Chapter 9

Modeling and Verification of Real-Time Systems using IF Toolbox

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 optimization 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.

The technical work presented in this paper has been carried out mainly between 1997 and 2005. This presentation is for a large part a translation of a book chapter in French [BOZ 06].

Chapter written by Marius Bozga and Susanne Graf and Laurent Mounier and Iulian Ober.
9.1. Motivation

Modelling plays a central role in software engineering for real-time distributed systems. The use of models in the design process of a software is used to express both, the expected behavior of the system under development and that of its environment — consisting of the physical environment and the execution infrastructure or platform — and offers numerous advantages:

- Abstract descriptions using high-level concepts can be analyzed efficiently using appropriate formal methods, and this even when no actual execution environment is available yet.
- The use of executable models, whose behavior are defined by an operational semantics, allow to experiment with large models just like with an implementation, and in addition, it allows the inclusion of legacy code for analysis purpose.
- Moreover, accurate executable models of the environment may offer more accurate observation and control capabilities than the ones provided by a real experimentation platform. Indeed, a model allows controlling physical parameters that are not always controllable in real-life (like time progress), and avoids probe effects and disturbances due to experimentation.

Building faithful models of complex systems useful for efficient analysis is a non trivial problem and a prerequisite to the application of formal analysis techniques. Traditionally, modelling techniques are applied at early phases of system development and at high abstraction level. Then, several models may be used, one for each type of analysis required. Nevertheless, methods are then needed to guarantee the consistency between these models during the entire design process and also with the final implementation. In restricted contexts, and when a global view of the system under development is available, some solutions have been proposed: For instance the B method [ABR 88] provides a stepwise and tool guided model refinement technique, and the synchronous approach [BEN 93, BEN 03], allows the automatic production of executable code or integrated circuit layouts from abstract design models.

More recently, new model-based approaches have been proposed, to better take into account other kinds of applications, and able to cope with existing software components and complex distributed execution platforms with various scheduling policies. Among them, one can notice the OMG MDA initiative [OMG 03a], proposing to generate code from a description of the execution platform, a functional description of the software behavior, and a non-functional description of its (assumed and guaranteed) properties. Architecture description languages like AADL [SAE 04] focus on the description of existing execution architectures and allow analytic analysis and the generation of an implementation through the integration of existing pieces of code. Similarly, but at a more abstract level, a new paradigm for component composition or coordination [SIF 01, SIF 03, SCH 04, GÖS 05] offers a notion of rich component interfaces, including non functional aspects.
Such approaches heavily rely on the existence of modelling methods and tools to provide support and guidance for system design and validation, particularly regarding coordination and interaction between heterogeneous components.

The IF toolset is an environment for modelling and validation of heterogeneous real-time systems that has been developed for this purpose. It is characterized by the following features:

- Support for modelling with user level formalisms, such as SDL or UML, in commercial editing or CASE tools to describe both abstract models and concrete ones (used to generate an actual implementation). This is essential to ease usability by practitioners and to stick to 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 (over) simplified models constructed in an ad hoc manner, as it is often the case for other tools based on low level description languages e.g., finite automata.

- Transformations from high level modelling formalisms into an intermediate representation, the IF notation, that serves as a semantic model by still preserving the essential structure for allowing efficient analysis and verification. This representation combines the 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 behavior to model run-to-completion execution modes.

  We consider a class of timed automata which are well-timed by construction. This representation is rich and expressive enough to describe the main concepts and constructs of source languages by nevertheless preserving structural concepts required for being able to simplify models for validation. In particular, the concurrency level and the data structure of the initial model are explicitly preserved in the IF representation. Thus, the translation methods we developed 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.

- Support for a wide range of abstraction and validation techniques, including model-checking, static analysis, test case generation and simulation. A methodology has been studied at Verimag for complex real-time applications. Model simplifications based on static analysis techniques are applied directly on IF system descriptions, whereas dynamic verification techniques are applied on a global, possibly abstracted, execution graph, generated by an exploration platform from the IF description. Other simplifications like partial order reductions are applied during this exploration process.

- Expression of requirements to be validated on models by using observers. These observers can be considered as a special class of models equipped with primitives for monitoring and checking for divergence from some nominal behavior. Our choice for observers rather than declarative formalisms such as temporal logic or Live Sequence
charts [DAM 99] is motivated by our concern to be close to industrial practice and to avoid as much as possible inconsistency in requirements.

This chapter is organized as follows. Section 9.2 presents the overall architecture of the IF toolset. Sections 9.3 and 9.4 are the heart of the paper. The first one gives an overview on IF, including its main concepts and constructs and their semantics. The second one provides a detailed description of the functionality of the toolbox, and in particular the simulation, analysis and verification tools. The UML to IF translation principle is also explained, showing how the main UML concepts are mapped into IF. Section 9.6 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 9.7 presents concluding remarks about the toolset and the underlying modelling and validation methodology.

9.2. Architecture

Figure 9.1. IF toolset architecture

Figure 9.1 describes the overall architecture of the toolset, the most important components as well as their inter-connections. We distinguish four different description
levels: the user description level (UML, SDL), the intermediate description level (IF),
the exploration platform level and the explicit labelled transition system level.

**User description level**

This level 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 1.0 [OMG 01], the standard XML format form exchanging UML models at
the time we had built our tools and which is still used by a number of UML tools to-
day. The IF toolset includes a translator which for an UML model in the XMI format
produces an IF description. The translator has been tested for specifications produced
by RATIONAL ROSE [IBM ], RHAPSODY [ILO ] or ARGO UML [RAM ]. Regarding
SDL, the translator takes as input textual SDL’96 specifications and produces IF
descriptions.

**Intermediate description level (IF)**

IF is an intermediate representation based on timed automata extended with dis-
crete data variables, communication primitives, dynamic process creation and destruc-
tion. This representation is expressive enough to describe the basic concepts of mod-
elling and programming languages for distributed real-time systems.

The abstract syntax tree of an IF description can be accessed through an application
programming interface (API). Since all the data (variables, clocks) and the commu-
nication structure are still explicit, high-level transformations based on static analysis
[MUC 97] or program slicing [WEI 84, TIP 94] can be applied. All these techniques
\begin{Verbatim}
can be used to transform the initial IF description into a simpler one while preserving
safety properties [BOZ 03b, FER 03]. Moreover, this API is well-suited to implement
exporters from IF to other specification formalisms.
\end{Verbatim}

**Exploration platform and labelled transitions system level**

At exploration platform level, a system is seen as a collection of running com-
ponents which, in any global state, propose a set of possible steps. An exploration
API allows to represent and store explored states as well as to compute on demand the
successors of any given state. This API can be used by modules performing any kind
of on-the-fly analysis.
Using this exploration API, several validation tools have been developed. They cover a broad range of functionalities: interactive/random/guided simulation, on-the-fly model checking using observers, on-the-fly temporal logic model checking, exhaustive state space generation, scheduling analysis, test case generation [BOZ 02a, BOZ 04]. Moreover, through this API are connected the CADP toolbox [FER 96b] for the validation of finite models as well as TGV [FER 96a, JÉR 99] for test case generation using on-the-fly techniques.

At semantic level, the state space containing all possible executions of the system can be explicitly represented as a labelled transition system (LTS) where the labels correspond to observable actions in the IF transitions. Such transition systems can be compared or reduced various simulation and bisimulation equivalences, further composed or visualized, e.g. using the Aldebaran tool [BOZ 97] of the CADP toolbox.

9.3. The IF Notation

IF is a notation for systems of components (called processes), running in parallel and interacting either through global variables or via 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 labelled transition system obtained by interleaving the behaviour of its processes. However, the interleaving can be restricted using dynamic priorities, in order to enforce a scheduling policy or any other user specific execution policy.

In this section, we present the concepts provided by the IF notation to describe behaviour of systems by distinguishing between the functional and non-functional ones.

9.3.1. Functional Features

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 moves from one control state to another by executing some transition. Transitions may be triggered by signals in the input queue or may be spontaneous. In any case, transitions can be guarded by predicates on variables, where a guard is
a conjunction of a data guard and a time guard. A transition is enabled in a state if its trigger 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 involve external functions/procedures, written in an external programming language (C/C++).

As for state charts [HAR 87, HAR 98], 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.

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 its message queue. Signal routes represent specialized communication media transporting signals between processes. The behaviour of a signal route is defined by its delivery policy (FIFO or multi-set), its connection policy (peer to peer, unicast or multicast), its delaying policy and finally its reliability (reliable or lossy). More complex communication media must be specified explicitly as IF processes.

Concerning 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.

EXAMPLE. – 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 9.2.

```
# system Server; // parameterized signals
# signal request();
# signal done(pid);
# signalroute entry(1) from env to server

server;
    with request;
    signalroute cs(1) #delay[1,2]
        from thread to server
        with done;
    signalroute entry(1)
        from env to server
    process thread(0);
```
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 [BOZ 02b].
9.3.2. Non-Functional features

The time model of IF is that of timed automata with urgency [BOR 98, BOR 00]. As for regular timed automata [ALU 94], the execution of a transition is instantaneous, and time is progressing only in states. Moreover, an urgency attribute is associated to transitions to indicate their priority with respect to time progress. We distinguish between eager, lazy and delayable transitions. Eager transitions are urgent, that is they have to be executed at the point of time at which they become enabled - except if they are disabled by executing another transition. Lazy transitions are never urgent and they may be disabled by time progress. Delayable transitions are a combination of both eager and lazy: they are not urgent, except for the moment at which time progress would disable when.

Like in timed automata, the time distance between events is 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. – A timed version of the thread process of the previous example is given. An extra state work introduced for distinguishing the instant at which work starts and the instant at which it ends 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;
```

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 non-determinism 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 non-functional 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

\[ p_1 \prec p_2 \text{ if } p_1\.\text{group} = p_2\.\text{group} \text{ and } p_2\.\text{counter} < p_1\.\text{counter} \]

which for any process instances which are part of some “group”, gives priority to those with the smallest values of the variable \(\text{counter}\) (e.g., the less frequently served).

Finally, in order to express mutual exclusion it is possible to declare shared resources. These resources can be used through particular actions of the form “\text{require some-resource}” and “\text{release some-resource}”.

### 9.3.3. Expressing properties with Observers

An observer expresses in an operational way a safety property of a system by characterizing its acceptable executions. Observers 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, they 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 may be used to restrict the behaviour of the environment
- intrusive observers - which may also alter the system’s behaviour by sending signals and changing variables.

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.

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 beyond the reached state.

Example.— 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.

```plaintext
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;
  endobserver;
```

9.4. The IF Tools

9.4.1. Core components

The core components of the IF toolset are shown in Figure 9.3. The syntactic transformation component deals with the construction and the exploration of an abstract syntax tree (AST) from an IF description. This tree is a collection of C++ objects representing all the syntactic elements that may be present in an IF description: a system has processes, signalroutes, types; a process has states and variables; states include their outgoing transitions and so on. The core component has an interface giving access to this abstract syntax tree. Primitives are available to traverse the tree and to consult, to modify its elements or to print the tree (in the original syntax). The syntactic transformation component has been used to build several applications. The most important ones that we have built are code generators (either simulation code or application code), static analysis transformations (operating at syntactic level) and pretty printers.
Notice that this API has also been intended as an interface with other verification tools. And there have indeed been some external tools connected to IF via this API. For example, there are translations to Promela, the language of SPIN [HOL 91], an explicit state model-checker like IF but with a different performance profile. These translations have been defined in [BOS 00] and later in [PRI 02].

There is a translation from IF to LASH, the Liege Automata-based Symbolic Handler [BOI 02], a tool for exact symbolic reachability analysis for specification with integer variables. There is also a translation to TREX [ANN 01], a similar tool that handles specifications with queues. A translation to the PVS automatic abstraction tool INVEST [BEN 98] has been defined.

There exist translations to test case generators and test execution tools. There are transformations from IF to the input languages of the testing tools AGATHA [LUG 01] and SPIDER [HAR 04]. The Test case generation tool TGV [FER 96a, JÉR 99] has been modified to be able to work on IF specifications.

The exploration platform component has an API providing access to the LTS corresponding to an IF description. The interface offers primitives for representing and accessing states and labels as well as basic primitives for traversing an 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.

The main features of the platform are simulation of the process execution, resolution of non-determinism, management of simulation time and representation of the state space. In other words, 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). More precisely, the platform composes active instances in order to obtain global states and global behaviour of the system. Moreover, the platform stores the set of reachable states, as soon as they are reached.

As shown in Figure 9.3, the exploration platform consists of two layers sharing a common state representation:

- The asynchronous execution 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.

- The dynamic scheduling 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.

Simulation time is handled by a specialized process which manages clock allocation/deallocation, computes time progress conditions and fires 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 [YOV 97] and UPPAAL [LAR 98].

Global states are implicitly stored by the platform. The internal state representation is shown in Figure 9.4. It preserves the structural information and seeks for maximal sharing. The layered representation involves a unique message table. Queues are lists of messages, represented using 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 hash tables with collision or by binary trees. This scheme allows to explicitly represent several millions of structured states.
This architecture provides features for validating heterogeneous systems. Exploration is not limited a priori to IF descriptions: any components with an adequate interface can be executed — in parallel to IF components — by the exploration platform. It is indeed possible to use C/C++ code (either directly, or instrumented accordingly) of already implemented components. Moreover, 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 priority rules [ALT 00].

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.

The exploration platform and its interface has been used as back-end 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 optimization (shortest path computation). These back-end tools are described in Section 9.4.3.
9.4.2. **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 variable analysis** transforms an IF description into an equivalent smaller one by removing globally dead variables and signal parameters and by resetting locally dead variables [MUC 97]. 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** 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 proven to be unreachable from the initial control state. The analysis computes an upper approximation of the set of processes that can be effectively created.

- **Variable abstraction** allows to compute abstractions by eliminating variables which are not relevant to the user and their dependencies. The computation proceeds as for live variable analysis: processes are analyzed separately, and the results are propagated between processes 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. Variable abstraction allows to extract automatically system descriptions for symbolic verification tools accepting only specific types of data e.g., TREX [ANN 01] which accepts only counters, clocks and queues. Moreover, this technique allows to compute finite-state abstractions for model checking.

9.4.3. **Validation**

Simulation and validation with observers: the exploration platform (see paragraph 9.4.1) includes several options allowing for exhaustive simulation, random or guided simulation, with or without breakpoints. In the presence of observers expressing properties (see paragraph 9.3.3) the platform allows for the exhaustive exploration of the state space and stops whenever an error state is reached (the current explored path sequence is then given as a witness trace for the error). Moreover, the exploration can be guided using cut or intrusive observers.

**EVALUATOR** implements an on-the-fly model checking algorithm for the alternation free $\mu$-calculus [KOZ 83]. This is a branching time logic, based upon propositional calculus with fixpoint operators.
ALDEBARAN [BOZ 97] is a tool for the comparison of LTSs 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 [PAR 81], observational bisimulation [MIL 80], branching bisimulation [GLA 89], safety bisimulation [BOU 91], etc. The choice of the relation depends on the class of properties to be preserved.

TGV [FER 96a, JÉR 99] is a tool for test generation developed by IRISA and VER-IMAG. 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.

Let us note that on the fly verification is not systematically more efficient than global methods requiring the the complete construction of the state space. The main reason is that on-the-fly verification explores a product space, between the system and an observer or a formula. Moreover, the explicit construction of the state space of the system alone, allows for the use of minimization techniques such as the ones implemented within Aldebaran, and which usually cannot be applied on the fly.

9.4.4. Translating UML to IF

The toolset supports generation of IF descriptions from both SDL [BOZ 99] and UML [OBE 04, GRA 06b]. We describe the principles of the translation from UML to IF which has been developed in the OMEGA project [OME 05, GRA 04].

9.4.4.1. UML modelling

We have considered a subset of UML 1.4 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 a 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 [DAM 02, ZWA 06].

Additionally, we used a specialization of the standard UML 1.4 profile for Scheduling, Performance and Time [OMG 03b]. Our profile, formally described in [GRA 06a],
The IF Toolbox 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 [RAM], the toolset reads models using the standard XML representation for UML (XMI [OMG 01]).

9.4.4.2. The principles of the mapping from UML to IF

UML runtime 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.

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 sequentializing 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.

Concerning operations, 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.

Representing an operation activation as an object creation has several advantages. First, 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. Moreover, 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. Finally, it allows to distinguish the relevant instants in the context of timing analysis.

9.5. An overview on uses of IF in case studies

The IF toolset has been used in a number of case studies by using different input languages and the frontends connecting them to IF. We give here an overview on a part of them.

- We have used the SDL front-end mainly for modelling and validating communication protocols, such as the SSCOP protocol [BOZ 00]. In this case study, the challenge was to handle a large amount of data, where most of it could be eliminated using static analysis.

  The SDL frontend was used also for verifying a part of the management layer of the MASCARA protocol which is an extension of ATM to wireless connections. Here, a combination of partial order reduction, static analysis and compositional verification allowed to handle the protocol [GRA 01].

  We also have modelled in SDL the European Ariane 5 Launcher mission management and validated it with IF [BOZ 01]. this case study will be presented in more details in the next section.

- We have developed two UML frontends. The UML frontend developed in the AGEDIS project [AGE 03, HAR 04] is not described here; in AGEDIS, we have used IF as a intermediate format for generating test cases with TGV for UML specifications. We have applied the Agedis tool chain for generating test cases for an mq-broker protocol.

  The UML interface developed in the OMEGA project has been used for several case studies, an overview can be found in [OME 05, GRA 05]:

    - A first case study is a flight control mechanism that implements "sensor voting" and "sensors monitoring" operations in a typical flight control system.
- A second one involves the Medium Altitude Reconnaissance System (MARS), which counteracts the image quality degradation caused by the forward motion of an aircraft [OBE 06b].

- A third application is a telecom service built on top of embedded platform and service components which has been modelled and analysed using both IF and Live sequence charts [COM 07].

- Finally, we have taken up on our initial Ariane-5 case study, and made a more complete UML model of the flight software by focusing on the relevant real time behaviours [OBE 06a] which is presented in more detail in the next section.

- We used the IF toolbox to analyse specific real-time systems by using the IF language directly. In [BOZ 03a], we modelled the production capacity of a chemical plant, and used then a new module based on the IF exploration engine integrating specific optimisations to extract optimal schedules for a specific production task. In [SAL 03], we used IF to implement a compositional method for timing analysis of combinational circuits.

- The IF toolbox has been used to verify and test the K9 Rover Executive, an experimental Nasa software for autonomous wheeled vehicles targeted for the exploration of Martian surface [AKH 04, BEN 04].

- IF has also been used by other groups who have developed some specific tools either generating IF specifications or using the existing APIs which then have been used in case studies. For example, [DHA 06] discusses an experiment on the validation of the Air Traffic Control (ATC) data link protocol used in ACARS or ATN networks. This protocol is originally fully specified in SDL, and the authors verify an IF version against properties expressed by observers, and within contexts that are also defined by observers. The authors have developed tools for automatic generation of such observers from higher-level description, like particular temporal logics. [HÉD 05] reports on a use of IF for quality of service evaluation of a driver for a data acquisition system. Non-probabilistic QoS criteria like the minimum/maximum data acquisition delay or the maximum loss rate are derived by exhaustive simulation from a timed model of the driver.

9.6. Case Study: the Ariane-5 Flight Program

Here, we select one of the more complete case studies with respect to the analysis and also the used features, which is the modelling and validation of the flight software of Ariane 5 by using the IF toolset. The study consists in modelling a part of the existing software in a formalism amenable to validation with IF, and in specifying a set of critical properties to be verified on the model.

For historical reasons, this study was performed on two similar models provided by EADS SPACE Transportation, one in SDL and one in UML. The two models were
verified for different properties, and using different techniques. To simplify the presentation, except when stated, we do not distinguish between the two models.

We summarize the relevant results of both experiments, and we give the 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.

9.6.1. Overview of the Ariane 5 Flight Program

Ariane 5 is the European heavy-lift launcher, capable of placing in orbit payloads up to 9.5 tons. The flight software controls the launcher’s mission from lift-off to payload release. It operates in a completely autonomous mode and has to handle both external disturbances (e.g. wind) and different hardware failures that may occur during the flight.

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

- **Cyclic synchronous algorithms**, mainly dedicated to control-command algorithms. In the sequel they are called GNC for Guidance, Navigation and Control. The algorithms and their reactivity constraints are defined by the control engineers based on discretization of continuous physical laws.

- **Aperiodic, event driven algorithms**. These algorithms manage the mission phases and perform particular tasks when the spacecraft changes from one permanent mode to another (engine ignition and stop, stage release, etc.), or when hardware failures occur (alternative or abortion manoeuvres).

The software components implementing this functionality are physically deployed on a single processor and share a common bus for acquiring sensor data and sending commands to the equipment (actually, a set of replicated processors is used for guaranteeing some fault tolerance, but this is out of the scope of our study).

We identified a set of properties, which include functional ones concerning the correct execution of command sequences and flight phases, and non-functional ones concerning mostly the task architecture and the scheduling of the system.

The functional model is independent of the task architecture; it is structured around six main components, modelled by singleton classes in UML and by processes in SDL:
• *Acyclic* is the main mission management component, which handles the start of the flight sequence and the switching from one phase to another. Its behavior is described by a state machine reacting to event receptions from the GNC algorithms (e.g., end of thrust detection) or from the environment, and to time conditions (e.g., time window protections ensuring that the treatment associated to an external event is performed within a predefined time window even in case of failure of the event detection mechanism).

• A set of specific components handle each the acyclic activities related to a particular launcher stage. They react to events received from *Acyclic* or to internal time constraints. In the study, we considered only two stages: *EAP* (lateral boosters) and *EPC* (main stage of the Ariane 5 launcher).

• *Cyclics*: This component manages the activation of the cyclic control/command algorithms (GNC). The algorithms are executed in a predefined order, depending on the current state of the launcher, which is tracked by the *Acyclic* class. We consider in more detail two of the algorithms activated by *Cyclics*, each implemented by a separate object: *Thrust_Monitor*, responsible for the monitoring of the *EAP* thrust, and *Guidance_Task*, which has the particularity that its activation frequency is lower than that of the other GNC algorithms. The other GNC algorithms are only represented inside *Cyclics* by their non-functional characteristics, namely their worst case execution time (WCET), for the purpose of timing analysis – as the numerical correctness of the algorithms was not considered in our study.

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

The verification was aimed to prove the feasibility of the scheduling policy adopted and the absence of certain types of clashes in the use of the shared bus, as well as some twenty functional safety properties identified by the EADS engineers, which can be classified as follows:

• *general requirements*, not necessarily specific to Ariane 5 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 and Zeno behaviours;

• *overall system requirements*, specific to the flight software and concerning its global behaviour. For example, the global sequencing of the flight phases is respected: ground, vulcain ignition, booster ignition, ...;

• *local component requirements*, specific to the flight software and regarding the functionality of some of its subsystems. 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.)

9.6.2. Verification of functional properties

Validation is a complex activity, involving the iterated application of verification and analysis phases as depicted in figure 9.5.

![Validation Methodology in IF](image)

**Translation to IF and basic static analysis.** This 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 (uninitialized or unused variables, unused signals, dead code).

**Model exploration.** The validation process continues with a debugging phase. Without being exhaustive, the user begins to explore the model in a random or guided manner. Simulation states need not to be stored because exhaustive exploration of the model is not aimed for in this phase, but rather error detection. Its aim 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. We usually test only generic properties, such as absence of deadlocks and signal loss, or local assertions.

**Advanced static analysis.** Their aim is to simplify the IF description and to reduce state space explosion during later verification. We use the following static analysis
techniques to reduce both the state vector and the state space, while completely preserving its behaviour (with respect to observability criteria defined by active variables and communication occurrences):

- A specific analysis technique is the elimination of redundant clocks [DAW 96]. 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.

- A second optimization 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 (other than clocks). Nevertheless, our initial attempts to generate the model without live reduction failed, whereas using live reduction we were able to build the model; but still, it was of unmanageable size, about \(2 \times 10^6\) states and \(18 \times 10^6\) transitions.

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

State space generation. The LTS generation phase aims to build a state graph of the specification by exhaustive simulation. In order to cope with the complexity, the user may 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 and efficient for constructing 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.– By using partial order reduction on internal steps, we reduced the size of the model by 3 orders of magnitude i.e, from \(2 \times 10^6\) states and \(18 \times 10^6\) transitions to \(1.6 \times 10^3\) states and \(1.65 \times 10^3\) 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 exactly defined time points and a time-nondeterministic one where actions take place within predefined time intervals. Table 9.1 presents in each case the sizes of the models obtained depending on the generation strategy used. The number of states of the computed state graphs may be considered as relatively small with respect to the millions of states reached by some model checkers. But it should be noticed that each state is not a few words but about 10kB which can be
handled with an ordinary workstation only thanks to the efficient state space storage strategy explained in Section 9.4.1.

<table>
<thead>
<tr>
<th>Model Generation</th>
<th>Time Deterministic</th>
<th>Time Non-Deterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Reduction</td>
<td>State Explosion</td>
<td>State Explosion</td>
</tr>
<tr>
<td>Partial Order</td>
<td>2201760 st.</td>
<td>18706871 tr.</td>
</tr>
<tr>
<td>Live Reduction</td>
<td>1604 st.</td>
<td>195718 st.</td>
</tr>
<tr>
<td>Partial Order</td>
<td>1642 tr.</td>
<td>278263 tr.</td>
</tr>
</tbody>
</table>

| Model Minimization | ~1 sec. | ~20 sec. |
| Model Checking     | ~15 sec. | ~120 sec. |

**Table 9.1. Verification Results.** The model minimization and model checking experiments are performed on the smallest available models i.e., obtained using both live and partial order reduction.

*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 \( \mu \)-calculus formulae using Evaluator.

**Example.** 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 \( \mu \)-calculus formula, verified with Evaluator:

\[
\neg \mu X. <EPC!Stop_3 > T \lor <EAP!Fire > X
\]

This formula expresses the requirement that the system does not execute the stop sequence no. 3 without 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 visualizing the resulting model (if it is small enough).

**Example.** 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 9.6 generated by Aldebaran. It is the quotient model with respect to safety
equivalence [BOU 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 neither this action nor \( EAP!Fire \) action are executed and therefore the entire launch procedure is aborted.

Table 9.1 gives the average time required for verifying each kind of property by temporal logic model checking and model minimization respectively.

3. Model checking with observers. We also used \textit{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 \textit{quantitative} timing properties, something which is difficult to express with \( \mu \)-calculus formulas. Consequently, observers are usually preferred over \( \mu \)-calculus by non-experts in verification.

\textbf{EXAMPLE.}– Figure 9.7 shows one of the timed safety properties that was verified against the UML model: “between any two commands sent by the flight software to the valves, at least 50ms must elapse”.

\textbf{9.6.3. Verification of non-functional properties}

The IF toolbox may be used also for modeling and validating some non-functional properties, specifically properties related to time and schedulability. One objective of the Ariane 5 study was to validate the system under different hypotheses concerning the task architecture and scheduling policy.
The policy used in the system is based on a cyclic fixed-priority preemptive scheme, with priorities assigned by rate monotonic analysis (RMA, see for example [LAY 73]). Acyclic events are handled by cyclic sampling, and certain precautions are taken, like delaying the bus writing operations of each task until the end of the task’s cycle. In turn, this statically ensures some important properties like overall schedulability and mutual exclusion on bus writing, as long as the WCET estimations are correct.

The task architecture is however considered very restrictive by the engineers, in particular in the presence of acyclic events requiring tight response times, or in the presence of cyclic algorithms requiring to read or write data during their cycle. Moreover, rate monotonic analysis is done under very pessimistic hypotheses for the execution time, which, for certain tasks of the Ariane 5 system vary a lot depending on the flight phase.

One goal in our study was to loosen the hypotheses, namely those concerning the sampling of acyclic events and concerning the reading and writing of the bus. The hypotheses for RMA are therefore no longer fulfilled, and schedulability and mutual exclusion is to be verified based on (an abstraction of) the functional model. In order to do this, we integrated a model of a fixed priority preemptive scheduler in the system model, using the timed modelling artifacts offered by IF.

Once this is done, scheduling and mutual exclusion goals become safety properties of the obtained model, and can be formalized and verified using observers. It is
the case for example for the schedulability of the NC task, which can be expressed as: “the computation of the NC component finishes before the end of every period”, which in other terms can be put as “the Cyclics component receives the Synchro signal while being either in state Start_Minor_Cycle, or in Wait_Start or in Abort (the other states are intermediate computation states)”. This is formalized by the observer in Figure 9.8.

9.6.4. Modular verification and abstraction

The verification of large models like Ariane 5 is generally subject to combinatorial explosion of the state space. The explosion cannot always be avoided using strongly behavior-preserving reduction techniques like partial order reduction or the aforementioned static analysis methods. One way to deal with such case is to use compositional abstractions: when a property concerns only a part of the system, we can replace the other parts, which play then the role of environment, by a simplified model representing an abstraction of it. Such a simplified model can be obtained either automatically – for example using the variable abstraction tools of IF, or building abstractions manually. If the abstract model satisfies a safety property based on observations preserved by the abstraction, one can conclude that the initial model also satisfies the property. On the other hand, if the abstract model does not satisfy a property, the result may represent a false negative, and a more precise abstraction would allow to verify it.

To illustrate this verification based on modular abstraction, consider the safety properties of the acyclic part of the Ariane 5 model. To verify them, the cyclic part (GNC) is replaced by an abstraction which discards all behavior details of the GNC component, and sends messages to the acyclic part at arbitrary time points (within some pre-determined intervals) – rather than at moments precisely computed by the GNC. This abstraction, although very coarse and easy to construct, proved sufficient to verify the safety properties of the acyclic part.

More details on the non-functional analysis of the Ariane 5 model can be found in [OBE 06a].
9.7. Discussion

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. As mentioned in section 9.4.1, it has been connected to explicit model checking tools, to symbolic and regular model checkers and abstraction tools, as well as to the automatic test generation and test execution tools.

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 [BOR 98], [BOR 00], [ALT 02]. 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 [GÖS 03]. 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 [OME 05, GRA 04], AGEDIS [AGE 03] and ADVANCE [ADV 04].

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. The combined use of static analysis and model checking by skilled users proves to be a powerful means to break the 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
detailed models of real life systems whose faithful modelling requires dynamically changing models with a potentially 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 rather than invariants for expressing urgency. In this context, 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, an efficient state space storage algorithm for handling models with large states, and the toolset has an open architecture which eases the connection with other tools.

9.8. Bibliography


