclic part with a mission duration much greater than the basic cycle, but much shorter than the real mission duration.

In stable phases, all executions of the basic cycle in the cyclic part are identical with respect to the properties to be verified, in particular to the schedulability of all tasks in all relevant cycles, and to the observation of a certain time window for the commands sent from the synchronous to the asynchronous part (stable phases are outside this time window).

This suggests that it is sufficient to verify a reduced model, obtained by a drastic reduction of the overall flight duration, being careful to make sure that only stable phases are shortened, whereas all the critical transition phases are fully explored. The transition phases are defined by the flight phases defined in the acyclic part and by the occurrence of exception events. Exception events can occur at any time, but the correctness of the software must only be guaranteed for 2 exceptions for the entire flight, which means that it is enough to make the stable phases long enough to allow the occurrence of 2 exceptions with subsequent stabilization.

Using such a reduction of the real duration of the mission, the reachable state space for the entire flight could be explored, and all the properties could be finally validated.

4.3 Results and figures

In this section, we show the efficiency of the applied reductions. The table in Fig. 5 shows the verification time and the cardinality of the state space, using dead variable and partial order reductions, while using different mission durations (but always respecting the required stabilization times).

For the comparison, we have used a model with all the properties (observers) enabled simultaneously. There is no state explosion caused by the parallel composition of all properties, since properties are not completely independent.

A discussion is necessary as the figures presented here may seem low compared to other known examples in explicit state or symbolic model checking, which range beyond 10^7 states. One must consider the following:

- The Ariane-5 model (after UML translation) consists of 77 types of IF processes, each having (many) variables of complex types, and sometimes having dynamically created instances. The footprint of the system state is slightly variable, with an average of 10KB.
  In our view, one cannot compare the 10^6 order of magnitude of Ariane-5 (for the largest exploration, shown in Fig. 5), with results obtained on other systems, which may have higher combinatorics but smaller footprints. It is unfortunately impossible to propose this model as a basis for benchmarking, due to confidentiality reasons related to its industrial nature.
- Given that the (approximately) 6GB of Ariane's 658981 states have been explored on a machine with only 1GB of RAM, we see this as evidence of the efficiency of the sharing algorithms [4] of the IF exploration platform.
- Another characteristics of the state spaces obtained here is that they are very narrow and deep (almost a vertical string). This is not because the example is sequential by nature: concurrency is present in this model and the combinatorics is potentially very big. The linear form of the state spaces indicates the efficiency of the partial order reduction of the IF platform.

5 Comparison to other approaches, discussion and future work

Discussion of related work There exist already a number of tools proposed for the validation of UML models by translating a subset of UML into the input

Mission duration	Nr of states	Nr of transitions	Verif. time (min)
7 s	51 324	54 697	03:30
15 s	161 956	171 734	12:06
22 s	303 496	321 206	11:33
30 s	463 932	490 901	22:58
37 s	658 981	696 031	34:53

Fig. 5. Complexity for different mission durations (all properties combined)

language of some existing validation tool [16, 15, 13, 7, 21, 6, 2, 1] to mention only a 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 [12] (in [16, 15]) or COSPAN [11] (in [21] for non-timed systems, and Kronos [22] (in [2]) or Uppaal [14] (in [13, 6]) for the verification of systems with timing constraints. Also the translation into proof-based frameworks, such as PVS (e.g. in [1]) or B, has been proposed.

With respect to the *expressivity of the UML profile* accepted, the IFx framework goes beyond other existing ones, as it handles a rich subset of UML, including inheritance and dynamic object creation and powerful timing features. Most of the cited UML validation tools are restricted to static systems, fitting exactly the model of the underlying model-checker. Also, they usually handle properties written in the property language proposed by the underlying model-checker. The Omega UML profile proposes *observers* for this purpose.

The IFx tool does not push forward the *theoretical* boundaries of existing verification technology. However, the tool 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, the bisimulation based reduction techniques of Aldebaran [8], and adds verification-targeted optimizations based on static analysis, as well as support for industry-backed standards like SDL and UML.

Experience showed that the combination of these techniques allows to obtain feedback very rapidly on most models without much remodeling and adaptation effort by the user. Positive verification results required, for the bigger examples, some effort to find an appropriate property preserving abstraction and to apply it manually – which is a common limitation of major model-checking tools.

The model structuring concepts present in IF allow to limit the overhead induced by the translation a rich user level formalism like UML, and also make the translation more flexible. Consequently, we plan on moving towards UML 2.0 and to system-oriented formalisms like AADL, which are better suited for modeling architectural and non-functional problems in the space vehicle control domain.

References

[1] T. Arons, J. Hooman, H. Kugler, A. Pnueli, and M. van der Zwaag. Deductive verification of UML models in TLPVS. In *Proceedings UML 2004*, pages 335–349. LNCS 3273, 2004.

[2] Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. Model checking UML specifications of real time software. In *Proceedings of 8th International Conference on Engineering of Complex Computer Systems*. IEEE, 2002.

[3] M. Bozga, D. Lesens, and L. Mounier. Model-Checking Ariane-5 Flight Program. In *Proceedings of FMICS'01 (Paris, France)*, pages 211–227. INRIA, 2001.

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. 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.

[5] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Proc. of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*

[6] A. David, O. Möller, and W. Yi. Formal verification UML statecharts with real time extensions. In *Proceedings of FASE 2002 (ETAPS 2002)*, vol. 2306 of *LNCS*. Springer-Verlag, April 2002.

[7] M. del Mar Gallardo, P. Merino, and E. Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, August 2002.

[8] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Computer Aided Verification, 8th Int. Conf. CAV '96*, vol. 1102 of *LNCS*, 1996.

[9] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis using two clocks. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, 2003.

[10] S. Graf, I. Ober, and I. Ober. Timed annotations in UML. *Int. Journal on Software Tools for Technology Transfer*, Springer Verlag, 2006. (In print. Available on Springer On-line at http://dx.doi.org/10.1007/s10009-005-0219-x).

[11] Z. Har'El and R. P. Kurshan. Software for Analysis of Coordination. In *Conference on System Science Engineering*. Pergamon Press, 1988.

[12] G. J. Holzmann. The model-checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 1999.

[13] A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, volume 2469 of *LNCS*, September 2002.

[14] K.G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status & Developments. In O. Grumberg, editor, *Proceedings of CAV'97 (Haifa, Israel)*, volume 1254 of *LNCS*, pages 456–459. Springer, June 1997.

[15] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioral subset of UML statechart diagrams using the SPiN model-checker. *Formal Aspects of Computing*, (11), 1999.

[16] J. Lilius and I.P. Paltor. Formalizing UML state machines for model checking. In Rumpe France, editor, *Proceedings of UML'1999*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[17] C. L. Liu and J. W. Leyland. Scheduling algorithms for multiprogramming in a hard real-time environment,. *JACM*, 20(1):46–61, 1973.

[18] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Int. Journal on Software Tools for Technology Transfer*, Springer Verlag, 2006. (In print. Available on Springer On-line at http://dx.doi.org/10.1007/s10009-005-0205-x).

[19] SysML Partners. SysML specification v. 0.9 draft (10 jan. 2005). Available at `http://www.sysml.org/artifacts.htm`.

[20] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Formal Methods in System Design 2005*, LNCS. Springer Verlag, 2005.

[21] Fei Xie, Vladimir Levin, and James C. Browne. Model checking for an executable subset of UML. In *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01)*. IEEE, 2001.

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