A case study in UML model-based dynamic validation: the Ariane-5 launcher software⋆

Iulian Ober[1]**, Susanne Graf**[1]**, David Lesens**[2]

[1] VERIMAG
   2, av. de Vignate
   38610 Gières, France
   e-mail: {iulian.ober,susanne.graf}@imag.fr
[2] EADS SPACE Transportation
   66, route de Verneuil - BP 3002
   78133 Les Mureaux Cedex - France
   e-mail: david.lesens@space.eads.net

The date of receipt and acceptance will be inserted by the editor

Abstract   The aim of this paper is to show the usefulness of IFx, an extension of the IF validation toolset for timed and complex systems, for the validation of designs represented using the UML profile defined in the IST Omega project.

This demonstration is done on hand of a realistic case study, a model of the Ariane-5 launcher software. In this case study, we do not address the correctness of the control algorithms, which may be specified as global transactions and validated with the help of synchronous tools such as Scade. We are rather interested in the expression of non functional properties which allow to guarantee the correctness of this transactional view and their validation on a model of the distributed architecture on which the system is deployed.

We provide examples of complex properties and their expression by UML observer state-machines, where special attention is paid to the modelling of properties expressing time constraints and schedulability. Also, on hand of this example, a validation methodology is demonstrated which has been experimented already in a number of real-life examples.

This case study is representative for systems in the aerospace domain concerning the nature and complexity of the handled verification problems.

Code generation form the developed software model was not part of this case study, but the obtained model is such that it can be refined into a complete model - including also all functional aspects - from which existing code generation techniques may be applied.

1 Introduction

Model-driven engineering is making its way through the habits of software and system 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 if not all post-design activities like getting a platform-specific implementation, generating and executing tests, deployment, etc.

In this context, the model of a software system (or of a system in general) gathers different views ranging from the *system requirements* (in the form of use cases, of static or dynamic properties that the system has to satisfy, etc.), to *architecture*, to *behavior* of individual components and/or subsystems, to *platform-related information* (resources and their utilization), etc. Since the model is central to the whole development process, it is essential for its designers and its users to be able to check its correctness and coherence early in the development cycle (i.e., before the availability of an implementation on the target architecture and in the real environment) by providing models of the functional and non functional aspects of software, architecture and environment.

In the Omega project[1], we have developed a UML profile adapted for modelling real-time and embedded systems [DJPV03,DJPV05,GOO03,GOO05] and extended the IF validation toolset for real-time systems [BFG+99,BGM02] for handling UML models, resulting in the tool IFx [OGO05]. IFx allows *simulating* an operational UML system specification and *verifying* complex behavioral properties as defined by the Omega UML profile.

---
[1] http://www-omega.imag.fr

The Omega UML profile emphasizes operational models which can be further refined into complete models from which code can be generated. This profile represents an extension of the UML profile of Rhapsody[2] with real time features, in particular with the notion of timed observers for the expression of requirements.

The Omega UML profile has been used in several case studies by industrial users for modeling real-time and embedded systems, and the IFx toolset has been used for validating coordination and real-time aspects[3].

In this paper, we focus on the largest of these case studies: a model obtained by reverse-engineering of a representative part of the flight software of the Ariane-5 launcher[4].

The aim of the paper is to show on hand of the chosen case study the usefulness of the IFx toolset for the validation of properties and design models provided in the the form of UML models conforming to the Omega profile.

Section 2 provides a short overview on the Omega UML profile and the IF validation toolset. *Comment by Susanne: oui je suis d'accord il faut inverser; il faut aussi surtout racourcir ou ici ou dans la methodologie; car actuellement on raconte l'histoire deux fois.* In section 3, we describe the the problem to be solved and discuss the modeling and specification features of the Omega UML profile used. Section 4 discusses the expression of requirements concerning control-flow and timing related aspects and section 5 discusses the IFx tool workflow and the validation results for Ariane-5. In section 7, we show how our approach relates to other approaches and finally, in section 8, we discuss some methodological issues, on how results were obtained, what problems can be encountered and how they can be solved, and what has to be done in order to really integrate the validation approach into the development process.

2 Preliminaries on the UML profile and the IFx toolset

In this section, we present the toolset and the UML profile used for carrying out the Ariane-5 case-study.

Detailed descriptions of Omega UML profile can be found in [DJPV03,DJPV05] for the general profile and in [GOO03,GOO05] concerning the real-time aspects. In section 2.1, we introduce only the features needed for the example where the main emphasis is on the expression of timing related requirements.

The IFx validation toolset is described briefly in section 2.2, and a more detailed description is available in [OGO05].

2.1 UML features supported by IFx and semantics

The IFx toolset supports the constructs and the particular semantics of the Omega UML profile [OGO05, GOO05,DJPV03]. This section provides a small digest of the features of this profile, necessary to understand the model of the Ariane-5 case study.

The architecture and the behavior of a system are described using class diagrams, state diagrams and operation specifications. Class diagrams may use most of the concepts available in UML, such as: attributes, different types of associations and generalization relationships.

State diagrams may be used to define the dynamic behaviour of reactive classes; objects react either to asynchronous signals received from other objects, to conditions (change events) or to operation calls. Operation calls which are handled by the state machine associated with an object are called *triggered* operations in Omega UML. The behavior associated with an operation that is not *triggered* (called *primitive* operation) is described by an action. Actions are written in a syntax compliant with the UML action semantics, containing imperative constructs like assignments, operation calls, object creation, signal exchange, etc.

The profile supports the description of concurrent and distributed systems by means of *active* classes. Instances of active classes define a partition of the system objects; an active object together with its dependent passive objects are called an *activity group*. Each activity group has exactly one thread of control and handles requests (operation calls and signals) coming from the other groups in a FIFO run-to-completion manner. *Comment: [susanne: ca c'est pas vrai, la concurrence existe meme sans signaux, simplement parce que chaque class active a initialement un thread] Thus, concurrency is created by non-blocking requests (signals or operations which do not send a return value) between activity groups.*

This execution model is presented in more detail together with its motivations in [OGO05,DJPV03]. It corresponds to a particular choice of semantics in the spectrum allowed by the UML standard [OMG03], and is an extension of the execution model implemented by the Rhapsody tool.

On top of the concepts mentioned above, the Omega profile defines a set of time-related constructs [GOO05]. There are basic concepts like *timers* and *clocks* for describing time-driven behavior in an imperative style, as well as a mechanism for defining *duration constraints* which are *declarative* assumptions or requirements about how system execution relates to time passing. Requirements to be validated are expressed by UML state ma-

---

chines, with stereotype <<observer>> which are interpreted as acceptors of safety properties. We use observers mainly of the expression of timing properties (see section 4).

2.2 The IFx toolset

IFx [OGO05] is a toolset providing simulation and verification functionalities for UML models containing detailed, operational designs. IFx implements the UML features and their semantics defined in the Omega UML profile. This profile is targeted to the designers of real-time reactive and distributed embedded systems. For this purpose it includes extensions for expressing timing and concurrency related information.

2.2.1 Toolset architecture and functionality The architecture of IFx is shown in Figure 1. The toolset reuses state-of-the art validation techniques from the IF environment [BGM02, BGO+04]. It enables the use of UML models through a compiler that transforms them to IF specifications. Models may be edited with any XMI-compatible editor such as Rational Rose or I-Logix Rhapsody. The simulation and verification functionality of IF is wrapped by a UML-specific interface which hides most of the details of the IF format and tools from the user.

The functionalities of IF which are wrapped and provided at UML level by IFx, are the following ones:

– *Simulation* allows the user to execute a model and to debug it interactively. The user can execute the model randomly or step by step, inspect the system state, put conditional breakpoints, and also perform more complex operations not offered by common implementation-level debuggers: rewind/replay an execution, resolve non-determinism manually, control the scheduling policy and time related parameters, etc.
– *Verification of simple consistency conditions* allows checking in a model the absence of deadlocks and time-locks or the satisfaction of state invariants.
– *Verification of behavioral properties* by state-space exploration. Linear safety properties may be expressed by observers, and verification is performed by exploring the state space of the model and the oberver(s) run in parallel, using state of the art reduction techniques for efficient exhaustive exploration. Diagnostic traces are generated when errors are detected and may be replayed and explored using the simulation facilities.
Verification may also be done off-line by inspecting and manipulating a completely generated (possibly abstract) model state space. This is of particular interest for the extraction of properties (by minimization modulo a bisimulation relation), as it will be shown on the case study.
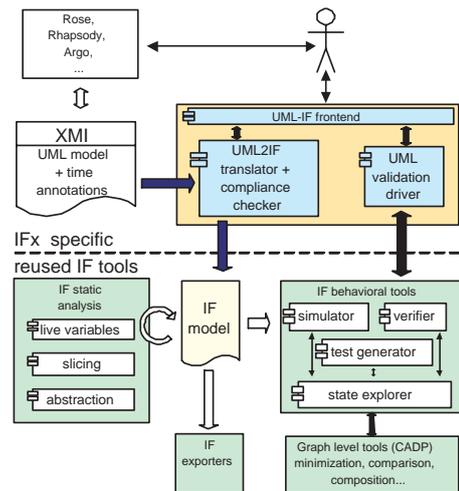

Fig. 1 Architecture of the IFx validation toolbox.

In order to scale to complex models, IF supports optimization and abstraction in several ways. There are "exact" static optimizations (like dead variable factorization and dead code elimination) which reduce the state space of the model while fully preserving its behavior (up to bisimilarity). Another exact dynamic optimization which is often very efficient is partial-order reduction during exhaustive state space exploration: this reduction renders deterministic the interleaving of parallel components whenever the non-deterministic interleaving cannot influence the verification of a given property. Data abstraction can be done either by static analysis (computing a slice and throw away a part of the system state which is irrelevant with respect to a set of properties defined by some observation criterion) or by abstract interpretation of some variables (e.g., symbolic handling of timers and clocks). Other techniques, such as input queue abstraction (a very efficient method for particular object topologies such as Kahn networks) are implemented in IF.

3 The Ariane-5 model

Ariane 5 is the European heavylift launcher. The objective of its Flight Software is to control the launcher mission from lift-off to payload release. This software operates in a completely autonomous mode and has to handle both external disturbances 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 behaviour. This description abstracts away both complex functionalities such as navigation and control algorithms and also implementation details, such as specific hardware and operating system dependencies. Nevertheless, it is fully representative of an operational space system.
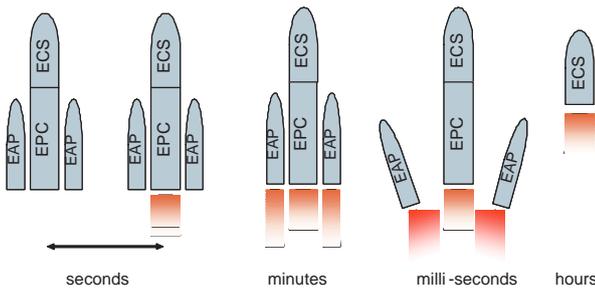
Fig. 2 Ariane 5 mission.

3.1 Ariane 5 Launcher presentation

The launcher is composed of 3 stages: EAP stage, EPC stage and ECS stage *Comment by Susanne: surement a racourcir, puisque redit par la suite*

- *EAP stage.* The two EAP boosters are ignited a few seconds after the main stage. They deliver about 90% of the global thrust at lift-off and the duration of their powder combustion is about two minutes. Then, they are separated from the main stage and fall down into the ocean.
- *EPC stage.* It is the main stage and is mainly composed of a LOX/ LH2 tank and an engine, which provides the main thrust during 8 minutes to reach the target orbit. At switch off, the stage is disconnected from the upper stage and falls down into the ocean.
- *ECS stage.* The objective of this last propulsion stage is to bring some additional energy to finetune the orbit and perform the payload release. The duration of this stage is about 15 minutes.

The *mission* is composed of several *phases* (see figure 2), each one corresponding more or less to the stabilized behaviour of a launcher stage. At the end of a permanent working of the launcher, a transition is performed to reach a new permanent working.

3.2 Case study description

An embedded space software such as the one of the Ariane 5 launcher consists of several modules which can have different, strongly interacting, types of behaviours. From the functional point of view, there are two main types of functional behaviour:

- cyclic synchronous algorithms. These are principally the control/command algorithms (Guidance, Navigation control,..) and some monitoring algorithms. They are obtained by discretisation of continuous physical laws, that is, they have a specific period and phase; they receive their inputs at the start of their period and shall produce their output before a given delay, not longer than their execution

period. Notice, that these algorithms as well as the reactivity constraints are defined by the control engineers and come as an input for the software engineer.

- aperiodic, event- or datum-driven algorithms, corresponding mainly to the spacecraft mission management (motor ingnition and stop, stage release ...) or error recovery algorithms, which interact strongly with the control command algorithms.

When designing the software architecture, the software engineer has to take into account a number of constraints

1. The application software has to be statically proven correct.
2. In the case of Ariane-5, all parts of the application software will finally be executed on the same processor[5] and share a common bus for aquiring sensor data from and sending commands to a set of physically distributed equipments
3.

In addition to these external constraints, there are additional constraints which provide a kind of *standard framework* for solving the initial problem

1.

The proof of the correctness of the implemented algorithms is done using a synchronous reactive view, that is under the assumption that at the beginning of each global cycle, all the data required by all the algorithms activated in this cycle are available and that the outputs are produced at the end of the cycle.

In a distributed architecture, several problems may arrise for guaranteeing the above assumptions.

- a data produced in a given cycle, may not be ready at the beginning of the next cycle because of a communication delay.
-

In the case where the reactivity constraints are relaxed enough Depending on the required reactivity, their implementation may be integrated into the cyclic synchronous process or not.

The software model contains 6 main classes, each having a single instance. To simplify the description, no distinction will be made between classes and instances in the following description.

- *Acyclic*: It is the main class of the software. This class manages the start of the software and the flight sequence and the associated automaton. The transitions of the automaton can be triggered
  1. on event reception from the GNC algorithm (e.g. end of thrust detection)

---

[5] in fact, a set of replicated processors, but this is out of the scope of our case study

2. on event reception from the environment
3. on timed conditions including time windows protection (allowing to assure that the treatment associated to an external event will be performed at coherent time, even in case of failure of the event detection mechanism)

- *EAP*: This class manages the behaviour of the *EAP* stage: ignition and release. The sequences described in this class are driven by events received from other classes and by internal time constraints.
- *EPC*: This class manages the behaviour of the *EPC*: ignition, monitoring of correct working, alarm raising and stop. The sequences described in this class are driven by events received from other classes and by internal time constraints.
  [IO]: il me semble que EPC est purement time-driven dans notre modle. Je me trompe?
- *Cyclics*: This class manages the activation of the cyclical control command algorithms. These algorithms are related to navigation, guidance, control, thermal control, etc. The algorithms are executed in a predefined order (depending on the current state of the launcher, given by the *Acyclic* class). Its state machine appears in an example later on in Figure 9.
- Thrust_Monitor: This is one of the algorithmic classes. It is responsible for the monitoring of the *EAP* thrust. It is activated by the *Cyclics* class.
- *Guidance_Task*: This is another algorithmic class activated by *Cyclics*. It has the particularity that its activation frequency is very low. In the real system, it is implemented in a specific ADA task.

In order to validate (by simulation or by proof) the software behaviour, a part of the environment is described. The environment can contain parts of the spacecraft as defined in the spacecraft design, the physical environment (ground control centre, wind for an atmospheric phase, other spacecraft behaviour, etc.), as well as abstractions of parts of the software which are not described in the model (such as: a numerical algorithm, a bus protocol, etc). In our model, we have used the following environment elements:

- *Ground*: It is the main class of the model. This class, representing the control centre, sends the start signal toward the launcher (and its software).
- *Bus*: This class describes the behaviour of the 1553 MIL bus allowing the communication between the main software and the equipment.
- *Valve*: This class describes a specific type of equipment. The hardware failures are modeled by an nondeterministic choice in order to verify the correct management of such a failure by the flight software.
- *Pyro*: This class describes another specific type of equipment.

The multitasking policy and the CPU consumption of each function is also modeled (see details section 4.3).

4 Capturing functional and non-functional requirements

In the initial phases of a project, functional and non-functional requirements are captured through use cases, through high-level activity diagrams, using domain-specific notations or just informally. As the system model becomes more precise requirements can be refined, formalized and used for validating the design model.

Formalization of properties in the Omega UML framework is based on the concept of *observer*. Requirements which are purely concerned with timing can also be specified using a form of declarative *constraints*. In this section we discuss (briefly) these concepts, and we insist on how they can be put to work – with examples from the Ariane-5 model and some methodological hints.

4.1 Expressing complex behavioral requirements

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, which has the purpose to give verdicts, is described by a special kind of state machine, in which some states may be labeled with the stereotypes ≪ *success* ≫ or ≪ *error* ≫.

Monitoring model execution is done either 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.

In the following we discuss some of the properties verified on the Ariane-5 example and alongside we give some methodological guidelines for writing observers.

Property 1. *The software shall not send an* Open *command to an already open valve.*

Valves are used in the main engine of the launcher to command the required thrust. Opening an already opened valve is usually an error in the software logic. This is one of the simplest safety properties that may be expressed with an observer. It requires that a certain condition never occurs during the system execution: software sends *Open* command to valve v and v is open. The only problem raised by this property, which comes back in every other formalized requirement, is to relate the informal condition expressed above with some formal event or condition occurring in the system.

In our case, the sending of an *Open* command means the call of the triggered operation *Open* defined in the class *Valve* (from package *Environment*). The "match" clause visible in Figure 3[6] matches the invocations of

---

[6] Because the properties presented in this section are taken directly from the UML model developed with Rational Rose (v.7.0), they are not completely conforming to the UML standard. In particular, we note the use of a *branch* symbol instead of a *choice* pseudo-state.

Fig. 3 Property 1.

Fig. 4 Property 2.
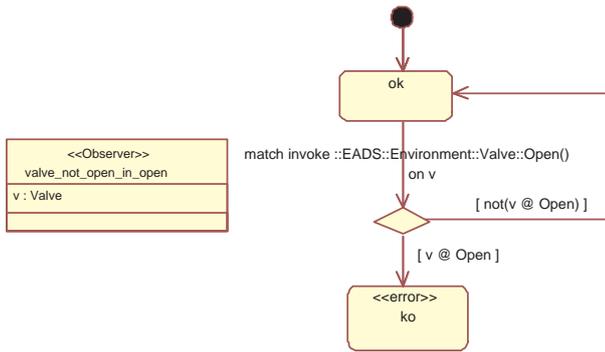
this operation, and every time the operation is invoked the reference to the callee object is stored in attribute v.

Then the *Valve* v is tested if open by simply testing whether its state machine is in state *open* (guard $v@open$). The state *ko* labeled with $\ll error \gg$ is entered if the above event occurs while the above condition is true.

Property 2. *The software shall not send two commands to the same valve at less than 50ms of interval.*

This property, required by electrical constraints on the hardware, needs a more complex formalization, since it talks about the distance between pairs of events corresponding to each of the instances of class *Valve*. A first idea is to use a different observer for each instance of class *Valve*, which measures the distance between every two consecutive commands on that *Valve*. This solution is very impractical, especially if we imagine that instances of class *Valve* could be created dynamically (although this is not the case in our model), or if the number of such instance becomes too important.

However, the following remark helps us designing a very simple observer for property 2: if several transitions are enabled in an observer at the same time, *all* the possibilities will be explored by the model checker. This obvious remark helps in writing properties over a finite sequence of events which occurs several times in the execution of the system: non-determinism can be used to pick each particular occurrence at a time and verify it.

The observer in Figure 4 works as follows: in state *initial* it waits for a command to be sent to a *Valve*, stores the reference of the concerned *Valve* in $v1$ and proceeds to state *nondet*.

In state *nondet* the observer chooses non-deterministically whether to proceed by verifying the timing of the next command sent to $v1$, or to return to *initial* and wait for another command to any *Valve*. Thus, when the observer is model-checked against the system specification, both options will be explored and all pairs of commands sent to any *Valve* will be covered.

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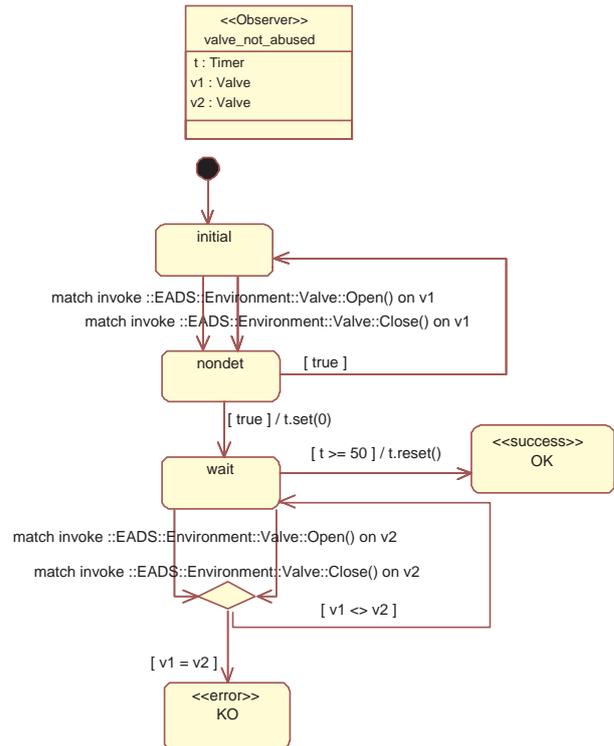
not come before 50ms. The clock t is used to measure the 50ms. In state *wait*, other commands may come, but they cause an *error* only if they concern the same valve $v1$. If more than 50ms elapse without error, the observer may go to a success state and consider the property verified for this particular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

Stereotyping the *OK* state with $\ll success \gg$ also allows to make the model checking more efficient: the execution of the system after the observer has reached *OK* cannot lead to an error anymore and may safely be ignored by the model checker.

This suggests a more general methodological issue: very often a safety property has the form $P \Rightarrow Q$ where P is an invariant or a simpler safety pre-condition on the prefixes of execution traces. For example, the property can be "if event A never happens, then an event B is never preceded by a C". P is in this case the predicate "if event A never happens". In this case, ignoring irrelevant scenarios in which P is not satisfied may strongly improve the performance of model checking. One can do that by introducing a sink state stereotyped with $\ll success \gg$ in which the observer goes every time P is broken (e.g., when an event A occurs), as shown in Figure 5.

Property 3. *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 and the pyrotechnic command shall not be ignited.*
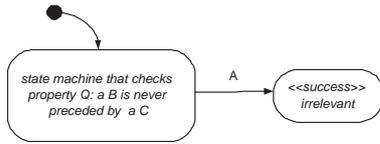
Fig. 5 Example of using a ≪ *success* ≫ state to cut off irrelevant parts of the state space
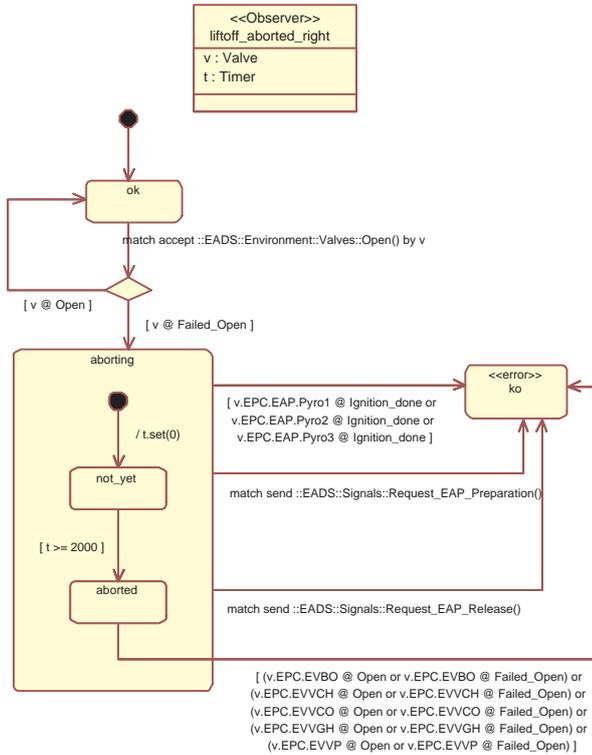
**Fig. 6** Property 3.

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 stay forever in the state *Wait_ignition*.
– 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 expressed in a purely black-box way. However, since several components are involved in aborting the lift-off, the observation of the internal signals *Request_EAP_Preparation* and *Request_EAP_Release*, which is supported in our framework, allows performing on the model level the equivalent of a mixed *white-box* and *black-box* testing activity.

We complete thus the previous property in the following way:

– The events *Request_EAP_Preparation* and *Request_EAP_Release* are never emitted.

The formal description of this property is shown in Figure 6. The observer functions as follows: every time an *Open* command is handled by a valve *v*, we test whether *v* reaches the state *Open* or *Failed_Open*. In the latter case, the observer enters state *aborted*, in which Pyro ignition (i.e. Pyro objects entering state *Ignition_done*) as well as 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* are required to remain closed (i.e. never reach the states *Open* or *Failed_Open*).

Property 4. *If the lift-off is performed, all the valves shall be opened, and the EAP stage shall be released on time.*

The lift-off is characterised by the ignition of the pyrotechnic command *Pyro1* (implying the booster ignition), i.e. object entering state *Ignition_done*. The separation on the EAP stage involves the ignition of *Pyro2* and *Pyro3* in a very precise timing.

This property can be broken into four separate observers which check that, if the *Pyro1* object enters state *Ignition_done*, then respectively:

– All the instance of the *Valve* class shall be in the *Open* state 2 seconds after, and then remain in this state forever.
– The instance *Pyro2* of the class *Pyro* shall enter *Ignition_done* in a predefined time window (relative to the start date H0).
– The instance *Pyro3* shall enter *Ignition_done* in a predefined time window (relative to the start date H0).
– The duration between the entry of *Pyro2* in the state *Ignition_done* and the entry of *Pyro3* in the state *Ignition_done* shall also be in a predefined time window.

We present in Figure 7 only the observer which checks the last of the four properties.

Although this feature is not used in the Ariane-5 model, let us note that it is possible for very complex properties to be described using a set of communicating observers. Communication is then done by shared (public) observer attributes.

4.2 Timing requirements

For a real-time system like the Ariane-5 software, certain system requirements are concerned purely with the timing of some events during the execution. This is the case for example in the Property 2 introduced in section

Fig. 7 Property 4.

Fig. 8 Property 2 as a timing constraint.

4.1: *The software shall not send two commands to the same valve at less than 50ms of interval.* Such simple duration conditions may be more easily specified using the declarative constructs of the Omega UML profile.

4.2.1 Some background on timing constraints The basic concept behind specifying constraints is the *event type*. An event type defines a pattern for matching significant events in the system execution, like an operation invocation, an object creation, etc. (the event kinds are actually the same ones that can be observed by observers, see section 4.1).

Based on an event type, one can define an *event instance*, which will capture all events matching the type or just a subset, depending on the scope in which the event instance is defined. Finally, at execution time, an event instance will represent the list of *event occurrences* that are captured.

Event instances are used to write *duration constraints* of the form $duration_{type}(ei_1, ei_2)$. There are several ways of pairing event occurrences to be constrained, each one represented by a specific *type* of operator. The simplest one (simply denoted $duration(ei_1, ei_2)$) constrains the time passed between the last occurrence of ei_1 and the last occurrence of ei_2.

For further detail on Omega timing constraints and their semantics, the reader is referred to [GOO05].

4.2.2 Property 2 as a local constraint Property 2 can be formalized as a local constraint attached to the *Valve* class, as shown in Figure 8.

The event types are *EInvOpen* and *EInvClose*, defined by a matching clause identical to the one used in observer transitions. Event instances *eo* and *ec* are declared within the scope of the class *Valve*. In this case, the runtime semantics is that there will be one instance of *EInvOpen* and one instance of *EInvClose* for every object of class *Valve*, and these two event instances will capture only event occurrences concerning their "parent" *Valve* object. This solves automatically the problem of matching events concerning the same valve that we had in the specification of property 2 by an observer (Figure 4).

Finally, the requirement that consecutive commands do not come at less than 50ms of interval is described by two declarative constraints: $duration(eo, ec) >= 50$ and $duration(ec, eo) >= 50$ (this is based on the hypothesis that there are no two consecutive *Open* commands or two consecutive *Close* commands, as required by property 1 from section 4.1).

4.3 Scheduling constraints and objectives

During the design of the Ariane-5 software, an architecture of tasks (or threads) has been constructed. Each function is assigned to a specific task. Let us note that in this system there are both cyclic control functions and sporadic regulation and configuration functions which are triggered by some event.

The tasks are all executed on the same processor, using a pre-defined fixed-priority preemptive scheduling policy. One of the goals of the model-based validation was to verify that the scheduling policy meets some consistency constraints, for example that the cyclic control functions finish in time at each cycle.

The main difficulty in using classical scheduling analysis methods such as RMA[?] to analyze this system

comes from the intervention of sporadic tasks. One cannot simply consider at each cycle the worst case execution time of sporadic tasks, as this would lead to a big over-approximation of resource occupation. What we propose instead is to take into account in the analysis the functional behavior of the system and its impact on resource consumption.

4.3.1 The problem   The scheduling policy that is used is a 3-level fixed priority preemptive scheduling:

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

There are several objectives that have to be attained by the scheduling:

- The Navigation-Control functions have to finish within the 72ms cycle and the Guidance functions have to finish within the 576ms cycle.
- The application uses a 1553 MIL bus. In this protocol, all the data transfers are performed under the supervision of a bus controller (the main onboard 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). A consistency condition is that the software components do not read or write the bus during the low-level transfer time frames.

The difficulty in analyzing the scheduling of Ariane-5 lies in that the execution time of the different tasks varies depending on the current flight phase. Figure 9 shows the statechart of the control cycle. One can see that there are optional paths which take a lot more time than others. The worst case execution time of this cycle is 64ms, while the best case is 37ms and the average measured by simulation is around 42ms.

4.3.2 Modeling the scheduling policy in Omega UML   It has been possible to model the scheduling policy and resource consumption using the low level constructs of the Omega profile (clocks). The IFx tool provides models for different types of schedulers as elements of a predefined library *Scheduling*. This solution is reusable and

open, the modeler can use the predefined scheduler models and ignore the internals of the *Scheduling* library, or alternatively extend the library with new schedulers.

The Scheduling library (see Figure 10) contains two types of classes organized in two hierarchies:

- *Task* classes used to annotate the *System* with requests for execution time, parameterized depending on the scheduling policy. Each object of the system that executes actions which take up a significant processing time, will use an instance of the class *Task* on which it will call the operation *exec* with a duration parameter. Depending on the scheduling policy, other parameters may have to be passed to *exec*, e.g., *priority* for fixed priority scheduling (FPPS), or *deadline* for earliest deadline first scheduling (EDF).
  Note that instances of class *Task* can be shared by several objects and can be used multiple times to consume processor time with *exec*. The only restriction is that there are no re-entrant calls to *exec* on the same *Task*.
- *Scheduler* classes are used to model the different scheduling policies. The behavior of these classes is transparent to the designer, who has to create an instance of a *Scheduler* class for each processing resource used by its system. Furthermore, each created *Task* has to be mapped to a *Scheduler* (when the *Task* is created). Subsequently, every time a *Task* is requested to consume processing time using *exec*, it will communicate with its *Scheduler* in order to determine the time when *exec* will finish based on the task duration and on the state of the *Scheduler* – i.e. the scheduling policy and the charge at that moment.

As an example, we describe in the following the behavior of the fixed priority preemptive scheduler. We use the scheme proposed in [**?**] (see also Figure 11). The scheduler works for a predefined range of priority from 0 (highest priority) to a constant N (lowest priority). At any time, there is at most one task executing on each level of priority; if a request comes on a level which is already occupied, this leads to an error. The scheduler keeps track of the following information:

- An array t_i of clocks measuring the time since the task on level i started its execution, including time when it was preempted.
- An array d_i with the foreseen end time for task i.

  The data is updated as follows:

- when a new task i arrives:
  - d_i stores its duration
  - let j be the currently executed task
    - if $i < j$ or j is inexistent then t_i is set to 0.
    - $\forall k > i$, if t_k is started then d_k is increased with the value of d_i.
- when the highest priority task i finishes, i.e. when $t_i = d_i$:
  - t_i is deleted
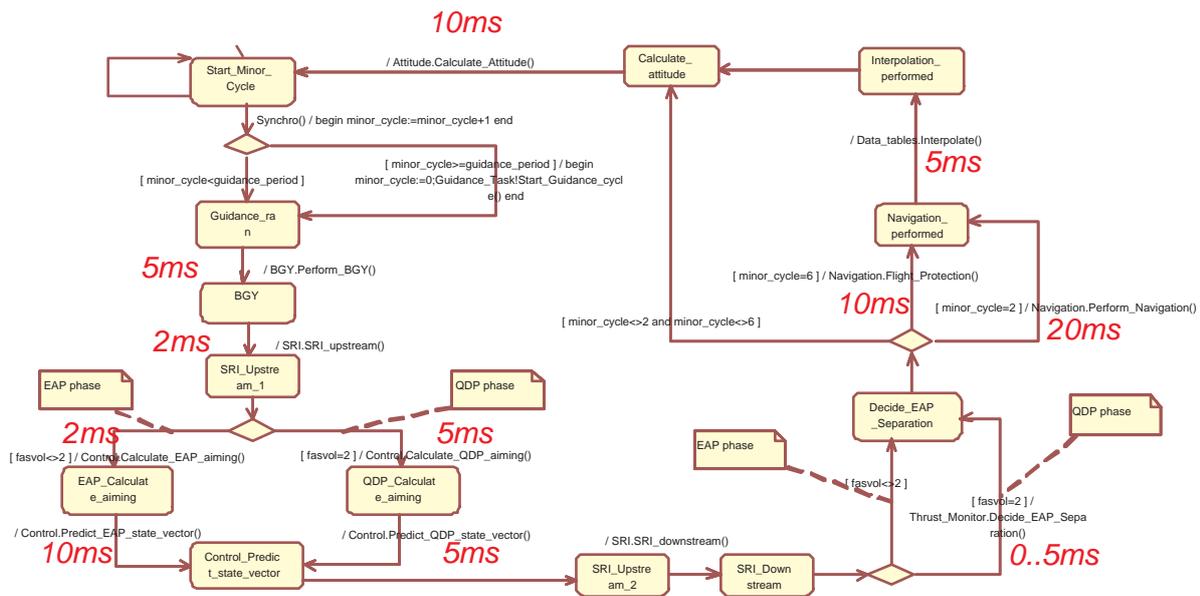
Fig. 9 Statechart of the Control cycle with unitary execution times.
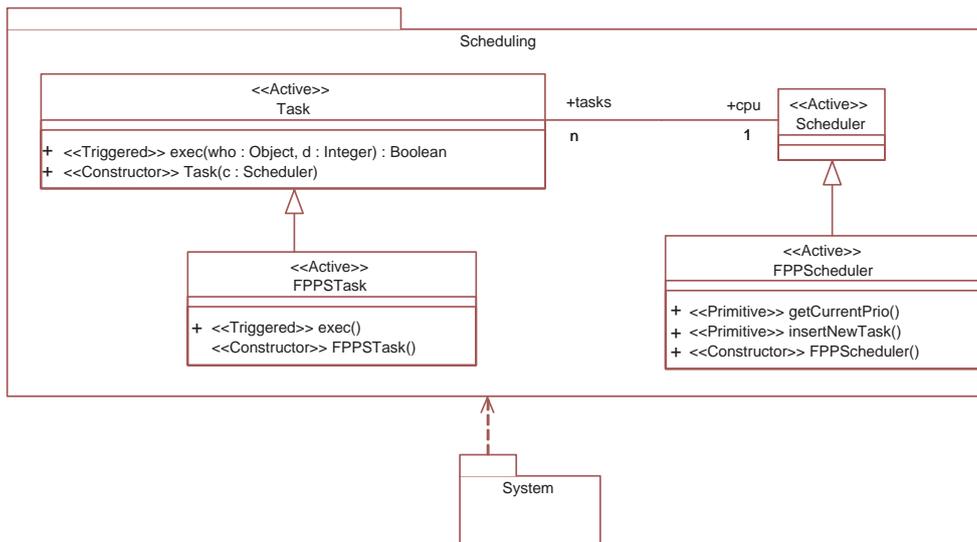
Fig. 10 Scheduling library.

    – if another task j is waiting (with the highest priority after i), and if t_j is not yet started, then t_j is set to 0.

4.3.3 Modeling the scheduling objectives Scheduling objectives are modeled using observers. As mentioned in section 4.3.1, there are three scheduling objectives:

– The control functions have to finish within the 72ms cycle. This property is formalized in the observer in Figure 12, 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 *Guidance* tasks have to finish within the 576ms cycle. This property is expressed with a similar observer.

– The bus transfer windows have to be observed. This is formalized in a similar manner by the fact that calls to *Bus read* and *write* operations do not occur while the *Bus* is in a *Transfer* state.
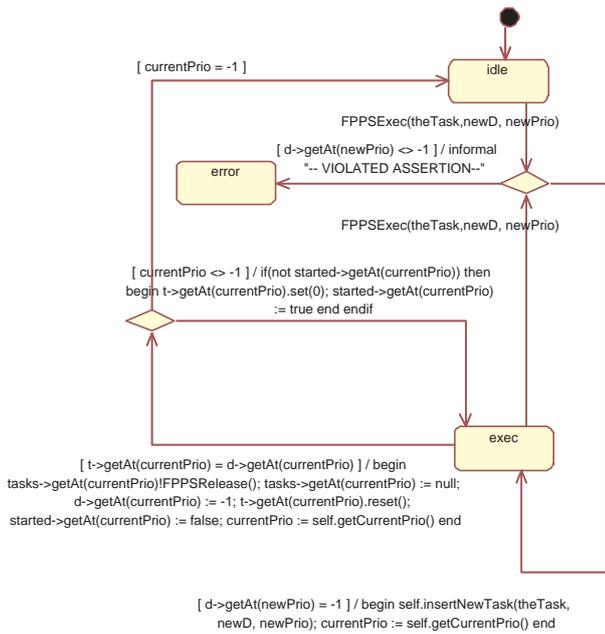
Fig. 11 Behavior of the fixed priority preemptive scheduler.
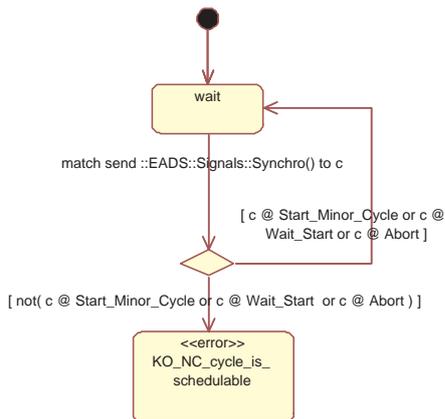
Fig. 12 Scheduling objective: the control cycle finishes in time.

5 Validation methodology

Validation of UML models with the IFx toolset involves several activities, which range from simple syntactic and static semantic checking to dynamic property verification. These activities are supported by different tools. In this section, we discuss a *standard workflow* that defines several validation phases. For each phase, we discuss the main difficulties that may occur as well as possible solutions.

5.1 Translation from UML to IF

In this phase, the *uml2if* tool is used to transform a UML model into an IF specification with an equivalent

semantics (see [OGO05] for the details of the translation). This phase performs simple static (syntactic and semantic) checks of the UML model. One can discover basic errors such as:

– syntax errors in actions
– use of undefined or uninterpreted UML constructs (e.g., unknown stereotypes or data types, some UML constructs or features not interpreted in the Omega profile, etc.)
– naming errors (e.g., use of undefined attributes, signals, classes, etc.)
– type errors (e.g. operation signature mismatch, etc.)
– violations of other well formedness constraints (e.g., root class is not active, a class has several state machines, etc.)

5.2 Static analysis

In this phase, the *dfa* tool is applied in order to analyze the IF specification, simplify it and optimize it for verification. Several types of transformations are possible:

– *State factorization* introduces systematic *reset* instructions for variables which are *dead* in a certain control state of the specification. In this way, it prevents the model checking tools to distinguish between execution states which differ only by values of dead variables. This technique is very effective, given that it can be applied locally at control-state level, and may collapse large (bisimulation equivalent) parts of the state graph.
This transformation is especially useful to reset clocks or counters which are not useful any more from a certain execution point on. In this case, the transformation may render finite a system with an infinite state space.
State factorization is recommended to be used in all cases before going on to model checking, since it is a low complexity transformation that preserves all properties.
– *Elimination of dead elements* such as unused signals and variables, unreachable states in process state machines, or process types which are never instantiated. This optimization simplifies the IF source code and can dimuinish the size of the individual system states (by eliminating variables) but has no impact on the size of the state space.
– *Slicing* is a static analysis method used to eliminate elements (variables and other objects) of the model which do not affect the validity of a set of properties, where the set of properties is defined by a set of *observable elements*.
This technique allows to explore a subset of the complete state space still guaranteeing that all properties which depend only on observations of the selected set of elements hold on the original model if and only if they hold on the reduced model.

In the Ariane-5 case study, the usefulness of factorization is limited due to the relatively small number of loops in the system state graph. *Comment by Susanne: je ne comprends pas, la factorization est lie au nombre de variables, non? p.e. on peut dire un peu plus sur ce qui etait util* Nevertheless factorization was useful to automatically reset clocks which were not useful from some point on. This was in particular the case in properties, where this simplification can be applied in the same way as in the system model. For example, clock t in Property 3 (see Fig. 6) is automatically reset upon entry in state *aborted*).

5.3 Model exploration through simulation

Simulation allows the user to explore the model in a guided or random manner, without being exhaustive. Simulation states need not to be stored as long as exhaustive state space exploration is not aimed at.

This lightweight method can be used for example to validate the existence of expected (nominal) executions of the system. It allows the user also to *disprove* simple safety properties, which must hold on all execution paths and all properties that can be invalidated on a single execution. They range from generic ones, such as absence of deadlocks or signal loss, to more specific and application dependent ones, e.g., conditional invariants, which are tested using conditional breakpoints.

The simulation tool allows performing in a simple manner usual debugging tasks: saving and re-loading a played scenario, stepping back and forward through it, inspecting the system state (possibly by defining custom views of the system state through XSLT style sheets) and inserting conditional breakpoints.

The simulation tool is mainly used at an early stage when the model has still to be debugged, or after some important changes. But it is also useful later for analyzing error traces generated by the model checker when a property is violated (replay of counter-example).

5.4 State space exploration and model checking with observers

In this phase, which should be applied only on a reasonnably debugged model, the IF model is compiled into an executable component which can be used to exhaustively explore the state space of the system, and possibly store it in the form of a labeled graph. The vertices of the graph are the system states reachable during execution of the system composed of the model under study and possibly a set of observers expressing properties, and edges represent transitions and are labeled with the event names associated with each transition.

Model checking can be performed either by analysing a completely generated and stored graph obtained by the execution of the system alone (including always also an explicit or implicit environment) as shown later on. When properties are expressed by observers, the error analysis is usually done *on-the-fly* during the complete traversal of the state space — which requires to store only the states encountered during the traversal, but not the transitions — by signalling the non satisfaction of a property each time that an <<error>> state of its observer is encountered. The IF tool stops exploration after the first error encountered. *Comment by Susanne: c'est vrai? ou seulement avec depth first*

In an early verification phase, where the goal is to quickly find errors, on-the-fly verification is the most efficient method, as usually errors are found (and then eliminated) before the state space is fully explored. The difference with simple simulation is that visited states are stored, thus enabling exploration strategies depending on the set of earlier visits of the same state.

Notice that the exploration of the entire state space[7], which is done by storing is a pre-requisite to positively prove the satisfaction of properties. In this case, a posteriori verification of a completely generated state graph is sometimes the better choice, especially for complex properties expressed by observers with many states.

Comment by Susanne: j'ai essaye de rester sur la ligne originale, mais finalement je me demande si j'en n'ai pas trop fait et il ne faudrait pas ou (1) racourcir parce que ce sont des generalites qui iront bien dans un papier tutorial mais pas tellement dans un "papier recherche" ou (2) fournir des examples venant de l'etude de cas qui affirment tout ca. Surtout concernant l'affirmation que la verif a posteriori est parfoi superieur devrait etre accompagne d'un exemple sinon on laisse tomber la phrase

5.4.1 Exploration strategies
Two main exploration strategies may be used with IF:

– Depth first exploration. With this strategy, when an <<error>> state is detected, the current "*state stack*" represents a diagnostic trace, that is, an execution leading from the initial state to a property violation which can then be analyzed and debugged using the simulator.
During depth first exploration, it is also possible to apply the so-called *partial order reduction* on-the-fly; meaning that only subset of executions which differ only by the execution order of *internal* transitions occurring in different — that is concurrent — processes. An *internal step* of a (set of) process(es) is one that does not affect the behaviour outside the process(es), that is does not perform any signal sending nor access any shared variables. Partial order reduction imposes a particular exploration order on internal steps and preserves *all* properties not observing internal actions.

---

[7] up to partial order reduction, and possibly of a model simplified by some static analysis method

Example: In the Ariane-5 model, the use of partial order reduction was the key for constructing tractable models. Without this reduction, even for flight parameters that yield the simplest behaviors, the complete exploration of the state space did not succeed (it had more than 10^6 states, with a state size of about 10KB). By using partial order reduction of internal steps, we reduced the size of the model by more than 3 orders of magnitude, that is, to about 1000 states for certain flight configurations.

Depth first exploration is in most cases the recommended strategy: it can be combined with partial order reduction and it also allows often detecting anomalous behaviors of the system very early (i.e. after generating only a fraction of the state space). During exploration, the number of already generated states and transitions as well as the "current depth" are displayed.

Notice that in cases in which the depth grows very fast or is (almost) equal to the number of explored states, one can suspect that the system is diverging, i.e. that any transition execution leads to a new state and there are no cycles. While this may be a normal behavior, in most systems this is caused by an error due to the fact that some clock or counter is never reset, or that a process signal queue is getting flooded by signals which are produced more often than they are handled. The causes of such anomalies can often be discovered and eliminated by analyzing a few "deep" states.

– Breadth first exploration. The main advantage of this strategy is that it allows finding the *shortest path* to some property violation. However, it cannot be combined with partial order reduction and the generation of diagnostics traces is much more complex *Comment by Susanne: le suivcant, je ne comprends pas – cannot be generated in a form amenable to debugging* and divergence problems can in not be detected and diagnosed early. *Comment by Susanne: autre question: est-ce bien vrai que po ne peut pas etre appliquee? ou est-ce simplement bien plus cher puisqu'il faut garder l'information normalement seulement pour l'exploration courante pour tous les etats non encore completement explores?*

5.4.2 Representations for time The time model underlying IF is that of timed automata [ACH$^+$95], which are based on the use of *clocks* which can be reset to zero and progress in states, all at the same rate. Thus, clocks can be used for measuring the time progress between the event in which a given clock has been set and the event in which it is measured. In order to enforce upper bounds on the time distance between events, a notion of *urgency* is needed. IF uses timed automata with urgency [BS97] which provides means to force the occurrence of transitions to the earliest time point at which they are enabled (*eager* transitions) and to forbid that enabled transitions are disabled by time progress beyond the validity of their guard (*delayable* transitions).

In IFx, all timing constraints of the UML model are translated into constraints on clocks in the corresponding IF model. IF allows two methods for representing clock values during state space exploration which can be chosen independently of the exploration strategy:

– symbolic or dense time representation: a state consists of an untimed state and a normalized constraint on clock values — represented by a so-called difference bound matrix (DBM) — representing the the values of the clocks at the point of time at which the state is entered, and how far time can progress without any change of the set of enabled transitions. Notice that the time unit has to be chosen in such a way that all relevant clock constraints, and therefore also bounds in the DBMs can be expressed by integer values.
– discrete time representation: in this case, in each state the clock variables have some (integer) value, and time progress is represented by *tick* events increasing all clock values by 1.

Details on the two representations and how they compare may be found in the literature on timed automata. The advantage of the symbolic representation is that it leads often to much smaller models. When the occurrence time constraints of some transitions depend on earlier measured time distances between events (that is clock differences), the use of the discrete representation is necessary as in this case the constraint on a successor state may not be representable by a DBM. When the symbolic states represent (close to) unit time progress intervals, the discrete representation is generally more efficient.

Notice that there are no simple guidelines depending on the form of the model allowing to make always the right choice. A general observation is that the presence of many *eager* transitions or the presence of "independently evolving" clocks leads often to small symbolic states. Notice that for the Ariane-5 model as well as other UML models verified with IFx, the symbolic time representation performed always bette, both in terms of state space size and generation time.

5.5 Other verification techniques

Several other techniques for property verification are available in IFx. We discuss here two of them:

– *Verification by model minimization* is an intuitive method for a non expert end-user. It consists in computing a reduced graph (with respect to a given set of observations) of the overall behavior of the specification. Such a model can be visually examined by the user.

Nevertheless, in order to obtain an abstract model, the complete state space must be stored in memory. If this is possible, it can be minimized modulo a bisimulation using ALDEBARAN (a tool connected to IFx [BFKM97]), and depending on the kind of bisimulation considered, the reduced graph strongly preserves different classes of properties (e.g., safety, absence of deadlocks, etc.). The set of observations must include all events relevant for the property being verified.

Example: The graph in figure 13 is the quotient model of Ariane-5 with respect to branching bisimulation [**?**], in which the only observable events are opening/closing the EPC valves, igniting the EPC stage and detecting anomalies.

The branching structure and all safety properties involving these actions are preserved on the graph from figure 13. It is easy to check by inspection on this abstract model that if an EAP anomaly occurs, then all the valves are closed and afterwards an EPC anomaly is signaled. Also, it is easy to check that the EPC sends the *Ignition* signal only after all valves have been (correctly) opened.

– Verification of properties expressed by μ-calculus formulas. Alternation-free μ-calculus is a temporal logic which allows expressing properties depending on the branching structure of the system state graph. Its expressive power is incomparable to that of observers (representing languages). It allows the expression of branching dependent properties such as "there is no deadlock", or fair liveness such as "after each A-transition, as long as no B-transition has occurred, there exists always at least one successor from which a B-transition can be reached".

There exists extensions of the μ-calculus allowing the expression of time dependent constraints, but this is not implemented in IF. Anyway, using μ-calculus is non trivial, even for specialists. Moreover, in timed systems, liveness properties are replaced by time-bounded safety properties for which the use of observers is almost always more convenient. Truely branching dependent properties, like the one above can only be expressed by modifying the model by adding boolean variables coding the existence of a required execution as a part of the state.

5.6 Reduction techniques with weak property preservation

Until now, we have only considered methods preserving both the satisfaction *and* the non satisfaction of a set of properties: that is, for some set of properties, the application of a reduction technique will conclude that the original system satisfies the property if and only if the reduced model does. IF allows also some reductions in which only one of the two implications holds, that is properties are only *weakly preserved*.
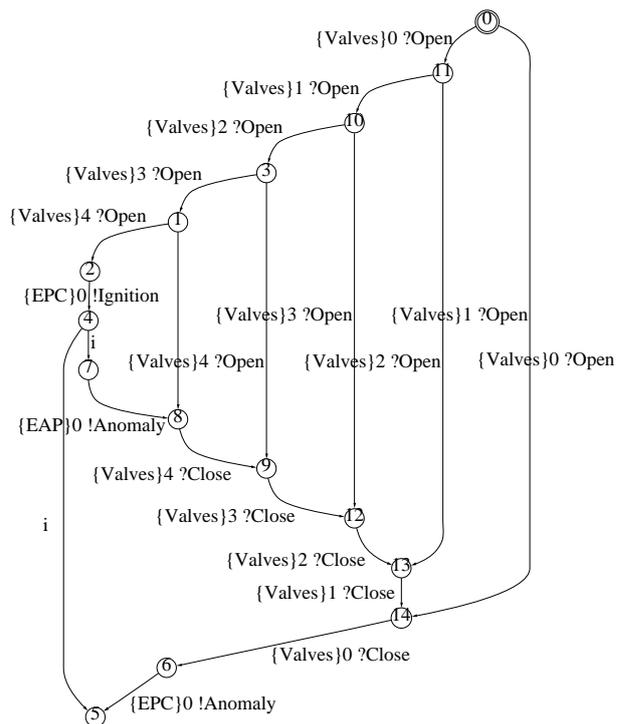
Fig. 13 A minimal model generated with ALDEBARAN.

– *existential abstraction* consists in eliminating a set of elements (variables, processes, messages) and all statements changing their value, together with all those elements whose values may depend on some already eliminated element. This transformation leads often to much smaller models as a smaller state space is considered, but it may lead also to a huge over-approximation of the behavior. Over-approximations preserve the satisfaction of safety properties, but not their non-satisfaction (i.e. they may generate false negatives). In the case of non satisfaction of a property on an abstract model, nothing can be concluded, and one has to reconsider the choice of the abstraction. IF does not (yet) provide support for this so-called abstraction refinement process [**?**].

A related abstraction of timed systems consists in relaxing the urgency type of transitions, for example transforming eager transitions into delayable or even lazy ones. This leads to a state graph allowing more behaviours but probably with much less states.

– introducing *priorities*: when property preserving partial order reduction does not eliminate enough concurrency-induced non determinism, IF proposes the addition of priority rules forcing particular execution orders for some conflicting transitions (for a description, see [OGO05,GS04]). In the case that confluence can shown for the prioritized transitions using stronger methods than those underlying the IF partial order reduction, strong property preservation

is still guaranteed. Otherwise, only a subset of the possible behaviours is explored, and only the violation of safety properties are preserved by the reduced model.

Notice that the simultaneous use of both over- and underapproximations is problematic as verification results are then meaningless. Notice also that most models use some form of over approximations due to the non representation of "irrelevant details". Thus, underapproaximations as they are introduced by priorities have to be used very carefully. When priority rules are however taken into account by the code production process, as this is for example the case in Rhapsody, then priorities are strictly property preserving with repsect to the code generation process and can be used to simplify models.

An example of using abstractions of this form is given in section 6.1. In the context of IF, where we are verifying mostly timed systems, a common abstraction consists in loosening the timing constraints of a component of the system. For example, a transition which is taken in a strictly defined time condition may be rendered time-nondeterministic by defining its urgency as *lazy*. This means that it can be delayed indefinitely instead of being executed as soon as it is enabled. While this introduces new behaviors in the model, the state space may shrink by an important factor because of the symbolic representation of time that is used.

6 Ariane-5 verification results

6.1 Abstractions in Ariane-5

Comment by Susanne: le problem que je vois ici, c'est qu'on a enumere tout un tas de techniques de reduction mais finalement on a utilise encore d'autres. Sans explication ca ne passe pas

The duration of a basic cycle of the cyclic behavior of the Ariane-5 flight software is about 100 ms. Each basic cycle contains about 100 steps. This implies that the generated model will have a depth of about 3 600 000 steps for 1 hour mission, and 15 000 000 000 steps for a 6 months mission, etc ...

Current tools do not allow to exhaustively explore state spaces of this size, even after application of the previously presented automatic reduction techniques. 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. *Comment by Susanne: p.e. plutot dans les conclusions de la section: In our case study, the verification has been done by the designer with the guidance from the verification expert. ...*

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 description representing an abstraction, that is, allowing more behaviours. Nevertheless, the simple variable elimination implemented in IF did not succeed *Comment by Susanne: a-t-on essaye ?*. We had to manually provide abstractions, and in order to minimize the modelling effort, the existing decomposition of the system into a cyclic and an acyclic part, and the clear interface between them has been exploited:

- Abstraction of the cyclic behavior.
  To prove safety properties related only to the acyclic part, that is the flight programm, the cyclic GNC part has been abstracted. This has been done by eliminating all the internal behaviour of the cyclic part, and by sending the events generated by the GNC (flight phase change commands) at arbitrary moments, rather than at an "optimal" time point computed by the concrete GNC. This simplified GNC is clearly an abstraction, and it was sufficient to show the satisfaction of all the properties of the asynchonous part. *Comment by Susanne: c'est finalement une abstraction qu'on saurait construire presque automatiquement?. Si je comprends bien on a utilise une elimination des time guards — affaiblir des gardes. cest clairement une abstraction — suivi d'une abstraction existentielle de tout le comportement interne de GNC, qu'on aurait pu obtenir automatiquement, p.e. meme en applicant le slicing ?*
- Abstraction of the asynchronous behavior.
  Comment: [IO]: est-ce qu'on peut dire plus prcisement ce qu'on a fait avec a? je ne me rapelle plus si je l'avais utilis vraiment. [S]: oui ca serait bien d'expliquer un peu en detail en quoi consiste l'abstraction
  In this second step, the asynchronous part has been abstracted. Also in this case, we mainly relaxed time constraints: The cyclic behavior received asynchronous events generated at a non deterministic time. No hardware failure can occur. Even if this abstraction is not completely realistic (especially for a CPU consumption point of view), it has allowed the detection and the correction of several errors in the model. *Comment by Susanne: la, on melange tout: interdire l'occurrence de "fault events" est une sous approximation, qui en effet p.e. utilise pour la detection d'erreur. Mais le melange avec l'abstraction (relacher des time constraints) fait qu'il peut s'agir de spurious errors; donc il faut argumenter pourquoi on est content quand meme.*
  ensuite on a bien du montrer des proprietes, ce qui est a priori impossible avec un melange de sur et sous approximations

We have also used an alternative reduction without behavioral abstraction, in order to show global correction of the software, which allows also checking some global properties on the time points at which the event exchanges between the cyclic and the acyclic part take place. *Comment by Susanne: est-ce vrai? sinon il faut dire pourquoi c'est bien, ca permet une simulation plus realist qui p.e. utilisee aussi pour une inspection manuelle? c'est plus simple a construire ? autre?*

Reduction of the durations of the flight phases   .

A huge source of state explosion is the difference of the timing scale between the asynchronous behaviour and the cyclic one. The cyclic behaviour deals with durations of a few milliseconds, whereas the asynchronous one deals with durations of several hours (and up to some months for other types of missions like the ATV[8] project).

Comment by Susanne: ca ne me convainc pas, c'est plutot une justification pour l'abstraction de le partie cyclique comme avant; j'essais une explication alternative ci-dessous. est-elle correcte?
That means that asynchronous events are rare, and the system is working without occurrence of any asynchronous events during a great number of basic cycles. Moreover, most of the output of the cyclic part is irrelevant for the properties to be verified. Thus, it is sufficient to perform the proof with a mission duration much greater than the basic cycle, but shorter than the real mission duration.

That means that asynchronous events are rare, and the system is working without occurrence of any asynchronous events during a great number of basic cycles in what we call here "stable phases". In such stable phases, all executions of the basic cycle in the cyclic part are identical with respect to the properties that we want to verify: in particular,

– the schedulability of all tasks in all relvant cycles
– the reactivity to exceptional events (such events do not occur in stable phases)
– the respect of a certain time window for the commands send from the synchronous to the asynchronous part (stable phases are outside this time window

This suggests that the model can be reduced by drastically reducing the overall flight duration, by 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 exceptions, where the correctness of the software has only to be guaranteed if there are not more than 2 exceptions over the entire flight.

The correctnes of a chosen reduction of the duration of the different phases can be shown als follows *Comment*

----
[8] Automated Transfer Vehicle

by Susanne: ici ca serait sympa si on pouvait dire has been shown .... For a choice of a minimal duration of all phases which requires a good knowledge of the design, we show phase by phase by starting with the first one, that:

– after the expected stabilization time the system is indeed stable, that means, nothing changes from one cycle to the next; thus, only time progress (leading to the next phase change) or exceptions may lead to new states.
– In a similar way one has to validate on over approximation of the stabilisation time for all exceptions
– then, the duration of the first phase can be reduced to the minimal stabilisation time plus the duration allowing two exceptions to occur and to stabilize again. This allows to guarantee that all states that can be reached during the first flight phase with the original flight durations, will also be reached using the shorter duration.
– Now we can use a short first flight phase to verify the stabilisation delay for the second phase, and so on, until the last one.

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 been shown to hold. Notice that the properties of the cyclic part were taken unchanged, whereas the properties of the acyclic part had to be adapted to take into account the reduced duration of the different flight phases.

6.2 Results and figures

In this section, we want to show the efficiency of the applied reduction methods.*Comment by Susanne: il faudrait donc inclure des durees plus longues puis conclure que pour 50 minutes on n'y arrivera jamais?*

In order to test the tool, we performed the mission time reduction by using different mission durations (but always respecting the required stabilisation times) *Comment by Susanne: est-ce que les duress utilisees sont vraiment suffisantes? j'aurai la tendance de dire que tant que l'espace d'etat ne crois pas lineairement, il y des vrais nouveaux etats?.*

The table in Figure 14 shows the verification time and the size of the explored model using live variable and partial order reduction (why not slicing?) for a given mission duration (which one?) and for the verification of individual properties. The difference between the explored state spaces is small, meaning that all properties are of similar complexity.

Comment by Susanne: je comprends bien qu'ici on a utilise -live et -po, et la difference des taille vient de la difference des tailles des observateur ou plutot du produit?

est-ce que tu as une taille du modele sans observa-teur? pour voir l'influence des proprietes

je suppose aussi que les temps donnes sont des temps de generation, et qu'on peut mentionner cette histoire de temps de compilation qui est affreusement long dans les conclusions

We have also tried to verify all the properties at the same time by running all observers in parallel with the model. The array in Figure 15 gives the same information as above for all properties verified at the same time and by varying the choice of the mission duration.

Notice that the state explosion due to parallel composition of all properties is less important that one could expect, which means that the properties are not really independent. In fact, all the observers have just a few control statesand the state changes in different events are due to events which are either identical or strictly ordered in time. The state of most observers depends also on clocks. But looking more closely, the clocks of different observers are active in different model states.

That is, the methodological guideline proposing complex proeprties into more simple ones, has here relative little impact on the complexity of model-checking. Nevertheless, the use of many small properties is of great importance for the understanding of properties.

7 Comparison to other approaches

There exist already a number of tools proposed for the validation of UML models by translating a subset of UML into the input language of some existing validation tool [LP99a, LP99b, LMM99, Kwo00, KMR02, SKM01, DMY02a, dMGMP02, XLB01, DMY02b, BLM02, STMW04, AHK$^+$04] 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 [Hol99] (in [LP99a, LP99b, LMM99, SKM01]) or COSPAN [HK88] (in [XLB01]) for untimed systems, and Kronos [Yov97] (in [BLM02]) or Uppaal [LPY97] (in [KMR02, DMY02b]) for the validation of systems with timing constraints. Also the translation into proof-based frameworks, such as PVS [SOR93] (e.g. in [AHK$^+$04]) or B (in [LMS02]), has been proposed.

From the point of view of the technology used, the IF tool includes and combines the on-the-fly exploration of SPIN and the symbolic representation of time constraints of Kronos and Uppaal. It includes also the bisimulation based reduction techniques of Aldebaran [FGK$^+$96].

With respect to the expressivity of the UML profile accepted, the framework presented here goes beyond the existing tools. IFx handles a rich subset of UML, including inheritance and dynamic object creation as well as powerful timing features. Most other tools are restricted to static systems, often described by state-charts. [DMY02b] considers a real-time UML fitting the input language of Uppaal which is less expressive than that of IF.

The tools described in [STMW04, AHK$^+$04] are, like ours, based on the Omega profile, where the first one does not take into account timing extensions and the second one is based on deductive verification with PVS and could been applied so far only to quite simple examples (using a small, yet expressive, subset of the profile).

Also, most of the tools proposed for the validation of UML models rely on the expression of properties to be checked in some tool dependent formalism, in many cases some form of temporal logic which is difficult to handle by system designers. This means also that the properties to be validated are not really part of the model which is bad from the methodological point of view. The Omega UML profile proposes the use of *observers*

8 Conclusions and Future Work

In the Omega project, we have developed different types of validation engines and used them in several case studies. The evaluation of the contributions of the different tools shows their complementarity:

- The big advantage of the IF tool which is based on explicit state exploration combined with powerful reduction techniques, turned out to be the fact that it allows to obtain feedback very rapidly on all models without much remodelling and adaptation effort by the user. Obtention of positive verification results required in the case of the bigger examples some effort to find an appropriate property preserving abstraction or approximation. In the case of the Ariane-5 flight programme, this consisted ....
- The RUVE toolset [STMW04] working on the same profile is built upon a BBD-based symbolic model-checking engine.
- The third toolset handling the Omega profile [AHK$^+$04] translates a UML model into a set of expressions defining the set of executions of this model and defines a set of PVS strategies designed for helping the user to accelerate the interactive verification with PVS of properties - expressed either in OCL or in the expression language of PVS. The advantage of this tool is that positive verification results on parameterized and infinite systems can be obtained. Nevertheless, the experience with the case studies in Omega showed that this kind of verification applies best to abstract algorithms. The obtention of results is time consuming, and in general it is advisable to first validate a finite instance of the system with a model-checker before starting the proof process. An important problem with this tool is the readability of the expressions obtained by automatic translation from UML. This means in general that only

Property	Number of states	Number of transitions	Proof duration
liftoff_aborted_right	36037	38149	00:00:36
pyro_not_ignited_twice	35988	38092	00:00:42
valve_not_abused	36082	38210	00:00:37
valve_not_close_in_close	36010	38114	00:00:44
valve_not_open_in_open	35998	38102	00:00:38
liftoff_performed_right1	46075	48713	00:00:49
liftoff_performed_right2	37897	40550	00:00:55
liftoff_performed_right3	37961	40632	00:01:12
liftoff_performed_right4	35986	38090	00:00:38
CPU_not_in_error	35980	38084	00:00:53
G_cycle_is_schedulable	36012	38116	00:00:48
NC_cycle_is_schedulable	36380	38484	00:00:39
read_write_coherence	36618	38722	00:00:47

Fig. 14 State space size and times for the verification of safety properties

Mission duration	Number of states	Number of transitions	Proof duration
7 s	51 324	54 697	00:03:30
15 s	161 956	171 734	00:12:06
22 s	303 496	321 206	00:11:33
30 s	463 932	490 901	00:22:58
37 s	658 981	696 031	00:34:53

Fig. 15 State space size and times for different mission durations (all properties combined)

small systems - or components of it - can be used for verification. Which means that the decomposition of the system must be done with respect to the properties to be verified.

This tool has not been applied to the schedulability problem of the Ariane-5 flight program Also, the user has to come up with the necessary auxiliary invariants

References

ACH+95. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, January 1995.

AHK+04. 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, Springer-Verlag, 2004.

BFG+99. M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of Formal Methods'99, Toulouse, France*, number 1708 in LNCS. Springer Verlag, September 1999.

BFKM97. M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the ALDEBARAN toolset. *Software Tools for Technology Transfer*, 1:166–183, 1997.

BGM02. M. Bozga, S. 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*, 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*, number 3185 in LNCS, June 2004.

BLM02. 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.

BS97. S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop, HART'97, Grenoble*, LNCS 1201, pages 286–300. Spinger Verlag, March 1997.

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).

dMGMP02. Maria del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Ob-*

ject Technology, 1(2):101–117, August 2002. (http://www.jot.fm/issues/issue 2002 07/article1).

DMY02a.   Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, Fundamental Approaches to Software Engineering (FASE'2002), volume 2306 of LNCS, pages 218–232. Springer-Verlag, April 2002.

DMY02b.   Alexandre David, Oliver Mller, and Wang Yi. Formal verification UML statecharts with real time extensions. In Proceedings of FASE 2002 (ETAPS 2002), volume 2306 of LNCS. Springer-Verlag, April 2002.

FGK+96.   Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, volume 1102 of Lecture Notes in Computer Science, pages 437–440, 1996.

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. STTT, Int. Journal on Software Tools for Technology Transfer, 2005. under press.

GS04.   Gregor Gössler and Joseph Sifakis. Priority systems. In Formal Methods for Components and Objects 2003, number 3188 in LNCS. Springer Verlag, 2004.

HK88.   Z. Har'El and R. P. Kurshan. Software for Analysis of Coordination. In Conference on System Science Engineering. Pergamon Press, 1988.

Hol99.   G. J. Holzmann. The model-checker SPIN. IEEE Trans. on Software Engineering, 23(5), 1999.

KMR02.   Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, 7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), volume 2469 of Lecture Notes in Computer Science, pages 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.

Kwo00.   Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Bran Selic Andy Evans, Stuart Kent, editor, Proceedings of UML'2000, volume 1939 of Lecture Notes in Computer Science. Springer-Verlag, 2000.

LMM99.   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.

LMS02.   N. Levy, R. Marcano, and J. Souquieres. From requirements to formal specification using UML and B. In International Conference in Computer Systems and Technologies, CompSysTech 2002. Sofia, Bulgaria, 2002.

LP99a.   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.

LP99b.   Johan Lilius and Ivan Porres Paltor. vUML: A tool for verifying UML models. In Proceedings of 14th IEEE International Conference on Automated Software Engineering. IEEE, 1999.

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

OGO05.   Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. STTT, Int. Journal on Software Tools for Technology Transfer, 2004, 2005. Under press.

OMG03.   OMG. Omg unified modeling language specification, version 1.5. Technical report, March 2003. available through http://www.omg.org/cgi-bin/doc?formal/03-03-04.

SKM01.   Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science, 55(3):13 pages, 2001.

SOR93.   N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

STMW04.   Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), pages 174–183, Bejing, China, September 2004. IEEE. available at http://csdl.computer.org/comp/proceedings/sefm/2004/22

XLB01.   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.

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.