Tools and Applications II: The IF Toolset*

Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis

VERIMAG, 2 avenue de Vignate, F-38610 Gières

Abstract. This paper presents an overview on the IF toolset which is an environment for modelling and validation of heterogeneous real-time systems. The toolset is built upon a rich formalism, the IF notation, allowing structured automata-based system representations. Moreover, the IF notation is expressive enough to support real-time primitives and extensions of high-level modelling languages such as SDL and UML by means of structure preserving mappings.
The core part of the IF toolset consists of a syntactic transformation component and an open exploration platform. The syntactic transformation component provides language level access to IF descriptions and has been used to implement static analysis and optimisation techniques. The exploration platform gives access to the graph of possible executions. It has been connected to different state-of-the-art model-checking and test-case generation tools.
A methodology for the use of the toolset is presented at hand of a case study concerning the Ariane-5 Flight Program for which both an SDL and a UML model have been validated.

1 Introduction

Modelling plays a central role in systems engineering. The use of models can profitably replace experimentation on actual systems with incomparable advantages such as:

- ease of construction by integration of heterogeneous components,
- generality by using genericity, abstraction, behavioural non determinism
- enhanced observability and controllability, especially avoidance of probe effect and of disturbances due to experimentation
- finally, possibility of analysis and predictability by application of formal methods.

Building models which faithfully represent complex systems is a non trivial problem and a prerequisite to the application of formal analysis techniques. Usually, modelling techniques are applied at early phases of system development and at high abstraction level. Nevertheless, the need of a unified view of the various life-cycle activities and of their interdependencies, motivated recently, the so called model-based development [OMG03a,Sif01,STY03] which heavily

* This work was supported in part by the European Commission through the projects IST-1999-29082 ADVANCE, IST-1999-20218 AGEDIS and IST-2001-33522 OMEGA

relies on the use of modelling methods and tools to provide support and guidance for system design and validation.

Currently, validation of real-time systems is done by experimentation and measurement on specific platforms in order to adjust design parameters and hopefully achieve conformity to QoS requirements. Model based development intends to replace experimentation on real prototypes by validation on virtual prototypes (models). Furthermore, a key idea is the use of successive model transformations in design methodologies to derive from some initial high level description low level descriptions close to implementations. Achieving such ambitious goals raises hard and not yet completely resolved problems discussed in this section.

Heterogeneity. A real-time system is a layered system consisting of an application software implemented as a set of interacting tasks, and of the underlying execution platform. It continuously interacts with an external environment to provide a service satisfying QoS requirements characterising the dynamics of the interaction. Models of real-time systems should represent faithfully interactive behaviour taking into account implementation choices related to resource management and scheduling as well as execution speed of the underlying hardware

The models of real-time systems involve heterogeneous components with different execution speeds and interaction modes. There exist two main sources of heterogeneity: interaction and execution.

Heterogeneity of interaction results from the combination of different kinds of interaction.

Interactions can be *atomic* or *non atomic*. The result of atomic interactions cannot be altered through interference with other interactions. Process algebras and synchronous languages assume atomic interactions. Asynchronous communication (SDL, UML) or method call are generally non atomic interactions. Their initiation and their completion can be separated by other events.

Interactions can involve *strict* or *non strict* synchronisation. For instance, rendez-vous and method calls require strict interactions. On the contrary, broadcast of synchronous languages and asynchronous communication do not need strict synchronisation. A process (sender) can initiate an interaction independently of the possibility of completion by its environment.

Heterogeneity of execution results from the combination of two execution paradigms.

Synchronous execution is typically adopted in hardware, in synchronous languages, and in time triggered architectures and protocols. It considers that a system execution is a sequence of steps. It assumes synchrony, meaning that the system's environment does not change during a step, or equivalently "that the system is infinitely faster than its environment". The synchronous paradigm has a built-in strong assumption of fairness: in a step all the system components execute a quantum computation defined by using either quantitative or logical time.

The *asynchronous* paradigm does not adopt any notion of global execution step. It is used in languages for the description of distributed systems such as SDL and UML, and programming languages such as Ada and Java. The lack of built-in mechanisms for sharing resources between components can be compensated through scheduling. This paradigm is also common to all execution platforms supporting multiple threads, tasks, etc.

Modelling time. Models for real-time systems should allow modelling progress of time in order to express various kinds of timing information e.g., execution times of actions, arrival times of events, deadlines, latency.

Timed models can be defined as extensions of untimed models by adding time variables used to measure the time elapsed since their initialisation. They can be represented as machines that can perform two kinds of state changes: actions and time steps. Actions are timeless state changes of the untimed system; their execution may depend on and modify time variables. In a time step, all time variables increase uniformly. There exists a variety of timed formalisms extensions of Petri nets [Sif77], process algebras [NS91] and timed automata [AD94]. Any executable untimed description e.g., application software, can be extended into a timed one by adding explicitly time variables or other timing constraints.

Timed models use a notion of logical time. Contrary to physical time, logical time progress can block, especially as a result of inconsistency of timing constraints. The behaviour of a timed model is characterised by the set of its runs, that is the set of maximal sequences of consecutive states reached by performing transitions or time steps. The time elapsed between two states of a run is computed by summing up the durations of all the time steps between them. For a timed model to represent a system, it is necessary that it is *well-timed* in the sense that in all its runs time diverges.

As a rule, in timed models there may exist states from which time cannot progress. If time can progress from any state of a timed model, then it is always possible to wait and postpone the execution of actions which means that it is not possible to model action *urgency*. Action urgency at a state is modelled by disallowing time progress. This possibility of stopping time progress goes against our intuition about physical time and constitutes a basic difference between the notions of physical and logical time. It has deep consequences on timed systems modelling by composition of timed components.

Often timed extensions of untimed systems are built in an ad hoc manner at the risk of producing over-constrained or incomplete descriptions. It is essential to develop a methodology for adding compositionally timing information to untimed models to get a corresponding timed model.

The IF toolset is an environment for modelling and validation of heterogeneous real-time systems. It is characterised by the following features:

– Support for high level modelling with formalisms such as SDL, UML used by users in some CASE tool. This is essential to ease usability by practitioners and to allow the use of state-of-the-art modelling technology. Furthermore, the use of high level formalisms allows validating realistic models which can be simplified if necessary by using automated tools. This avoids starting with

simplified models constructed in an ad hoc manner as it is the case for other tools using low level description languages e.g., automata.
- Translation of high level models into an intermediate representation, the IF notation, that serves as a semantic model. This representation is rich and expressive enough to describe the main concepts and constructs of source languages. It combines composition of extended timed automata and dynamic priorities to encompass heterogeneous interaction. Priorities play an important role for the description of scheduling policies as well as the restriction of asynchronous behaviour to model run-to-completion execution. We consider a class of timed automata which are by construction well-timed.
  The developed translation methods for SDL and UML preserve the overall structure of the source model and the size of the generated IF description increases linearly with the size of the source model. IF is used as a basis for model simplification by means of static analysis techniques and the application of light structural analysis techniques e.g., checking sufficient conditions for deadlock-freedom of processes. It is also used for the generation of lower level models e.g., labelled transitions systems used for verification purposes.
- Combined use of various validation techniques including model-checking, static analysis on the intermediate representation and simulation. A methodology has been studied at Verimag for complex real-time applications.
- Expression of requirements to be validated on models by using *observers*. These can be considered as a special class of models equipped with primitives for monitoring and checking for divergence from some nominal behaviour. Our choice for monitors rather than declarative formalisms such as temporal logic or Live Sequence charts [DH99] is motivated by our concern to be close to industrial practice and to avoid as much as possible inconsistency in requirements.

The paper is organised as follows. Section 2 presents the overall architecture of the IF toolset. Section 3 is the main section of the paper. It starts with a presentation of IF including its main concepts and constructs and their semantics. Then the overall architecture of the toolset and its features for simulation, analysis and validation are described. Finally the translation principle from UML to IF is explained by showing how the main UML concepts and constructs can be mapped into IF.

Section 4 presents an example illustrating the application of the toolset to the modelling and validation of the Ariane-5 Flight Program. For this non trivial case study, we provide a validation methodology and results. Section 5 presents concluding remarks about the toolset and the underlying modelling and validation methodology.

2 Setting the context - the overall architecture

Figure 1 describes the overall architecture of the toolset, the most important components as well as their inter-connections. We distinguish three different de-

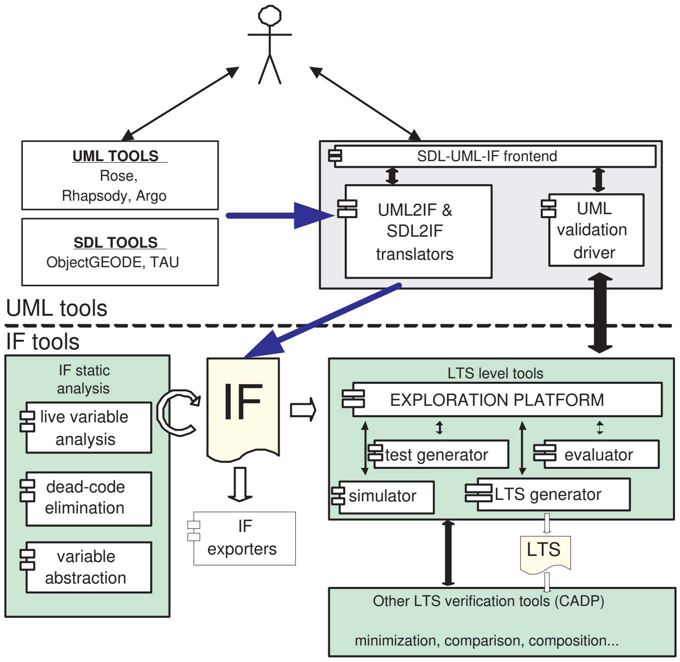

Fig. 1. IF toolset architecture.

scription levels: the *specification level* (UML, SDL), the *intermediate description level (IF)*, and the Labelled Transition Systems (LTS) level.

Specification level. This corresponds to the description provided by the user in some existing specification language. To be processed, descriptions are automatically translated into their IF descriptions. Currently, the main input specification formalisms are UML and SDL.

Regarding UML, any UML tool can be used as long as it can export the model in XMI [OMG01], the standard XML format. The IF toolset includes a translator from UML which produces IF descriptions. The translator accepts specifications produced by Rational Rose [IBM], Rhapsody [Ilo] or Argo Uml [RVR+].

Intermediate description level (IF). IF descriptions are generated from specifications. IF is an intermediate representation based on timed automata extended with discrete data variables, communication primitives, dynamic process creation and destruction. This representation is expressive enough to describe the basic concepts of modelling and programming languages for distributed real-time systems.

The abstract syntax tree of an IF description can be accessed through an API. Since all the data (variables, clocks) and the communication structure are still explicit, high-level transformations based on static analysis [Muc97] or program slicing [Wei84,Tip94] can be applied. All these techniques can be used to transform the initial IF description into a "simpler" one while preserving safety properties. Moreover, this API is well-suited to implement exporters from IF to other specification formalisms.

LTS level. The LTS are transition graphs describing the executions of IF descriptions. An *exploration API* allows to represent and store states as well as to compute on demand the successors of a given state. This API can be linked with "generic" exploration programs performing any kind of *on-the-fly* analysis.

Using the exploration API, several validation tools have been developed and connected to work on IF descriptions. They cover a broad range of features: interactive/random/guide simulation, on-the-fly model checking using observers, on-the-fly temporal logic model checking, exhaustive state space generation, scheduling analysis, test case generation. Moreover, through this API are connected the CADP toolbox [FGK+96] for the validation of finite models as well as TGV [FJJV96,JM99] for test case generation using on-the-fly techniques.

3 Description of the formalism/technique/system/tool

3.1 The IF notation

IF is a notation for systems of components (called *processes*), running in parallel and interacting either through shared variables or asynchronous signals. Processes describe sequential behaviours including data transformations, communications and process creation. Furthermore, the behaviour of a process may be subject to timing constraints. The number of processes may change over time: they may be created and deleted dynamically.

The semantics of a system is the LTS obtained by interleaving the behaviour of its processes. To enforce scheduling policies, the set of runs of the LTS can be further restricted using dynamic priorities.

Processes. The behaviour of a *process* is described as a timed automaton, extended with data. A process has a unique process identifier (*pid*) and local memory consisting of variables (including clocks), control states and a queue of pending messages (received and not yet consumed).

A process can move from one control state to another by executing some *transition*. As for state charts [Har87,HP98], control states can be hierarchically structured to factorize common behaviour. Control states can be *stable* or *unstable*. A sequence of transitions between two stable states defines a *step*. The execution of a step is *atomic*, meaning that it corresponds to a single transition in the LTS representing the semantics. Notice that several transitions may be enabled at the same time, in which case the choice is made non-deterministically.

Transitions can be either *triggered* by signals in the input queue or be *spontaneous*. Transitions can also be *guarded* by predicates on variables, where a guard is the conjunction of a data guard and a time guard. A transition is enabled in a state if its trigger signal is present and its guard evaluates to true. Signals in the input queue are a priori consumed in a fifo fashion, but one can specify in transitions which signals should be "*saved*" in the queue for later use.

Transition *bodies* are *sequential programs* consisting of elementary actions (variable or clock assignments, message sending, process creation/destruction, resource requirement/release, etc) and structured using elementary control-flow statements (like if-then-else, while-do, etc). In addition, transition bodies can use external functions/procedures, written in an external programming language (C/C++).

Signals and signalroutes. *Signals* are typed and can have data parameters. Signals can be addressed directly to a process (using its *pid*) and/or to a signal route which will deliver it to one or more processes. The destination process stores received signals in a fifo buffer.

Signalroutes represent specialised communication media transporting signals between processes. The behaviour of a signalroute is defined by its delivery policy (FIFO or multi-set), its connection policy (peer to peer, unicast or multicast), its delaying policy ("zero delay", "delay" or "rate") and finally its reliability ("reliable" or "lossy"). More complex communication media can be specified explicitly as IF processes.

In particular, signalroutes can be connected at one end with an implicitly defined "environment process" **env**. In transitions triggered by signals from the environment, the trigger signal is considered as present whenever the transition guard evaluates to true.

Data. The IF notation provides the *predefined basic types* bool, integer, real, pid and clock, where *clock* is used for variables measuring time progress. Structured data types are built using the *type constructors* enumeration, range, array, record and abstract. Abstract data types can be used for manipulating external types and code.

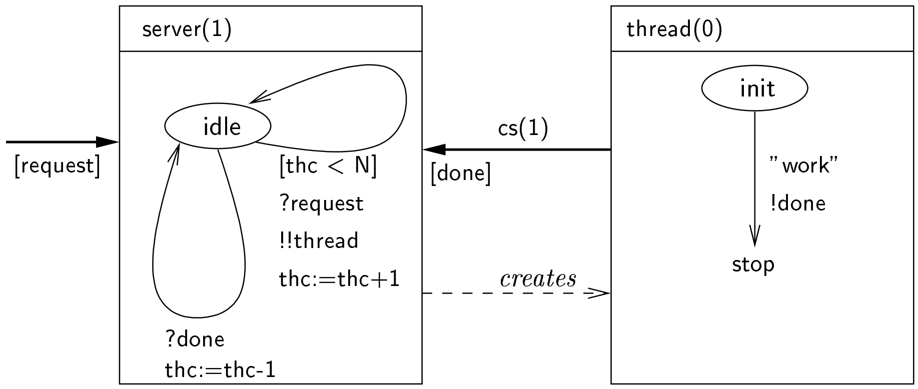

Fig. 2. Illustration of the multi-threaded server example.

Example 1. The IF description below describes a system consisting of a `server` process creating up to N `thread` processes for handling `request` signals. A graphical representation of the system is given in Figure 2.

```
system Server;
signal request();              // signals with parameter types
signal done(pid);

signalroute entry(1)           // signalroutes and their signals
   from env to server
   with request;

signalroute cs(1) #delay[1,2]
   from thread to server
   with done;
                               // definition of process types
process thread(0);             // and initial number of instances
  fpar parent pid, route pid;  // formal parameters received at creation

   state init #start ;         // 1 state + 1 outgoing transition
     informal "work";          // informal action labelled "work"
     output done()             // sending of the done signal
      via route to parent;     // received by parent
     stop;                     // terminate process, destroy instance
  endstate;
endprocess;

process server(1);
  var thc integer;             // local variables
  state idle #start ;          // 1 state + 2 outgoing transitions
    provided thc < N;          // first transition: guard
    input request();           // trigger
      fork thread(self, {cs}0); // create thread process and passing
                               // own pid and signalroute cs as params
      task thc := thc + 1;
      nextstate -;             // end of transition - back to idle

    input done();              // second transition
      task thc := thc - 1;
      nextstate -;
  endstate;
endprocess;
endsystem;
```

Composition (System). The semantics associates with a system a global LTS. At any point of time, its state is defined as the tuple of the states of its living components: the states of a process are the possible evaluations of its attributes (control state, variables and signal queue content). The states of a signalroute are lists of signals "*in transit*". The transitions of the global LTS representing a system are steps of processes and signal deliveries from signalroutes to signal queues where in any global state there is an outgoing transition for all enabled transitions of all components (interleaving semantics). The formal definition of the semantics can be found in [BL02b].

System models may be highly nondeterministic, due to the nondeterminism of the environment which is considered as open and to the concurrency between their processes. For the validation of functional properties, leaving this second type of nondeterminism non resolved is important in order to verify correctness independently of any particular execution order. Nevertheless, going towards an implementation means resolving a part of this non determinism and choosing an execution order satisfying time related and other nonfunctional constraints.

In IF, such additional restrictions can be enforced by dynamic priorities defined by rules specifying that whenever for two process instances some condition (state predicate) holds, then one has less priority than the other. An example is

$$p1 \prec p2 \text{ if } p1.group = p2.group \text{ and } p2.counter < p1.counter$$

which for any process instances which are part of some "*group*", gives priority to those with the smallest values of the variable *counter* (e.g., the less frequently served).

Time. The time model of IF is that of *timed automata with urgency* [BST98], [BS00] where the execution of a transition is an *event* defining an *instant* of state change, whereas time is progressing in states. Urgency is expressed by means of an *urgency* attribute of transitions. This attribute can take the values *eager*, *lazy* or *delayable*. *Eager* transitions are executed at the point of time at which they become enabled - if they are not disabled by another transition. *Delayable* transitions cannot be disabled by time progress. *Lazy* transitions may be disabled by time progress.

Like in timed automata, time distances between events are measured by variables of type "clock". Clocks can be created, set to some value or reset (deleted) in any transition. They can be used in time guards to restrict the time points at which transitions can be taken.

Local clocks allow the specification of timing constraints, such as durations of tasks (modelled by time passing in a state associated with this task, see example below), deadlines for events in the same process. Global time constraints, such as end-to-end delays, can be expressed by means of global clocks or by observers (explained in the next section).

Example 2. A timed version of the `thread` process of the example 1 is given. An extra state `work` introduced for distinguishing the instant at which work starts and the instant at which it ends and and to constrain the duration between them. The intention is to model an execution time of "*work*" of 2 to 4 time units.

The *thread* process goes immediately to the `work` state - the start transition is **eager** - and sets the clock *wait* is set to 0 in order to start measuring time progress. The transition exiting the `work` state is **delayable** with a time guard expressing the constraint that the time since the clock `wait` has been set should be at least 2 but not more than 4.

```
process thread(0);
  fpar parent pid, route pid;
  var wait clock;
  state init #start ;
    urgency eager;
    informal "work";
    set wait := 0;
      nextstate work;
  endstate;
```

```
state work ;
  urgency delayable;
  when wait >= 2 and wait <= 4;
    output done()
      via route to parent;
    stop;
  endstate;
endprocess;
```

Resources. In order to express mutual exclusion it is possible to declare shared *resources*. These resources can be used through particular actions of the form "**require** `some-resource`" and "**release** `some-resource`".

Observers. Observers express in an operational way safety properties of a system by characterising its acceptable execution sequences. They also provide a simple and flexible mechanism for controlling model generation. They can be used to select parts of the model to explore and to cut off execution paths that are irrelevant with respect to given criteria. In particular, observers can be used to restrict the environment of the system.

Observers are described in the same way as IF processes i.e., as extended timed automata. They differ from IF processes in that they can react *synchronously* to events and conditions occurring in the observed system. Observers are classified into:

- *pure* observers - which express requirements to be checked on the system.
- *cut* observers - which in addition to monitoring, guide simulation by selecting execution paths. For example, they are used to restrict the behaviour of the environment
- *intrusive* observers - which may also alter the system's behaviour by sending signals and changing variables.

Observation and intrusion mechanisms. For monitoring the system *state*, observers can use primitives for retrieving values of *variables*, the *current state* of the processes, the contents of *queues*, etc.

For monitoring *actions* performed by a system, observers use constructs for retrieving events together with data associated with them. Events are generated whenever the system executes one of the following actions: signal output, signal delivery, signal input, process creation and destruction and informal statements.

Observers can also monitor time progress, by using their own clocks or by monitoring the clocks of the system.

Expression of properties. In order to express properties, observer states can be marked as *ordinary*, *error* or *success*. *Error* and *success* are both terminating states. Reaching a success state (an error state) means satisfaction (non satisfaction). *Cut* observers use a *cut* action which stops exploration.

Example 3. The following example illustrates the use of observers to express a simple safety property of a protocol with one transmitter and one receiver, such as the alternating bit protocol. The property is: *Whenever a* ***put(m)*** *message is received by the* ***transmitter*** *process, the* ***transmitter*** *does not return to state* ***idle*** *before a* ***get(m)*** *with the same m is issued by the* ***receiver*** *process.*

```
pure observer safety1;
  var m data;
  var n data;
  var t pid;
  state idle #start ;
    match input put(m) by t;
      nextstate wait;
  endstate;
  state wait;
    provided ({transmitter}t)
      instate idle;
      nextstate err;
    match output put(n)
      nextstate err;
    match output get(n);
      nextstate decision;
  endstate;
  state decision #unstable ;
    provided n = m;
      nextstate idle;
    provided n <> m;
      nextstate wait;
  endstate;
  state err #error ;
  endstate;
endobserver;
```

3.2 Simulation, Analysis and Validation

Core components of the IF toolset. The core components of the IF toolset are shown in Figure 3.

Syntactic Transformations Component. This component deals with syntactic transformations including the construction of an abstract syntax tree (AST) from an IF description. The tree is a collection of C++ objects representing all the syntactic elements present in IF descriptions. The AST reflects precisely the syntactic structure of IF descriptions: a system includes processes, signalroutes, types; a process includes states and variables; states include their outgoing transitions and so on.

This component has an interface giving access to the abstract syntax tree. Primitives are available to traverse the tree and to consult or to modify its elements. There are primitives allowing to write the tree back as an IF textual description. The syntactic transformation component has been used to build several applications. The most important ones are code generators (either simulation code or application code), static analysis transformations (operating at syntactic level), translations to other languages (including a translation to the Promela language of SPIN [Hol91]) and pretty printers.

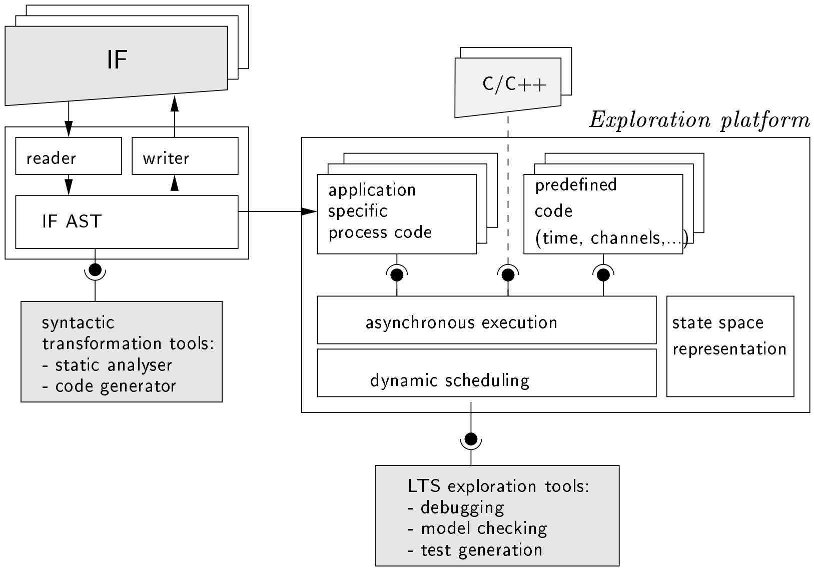

Fig. 3. Functional view of the IF Core Components.

Exploration Platform. This component has an API providing access to the LTS corresponding to IF descriptions. The interface offers primitives for representing and accessing states and labels as well as basic primitives for traversing LTS: an *init* function which gives the initial state, and a *successor* function which computes the set of enabled transitions and successor states from a given state. These are the key primitives for implementing any on-the-fly forward enumerative exploration or validation algorithm.

Figure 3 shows the structure of the exploration platform. The main features of the platform are simulation of the process execution, non-determinism resolution, management of time and representation of the state space.

The exploration platform can be seen as an operating system where process instances are plugged-in and jointly executed. Process instances are either application specific (coming from IF descriptions) or generic (such as time or channel handling processes).

Simulation time is handled by a specialised process managing clock allocation/deallocation, computing time progress conditions and firing timed transitions. There are two implementations available, one for discrete time and one for dense time. For discrete time, clock values are explicitly represented by integers. Time progress is computed with respect to the next enabled deadline. For dense time, clock valuations are represented using variable-size Difference Bound Matrices (DBMs) as in tools dedicated to timed automata such as Kronos [Yov97] and Uppaal [LPY98].

The exploration platform composes all active processes and computes global states and the corresponding system behaviour. The exploration platform consists of two layers sharing a common state representation:

- *Asynchronous execution layer.* This layer implements the general interleaving execution of processes. The platform asks successively each process to execute its enabled steps. During a process execution, the platform manages all inter-process operations: message delivery, time constraints checking, dynamic creation and destruction, tracking of events. After a completion of a step by a process, the platform takes a snapshot of the performed step, stores it and delivers it to the second layer.
- *Dynamic scheduling layer.* This layer collects all the enabled steps. It uses a set of dynamic priority rules to filter them. The remaining ones, which are maximal with respect to the priorities, are delivered to the user application via the exploration API.

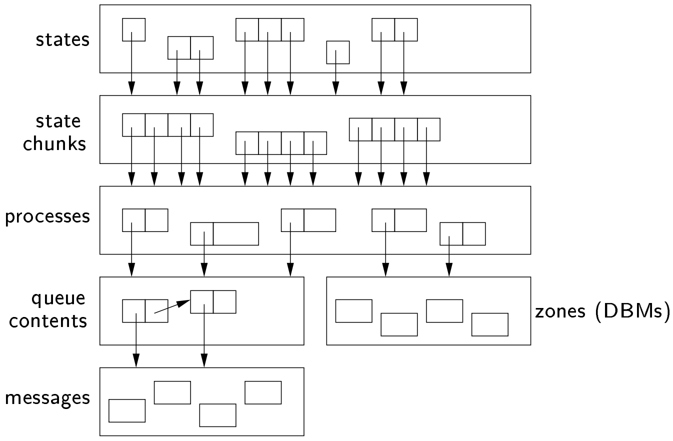

Fig. 4. Internal state representation.

- *State representation.* Global states are implicitly stored by the platform. The internal state representation is shown in figure 4. It preserves the structural information and seeks for maximal sharing. The layered representation involves a unique table of messages. Queues are lists of messages, represented by suffix sharing. On top of them, there is a table of process states, all of them sharing queues in the table of queues. Processes are then grouped into fixed size state chunks, and finally, global states are variable-size lists of chunks. Tables can be represented either by using hash tables with collision or by binary trees. This scheme allows to explicitly represent several millions of structured states.

The exploration platform and its interface has been used as back-ends of debugging tools (interactive or random simulation), model checking (including exhaustive model generation, on the fly μ-calculus evaluation, model checking with observers), test case generation, and optimisation (shortest path computation).

This architecture provides features for validating heterogeneous systems. Exploration is not limited to IF descriptions: all kinds of components with an adequate interface can be executed in parallel on the exploration platform. It is indeed possible to use C/C++ code (either directly, or instrumented accordingly) of already implemented components.

Another advantage of the architecture is that it can be extended by adding new interaction primitives and exploration strategies. Presently, the exploration platform supports asynchronous (interleaved) execution and asynchronous point-to-point communication between processes. Different execution modes, like synchronous or run-to-completion, or additional interaction mechanisms, such as broadcast or rendez-vous, are obtained by using dynamic priorities [AGS00].

Concerning the exploration strategies, reduction heuristics such as partial-order reduction or some form of symmetry reduction are already incorporated in the exploration platform. More specific heuristics may be added depending on a particular application domain.

Static Analysis. Practical experience with IF has shown that simplification by means of static analysis is crucial for dealing successfully with complex specifications. Even simple analysis such as live variables analysis or dead-code elimination can significantly reduce the size of the state space of the model. The available static analysis techniques are:

Live variables analysis This technique transforms an IF description into an equivalent smaller one by removing globally dead variables and signal parameters and by *resetting* locally dead variables [Muc97]. Initially, all the local variables of the processes and signal parameters are considered to be dead, unless otherwise specified by the user. Shared variables are considered to be always live. The analysis alternates local (standard) live variables computation on each process and inter-process liveness attributes propagation through input/output signal parameters until a global fixpoint is reached.

Dead-code elimination. This technique transforms an IF description by removing unreachable control states and transitions under some user-given assumptions about the environment. It solves a simple static reachability problem by computing, for each process separately, the set of control states and transitions which can be statically reached starting from the initial control state. The analysis computes an upper approximation of the set of processes that can be effectively created.

Variable abstraction. This technique allows to compute abstractions by eliminating variables and their dependencies which are not relevant to the user. The computation proceeds as for live variables analysis: processes are analysed separately, and the results obtained are propagated between them by using the input/output dependencies. Contrary to the previous techniques which are exact, simplification by variable abstraction may introduce additional behaviours. Nevertheless, it always reduces the size of the state representation.

By using variable abstraction it is possible to extract automatically system descriptions for symbolic verification tools accepting only specific types of data e.g., TReX [ABS01] which accepts only counters, clocks and queues. Moreover, this technique allows to compute finite-state abstractions for model checking.

Validation components.

Model-checking using EVALUATOR. The EVALUATOR tool implements an on-the-fly model checking algorithm for the alternation free μ-calculus [Koz83]. This is a branching time logic, based upon propositional calculus with fixpoint operators. The syntax is described by the following grammar:

$$\varphi ::= T \mid X \mid \neg\varphi \mid \varphi \wedge \varphi \mid\ < a > \varphi \mid \mu X.\varphi$$

For a given LTS representing a specification, the semantics of a formula is defined as the set of states satisfying it, as follows:

- T (true) holds in any state
- $\neg$ and $\wedge$ are the usual boolean operators
- $< a > \varphi$ is true in a state if there exists a transition labelled by a leading to a state which satisfies φ
- $\mu X.\varphi$ denotes the usual least fix point operator (where X is a free variable of φ representing a set of states)

This logic can be used to define macros expressing usual requirements such as: "there is no deadlock", "any action a is eventually followed by an action b", "it is not possible to perform an action a followed by an action b, without performing an action c in between", etc.

Comparison or minimisation with ALDEBARAN. ALDEBARAN [BFKM97] is a tool for the comparison of LTS modulo behavioural preorder or equivalence relations. Usually, one LTS represents the system behaviour, and the other its requirements. Moreover, ALDEBARAN can also be used to reduce a given LTS modulo a behavioural equivalence, possibly by taking into account an observation criterion.

The preorders and equivalences available in ALDEBARAN include usual simulation and bisimulation relations such as strong bisimulation [Par81], observational bisimulation [Mil80], branching bisimulation [vGW89], safety bisimulation [BFG$^+$91], etc. The choice of the relation depends on the class of properties to be preserved.

Test case generation using TGV. TGV [FJJV96,JM99] is a tool for test generation developed by IRISA and VERIMAG. It is used to automatically generate test cases for conformance testing of distributed reactive systems. It generates test cases from a formal specification of the system and a test purpose.

3.3 Translating UML to IF

The toolset supports generation of IF descriptions from both SDL [BFG+99] and UML [OGO04]. We describe the principles of the translation from UML to IF.

UML modelling. We consider a subset of UML including its object-oriented features and which is expressive enough for the specification of real-time systems. The elements of models are classes with structural features and relationships (associations, inheritance) and behaviour descriptions through state machines and operations.

The translation tool adopts a particular semantics for concurrency based on the UML distinction between active and passive objects. Informally, a set of passive objects form together with an active object an *activity group*. Activity groups are executed in run-to-completion fashion, which means that there is no concurrency between the objects of the same activity group. Requests (asynchronous signals or method calls) coming from outside an activity group are queued and treated one by one. More details on this semantics can be found in [DJPV02,HvdZ03].

The tool resolves some choices left open by UML, such as the concrete syntax of the action language used in state machines and operations.

Additionally, we use a specialisation of the standard UML profile for Scheduling, Performance and Time [OMG03b]. Our profile, formally described in [GOO03], provides two kinds of mechanisms for timing: imperative mechanisms including timers, clocks and timed transition guards, and declarative mechanisms including linear constraints on time distances between events.

To provide connectivity with existing CASE tools such as RATIONAL ROSE [IBM], RHAPSODY [Ilo] or ARGO UML [RVR+], the toolset reads models using the standard XML representation for UML (XMI [OMG01]).

The principles of the mapping from UML to IF. Runtime UML entities (objects, call stacks, pending messages, etc.) are identifiable as a part of the system state in IF. This allows tracing back to UML specifications from simulation and verification.

Objects and concurrency model. Every UML class X is mapped to a process P_X with a local variable for each attribute or association of X. As inheritance is flattened, all inherited attributes and associations are replicated in the processes corresponding to each subclass. The class state machine is translated into the process behaviour.

Each activity group is managed at runtime by a special IF process, of type *group manager*, which is responsible of sequentialising requests coming from objects outside the activity group, and of forwarding them to the objects inside when the group is stable. Run-to-completion is implemented by using the dynamic priority rule

$$y \prec x \text{ if } x.leader = y$$

which means that *all objects of a group have higher priorities than their group manager*. For every object x, $x.leader$ points to the manager process of the object's activity group. Thus, as long as at least one object inside an activity group can execute, its group manager will not initiate a new run-to-completion step. Notice that adopting a different execution mode can be done easily by just eliminating or adding new priority rules.

Operations and polymorphism. The adopted semantics distinguishes between *primitive operations* - described by a method with an associated action - and *triggered operations* - described directly in the state machine of their owner class. Triggered operations are mapped to actions embedded directly in the state machine of the class.

Each primitive operation is mapped to a handler process whose run-time instances represent the activations and the stack frames corresponding to calls.

An operation call (either primitive or triggered) is expressed in IF by using three signals: a *call* signal carrying the call parameters, a *return* signal carrying the return value, and a *completion* signal indicating completion of computation of the operation, which may be different from *return*. Therefore, the action of invoking an operation is represented in IF by sending a *call* signal. If the caller is in the same activity group, then the *call* is directed to the target object and is handled immediately. Alternatively, if the caller is in a different group, the *call* is directed to the object's *group manager* and is handled in a subsequent run-to-completion step.

The handling of incoming primitive calls by an object is modelled as follows: in every state of the callee object (process), upon reception of a call signal, the callee creates a new instance of the operation's handler. The callee then waits until completion, before re-entering the same stable state in which it received the call.

Mapping operation activations into separate processes has several advantages:

– It provides a simple solution for handling *polymorphic* (dynamically bound) calls in an inheritance hierarchy. The receiver object knows its own identity, and can answer any *call* signal by creating the appropriate version of the operation handler from the hierarchy.
– It allows for extensions to other types of calls than the ones currently supported by the semantics (e.g. non-blocking calls). It also preserves modularity and readability of the generated model.
– It allows to distinguish the relevent instants in the context of timing analysis.

Mapping of UML observers. In order to specify and verify dynamic properties of UML models, we define a notion of *UML observer* [OGO04] which is similar to IF observers (see section 3.1).

Observers are described by classes stereotyped with ≪*observer*≫. They can own attributes and methods, and can be created dynamically. We defined in [OGO04] event types such as operation invocation, operation return, object creation, etc.

Several examples of observers are provided in section 4.3.

Mapping of real-time concepts. The mapping of UML timers and clocks to IF is straightforward. Declarative constraints on duration between events are expressed by means of clocks and time guards or observers [OGO04].

4 An example: the Ariane-5 Flight Program[1]

We present a real-world case study on the modelling and validation of the Flight Program of Ariane-5 by using the IF toolset.

This work has been initiated by EADS Launch Vehicles in order to evaluate the maturity and applicability of formal validation techniques. This evaluation consisted in formally specifying some parts of an existing software, on a re-engineering basis, and verifying some critical requirements on this specification. The Ariane-5 Flight Program is the embedded software which autonomously controls the Ariane-5 launcher during its flight, from the ground, through the atmosphere, and up to the final orbit.

The specification and validation have been studied in two different contexts:

- A first study carried out on a re-engineered SDL model has been conducted in 2001. The SDL model was translated automatically to IF, simplified by static analysis, simulated and verified using μ-calculus properties as well as behavioural model minimisation and comparison.
- A second study carried out on a re-engineered UML model, has been conducted more recently in the framework of the IST OMEGA project [Con03]. The goal was to evaluate both the appropriateness of extensions of UML to model this type of real-time system, and the usability of IF validation tools. In this study, the UML model has been translated automatically to IF, simplified by static analysis, simulated and verified against properties expressed as *observers*.

We summarise the relevant results of both experiments, and we give principles of a *verification methodology* that can be used in connection with the IF toolset. For such large examples, push-button verification is not sufficient and some iterative combination of analysis and validation is necessary to cope with complexity.

[1] Ariane-5 is an European Space Agency Project delegated to CNES (Centre National d'Etudes Spatiales).

4.1 Overview of the Ariane-5 Flight Program

The Ariane-5 example has a relatively large UML model: 23 classes, each one with operations and a state machine. Its translation into IF has 7000 lines of code.

The launcher flight. An Ariane-5 launch begins with ignition of the main stage engine (EPC - *Etage Principal Cryotechnique*). Upon confirmation that it is operating properly, the two solid booster stages (EAP - *Etage Accélérateur à Poudre*) are ignited to achieve lift-off.

After burn-out, the two solid boosters (EAP) are jettisoned and Ariane-5 continues its flight through the upper atmosphere propelled only by the cryogenic main stage (EPC). The fairing is jettisoned too, as soon as the atmosphere is thin enough for the satellites not to need protection. The main stage is rendered inert immediately upon shut-down. The launch trajectory is designed to ensure that the stages fall back safely into the ocean.

The storable propellant stage (EPS - *Etage à Propergol Stockable*) takes over to place the geostationary satellites in orbit. Payload separation and attitudinal positioning begin as soon as the launcher's upper section reaches the corresponding orbit. Ariane-5's missions ends 40 minutes after the first ignition command.

A final task remains to be performed - that of passivation. This essentially involves emptying the tanks completely to prevent an explosion that would break the propellant stage into pieces.

The Flight Program. The Flight Program entirely controls the launcher, without any human interaction, beginning 6 minutes 30 seconds before lift-off, and ending 40 minutes later, when the launcher terminates its mission.

The main functions of the Flight Program are the following ones:

- *flight control*, involves navigation, guidance and control algorithms,
- *flight regulation*, involves observation and control of various components of the propulsion stages (engines ignition and extinction, boosters ignition, etc),
- *flight configuration*, involves management of launcher components (stage separation, payload separation, etc).

We focused on *regulation* and *configuration* functions. The *flight control* is a relatively independent synchronous reactive control system.

The environment. In order to obtain a realistic functional model of the Flight Program restricted to regulation and configuration functionalities, we need to take into account its environment. This has been modelled by two external components abstracting the actual behaviour of the flight control part and the ground:

- the *flight control* includes several processes describing a nominal behaviour. They send, with some controlled degree of uncertainty, the right flight commands, with the right parameters at the right moments in time.

– the *ground* part abstracts the nominal behaviour of the launch protocol on the ground side. It passes progressively the control of the launcher to the on board flight program, by providing the launch date and all the confirmations needed for launching. Furthermore, it remains ready to take back the control, if some malfunctioning is detected during the launch procedure.

Requirements. With the help of EADS engineers, we identified a set of about twenty functional safety requirements ensuring the right service of the Flight Program. The requirements have been classified into three classes:

– *general requirements*, not necessarily specific to the Flight Program but common to all critical real-time systems. They include basic untimed properties such as the absence of deadlocks, livelocks or signal loss, and basic timed properties such as the absence of timelocks, Zeno behaviours or deadlines missed;
– *overall system requirements*, specific to the Flight Program and concerning its global behaviour. For example, the global sequence of the flight phases is respected: ground, vulcain ignition, booster ignition, ...;
– *local component requirements*, specific to the Flight Program and regarding the functionality of some of its parts. This category includes for example checking the occurrence of some actions in some component (e.g, payload separation occurs eventually during an attitudinal positioning phase, or the stop sequence no. 3 can occur only after lift-off, or the state of engine valves conforms to the flight phase, etc.)

4.2 UML model

The Ariane-5 Flight Program is modelled in UML as a collection of objects communicating mostly through asynchronous signals, and whose behaviour is described by state machines. Operations (with an abstract body) are used to model the guidance, navigation and control tasks. For the modelling of timed behaviour and timing properties, we are using the OMEGA real-time UML profile [GOO03], which provides basic primitives such as timers and clocks. The model shown in figure 5 is composed of:

– a global controller class responsible for flight configuration (*Acyclic*);
– a model of the regulation components (e.g. *EAP*, *EPC* corresponding to the launcher's stages);
– a model of the regulated equipment (e.g. *Valves*, *Pyros*);
– an abstract model of the cyclic GNC tasks (*Cyclics*, *Thrust_monitor*, etc.);
– an abstract model of the environment (classes *Ground* for the external events and *Bus* for modelling the communication with synchronous GNC tasks).

The behaviour of the flight regulation components (EAP, EPC) involves mainly the execution of the firing/extinction sequence for the corresponding stage of the launcher (see for example a fragment of the EPC stage controller's state machine

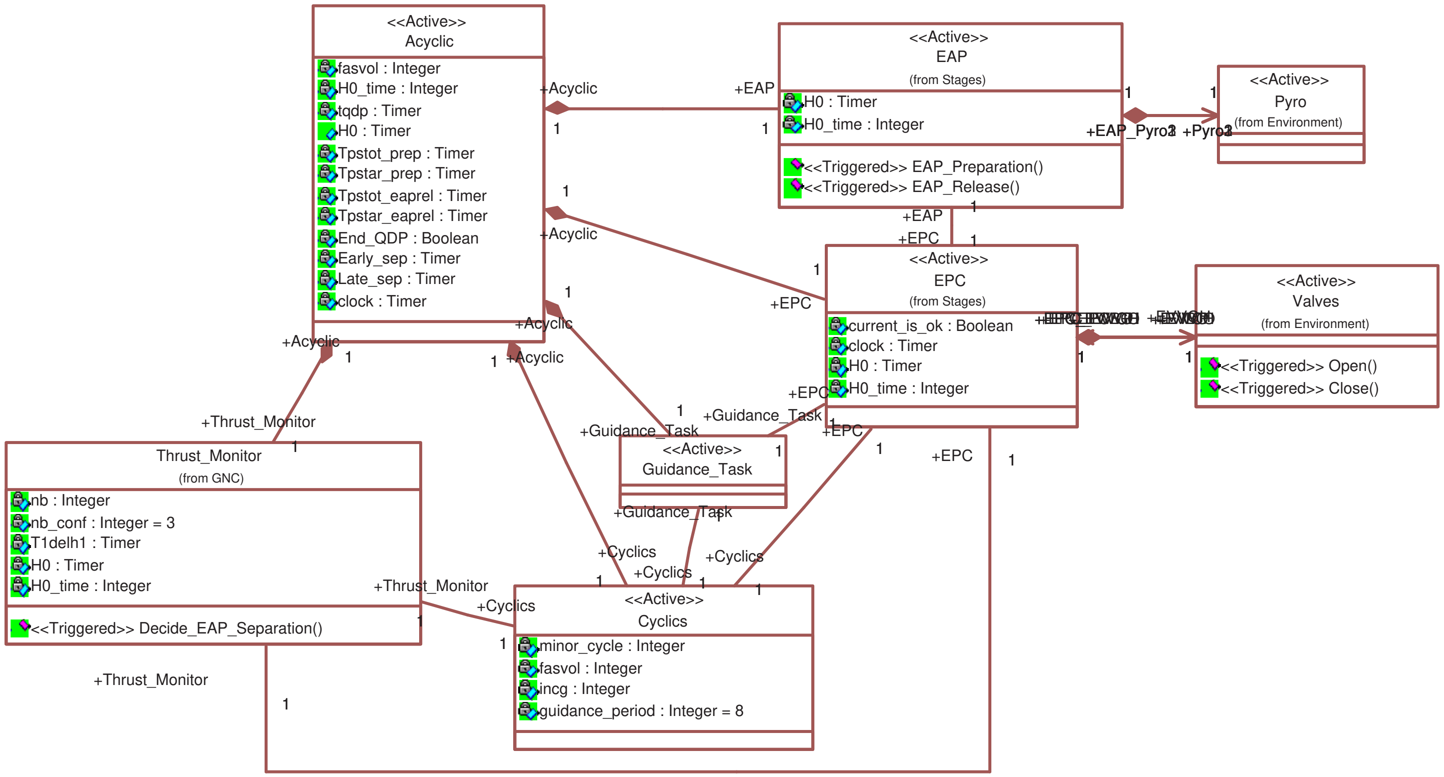

Fig. 5. Structure of the UML specification (part).

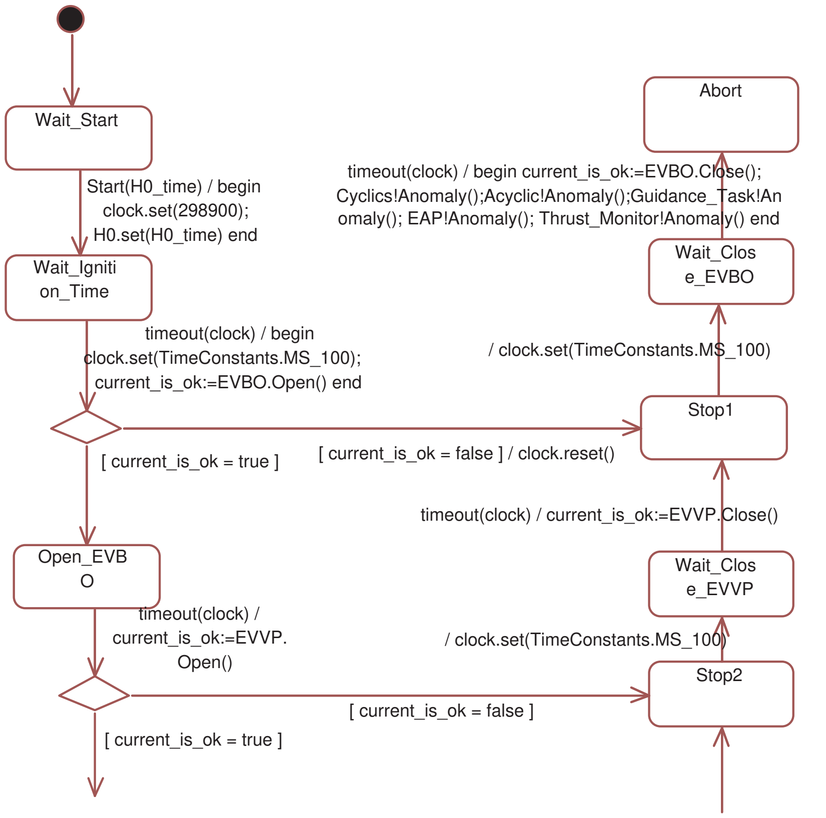

Fig. 6. Behaviour of the EPC regulation process (part).

in figure 6). The sequence is time-driven, with the possibility of safe abortion in case of anomaly.

The flight configuration part implements several tasks: EAP separation, EPC separation, payload separation, etc. In their case too, the separation dates are provided by the control part, depending on the current flight evolution.

4.3 Validation using the IF toolset

Validation is a complex activity, involving the iterated application of verification and analysis phases as depicted in figure 7.

Translation to IF and basic static analysis provides a first sanity check of the model. In this step, the user can find simple compile-time errors in the model (name errors, type errors, etc.) but also more elaborate information (uninitialised or unused variables, unused signals, dead code).

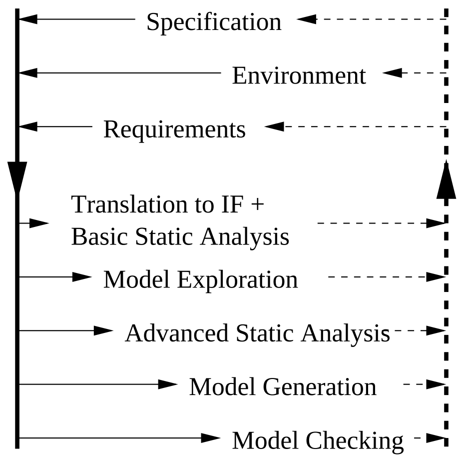

Fig. 7. Validation methodology in IF.

Model exploration. The validation process continues with a debugging phase. Without being exhaustive, the user begins to explore the model in a guided or random manner. Simulation states do not need to be stored as the complete model is not explicitly constructed at this moment.

The aim of this phase is to inspect and validate known nominal scenarios of the specification. Moreover, the user can *test* simple safety properties, which must hold on all execution paths. Such properties are generic ones, such as absence of deadlocks and signal loss, or more specific ones such as local assertions.

Advanced static analysis. The aim is to simplify the IF description. We use the following static analysis techniques to reduce both the state vector and the state space, while completely preserving its behaviour:

- A specific analysis technique is the elimination of redundant clocks [DY96]. Two clocks are *dependent* in a control state if their difference is constant and can be statically computed at that state.
  The initial SDL version of the Flight Program used no less than 130 timers. Using our static analysis tool we were able to reduce them to only 55 timers, functionally independent ones. Afterwards, the whole specification has been rewritten taking into account the redundancy discovered by the analyser.
- A second optimisation identifies live equivalent states by introducing systematic resets for dead variables in certain states of the specification.
  For this case study, the live reduction has not been particularly effective due to the reduced number of variables (others than clocks) used in the specification. Our initial attempts to generate the model without live reduction failed. Finally, using live reduction we were able to build the model but still, it was of unmanageable size, about $2 \cdot 10^6$ states and $18 \cdot 10^6$ transitions.

– The last optimisation is dead-code elimination. We used this technique to automatically eliminate some components which do not perform any relevant action.

LTS generation. The LTS generation phase aims to build the state graph of the specification by exhaustive simulation. In order to cope with the complexity, the user can choose an adequate state representation e.g., discrete or dense representation of time as well as an exploration strategy e.g., traversal order, use of partial order reductions, scheduling policies, etc.

The use of partial order reduction has been necessary to construct tractable models. We applied a simple *static* partial order reduction which 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 or access to shared variables. This partial order reduction imposes a fixed exploration order between internal steps and preserves *all* the properties expressed in terms of visible actions.

Example 4. By using partial order reduction on internal steps, we reduced the size of the model by 3 orders of magnitude i.e, from $2 \cdot 10^6$ states and $18 \cdot 10^6$ transitions to $1.6 \cdot 10^3$ states and $1.65 \cdot 10^3$ transitions, which can be easily handled by the model checker.

We considered two different models of the environment. A *time-deterministic* one, where actions take place at precise moments in time and a *time-nondeterministic* one where actions take place within predefined time intervals. Table 1 presents in each case the sizes of the models obtained depending on the generation strategy used.

		time deterministic	time non-deterministic
model generation	− live reduction − partial order	state explosion	state explosion
	+ live reduction − partial order	2201760 st. 18706871 tr.	state explosion
	+ live reduction + partial order	1604 st. 1642 tr.	195718 st. 278263 tr.
model verification	model minimisation	∼ 1 sec.	∼ 20 sec.
	model checking	∼ 15 sec.	∼ 120 sec.

Table 1. Verification Results. The model minimisation and model checking experiments are performed on the smallest available models i.e, obtained with both live and partial order reduction.

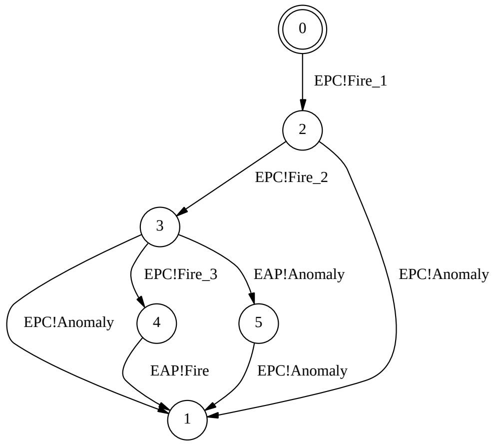

Fig. 8. Minimal model.

Model checking. Once the model has been generated, three model checking techniques have been applied to verify requirements on the specification:

1. Model checking of μ-calculus formulae using EVALUATOR.

Example 5. The requirement expressing that "*the stop sequence no. 3 occurs only during the flight phase, and never on the ground phase*" can be expressed by the following μ-calculus formula, verified with EVALUATOR:

$$\neg\ \mu X. < EPC!Stop\_3 > T \ \ \vee \ \ < \overline{EAP!Fire} > X$$

This formula means that the system cannot execute the *stop sequence no. 3* without executing the firing of the EAP first.

2. Construction of reduced models using ALDEBARAN. A second approach, usually much more intuitive for a non expert end-user, consists in computing an abstract model (with respect to given observation criteria) of the overall behaviour of the specification. Possible incorrect behaviours can be detected by visualising such a model.

Example 6. All safety properties involving the firing actions of the two principal stages, EAP and EPC, and the detection of anomalies are preserved on the LTS in figure 8 generated by ALDEBARAN. It is the quotient model with respect to safety equivalence [BFG$^+$91] while keeping observable only the actions above. For instance it is easy to check on this abstract model that, whenever an anomaly occurs *before* action *EPC!Fire_3* (ignition of the Vulcain engine), then nor this action nor *EAP!Fire* action are executed and therefore the entire launch procedure is aborted.

Table 1 gives the average time required for verifying each kind of property by temporal logic model checking and model minimisation respectively.

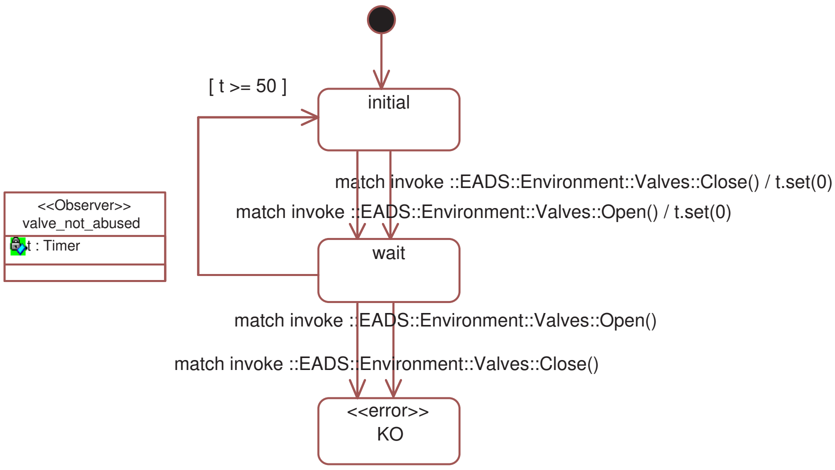

Fig. 9. A timed safety property of the Ariane-5 model.

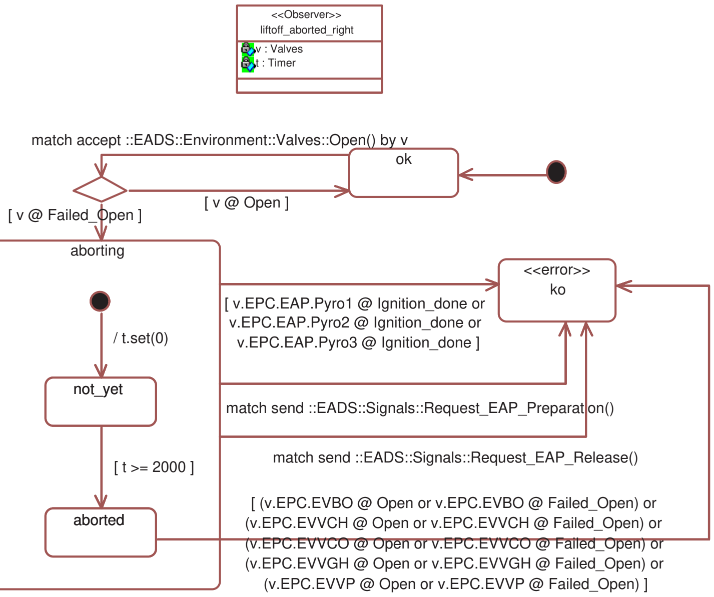

Fig. 10. A timed safety property of the Ariane-5 model.

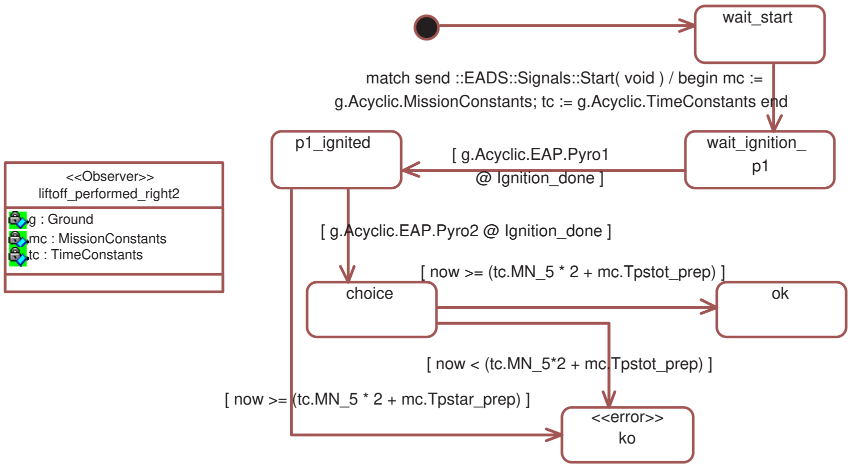

Fig. 11. A timed safety property of the Ariane-5 model.

3. Model checking with observers. We also used *UML observers* to express and check requirements. Observers allow us to express in a much simpler manner most safety requirements of the Ariane-5 specification. Additionally, they allow to express *quantitative* timing properties, something which is difficult to express with μ-calculus formulas.

Example 7. Figures 9 to 11 show some of the properties that were checked on the UML model:

Figure 9: between any two commands sent by the flight program to the valves there should elapse at least 50ms.

Figure 10: if some instance of class *Valve* fails to open (i.e. enters the state *Failed_Open*) then

- No instance of the *Pyro* class reaches the state *Ignition_done*.
- All instances of class *Valve* shall reach one of the states *Failed_Close* or *Close* after at most 2 seconds since the initial valve failure.
- The events *EAP_Preparation* and *EAP_Release* are never emitted.

Figure 11: if the *Pyro1* object (of class *Pyro*) enters the state *Ignition_done*, then the *Pyro2* object shall enter the state *Ignition_done* at a system time between $TimeConstants.MN\_5{*}2{+}Tpstot\_prep$ and $TimeConstants.MN\_5{*}2 + Tpstar\_prep$.

5 Conclusion

The IF toolset is the result of a long term research effort for theory, methods and tools for model-based development. It offers a unique combination of features for modelling and validation including support for high level modelling, static analysis, model-checking and simulation. Its has been designed with special care for openness to modelling languages and validation tools thanks to the definition of appropriate API's. For instance, it has been connected to explicit model checking tools such as Spin [Hol91] and Cadp [FGK+96], to symbolic and regular model checker tools such as TreX [ABS01], Lash [BL02a], the PVS-based abstraction tool Invest [BLO98] and to the automatic test generation and execution tools Tgv [FJJV96], Agatha [LRG01] and Spider [HN04].

The IF notation is expressive and rich enough to map in a structural manner most of UML concepts and constructs such as classes, state machines with actions, activity groups with run-to-completion semantics. The mapping flattens the description only for inheritance and synchronous calls and this is necessary for validation purposes. It preserves all relevant information about the structure of the model. This provides a basis for compositional analysis and validation techniques that should be further investigated.

The IF notation relies on a framework for modelling real-time systems based on the use of priorities and of types of urgency studied at Verimag [BST98], [BS00], [AGS02]. The combined use of behaviour and priorities naturally leads to layered models and allows compositional modelling of real-time systems, in particular of aspects related to resource sharing and scheduling. Scheduling policies can be modelled as sets of dynamic priority rules. The framework supports composition of scheduling policies and provides composability results for deadlock freedom of the scheduled system. Priorities are also an elegant mechanism for restricting non determinism and controlling execution. Run-to-completion execution and mutual exclusion can be modelled in a straightforward manner. Finally, priorities prove to be a powerful tool for modelling both heterogeneous interaction and heterogeneous execution as advocated in [GS03]. The IF toolset fully supports this framework. It embodies principles for structuring and enriching descriptions with timing information as well as expertise gained through its use in several large projects such as the IST projects OMEGA [Con03,GH04], AGEDIS [Con02] and ADVANCE [Con01].

The combination of different validation techniques enlarges the scope of application of the IF toolset. Approaches can differ according to the characteristics of the model. For data intensive models, static analysis techniques can be used to simplify the model before verification, while for control intensive models partial order techniques and observers are very useful to cope with state explosion. In any case, the combined use of static analysis and model checking by skilled users proves to be a powerful means to break complexity. Clearly, the use of high level modelling languages involves some additional cost in complexity with respect to low level modelling languages e.g., languages based on automata. Nevertheless, this is a price to pay for validation of real life systems whose faithful modelling requires dynamically changing models with infinite state space. In our methodology, abstraction and simplification can be carried out automatically by static analysis.

The use of observers for requirements proves to be very convenient and easy to use compared to logic-based formalisms. They allow a natural description, especially of real-time properties relating timed occurrences of several events. The "operational" description style is much more easy to master and understand by practitioners. The limitation to safety properties is not a serious one for well-timed systems. In fact, IF descriptions are by construction well-timed - time can always progress due to the use of urgency types. Liveness properties become bounded response, that is safety properties.

The IF toolset is unique in that it supports rigorous high level modelling of real-time systems and their properties as well as a complete validation methodology. Compared to commercially available modelling tools, it offers more powerful validation features. For graphical editing and version management, it needs a front end that generates either XMI or SDL. We are currently using RATIONAL ROSE and OBJECTGEODE. We have also connections from RHAPSODY and ARGO UML.

Compared to other validation tools, the IF toolset presents many similarities with SPIN. Both tools offer features such as a high level input language, integration of external code, use of enumerative model checking techniques as well as static optimisations. In addition, IF allows the modelling of real-time concepts and the toolset has an open architecture which eases the connection with other tools.

References

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *Proceedings of CAV'01, (Paris, France)*, volume 2102 of *LNCS*. Springer, 2001.

[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[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.

[AGS02] K. Altisen, G. Gössler, and J. Sifakis. Scheduler Modeling Based on the Controller Snthesis Paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing"*, 23(1/2):55–84, 2002.

[BFG+91] A. Bouajjani, J.Cl. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proceedings of ICALP'91*, volume 510 of *LNCS*. Springer, July 1991.

[BFG+99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. In R. Dssouli, G. Bochmann, and Y. Lahav, editors, *Proceedings of SDL FORUM'99 (Montreal, Canada)*, pages 423–440. Elsevier, June 1999.

[BFKM97] M. Bozga, J.Cl. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the Aldebaran Toolset. *Software Tools for Technology Transfer*, 1(1+2):166–183, December 1997.

[BL02a] B. Boigelot and L. Latour. The Liege Automata-based Symbolic Handler LASH. http://www.montefiore.ulg.ac.be/ boigelot/research/lash, 2002.

[BL02b] M. Bozga and Y. Lakhnech. IF-2.0: Common Language Operational Semantics. Technical report, Verimag, 2002.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In A. Hu and M. Vardi, editors, *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, pages 319–331. Springer, June 1998.

[BS00] S. Bornot and J. Sifakis. An Algebraic Framework for Urgency. *Information and Computation*, 163:172–202, 2000.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[Con01] ADVANCE Consortium. http://www.liafa.jussieu.fr/ advance - website of the IST ADVANCE project, 2001.

[Con02] AGEDIS Consortium. http://www.agedis.de - website of the IST AGEDIS project, 2002.

[Con03] OMEGA Consortium. http://www-omega.imag.fr - website of the IST OMEGA project., 2003.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 1999. Journal Version to appear in Journal on Formal Methods in System Design, July 2001.

[DJPV02] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In *Proceedings of FMCO'02*, LNCS. Springer Verlag, November 2002.

[DY96] C. Daws and S. Yovine. Reducing the Number of Clock Variables of Timed Automata. In *Proceedings of RTSS'96 (Washington, DC, USA)*, pages 73–82. IEEE Computer Society Press, December 1996.

[FGK+96] J.Cl. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of CAV'96 (New Brunswick, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer, August 1996.

[FJJV96] J.C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In *Proceedings of CAV'96*, number 1102 in LNCS. Springer, 1996.

[GH04] S. Graf and J. Hooman. Correct development of embedded systems. In *European Workshop on Software Architecture: Languages, Styles, Models, Tools, and Applications (EWSA 2004), co-located with ICSE 2004, St Andrews, Scotland*, LNCS, May 2004.

[GOO03] S. Graf, I. Ober, and I. Ober. Timed Annotations in UML. In *Workshop SVERTS on Specification and Validation of UML models for Real Time and Embedded Systems, a satellite event of UML 2003, San Francisco, October 2003*, Verimag technical report 2003/10/22 or http://www-verimag.imag.fr/EVENTS/2003/SVERTS/, October 2003.

[GS03] G. Gössler and J. Sifakis. Composition for Component-Based Modeling. In *proc. FMCO'02*, volume 2852 of *LNCS*. Springer-Verlag, 2003.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming 8, 231-274*, 1987.
[HN04] A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In *Proceedings of ISSTA'2004*, 2004.
[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.
[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.
[HvdZ03] J. Hooman and M.B. van der Zwaag. A Semantics of Communicating Reactive Objects with Timing. In *Proceedings of SVERTS'03 (Specification and Validation of UML models for Real Time and Embedded Systems)*, San Francisco, October 2003.
[IBM] IBM. Rational ROSE Development Environment.
[Ilo] Ilogix. Rhapsody Development Environment.
[JM99] T. Jéron and P. Morel. Test Generation Derived from Model Checking. In N. Halbwachs and D. Peled, editors, *Proceedings of CAV'99 (Trento, Italy)*, volume 1633 of *LNCS*, pages 108–122. Springer, July 1999.
[Koz83] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 1983.
[LPY98] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, 1:134–152, 1998.
[LRG01] D. Lugato, N. Rapin, and J.P. Gallois. Verification and tests generation for SDL industrial specifications with the AGATHA toolset. In *Real-Time Tools Workshop affiliated to CONCUR 2001, Aalborg, Denmark*, 2001.
[Mil80] R. Milner. *A Calculus of Communication Systems*, volume 92 of *LNCS*. Springer, 1980.
[Muc97] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.
[NS91] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In *Proc. CAV'91*, volume 575 of *LNCS*. Springer-Verlag, July 1991.
[OGO04] I. Ober, S. Graf, and I. Ober. Model Checking of UML Models via a Mapping to Communicating Extended Timed Automata. In *11th International SPIN Workshop on Model Checking of Software, 2004*, volume LNCS 2989, pages 127–145, 2004.
[OMG01] OMG. Unified Modeling Language Specification (Action Semantics). OMG Adopted Specification, December 2001.
[OMG03a] OMG. Model Driven Architecture. http://www.omg.org/mda, 2003.
[OMG03b] OMG. Standard UML Profile for Schedulability, Performance and Time, v. 1.0. OMG document formal/2003-09-01, September 2003.
[Par81] D. Park. Concurrency and Automata on Infinite Sequences. *Theoretical Computer Science*, 104:167–183, March 1981.
[RVR+] A. Ramirez, Ph. Vanpeperstraete, A. Rueckert, K. Odutola, J. Bennett, and L. Tolke. ArgoUML Environment.
[Sif77] J. Sifakis. Use of Petri Nets for Performance Evaluation. In *Proc. 3rd Intl. Symposium on Modeling and Evaluation*, pages 75–93. IFIP, North Holland, 1977.
[Sif01] J. Sifakis. Modeling Real-Time Systems — Challenges and Work Directions. In T.A. Henzinger and C. M. Kirsch, editors, *Proc. EMSOFT'01*, volume 2211 of *LNCS*. Springer-Verlag, 2001.

[STY03] J. Sifakis, S. Tripakis, and S. Yovine. Building Models of Real-Time Systems from Application Software. *Proc. IEEE*, 91(1):100–111, 2003.

[Tip94] F. Tip. A Survey of Program Slicing Techniques. Technical Report CS-R9438, CWI, Amsterdam, The Netherlands, 1994.

[vGW89] R.J. van Glabbeek and W.P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. Technical Report CS-R8911, CWI, Amsterdam, The Netherlands, 1989.

[Wei84] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[Yov97] S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.