Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A case study in UML model-based 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. This paper presents ...

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.

Recently we have been developing a validation toolset for UML models, IFx [OGO05], which allows *simulating* an operational system specification and *verifying* complex behavioral properties expressed formally within the model. This toolset, which is particularly well adapted to real-time and concurrent systems, is described briefly in section 1.2.

The focus of this paper is on one of the largest case studies on which our toolset has been applied: a model obtained by reverse-engineering of a representative part of the flight software of the Ariane-5[1] launcher. Along the lines we discuss the modeling and specification features used in connection with IFx, the normal tool workflow and the validation results for Ariane-5, and also some methodological issues: how results were obtained, what problems can be encountered and how they can be solved.

1.1 *State-of-the-art and current practice in model based validation*

[DL]: it is not a global state of the art (i.e. the descripion of all existing methods and tools), but only the IF state of the art

1.2 *The IFx toolset*

IFx [OGO05] is a toolset providing simulation and verification functionality for operational UML models. Implementing specific UML semantics and extensions for expressing timing-related information [OGO05, GOO05], IFx is targeted to designers of safety critical embedded real-time systems.

1.2.1 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

[1] Ariane 5 is the european heavylift launcher, with payload capacity of 10,000 kilogrammes on dual launches into GTO (geostationary transfer orbit). EADS SPACE Transportation, the European space transportation and orbital systems specialist, is now single Prime Contractor for the Ariane 5 system.

Rhapsody. The simulation and verification functionality of IF is wrapped by a UML-specific interface which hides the details of the IF format and tools from the designer.

The functionality of IF, wrapped and provided at UML level by IFx, is the following:

- *Simulation* allows the user to execute a model and to debug it interactively. The user can execute the model step by step, inspect the system state, put conditional breakpoints, and also perform more complex operational 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 the absence of deadlocks and time-locks in a model or the satisfaction of some state invariants.
- *Verification of behavioral properties* by model checking. The properties may be expressed as simple timing constraints using the Omega time extensions or as more elaborated observer objects (see next paragraph). Verification is performed by (and during) the exhaustive construction of the model's state space. Diagnostic traces are generated upon errors, and may be used in simulation.
  Verification may also be done off-line by inspecting and manipulating the model state space (e.g. minimizing it modulo a bisimulation relation), as it will be shown on the Ariane-5 case study.
  [DL]: je pense que le paragraph suivant peut-tre supprim car il est redondant de ce qu'il y a ci-dessous
- *Static analysis* may be used to optimize an IF specification for subsequently verifying it. Static analysis computes exact or approximating abstractions of a model, [DL]: I do not understand how an abstraction can be exact (it is no more an abstraction in this case). I assume it means that all the properties without exception (?) are preserved. chosen to preserve certain types of properties (e.g. safety properties). Their use is exemplified in the sequel (SECTION XXX??).

Additionally to these, IF offers other functions that may in the future be interesting for the UML modeler: automatic test case generation, connections with other validation tools like Spin or Agatha, etc.

In order to scale up to complex models, IF supports abstraction in several ways. For example, 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 an observation criterion) or by abstract interpretation of some variables (e.g., symbolic handling of timers and clocks). An exact [DL]: idem as before. It is not an abstraction in this case? it is only a translation abstraction 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. Other techniques, such as input queue abstraction (a very efficient method for particular object topologies such as Kahn networks) are implemented in IF.

1.2.2 Supported UML features and semantics

The IFx toolset supports the constructs and the particular semantics of the Omega UML profile [OGO05, GOO05, DJPV03]. In this section we make 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, generalization relationships.

State diagrams may be used to define reactive class behavior; they react either to asynchronous signals exchanged between objects, to conditions (change events) or to operation calls (operation calls which are handled by the state machine are known as *triggered* operations in Omega UML). The behavior of an operation that is not *triggered* (called *primitive* operation) is described by an action. Actions are written in a syntax compliant to UML1.4 action semantics, containing imperative constructs like assignments, operation calls, object creation, signal exchange, etc.

The profile supports the description of concurrent systems, by using 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. Thus concurrency is created by non-blocking requests between activity groups (i.e. signals or operations which do not send a return value).

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 [?], 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.

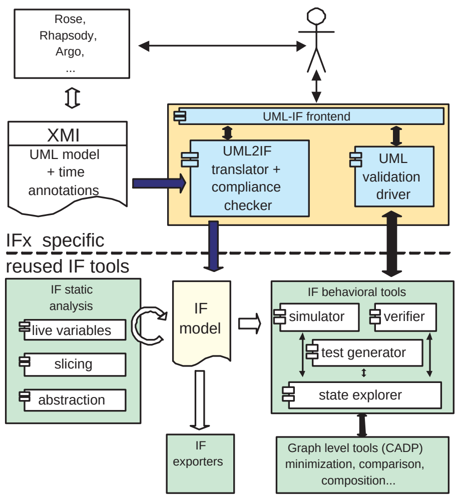


Fig. 1. Architecture of the IFx validation toolbox.

2 The Ariane-5 model

Ariane 5 is the european heavylift launcher. It has a payload capacity of 10,000 kilogrammes on dual launches into GTO (geostationary transfer orbit). The objective of the Ariane 5 Flight Software is to control the launcher mission from lift-off to payload release. This software operates in a completely automatic mode and has to handle both the external disturbances and the hardware different 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 system.

2.1 Ariane 5 Launcher presentation

The launcher is composed of 3 stages: EAP stage, EPC stage and ECS stage

- EAP stage
  The two EAP boosters are ignited a few seconds after the main stage. They deliver about 90of the global thrust at lift-off and the duration of their powder combustion is about two minutes. Thus, they are separated from the main stage, and fall down in 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 fall down in the ocean.

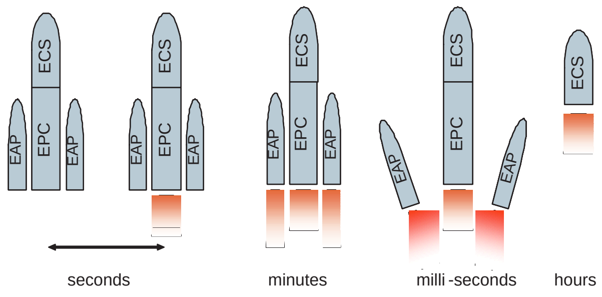


Fig. 2. Ariane 5 mission.

- ECS stage
  It is the upper stage and the objective is to bring a supplementary benefit in energy to perform the payload release. The duration of the permanent functioning 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 permanent working point of a launcher stage. At the end of a permanent working of the launcher, a transition is performed to reach a new permanent working.

2.2 Case study description

An embedded space software such as the one of the Ariane 5 launcher is composed by several modules which can have different types of behaviours strongly interacting. Roughly speaking, it exist two main types of functional behaviours:

- cyclical synchronous execution (i.e. all the processes have a specific period and phase; they received their inputs at the start of their periods and shall produce their outputs before a given delay, which shall not be greater than their execution period).
- non cyclical execution (synchronised or not with the cyclical synchronous process depending of their required precision; their execution can depend from a date or from a process external event).

The cyclical synchronous behaviour corresponds mainly to the navigation, guidance and control algorithms (GNC, i.e. the control command of the spacecraft). The asynchronous behaviour corresponds mainly to the spacecraft mission management (motors ingnition and stop, stage release...).

The software model contains mainly 6 classes corresponding to 6 main objects (each class has a single instance). To simplify the description, no distinction will be performed 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.

- EAP: This class describes the behaviour of the EAP stage: ignition and release. The sequences described in this class are driven by event received by other classes and by internal time constraints.
- EPC: This class describes the behaviour of the EPC: ignition, monitoring of correct working, alarm raising and stop. The sequences described in this class are driven by event received by other classes and by internal time constraints.
- Cyclics: This class manages the activation of the cyclical control command algorithms. These algorithms can be navigation, guidance, control, thermal control... This class executes these algorithms in a predefined order (depending of the current state of the launcher, given by the Acyclic class). See an example figure**??**
- Thrust_Monitor: This class is one of the algorithmic classes. It is responsible for the monitoring of the EPA thrust. It is activated by the Cyclics class.
- Guidance_Task: The guidance activation is a particular case as its frequency is very low. Thus, it is implemented in a specific ADA task. This is modelled by the Guidance_Task, which is activated by the Cyclics class.

The Acyclic object creates (via aggregation links) all the over main objects. The constructor of the Ground class (see below) creates the links between sub-object.

In order to validate (by simulation or by proof) the software behaviour, a part of the environment is described. The environment can contain part of the spacecraft as defined in the spacecraft design, the physical environment (ground control centre, wind for an atmospheric phase, star and moon for some sensors, other spacecraft?), and part of the software (or more generally of the computer based system) which is not described in the model (as a numerical algorithm, a bus protocol, etc).

- Ground: It is the main class of the model. This class, representing the control centre send the start signal toward the launcher (and its software).
- Bus: This class describes the behaviour of 1553 bus allowing the communication between the main software and the equipment.
- Valve: This class describes a specific type of equipment.
- Pyro: This class describes a specific type of equipment.

3 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.

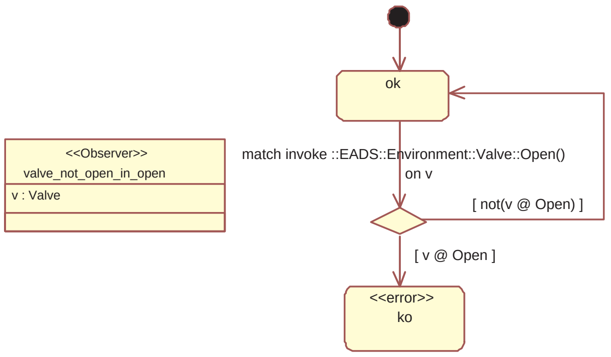


Fig. 3. Property 1.

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.

3.1 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 object state and attribute values, contents of queues, 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[2] matches the invocations of 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 ≪ *error* ≫ 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*, with which we measure the distance between every two consecutive *Valve* commands. 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.

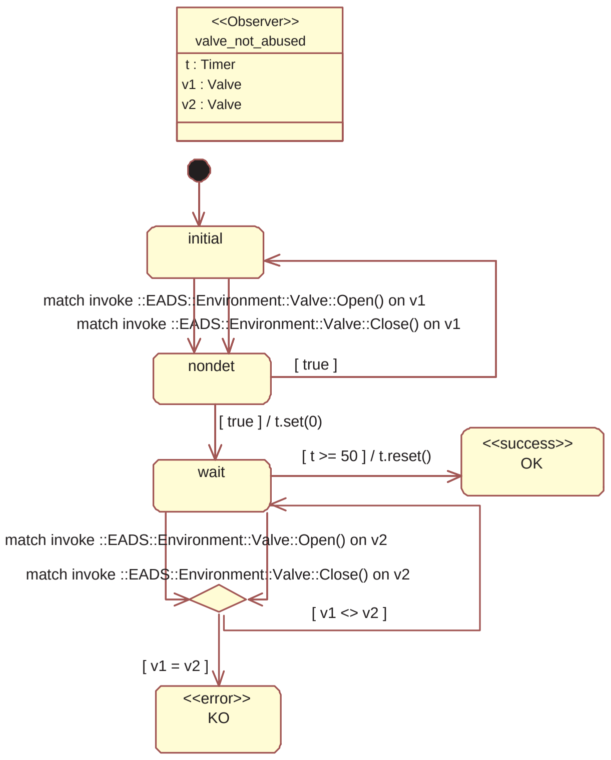
However, the following remark helps us designing a very simple observer for property 2: if several transitions are enabled in an observer at the same time, *all* the possibilities will be explored by the IFx model checker. This obvious remark helps in writing properties over a finite sequence of events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

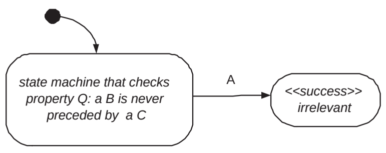
The rest of the observer tests a simple safety condition: the second command sent to the *Valve* $v1$ will not come before 50ms. The clock t is used to measure the 50ms. In state *wait*, other commands may come, but they cause an *error* only if they concern the same valve $v1$. If more than 50ms elapse 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.

[2] 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, which is not supported by Rational Rose.??



<<Observer>>
valve_not_abused
t : Timer
v1 : Valve
v2 : Valve
initial
match invoke ::EADS::Environment::Valve::Open() on v1
match invoke ::EADS::Environment::Valve::Close() on v1
nondet
[true]
[true] / t.set(0)
wait
[t >= 50] / t.reset()
<<success>>
OK
match invoke ::EADS::Environment::Valve::Open() on v2
match invoke ::EADS::Environment::Valve::Close() on v2
[v1 <> v2]
[v1 = v2]
<<error>>
KO

Fig. 4. Property 2.



state machine that checks property Q: a B is never preceded by a C
A
<<success>>
irrelevant

Fig. 5. Example of using a ≪ *success* ≫ state to cut off irrelevant parts of the state space

Stereotyping the *OK* state with ≪ *success* ≫ 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 ≪ *success* ≫ 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*

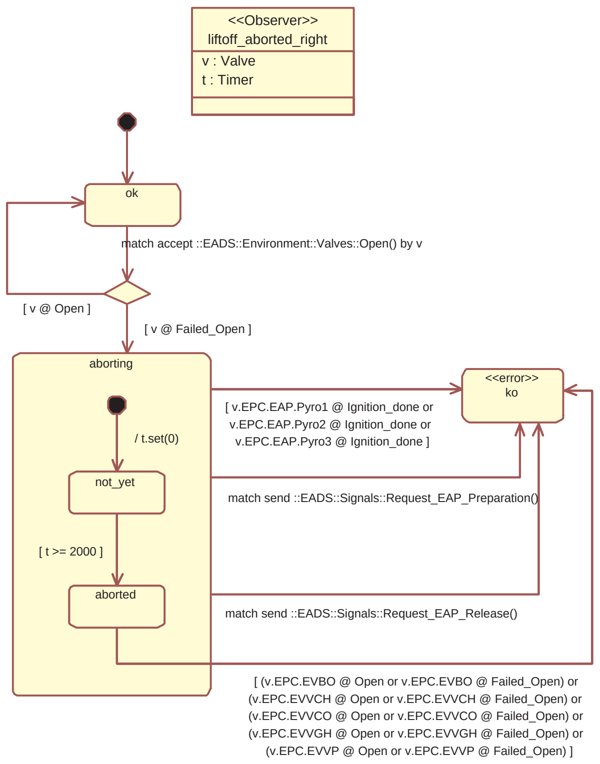

<<Observer>>
liftoff_aborted_right
v : Valve
t : Timer
ok
match accept ::EADS::Environment::Valves::Open() by v
[v @ Open]
[v @ Failed_Open]
aborting
/ t.set(0)
not_yet
[t >= 2000]
aborted
<<error>>
ko
[v.EPC.EAP.Pyro1 @ Ignition_done or
v.EPC.EAP.Pyro2 @ Ignition_done or
v.EPC.EAP.Pyro3 @ Ignition_done]
match send ::EADS::Signals::Request_EAP_Preparation()
match send ::EADS::Signals::Request_EAP_Release()
[(v.EPC.EVBO @ Open or v.EPC.EVBO @ Failed_Open) or
(v.EPC.EVVCH @ Open or v.EPC.EVVCH @ Failed_Open) or
(v.EPC.EVVCO @ Open or v.EPC.EVVCO @ Failed_Open) or
(v.EPC.EVVGH @ Open or v.EPC.EVVGH @ Failed_Open) or
(v.EPC.EVVP @ Open or v.EPC.EVVP @ Failed_Open)]

Fig. 6. Property 3.

of lift-off abort, the valves shall all be closed and the pyrotechnic command 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 entries 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 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.

3.2 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 3.1: *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.

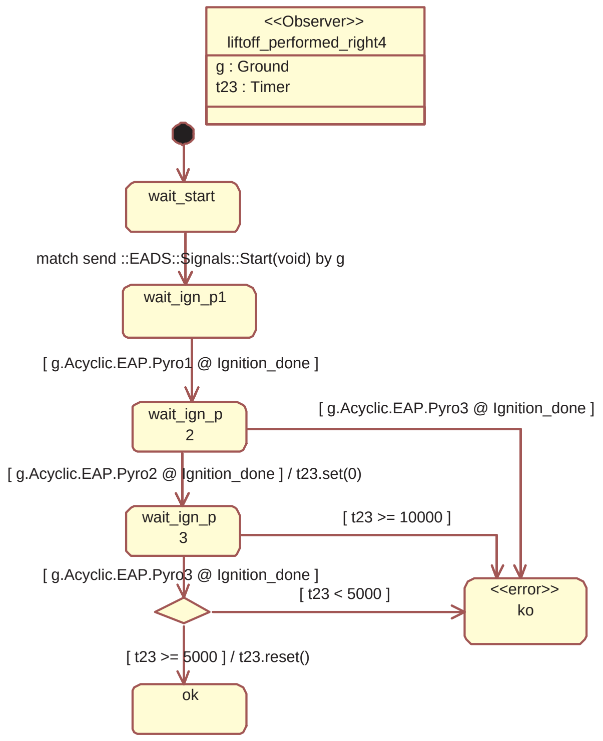

<<Observer>>
liftoff_performed_right4
g : Ground
t23 : Timer
wait_start
match send ::EADS::Signals::Start(void) by g
wait_ign_p1
[g.Acyclic.EAP.Pyro1 @ Ignition_done]
wait_ign_p 2
[g.Acyclic.EAP.Pyro3 @ Ignition_done]
[g.Acyclic.EAP.Pyro2 @ Ignition_done] / t23.set(0)
wait_ign_p 3
[t23 >= 10000]
[g.Acyclic.EAP.Pyro3 @ Ignition_done]
[t23 < 5000]
<<error>> ko
[t23 >= 5000] / t23.reset()
ok

Fig. 7. Property 4.

3.2.1 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 3.1).

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].

3.2.2 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

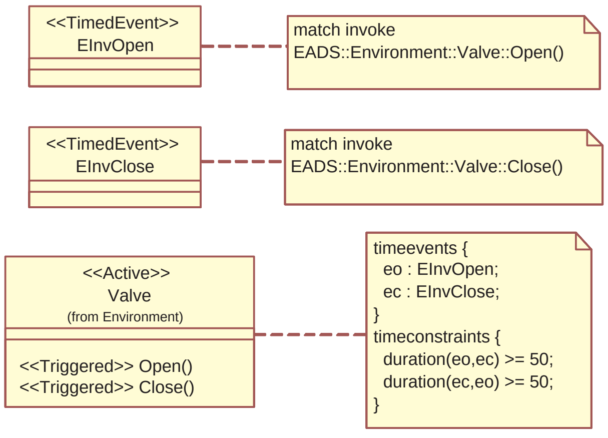


Fig. 8. Property 2 as a timing constraint.

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 3.1).

3.3 Scheduling constraints and objectives

During the design of the Ariane-5 software, an architecture of tasks (or threads) is 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.

3.3.1 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.

3.3.2 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.
    - $\forall k > i$, then d_k is incremented with d.
- when the highest priority task i finishes, i.e. when $t_i = d_i$:
  - t_i is deleted ($started_i$ is set to false);
  - 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.

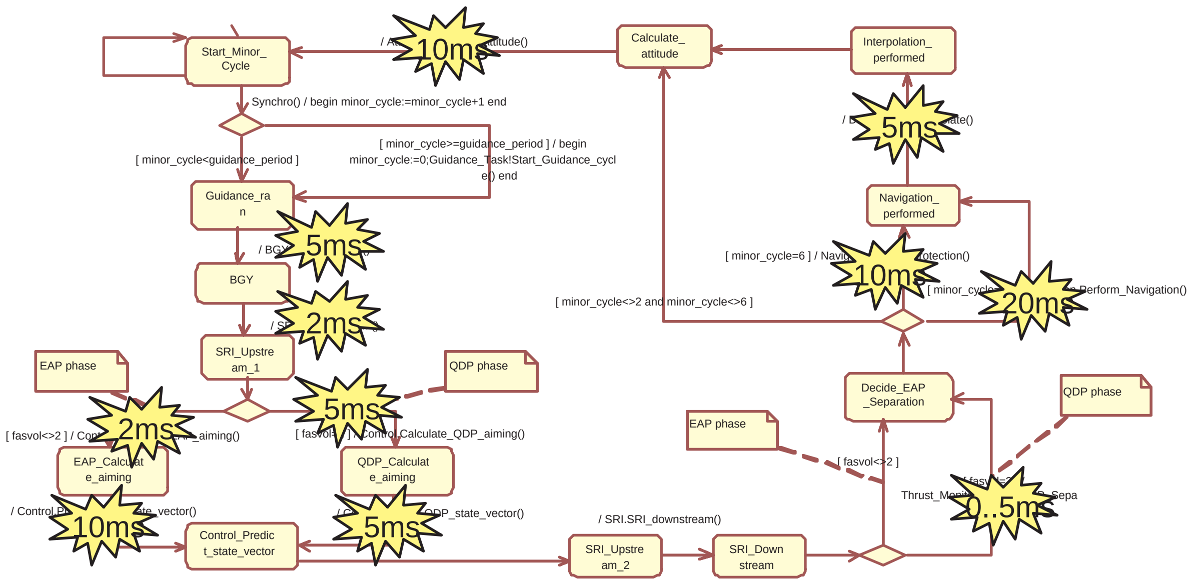
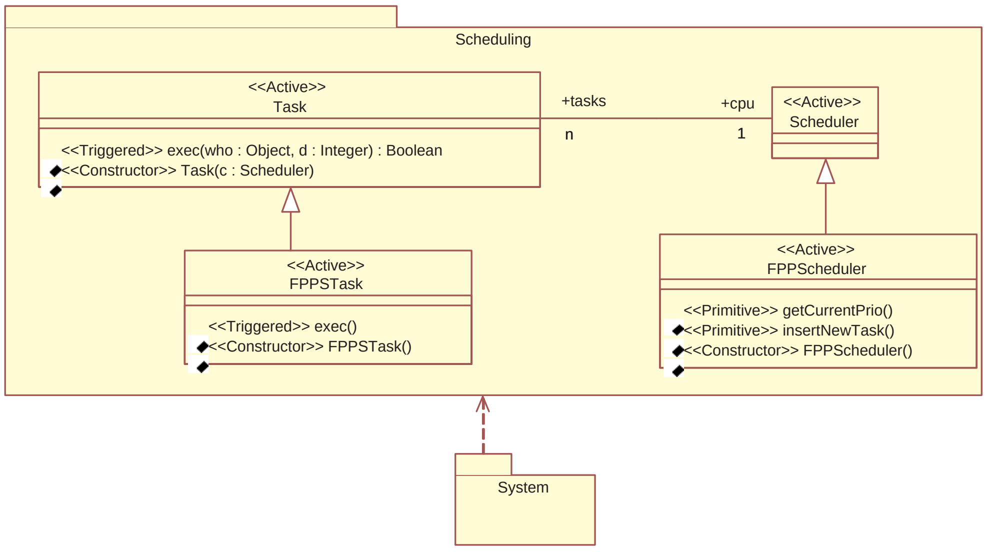


Fig. 9. Statechart of the Control cycle with unitary execution times.


Scheduling
<<Active>>
Task
<<Triggered>> exec(who : Object, d : Integer) : Boolean
<<Constructor>> Task(c : Scheduler)
+tasks
+cpu
n
1
<<Active>>
Scheduler
<<Active>>
FPPSTask
<<Triggered>> exec()
<<Constructor>> FPPSTask()
<<Active>>
FPPScheduler
<<Primitive>> getCurrentPrio()
<<Primitive>> insertNewTask()
<<Constructor>> FPPScheduler()
System
Fig. 10. Scheduling library.

3.3.3 Modeling the scheduling objectives

Scheduling objectives are modeled using observers. As mentioned in section 3.3.1, 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.

4 Validation methodology and results

Validation of UML models with the IFx toolset involves several activities supported by different tools. At a very abstract level, the workflow for using IFx is shown in Figure **??**. The validation activities, depicted in the lower

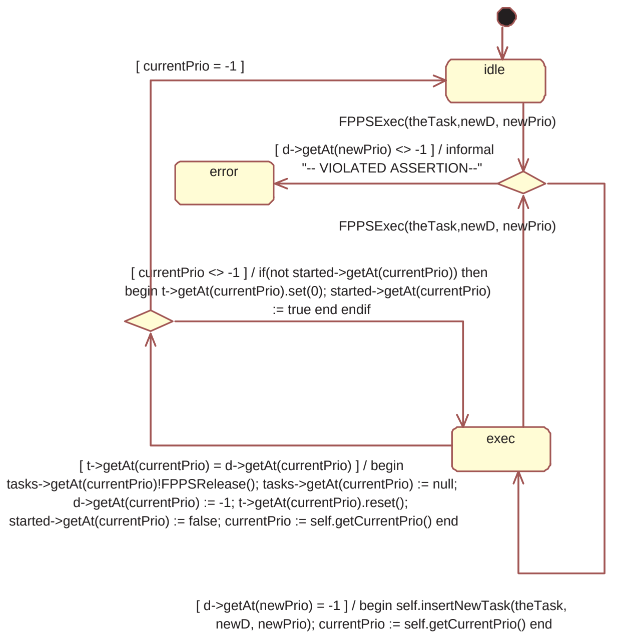

idle
[currentPrio = -1]
FPPSExec(theTask,newD, newPrio)
[d->getAt(newPrio) <> -1] / informal "-- VIOLATED ASSERTION--"
error
FPPSExec(theTask,newD, newPrio)
[currentPrio <> -1] / if(not started->getAt(currentPrio)) then begin t->getAt(currentPrio).set(0); started->getAt(currentPrio) := true end endif
exec
[t->getAt(currentPrio) = d->getAt(currentPrio)] / begin tasks->getAt(currentPrio)!FPPSRelease(); tasks->getAt(currentPrio) := null; d->getAt(currentPrio) := -1; t->getAt(currentPrio).reset(); started->getAt(currentPrio) := false; currentPrio := self.getCurrentPrio() end
[d->getAt(newPrio) = -1] / begin self.insertNewTask(theTask, newD, newPrio); currentPrio := self.getCurrentPrio() end

Fig. 11. Behavior of the fixed priority preemptive scheduler.

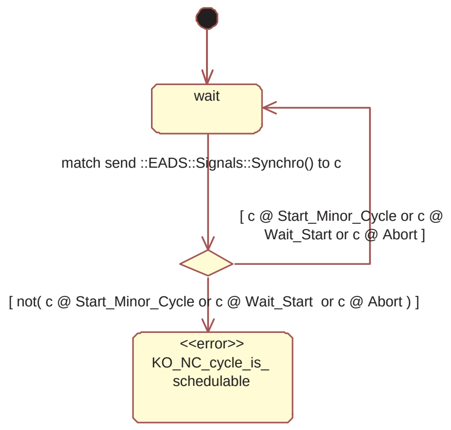

wait
match send ::EADS::Signals::Synchro() to c
[c @ Start_Minor_Cycle or c @ Wait_Start or c @ Abort]
[not(c @ Start_Minor_Cycle or c @ Wait_Start or c @ Abort)]
<<error>>
KO_NC_cycle_is_schedulable

Fig. 12. Scheduling objective: the control cycle finishes in time.

part, range from simple syntactic and static semantic checking to dynamic property verification. These activities help improving the system model and its requirements. In the following we discuss each phase in this workflow with its specific difficulties and their possible solutions.

4.1 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 also allows performing a simple static (syntactic and semantic) checking of the UML model. One can discover 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.)
– name 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.)

4.2 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* is the most useful transformation in order to optimize the IF specification for model checking, while fully preserving its behavior. This transformation introduces systematic resets 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 for example to reset clocks or other types of counters which are not useful any more from a certain execution point forward, but which have been forgot running in the specification. 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 preserves all model 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 simplify the source of the IF system and can diminish the size of the individual system states (by eliminating variables) but has no impact on the size of the state space.
– *Slicing* is used to eliminate a part of the model variables which is considered irrelevant for the verification of a certain property, together with all the variables depending on them.
  This is an abstraction technique which allows to compute an over-approximation of the behavior of a system (component). Over-approximations preserve the satisfaction of safety properties, but does not preserve their non-satisfaction (i.e. may generate false negatives).
  There is however no automatic support in IFx to guarantee that the property which has to be verified is preserved. [DL]: cette phrase est extrmement

ambige!!!!! Elle laisse sous-entendre qu'on applique des abstractions sans tre sr que les proprits spcifies sont prserves. J'espre que toutes les proprits qu'on a crites sont prserves. Il faut absolument revoir cette explication.

In the Ariane-5 case study, however, the factorization is limited due to the relatively small number of loops in the system state graph. Nevertheless factorization has been used to automatically reset forgotten clocks, especially in properties. For example, clock t in property 3 is reset upon entry in state *aborted*).

4.3 Model exploration through simulation

Simulation allows the user to explore the model in a guided or random manner, without being exhaustive. Simulation states do not need to be stored as the complete state space is not explicitly constructed at this moment. The user can also *test* simple safety properties, which must hold on all execution paths. Properties range from generic ones, such as absence of deadlocks or signal loss, to more specific and application dependent ones, e.g., invariants tested using conditional breakpoints.

The simulation tool allows performing 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 stylesheets), inserting conditional breakpoints (defined through XPath conditions on the XML trees which represents the IF system state or the fireable transitions).

The simulation tool can also be used later on for analyzing the error traces generated by the model checker when a property is violated (replay of counter-example). **XXXX??**

4.4 State space generation and observer model checking

In this phase the IF model is compiled into an executable component which can be used to exhaustively generate the state space of the system, in the form of a labeled graph. The vertices of the graph are the system states reachable during execution (including the states of observers executed in parallel with the system), and edges represent transitions and are labeled with the occurring events.

Observer model checking is performed by signaling the system states in which an observer resides in an $\ll$ *error* $\gg$ or a $\ll$ *success* $\gg$ state.

[DL]: la phrase suivante n'est pas claire. Il faut la clarifier ou la supprimer. Doit on ou pas construire l'espace d'tat complet ou pas? Les deux phrases se contredisent! Generating the whole state space is a prerequisite for positively verifying properties. However, if a goal is to quickly find errors, it is possible to obtain results without fully generating the state space since observers (and other forms of properties like μ-calculus formulas) are verified on the fly.

4.4.1 Exploration strategy

Two exploration strategies are possible:

- Depth first exploration. With this strategy, when a property violation is detected, it is possible to generate a diagnostics trace which can be debugged with the simulator.
  During depth first exploration, it is also possible to apply *partial order reduction* on the fly; this eliminates spurious interleaving between internal steps occurring in different processes at the same time. Internal steps are those which do not perform visible communication actions, neither signal emission nor access to shared variables. Partial order reduction imposes a fixed exploration order on internal steps and preserves *all* properties expressed in terms of visible action

  Example: In the Ariane-5 model, the use of partial order reduction has allowed constructing tractable models. Without this reduction, even for flight parameters that yield the simplest behaviors, the generation of the state space did not terminate (its size was greater than10^6 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, i.e. to about 1000 states for certain flight configurations.

  Depth first exploration is the recommended strategy as it also allows detecting some types of 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. [DL] le suivant est mon avis en trop. Il faut le laisser que si il y a suffisement de place, mais ne pas hsiter le supprimer sinon
- Breadth first exploration. This strategy allows to find the shortest path to a property violation. However, there are some technical limitations of this strategy: partial order reductions cannot be applied, the diagnostics trace cannot be generated in a form amenable to debugging, state space explosion cannot be detected and diagnosed early.

4.4.2 Representations for time

Orthogonally to the exploration order, a time representation scheme has to be chosen. In IFx (and consequently in Omega UML), time in a particular system state is represented by the values of active clock variables in that state. The values of these clocks may be represented either explicitly as integer values (we talk then about *discrete* time), or implicitly using linear constraints on the values (from $\mathbb{R}$) [DL]: la note de $\mathbb{R}$ n'existe pas en informatique. Mme si je comprend l'approche thorique de temps dense, je prfrerai quand mme parler de dcimaux plutt que de rels. of each pair of clocks (we talk then about *dense* or *symbolic* time). Details on the two representations and how they compare may be found in the literature on timed automata [**?**].

There are no simple methodological guidelines on when to use either one of these representations, and the choice has to be done based on trial. However, for the Ariane-5 model as well as other UML models verified with IFx, the symbolic time representation performed better both in terms of state space size and generation time.

4.4.3 Improving partial order reductions

In some cases, automatic partial order reductions are not sufficiently strong and may be improved by manually adding hypotheses on the order of execution of some actions.

For example, consider that a component A sends messages to a component B, which consumes them but also performs some visible action for each consumed message. Because of the presence of visible actions in both A and B, there will be no automatic partial order reduction. However, based on some system-specific knowledge, the designer may want to consider that B always consumes the message before A sends a new one.

4.5 Other verification techniques

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

– *Verification by graph minimization* is an intuitive method for a non expert end-user. It consists of computing an abstract model (with respect to given set of observations) of the overall behavior of the specification. Such a model can be visualised and possible incorrect behaviors detected by the user.
  In order to obtain an abstract model, the state space must be generated first by exhaustive simulation. After that, it can be minimized modulo a bisimulation using Aldebaran (a tool connected to IFx [BFKM97]), and depending on the relation that is used the minimized graph preserves different classes of properties (e.g., safety, absence of deadlocks, etc.). The minimization takes into account the observations which are relevant for the property being verified (i.e. the actions that have to remain visible).

  Example: The graph in figure 13 is the quotient model of Ariane-5 with respect to branching bisimulation [vGW96], 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 (i.e., the "i" transition).

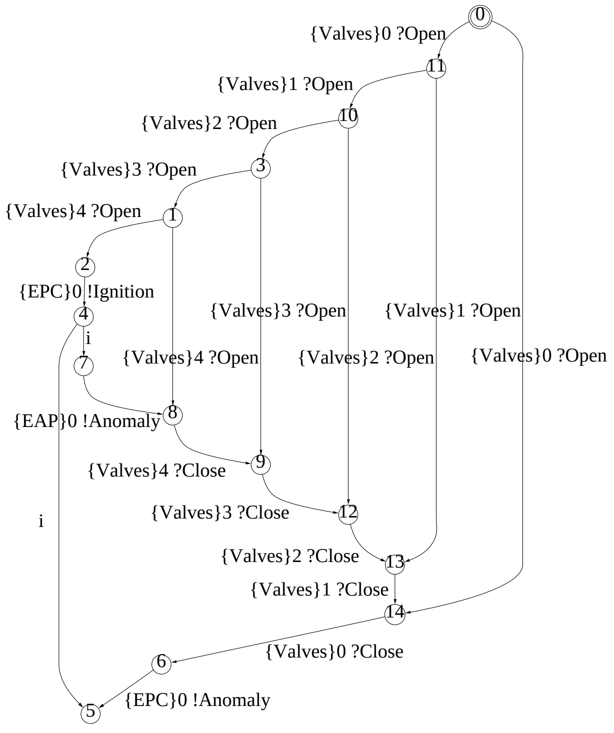


Fig. 13. A minimal model generated with Aldebaran.

– Verification of μ-calculus formulas. Alternating-free μ-calculus is a temporal logic which allows expressing properties about the branching structure of the system state graph. Its expressive power is thus incomparable to that of observers. It allows for example to express properties such as "there is no deadlock", or liveness properties such as "every action A is eventually followed by an action B".
  Using μ-calculus is very difficult for non-specialists. The formalism also presents some technical drawbacks, like the impossibility to reason quantitatively about time. Since in timed systems most liveness properties may be expressed as time-bounded safety properties (e.g., for the above property, a time-bounded version is "every action A is eventually followed by an action B within t time"), it is more convenient to use observers for most properties.

4.6 Introducing abstractions

Often the state space of a very detailed system design cannot be generated because of its big size. The designer has to use some form of abstraction in order to reduce the size while still preserving the properties that need to be validated. Some exact abstractions which preserve all properties have already been described: state factorization, partial order reduction. When these are not sufficient, the user will have to make the model itself more abstract, usually by describing an over-approximation (i.e. a model which presents all executions of the initial model and some additional ones). Such an abstraction preserves the satisfaction of safety properties, but does not preserve their non-satisfaction, i.e. it may lead to false negatives.

An example of using abstractions of this form is given in section 4.7.1. 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 and of the fact that there are fewer combinations of clock constraints that are reachable.

4.7 Ariane-5 verification results

4.7.1 Abstractions in Ariane-5

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 about 3 600 000 steps for 1 hour mission, and 15 000 000 000 steps for a 6 months mission, etc...

The state-of-the-art does not allow today proving models with a such big size. To solve this problem of explosion of the number of states, we apply different types of abstraction on this model.

– Abstraction of the cyclic behavior
The cyclic synchronous behaviour of the software has first been abstracted. The events generated by the GNC to command the booster released are generated at a non deterministic time. Using this abstract cyclical part, all the properties of the asynchonous part have been formally proved correct.
– Abstraction of the asynchronous behavior
In this second step, the asynchronous part has been abstracted. 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 of several errors.
– Abstraction of long duration
Finally, the model has been analyzed without abstraction. The problem for this last step is the difference of the timing scale between the asynchronous behaviour and the synchronous one. The asynchronous behaviour deals with durations of some milliseconds, whereas the asynchronous one deals with durations of several hours (for Ariane 5 launcher) up to some months (for ATV project) or some years (for some deep space missions).
In practice, the system is working without occurrence of any asynchronous events during a great number of basic cycles, and the most of the output of the cyclic part is irrelevant for our verification. Thus, it is sufficient to perform the proof with a mission duration much greater than the basic cycle, but not necessarily with the real mission duration.
Using such a reduction of the real duration of the mission, all the properties (synchronous and asynchronous) have been proved correct. In order to test the tool, proofs have been performed with several mission durations (see next section).

4.7.2 Results and figures

With a reduced mission time, the following table gives the duration of the proof of each property. It also gives the number of states and the number of transitions of the produced automaton.

Property	Number of states	Number of trans
liftoff_aborted_right	36037	38149
pyro_not_ignited_twice	35988	38092
valve_not_abused	36082	38210
valve_not_close_in_close	36010	38114
valve_not_open_in_open	35998	38102
liftoff_performed_right1	46075	48713
liftoff_performed_right2	37897	40550
liftoff_performed_right3	37961	40632
liftoff_performed_right4	35986	38090
CPU_not_in_error	35980	38084
G_cycle_is_schedulable	36012	38116
NC_cycle_is_schedulable	36380	38484
read_write_coherence	36618	38722

We have also tried to verify all the properties at the same time by running all observers in parallel with the model. The following array gives the same information as above for different mission durations.

Mission duration	Number of states	Number of transitions
7 000 ms	51 324	54 697
15 000 ms	161 956	171 734
22 000 ms	303 496	321 206
30 000 ms	463 932	490 901
37 000 ms	658 981	696 031

5 Lessons learned and future work

References

References

[AGS00] K. Altisen, G. Gössler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[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*, LNCS, June 2004.

[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.

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

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

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

[vGW96] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.