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

Validating timed UML models by simulation and verification*

Iulian Ober
Susanne Graf
Ileana Ober

VERIMAG
2, av. de Vignate
38610 Gières, France
e-mail: {ober,graf,iober}@imag.fr

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

Abstract. This paper presents a technique and a tool for model-checking operational (design level) UML models based on a mapping to a model of communicating extended timed automata. The target language of the mapping is the IF format, for which existing model-checking and simulation tools can be used.

Our approach takes into consideration most of the structural and behavioural features of UML, including object-oriented aspects. It handles the combination of operations, state machines, inheritance and polymorphism, with a particular semantic profile for communication and concurrency. We adopt a UML profile that includes extensions for expressing timing. The breadth of concepts covered by our mapping is an important point, as many previous approaches for applying formal validation to UML put much stronger limitations on the considered models.

For expressing properties about models, a formalism called *UML observers* is defined in this paper. Observers reuse existing concepts like classes and state machines, and they allow expressing a significant class of linear temporal properties.

The approach is implemented in a tool that imports UML models from an XMI repository, thus supporting several editors like Rational Rose, Rhapsody or Argo. The generated IF models may be simulated and verified via an interface that presents feedback in the vocabulary of the original UML model.

1 Introduction

This paper presents a technique and a tool for validating UML models by simulation and property verification. We are focusing on UML as we feel some of the techniques that emerged in the field of formal validation are both essential to the reliable development of real-time and safety critical systems, and sufficiently mature to be integrated in a real-life development process.

Our past experiences (for example with the SDL language [?]) show that this integration can only work if validation takes into account widely used modelling languages. Currently, UML based model driven development encounters a big success in the industrial world and is supported by several CASE tools furnishing editing, methodological help, code generation and other functions, but very little support for validation.

This work is part of a broader project (IST OMEGA [?]) which aims at building a UML-based methodology and a validation environment for real-time and embedded systems. An important part of this project was concerned with defining a suitable operational UML profile for real-time applications [?,?], and a formal semantics of it [?] as well as real-time extensions [?]. The work presented in this paper builds upon the foundation of this profile (called OMEGA UML in the following) and is concerned only with validation and tool-related issues such as: implementing the semantics, defining a property specification formalism and applying model-checking techniques. The choices and the semantics of the profile itself are explained only to the extent necessary for understanding the paper.

1.1 Basic assumptions

The following *assumptions* provide the starting point for this work :

– *UML is broader than what we need or can handle in automatic validation.* In UML 1.4 [?] there are 9 types of diagrams and about 150 language concepts (meta-classes). Some of them are too informal to be

* This work is supported by the OMEGA European Project (IST-33522). See also http://www-omega.imag.fr

useful in validation (for example use cases) while for others the relationships and the consistency with the rest of the UML model are not clearly (nor uniquely) defined (for example collaborations, system-level activity diagrams, deployment diagrams).
As a consequence, in this work we focused on a subset of UML concepts that define an operational view of the modelled system: objects, their structure and their behaviour.

– *UML has neither a standard nor a broadly accepted dynamic semantics.* The OMEGA profile used in this work defines a semantics for UML which is suitable for distributed real-time applications. It identifies necessary concepts such as the mechanisms of communication between objects, the concurrency model, the formalism for specifying actions and timing. The main aspects of this semantics are presented in section 2.
– *To produce powerful tools we have to build upon the existing.* This motivates our choice to do a translation to the IF language [?], for which there exists a rich set of tools performing static analysis, model checking, model construction and manipulation, etc. The experiments performed so far confirm that many of these tools can handle UML-generated models. Moreover, mapping UML to IF yields a flexible implementation of the OMEGA semantics in which one can test semantic choices and propose improvements.
On the side of model editing, we are relying on common UML CASE tools such as Rational Rose or I-Logix's Rhapsody, via the standard XML representation for UML (XMI).

1.2 Overview of our approach

The approach presented here covers an operational subset of UML (presented in section 2). The *structure* of models is captured through class definitions, linked by association relationships, aggregation or inheritance. The *behaviour* of each class is described in the standard way by means of state machines and operations, containing structured imperative actions. A particular model of *concurrency* and *communication* is adopted. The combination of all these features, goes beyond previous work done in this area (see section 1.3), which has until now mainly focused on verification of statecharts.

In order to analyse the potential behaviours of UML models, we are translating them into the input language of the IF toolset [?,?]. The translation, explained in section 3, does not yield a particular implementation of an abstract UML model, it rather yields another model which is semantically equivalent to the initial one. Abstractions, such as non-deterministic behaviour of certain objects or informal specification of certain actions are preserved in the IF model.

IF is a formal language based on *communicating extended timed automata* (CETA), for which powerful simulation and verification tools exist. It has been previously used in a number of research projects and case studies. Its main features are presented in section 1.4.

On the level of UML modelling, an important issue in designing real-time systems is the ability to capture quantitative timing requirements and assumptions as well as time dependent behaviour. A set of *timing extensions* for UML are defined in the OMEGA profile [?], and are summarised in section 4 together with their mapping to IF.

Section 5 presents a lightweight extension of UML (*observer classes*) which is used as a *property* description language. Instances of observer classes allow expressing linear temporal property by using a specific semantics for their state machines. Experience shows that the use of such familiar concepts diminishes the shock of introducing formal verification to UML users.

Section 6 presents the UML validation toolset IFx. The functionalities of the tool, ranging from static analysis and optimisations to model generation and model checking, are presented in section 7 on a concrete and complex example – a model of the Ariane 5 flight configuration software.

1.3 Related work

Work on formalising and reasoning with the semantics of UML appeared in the literature during the late 90's (see for example [?,?,?]). During the more recent years, theoretical work, as well as tools supporting formal *analysis* (and particularly *model checking*) of UML models has become a very active field of study, as witnessed by a number of papers [?,?,?,?,?,?,?,?,?,?].

Like ourselves, many of these authors base their work on existing model checkers (SPIN [?] in the case of [?,?,?,?], COSPAN [?] in the case of [?], Kronos [?] for [?] and UPPAAL [?] for [?]), and on the mapping of UML to the input language of the respective tool.

As for specifying properties, some authors opt for the property language of the model checker itself, e.g., [?,?,?]. Others [?,?] use UML collaboration or sequence diagrams, which specify required or forbidden sequences of messages between objects, but are too weak to express stronger properties. We propose the use of a variant of UML classes and state machines to express properties.

Concerning language coverage, most previous approaches do not handle dynamic object creation, inheritance or behaviour described through operations. These are some of the features which make UML an object-oriented formalism. The approach presented in this paper is, to our knowledge, the first one to fill this gap. Our handling of UML state machines was inspired by the material cited above, together with previous work on statecharts [?,?,?].

The concurrency model of the OMEGA profile is inspired by the concurrency model adopted in the Rhapsody tool [?]. The improvements are the formalisation of its semantics, and a more relaxed interpretation of non-determinism which allows a higher level of abstraction and opening to different implementations (Rhapsody adopts an implicitly defined scheduling scheme). In the definition of the profile, we also took inspiration from our previous assessment of the UML concurrency model [?], and from other positions on this topic (see for example [?]).

Finally, the work presented in this paper is part of a broader effort [?,?] to produce a toolset and a methodology which integrate UML and formal techniques for the development of real-time and embedded systems. The framework supports activities like:

- static wellformedness checks
- checking (timed) models against (timed) observers as well as scheduling analysis, formulated as a model-checking problem on the model
- checking of (untimed) models against LTL formulas or Live Sequence Chart specifications (LSC, a variant of interaction diagrams with stronger structuring constructs [?])
- Consistency analysis of LSC (requirements analysis) [?] and state diagram synthesis from LSC specifications [?]
- deductive verification using the interactive theorem prover PVS: compositional verification, consistency checks and reasoning on requirements specified in OCL [?,?].

For more details, the reader is referred to [?,?].

1.4 The back-end: model, techniques, tools

The validation approach proposed in this work is based on the formal model of communicating extended timed automata and on the IF verification environment built upon this model [?,?,?]. We summarise the elements of this model below.

Modelling with communicating extended timed automata

The IF language and the associated toolset developed at Verimag are conceived for modelling and validating *distributed systems* which can manipulate *complex data*, and which involve *dynamic aspects* and *real time constraints*. The IF language is sufficiently expressive to describe the operational semantics of user level formalisms such as UML or SDL at a similar level of abstraction; IF has also been used as a format for inter-connecting model-based validation tools.

An IF description defines the structure of a system and the behaviour of its components. A *system* is composed of a set of communicating *processes* that run in parallel (see Figure 1). Processes are instances of *process types*. They have their own identity (Pid), they may own complex data variables (defined through ADA-like data type definitions), and their behaviour is defined by

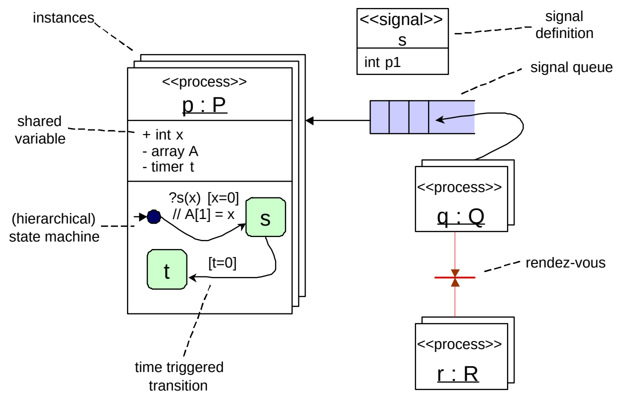

Fig. 1. Constituents of a communicating extended automata model in IF.

a *state machine*. The state machine of a process type may use composite states and the effect of transitions is described using common (structured) imperative statements.

The notion of process is similar to the notion of object from object-oriented languages. The difference is that a process type does not define *operations* and there is no notion of *inheritance*. Operations, inheritance and other notions may be layered on top of the IF model resulting in a more modular definition of the semantics of object models (see section 3).

Processes may communicate via *asynchronous signals* (similar to the UML 1.4 homonym), via shared variables (corresponding to public attributes in UML), or via synchronous rendez-vous. Asynchronous signals are buffered in input queues (one for each process). Parallel processes are composed asynchronously (i.e. by interleaving). The model allows *dynamic creation* of processes, which is an essential feature for modelling object systems.

IF provides support for real time constraints expressed using *clock* variables and guard conditions on them. The values of clocks increase all at the same rate as time progresses. The underlying semantics is based on *finite timed automata with urgency* [?,?].

For more details on the IF model and its semantics, the reader is referred to [?,?].

A framework for modelling priority

On top of the set of processes, one may specify a set of system-wide priority rules of the following form:

$$StateCondition(p_1, p_2) \Rightarrow p_1 \prec p_2$$

The rules are evaluated at each stable state of the system and they define a partial priority order between processes: for every pair of distinct Pids (p_1, p_2), if the condition $StateCondition(p_1, p_2)$ holds in the current system state then the process with ID p_1 has priority over p_2 for the next system step. This means that if p_1 has an enabled transition, p_2 is not allowed to execute.

This priority framework is formalised in [?,?,?].

Property description and verification with observers

Dynamic and time dependent safety properties may be expressed in IF using *observers*. These are special processes used as language acceptors, which execute synchronously with the system and can monitor changes of *state* (variable values, contents of queues, etc.) and *events* that occur (input and output of signals, creation and destruction of processes, etc.).

For expressing properties, some of the states of an observer may be classified (syntactically) as *error* or as *invalid* states. An execution that does not go through an invalid state but reaches an error state is an error trace. Thus, observers can be used to express *safety properties.* A re-interpretation of success states as accepting states of a Büchi automaton allows observers even to express liveness properties.

IF observers are inspired by the observer concept introduced by Jard, Groz and Monin in the VEDA tool [?]. This intuitive and powerful property specification formalism has been adapted over the past 15 years to other modelling languages (LOTOS, SDL) and implemented in industrial case tools like ObjectGEODE.

Analysis techniques and the IF-2 toolbox

The IF toolbox [?,?] is the validation environment built upon the language presented before. It is composed of three categories of tools (see also Figure 5):

1. **behavioural tools** for simulation, verification of properties, automatic test generation. The tools implement state of the art techniques such as *partial order reductions* and some form of *symbolic simulation*, and thus present a good level of scalability.
2. **static analysis tools** providing source-level optimisations that help reducing furthermore the state space of the models, and thus improve the chance of obtaining results from the behavioural tools. The implemented *data* and *control* flow analysis techniques, leading to exact abstractions of the initial model, are dead variable reduction, dead code elimination and *slicing.*
3. **front-ends and exporting tools** which provide an interface with higher-level languages (UML, SDL) and with other validation tools (Spin [?], Agatha [?], etc.).

The IF language allows its user to describe models ranging from very abstract specifications to detailed, directly implementable design models. In order to tackle the complexity of detailed models, the IF toolbox supports abstraction in several ways. For example, data abstraction can be done either by static analysis (computing a slice and throw away a part of the system state which is irrelevant with respect to an observation criterion) or by abstract interpretation of some variables (e.g., symbolic handling of timers and clocks). Another (exact) abstraction often used in IF is provided by partial-order reductions during exhaustive state space exploration; the effect of this reduction is to render deterministic the interleaving of parallel components whenever the non-deterministic interleaving cannot influence the result of the verification of a given property. Finally, other techniques such as input queue abstraction (a very efficient method for particular object topologies such as Kahn networks) have been experimented.

Compositional verification is not directly supported by IF, but some functionalities of the toolbox provide support for a more manual application of a compositional verification methodology. For example, minimal model generation with Aldebaran can be used to extract an abstract model of the behaviour of some component(s), which can then be used instead of the concrete models for constructing and verifying the model of the composed system, or user defined abstractions of subsystems can be checked conform to a more concrete version of a module.

The toolbox has already been used in a series of industrial-size case studies [?].

2 The OMEGA UML profile

This section outlines the main features of the operational OMEGA UML profile [?,?,?,?] implemented in our tools.

2.1 UML concepts covered

The operational subset of UML considered here consists of the following model element types:

– *Classes.* active or passive (see section 2.2).
– *Operations.* triggered/primitive (see section 2.2), constructors, destructors.
– *Signals* for asynchronous communication.
– *Attributes* with *basic types* or object *reference types.*
– *Basic data types.* currently Integer, Boolean, Real.
– *Associations.* simple and composite, with bounded multiplicity.
– *Generalisations.* Their semantics involves polymorphism and dynamic binding of operations.
– *Statecharts.* They are not presented in detail in this paper as already tackled in many previous works, such as [?,?,?,?,?,?,?,?,?].

In order to describe a meaningful behaviour for a UML model, one also needs to describe *actions.* Actions in UML describe the *effect* of a statechart transition, or the body of an operation. Thus, they allow the description of expressive control structure (not limited to finite automata) or even the description of the implementation of operations and transitions. Beginning with version 1.4 of UML, there is a standard for describing actions, but this standard is defined only in terms of a metamodel (giving the types of actions and their components). In order to make it usable, one still has to define a concrete syntax, but which is allowed to vary from one tool to another.

The OMEGA profile [?] defines a textual action language compatible with UML 1.4 which covers: *object creation and destruction*, *operation calls*, *expression* evaluation (including navigation expressions), variable *assignment*, *signal output*, *return action* as well as control flow structuring statements (*conditionals* and *loops*). The concrete syntax of this action language is not presented here as it has only been introduced as a common format to circumvent the problem that none of the today existing UML tools export actions in a structured form.

Additionally to the elements mentioned above, a number of UML extensions for describing timing constraints and assumptions are supported. They are discussed in section 4, and a more detailed description can be found in the companion paper [?].

2.2 The execution model

The purpose of this section is to illustrate the features and particularities of the OMEGA profile taken into account in our tool, not its totality and also not its complete formal semantics, which may be found in [?,?,?]. The execution model chosen in OMEGA and presented here is an extension of the execution model of the Rhapsody UML tool (see [?,?] for an overview), which is used in a large number of UML applications. Other execution models can be accommodated to our framework by adapting the mapping to IF accordingly.

Activity groups and concurrency.

There are two kinds of classes: *active* and *passive* ones. At execution, each instance of an active class defines a concurrency unit called *activity group.* Each instance of a passive class belongs to exactly one activity group, the one of the instance that has created it.

Apart from defining the partition of the system into activity groups, there is no difference between how active and passive classes (and instances) are defined and handled. Both kinds of classes are defined by their attributes, relationships, operations and state machine, and their operational semantics is identical.

Different activity groups are considered as *concurrent*, and each activity group treats external requests (all signals and operation calls from outside the group) one by one in a run-to-completion fashion. During a step, the above mentioned external requests are deferred and stored in the activity groups' *request queue* as long as the activity group is not *stable.*

An *activity group* is stable when all its objects are. An *object* is stable if it has no enabled spontaneous transition[1] and no pending operation call from inside its group.

The motivation for making activity groups working in run-to-completion steps is to be able to consider such a step as atomic from the point of view of the environment of the group. This interpretation of activity groups implies that every activity group has a single control thread, and the atomicity of steps allows preemptive scheduling at run-time. Notice however, that the atomicity of steps can only be guaranteed when some conditions on the outgoing communications hold in each step and if direct data access (through navigation) in between activity groups is not possible. The OMEGA profile does not enforce such a constraint, but the OMEGA methodology proposes to systematically use data access via *get* and *set* operations instead.

The semantics of activity groups described here corresponds to that of concurrent *components*, which make visible to the outside world only the stable states in-between two run-to-completion steps. Such a model has been already successfully used in many concurrent object oriented languages and in synchronous languages.

Operations, signals and state machines.

We consider only synchronous operation calls, where the caller (and its group) are blocked in a *suspended* state until the completion of the call. In the UML model we distinguish syntactically between two kinds of operations: *triggered* and *primitive* ones.

The body of *triggered operations* is described directly in the state machine of a class: the operation call is seen as a special kind of transition trigger. Triggered operations differ from *asynchronous signals* in that they may have a return value.

Primitive operations are closer to methods in usual object oriented programming language. They have a body described by an action. Their handling is more delicate since they may be overridden in the inheritance hierarchy and they are dynamically bound, like in all object-oriented models. When a call for a primitive operation is sent to an object, the appropriate operation implementation with respect to the actual type of the called object in the inheritance hierarchy has to be executed.

With respect to call initiation, when an object having the control in its activity group calls an operation on an(other) object *from the same group*, the call is handled immediately (i.e. on the same control thread), like in usual programming languages. Notice that in case of triggered operation calls, the dynamic call graph should be acyclic, since an object that is already waiting for the termination of a call — made from within a statemachine transition — is in a *suspended* state in which it is not able to handle any new calls. (This type of conditions may be verified using the IF mapping.)

[1] that is a transition which is guarded only by a boolean condition and not triggered by an event

Calls received *from other activity groups* are, independently of the type of operation call, queued by the receiving group and handled in a subsequent run-to-completion step.

Signals are always put in the target object's group queue for handling in a later run-to-completion step, regardless of whether the target is in the same group as the sender or not. This choice is made in order to be able to distinguish triggering an action within the same step (an operation call) and triggering an action in a later step (signal trigger). It has also the effect that concurrency within an activity group cannot be created by sending asynchronous signals.

3 Mapping UML models to IF

In this section, we give the main lines of the mapping of a UML model to an IF system. The intermediate layer of IF helps us tackle the complexity of UML and provides a semantic basis for re-using our existing model checking tools (see section 6).

The mapping is done in such a way that all run-time UML entities (objects, call stacks, pending messages, etc.) are identifiable as a part of the IF state. In simulation and verification, this allows tracing back to the UML specification.

3.1 Mapping the object domain to IF

Mapping of attributes and associations. Every class X is mapped to a process type P_X that has a local variable corresponding to each attribute or association of X. Inheritance is flattened and all inherited attributes and associations are replicated in the process type corresponding to a class.

Activity group management. Each *activity group* is managed by a special *group manager* process (of type *GM*). This process dispatches all signals and external operation calls and thus contributes to ensure the run-to-completion policy. Each process P_X has a local variable *leader*, which points to the *GM* process managing its activity group.

Mapping of operations and call polymorphism. For each operation $m(p_1 : t_1, p_2 : t_2, ...)$ in class X, the following components are defined in IF:

– a signal $call_{X::m}(waiting : pid, caller : pid, callee : pid, p_1 : t_1, p_2 : t_2, ...)$ used to indicate an operation call. The parameter *waiting* holds the Pid of the process that waits for the completion of the call (the *caller* if it is in the same group as the *callee*, the group manager of *callee*, otherwise). The parameter *caller* designates the process waiting for a return value, while *callee* designates the process receiving the call (a P_X instance).
– a signal $return_{X::m}(r_1 : tr_1, r_2 : tr_2, ...)$ used to indicate the return of an operation call (sent to the *caller*). Several return values may be sent with this signal.
– a signal $complete_{X::m}()$ used to indicate *completion* of computation in the operation (may differ from return, as an operation is allowed to return a result and continue computation). This signal is sent to the *waiting* process (see $call_{X::m}$).
– for a *primitive* operation (see section 2.2), a process type $P_{X::m}(waiting : pid, caller : pid, callee : pid, p_1 : t_1, p_2 : t_2, ...)$ is defined which describes the behaviour of the operation using an automaton. The parameters have the same meaning as in the $call_{X::m}$ signal. The *callee* Pid is used to access local attributes of the called object, via the shared variable mechanism of IF.
– the implementation of a *triggered* operation (see section 2.2), is modelled in the state machine of P_X and it is required that there exists an explicit *return* action in the state machine. Transitions triggered by a $X :: m$ call event in the UML state machine are triggered by $call_{X::m}$ in the IF automaton.

The action of invoking an operation $X :: m$ is mapped to sending a signal $call_{X::m}$. The signal is sent either directly to the concerned object (if the caller is in the same group) or to the object's *active group manager*. This group manager queues the call and forwards it to the destination when the group becomes stable. As operation calls are blocking, the sender of a *call* signal enters a state in which it expects, in order to be unblocked, either a *return* signal (if X and Y are in different activity groups) or a *complete* signal (if X and Y are in the same group).

The handling of incoming primitive calls is modelled by transition loops in every state[2] of the processes P_X, which, upon reception of a corresponding $call_{X::m}$ signal create a new instance of $P_{X::m}$ and wait for it to terminate (see sequence diagram in Figure 2).

In general, the mapping of primitive operation (activations) into separate automata created by the called object has several advantages:

– it allows extensions to non-usual types of calls, such as non-blocking calls. It also preserves modularity and readability of the generated model.
– it provides a simple solution for handling *polymorphic* calls in an inheritance hierarchy: if A is a base class and B is on of its heirs, both implementing the method m, then P_A responds to $call_{A::m}$ by creating a handler process $P_{A::m}$, while P_B responds to both $call_{A::m}$ and $call_{B::m}$, in each case creating a handler process $P_{B::m}$ (Figure 3).
This solution is similar to the one used in most object oriented programming language compilers, where a

[2] This is eased by IF's support for hierarchical automata.

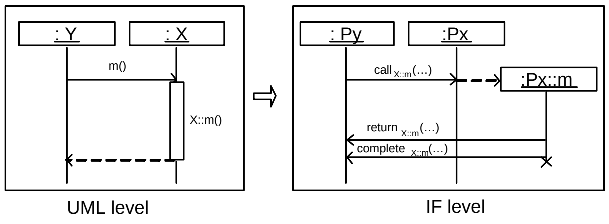

Fig. 2. Handling primitive operation calls using dynamic creation.

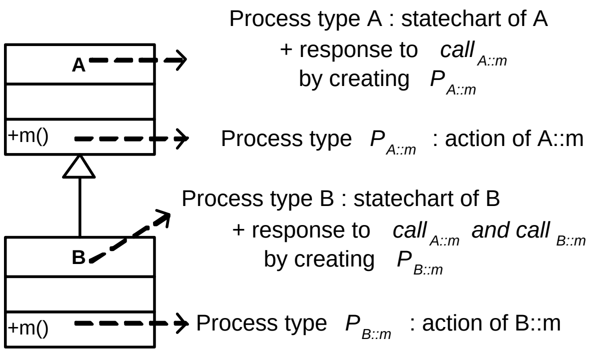
A
+m()
B
+m()
Process type A : statechart of A
+ response to call A::m
by creating P A::m
Process type P A::m : action of A::m
Process type B : statechart of B
+ response to call A::m and call B::m
by creating P B::m
Process type P B::m : action of B::m

Fig. 3. Mapping of primitive operations and inheritance.

"method lookup table" is used for dynamic binding of calls to operations; here, the object's state machine plays the role of the lookup table and the dynamically created method instances represent the call stack.

Mapping of constructors. Constructors differ from primitive operations in that their binding is static. Consequently, they do not need the definition of the $call_{X::m}$ signal and the call action is mapped directly to the creation of the handler process $P_{X::m}$. The handler process begins by creating a P_X object and its components (i.e. all the aggregate objects defined by UML composition relationships), after which it continues execution as a normal operation.

Mapping of state machines. UML state machines are mapped almost syntactically to IF. Several prior research papers tackle the problem of mapping statecharts to (hierarchical) automata (e.g., [?]). The method we apply is similar.

Actions. The action kinds enumerated in section 2.1 are supported as follows:

- *object creation* is modelled by the creation of the constructor's handler process
- *method call* is modelled by sending a *call* signal and waiting for a *return/complete* signal
- *assignment* is directly supported in IF. Access to attributes is supported by the shared variable mechanism.
- *signal output* is directly supported in IF.
- *return action* is modelled by the sending of a *return* signal.
- *control structure actions* are directly supported in IF.

3.2 Modelling run-to-completion with dynamic priorities

The concurrency model introduced in section 2.2 is realized using the dynamic partial priority order mechanism presented in 1.4. As already mentioned, the calls or signals coming from outside an activity group are placed in the group's queue and are handled one by one in run-to-completion steps. In IF, the group management processes (of type GM) handle this queueing and forwarding behaviour.

In order to obtain the desired run-to-completion (RTC) semantics, the following priority rules are applied (the rules concern processes representing instances of UML classes, and not the processes representing operation handlers, etc.):

- All objects of a group have higher priority than their group manager:

  $$(x.leader = y) \Rightarrow x \prec y$$

  This guarantees that *as long as an object inside a group may execute, the group manager will not initiate a new RTC step.*
- Each GM object has an attribute *running* which points to the presently running or most recently run object in the group. This attribute behaves like a token that is taken or released by the objects having something to execute. The priority rule:

  $$(x = y.leader.running) \wedge (x \neq y) \Rightarrow x \prec y$$

  ensures that *as long as an object that is already executing has something more to execute (the continuation of an action, or the initiation of a new spontaneous transition), no other object in the same group may start a transition.*
- Every object x with the behaviour described by a state machine in UML will execute $x.leader.running := x$ at the beginning of each transition. As a consequence of the previous rule, such a transition may be executed only when the previously running object of the group has reached a stable state, which means that the current object may take the *running* token safely.
  The non-deterministic choice of the next object to execute in a group (stated in the semantics) is ensured by the interleaving semantics of IF.

3.3 Preserving non-determinism

High level specifications described in UML usually abstract away from implementation details or scheduling policies of concurrent computations. Such aspects may appear as non-deterministic choices. When verifying properties of the dynamics of a model, it can be important to take into account all possible resolutions of this non determinism by an implementation.

The translation of UML to IF preserves non-determinism at several levels:

- the non-deterministic interleaving of actions in parallel activity groups is preserved by the non-deterministic interleaving of processes in IF (modulo the restrictions induced by priority orders described above).
- the non-deterministic choice of the executing object inside an activity group, when several can be activated (as mentioned above).
- the non-deterministic choices described explicitly by the designer in the behavioural model of an object (e.g. in the state machine).

Notice that this preservation of non determinism holds for interactive simulation where the user might want to see all possible orders and when executions are assumed to take time. For verification purposes, we generally eliminate as much non determinism as allowed for still preserving all properties under consideration (partial order reduction).

4 UML extensions for capturing timing

To build a faithful model of a *real-time* system, one needs to represent different types of timing information:

- time-triggered behaviour (*prescriptive modelling*). For example, it is common practice in real-time programming environments to limit the execution of an action or waiting for a signal by a delay which can be represented by a *timer* object.
- knowledge about the timing of events (*descriptive modelling*). Such information is taken either as a hypothesis under which the system works (e.g., worst case execution times of system actions, scheduler latency, etc.) or as a *requirement* to be imposed upon the system (e.g., required end-to-end response time).

Different UML tools targeting real-time systems adopt different extensions for expressing such timing information. A standard profile targeting real-time applications was defined by the OMG [?] and provides a common set of concepts for modelling timing.

In this work, we are using a subset of the concepts of [?] (timers, clocks, time-related data types and timed events). Concerning timed events, we refine the profile in order to gain some flexibility by identifying a number of event types (e.g. message reception, object creation) and by differentiating event types and their occurrences. We also allow the definition of *duration constraints* between arbitrary events occurring in the system. This framework is described in more detail in [?] and accompanied by a formal semantics in terms of OCL[3].

4.1 Features for modelling timing

Here, we present the subset of the OMEGA time extensions taken into account in the tool. We introduce two time related types: *time* — representing absolute time points — and *duration* — representing time differences or relative time — and a global operator *now* for retrieving the current time (since system start).

The following concepts are used for modelling *time-triggered* behaviour:

- *timer* objects, which measure durations. They may be set to a relative deadline, reset, and they send an asynchronous signal when the deadline is reached.
- *clock* objects, which measure also durations; their value may be consulted by other objects.

For modelling *descriptive timing information*, the extensions defined in [?] allow to:

- identify syntactically many of the meaningful **events** of a system execution. An event has an occurrence time, a type and a set of related information depending on its type. The event types that can be identified are listed in section 5.1, as they also constitute an essential part of our property specification language.
- express **duration constraints** between events identified as above. The constraints may be either *assumptions* (hypotheses to be enforced upon the system runs) or *assertions* (properties to be verified on system runs).
  If several events of the same type and with the same parameters may occur during a run, there are mechanisms for identifying the particular event occurrence that is relevant in a certain context.
- finally, we introduce scheduling related concepts such as resources, execution times and priorities.

The class diagram in Figure 4 shows an example using these features. This model describes a client-server architecture. When a client connects to the server (modelled by *LicServer*) by calling the method *connect*, a worker object (*LicClientWorker*) is created to handle the requests from that specific client. The worker object is supposed to expire after a fixed delay of 10 seconds.

A timing assumption attached to the client (*LicClient*) says that: *"whenever a client connects to the server, it will make a request before its worker object expires, that is before 10 seconds"*. This is specified using two event types, one corresponding to calls (actually, returns from calls) to *LicServer::connect*, and the other corresponding to calls to *LicClientWorker::request*. Using instances of these event types (*ec*, *er*), a duration constraint is specified.

[3] In an earlier version of [?] which can be found in [?], we used timed automata to define the semantics of the real-time profile which makes the translation to IF easier

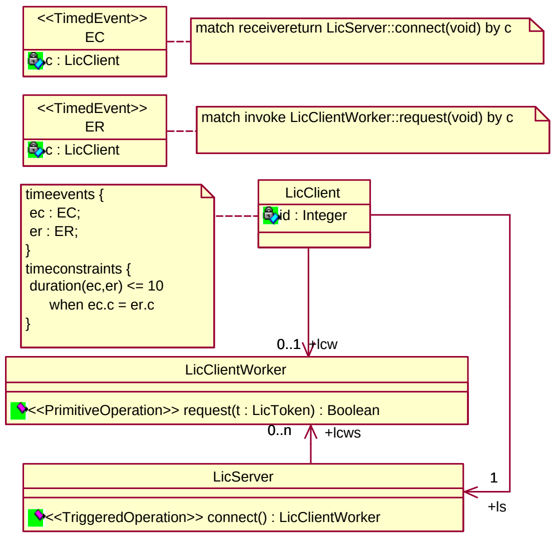

Fig. 4. Using events to describe timing constraints.

4.2 Mapping timed specifications to IF

The time-related concepts presented in the previous section are mapped to IF as follows. *Clock*s exist as a native concept, while *Timer*s are implemented using a clock and a timer process sending timeout signals. *Events* and their associated parameters correspond to transitions of the IF model: for example, the event of invoking an operation $X :: m$ corresponds to the transition in which the $call_{X::m}$ signal is sent.

For expressing timing constraints, there are two alternatives:

– the constraint is *local* to some IF process, in the sense that all involved events are directly observed by the process (e.g. its inputs, outputs, etc.). This is the case in Figure 4. In this case, the constraint may be tested or enforced by looking at the process alone, and by using an additional clock for measuring the duration to be constrained.
– the constraint is *global*, that is depending on events attached to several objects. In that case, the constraint will be tested or enforced by an *observer* or a set of observers, which may possibly be dynamically created, running in parallel with the system.

The tools ensure that runs not satisfying a constraint are either ignored – if it is an assumption, or diagnosed as error – if it is an assertion.

5 Expressing properties by UML observers

For specifying and verifying dynamic properties of UML models, we introduced the notion of *UML observers* analogously to IF observers (section 1.4): they are special objects monitoring run-time *state* and *events*. Observers are described by classes stereotyped with $\ll$ *observer* $\gg$. They may have local memory (*attributes*) and their behaviour is described by a state machine.

As for IF observers, we use sates classified as $\ll$ *error* $\gg$ states or $\ll$ *invalid* $\gg$ states to express properties and hypotheses.

Several examples of properties specified by observers can be found in section 7. For the designer, the advantage of observers compared to other property specification languages is that they use already known concepts while remaining formal and and allow the expression of any safety properties.

5.1 Observations

The main issue in defining observers is the choice of events which trigger their transitions, and which must include specific UML event types. We use the timed events introduced in the OMEGA time extensions [**?**] from which we mention here the most important ones:

– Events related to *signal exchange*: **send**, **receivesignal**, **consumesignal**.
– Events related to *operation calls*: **invoke**, **receive** (reception of call), **accept** (start of actual processing of call – may be different from **receive**), **invokereturn** (sending of a return value), **receivereturn** (reception of the return value), **acceptreturn** (actual consumption of the return value).
– Events related to the execution of *actions* or *transitions*: **start**, **end** and **startend** for instantaneous actions.
– Events related to *states*: **enter**, **exit**.
– Events related to *timers*: **occur**, **timeout** as well as **startend** events associated with timer specific actions **set** and **reset** which are considered instantaneous.

The trigger of a transition is a **match** clause specifying the type of event (e.g., **receive**), some related information (e.g., the operation name) and observer variables that may receive related information (e.g., variables receiving the values of operation call parameters).

Besides events, an observer may access any part of the state of the UML model: object attributes and state, signal queues. In order to express quantitative timing properties, observers may use the concepts available in the OMEGA profile such as *clocks*.

6 A simulation and verification toolset

The translation of UML models to IF models and the validation techniques presented in the previous sections are implemented in an extended version of the IF toolbox - IFx[4]. The architecture of the toolbox is shown in

[4] http://www-verimag.imag.fr/˜ober/IFx.

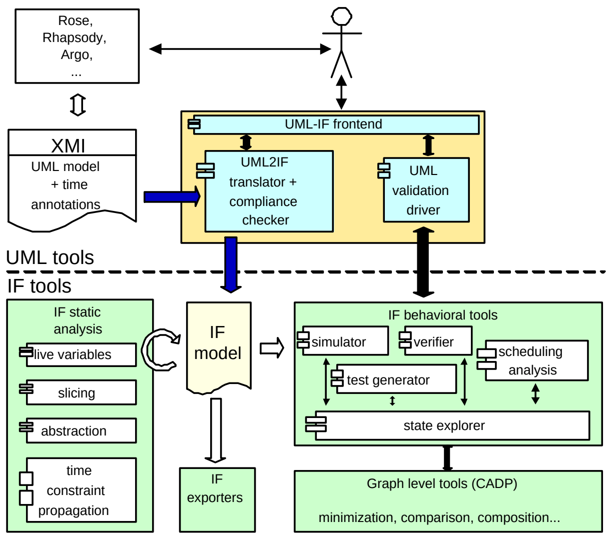

Fig. 5. Architecture of the IFx validation toolbox.

Figure 5. It allows a designer to simulate and verify UML models and observers developed in third-party editors[5] and stored in XMI[6] format.

In a first phase, the tool takes as input a UML model and generates an IF specification and a set of observers by applying the translation rules presented before. During this phase, a first sanity check is performed on the model and results are provided in the form of compiler warnings and errors. They concern action syntax, timing annotation syntax, type errors, etc.

In a second phase, the tool drives the back-end IF simulation and verification tools, and translates the validation results back to the UML level of the original model. The idea is to make the back-end formalism and tools invisible to the designer, but also to enhance the functionality of the IF toolbox by providing more complex interactive simulation features like conditional breakpoints, scenario persistence, custom views for the system state, etc.

Using the IF toolbox as underlying engine gives access to several existing state space reduction and analysis techniques: *static analysis* and *partial order* optimisations for state-space reduction, *symbolic* model exploration, model minimisation and comparison [?]. The use of reduction techniques improves the scalability of the tools, which is an essential feature in the context of UML where large design models are often manipulated.

The tool is being applied on several case studies in the context of the OMEGA project. One of them is presented in some detail in the next section.

[5] The CASE tools that have been tested for compatibility are: Rational Rose Enterprise Edition 2002 / Unisys Rose XMI Add-in 1.3.6 and I-Logix Rhapsody Developer Edition, v4.1, v4.2 and v5.2

[6] XMI 1.0 or 1.1 for UML 1.4

7 Modelling and verification methodology illustrated by the Ariane-5 case study[7]

In this section, we outline a *verification methodology* that may be used when working with the IFx toolbox. We illustrate the steps of our methodology on hand of examples from the Ariane-5 case study.

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. This case study has been performed in collaboration with EADS Space Transportation in the IST OMEGA project, in order to evaluate the applicability of the UML profile and of the validation tools. The study consists in formally specifying some parts of an existing software in UML with Rational Rose and in verifying a set of critical properties on this specification.

7.1 Overview of the Ariane-5 Flight Program

The Ariane-5 example is a non-trivial UML model (23 classes, each one with operations and a state machine) translated into 77 IF processes and about 7000 lines of IF code.

The phases of the flight.

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

After burn-out, the two EAP boosters 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 payload 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) 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 mission ends about 40 minutes after the first ignition command.

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 as follows:

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

- *flight control*, involves guidance, navigation and control algorithms (*GNC*),
- *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).

The UML description models the *regulation* and *configuration* parts in detail. For the *flight control* part, the computational aspects are abstracted to a set of empty control functions, and only the structure of the control cycle (i.e. the flowchart according to which the functions are called) and timing are modelled in detail.

The environment.

In order to obtain a realistic functional model of the flight program, the environment of the launcher software must also be modelled. The ground component abstracts the nominal behaviour of the launch protocol on the ground side, by providing the launch date and confirmations needed for launching. Furthermore, the equipment controlled by the flight program (like valves and pyros) is modelled to allow both success and hardware failure scenarios.

Requirements.

Several safety requirements ensuring the right service of the flight program have been identified and verified on the UML model. The requirements can be classified as follows:

- *general requirements*, not necessarily specific to the flight program but general for all critical real-time systems, such as the absence of deadlocks, signal loss, timelocks;
- *overall system requirements*, specific to the flight program and concerning its global behaviour. For example, it is required that the firing and the extinction functions perform a series of actions in a specific order;
- *local component requirements*, concerning the functionality of some part. For example, it is required that the opening and closing commands sent to the valves conform to their state.
- *scheduling requirements*, concerning the preservation of scheduling objectives under the given the scheduling policy and action execution times.

7.2 UML modelling

The Ariane-5 flight program is modelled as a collection of objects communicating mostly through asynchronous signals, and the behaviour of which is described by state machines. Operations (with an abstract body) are used to model the guidance, navigation and control (GNC)

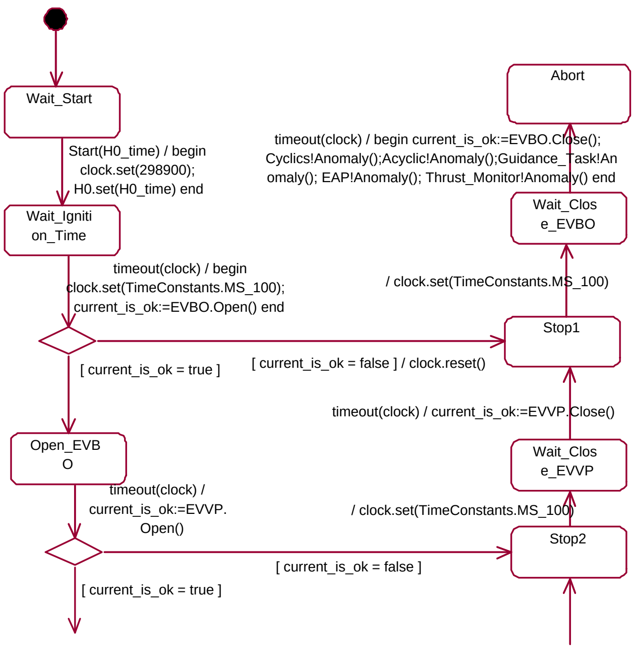
Abort
Wait_Start
Start(H0_time) / begin clock.set(298900); H0.set(H0_time) end
timeout(clock) / begin current_is_ok:=EVBO.Close(); Cyclics!Anomaly();Acyclic!Anomaly();Guidance_Task!Anomaly(); EAP!Anomaly(); Thrust_Monitor!Anomaly() end
Wait_Ignition_Time
Wait_Close_EVBO
timeout(clock) / begin clock.set(TimeConstants.MS_100); current_is_ok:=EVBO.Open() end
/ clock.set(TimeConstants.MS_100)
Stop1
[current_is_ok = true]
[current_is_ok = false] / clock.reset()
timeout(clock) / current_is_ok:=EVVP.Close()
Open_EVBO
Wait_Close_EVVP
timeout(clock) / current_is_ok:=EVVP.Open()
/ clock.set(TimeConstants.MS_100)
Stop2
[current_is_ok = false]
[current_is_ok = true]

Fig. 7. Behaviour of the EPC regulation process (part).

tasks. For modelling timed-dependent behaviour, timers and clocks are being used.

The model (a partial view of its structure is visible in Figure 6) is composed of:

- a global controller class responsible of flight configuration (*Acyclic*);
- a model of the regulation components (e.g., *EAP*, *EPC* corresponding to the launcher stages);
- a model of the regulated equipment (e.g., *Valves*, *Pyros*);
- an abstract model of the cyclic GNC tasks (*Cyclics*, *Thrust_monitor*, etc.);
- a 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 the partial view of the EPC stage controller's behaviour in Figure 7). The behaviour 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. The separation dates are provided by the control part, based on the current flight evolution.

7.3 Validation methodology and results

Formal validation is a complex activity, which may be structured into several tasks as depicted in Figure 8.

Translation to IF and basic static analysis.

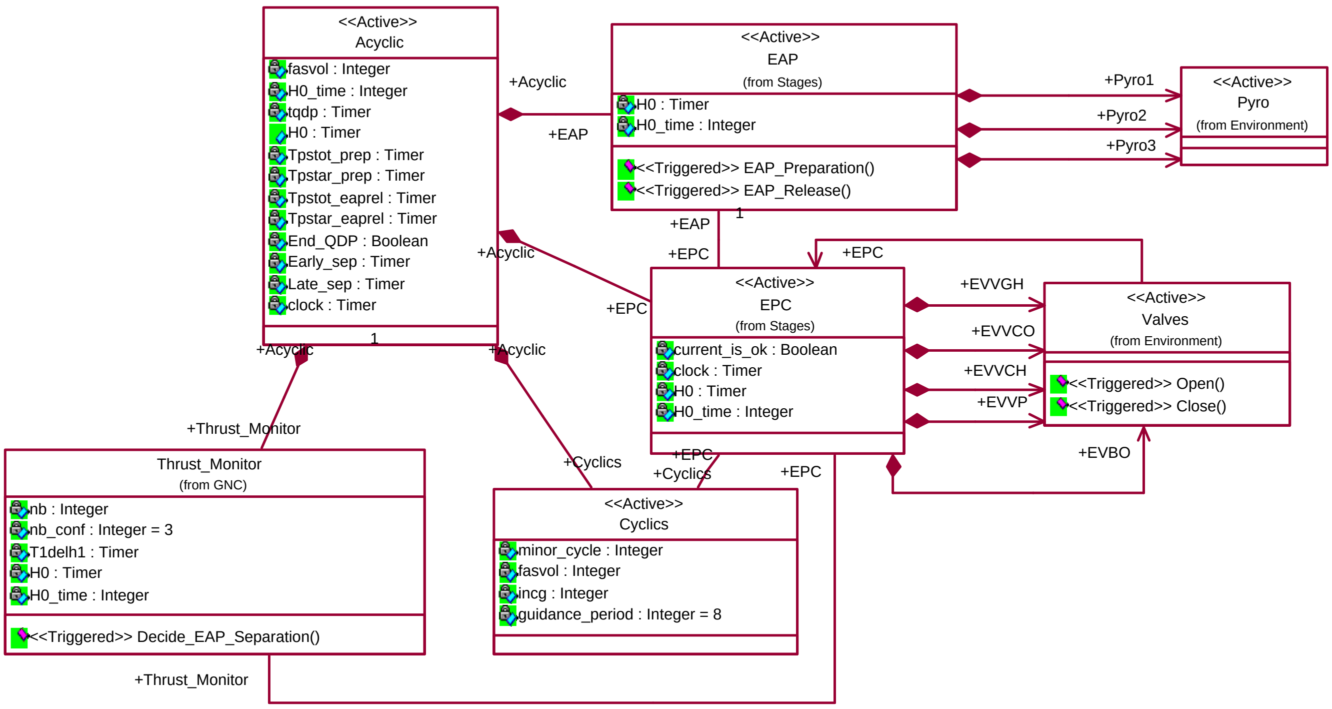

Fig. 6. Structure of the UML specification (part).

[t]

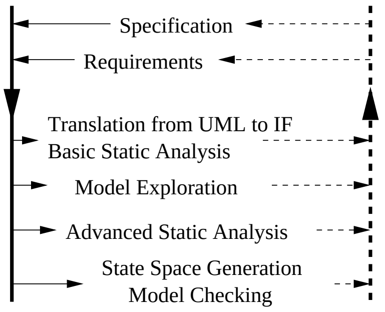

Fig. 8. Verification methodology in IF.

This phase provides a first sanity check of the model. 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).

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. The user can also *test* simple safety properties. Such properties range from generic ones, such as absence of deadlocks or signal loss, to more specific and application dependent ones, e.g., invariants tested using conditional breakpoints.

Advanced static analysis.

The aim at this phase is to prepare the specification to an exhaustive simulation. Optimisation based on static analysis (see section 1.4) are applied in order to reduce both the state vector and the state space, while completely preserving its behaviour.

For example, one possible optimisation introduces systematic resets for variables which are dead in certain control states of the specification. In this way, it prevents the tool to distinguish between simulation states which differ only by values of variables which are dead in a given state. This technique is very effective, given that it can be applied locally at control-state level, and may collapse large (bisimulation equivalent) parts of the state graph. For this case study, however, the live reduction was not impressive due to the relatively small number of loops in the simulation graph of the system.

State space generation and model checking.

Some verification techniques implemented in IFx, like observer and μ-calculus based model checking, work on-the-fly without the need of generating the state space beforehand. Others, like minimisation, work on an already generated state space.

In the context of UML models, the most intuitive verification techniques presented in the following are *model minimisation* and *observer based model checking.*

Model minimisation is an intuitive method for a non expert end-user. It consists in computing an abstract model (with respect to given set of observations) of the overall behaviour of the specification. Such a model can be visualised and possible incorrect behaviours detected by the user. These abstract models are computed by ALDEBARAN (a tool connected to IFx [?]) and, depending on the (bi)-simulation relation used, they preserve different classes of properties.

In order to obtain an abstract model, the state space mist first be generated by exhaustive simulation. In order to cope with the complexity in this phase, 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 property preserving partial order reductions, under-approximating scheduling policies, etc.

Example 1. For Ariane-5, the use of partial order reduction has been useful 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, no signal emission nor access to shared variables. This partial order reduction imposes a fixed exploration order on internal steps and preserves *all* properties expressed in terms of visible actions.
By using partial order reduction of internal steps, we reduced the size of the model by 3 orders of magnitude, i.e, from more than 10^6 states (model generation did not terminate, due also to the large size – about 1KB – of the system state) to about 1000 states and 1200 transitions, which can be easily handled by the minimisation tool.

After the generation of the state space, it can be minimised modulo bisimulation using ALDEBARAN. Minimisation takes into account the observation criteria which are relevant for both, the observations relevant for the property being verified (i.e. the actions that have to remain visible) and the type of property (e.g., safety, absence of deadlocks, etc.).

Example 2. The graph in Figure 9 is the quotient model of Ariane-5 with respect to branching bisimulation [?], in which the only observable events are opening/closing the EPC valves, igniting the EPC stage and detecting anomalies.
The branching structure and all safety properties involving these actions are preserved on the graph from Figure 9. It is easy to check by inspection on this abstract model that if an EAP anomaly occurs, then all the valves are closed and afterwards an EPC anomaly is signalled. Also, it is easy to check that the EPC sends the *Ignition* signal only after all valves have been (correctly) opened.

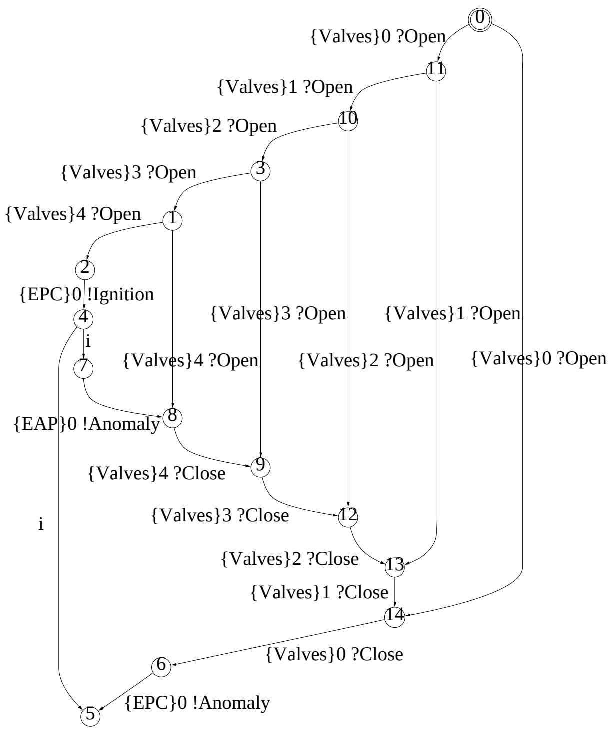

Fig. 9. A minimal model generated with ALDEBARAN.

Observer based model-checking is useful for more complex safety properties, which depend on *quantitative time* or on the values of system variables, signal parameters, etc. This type of verification is done on the fly, while the state graph is generated.

Example 3. Figures 10 to 12 show some of the timed safety properties of Ariane-5 that were checked over the UML model using observers:

Figure 10: between any two commands sent by the flight program to the valves there should elapse at least 50ms.

Figure 11: 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 12: 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 $TimeConstants.MN\_5 * 2 + Tpstar\_prep$.

Scheduling analysis. A particular type of property that can be checked using observers is schedulability of a

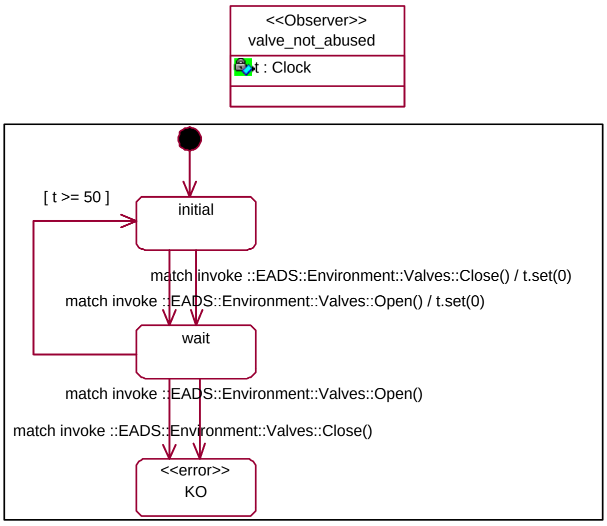
<<Observer>>
valve_not_abused
t : Clock
[t >= 50]
initial
match invoke ::EADS::Environment::Valves::Close() / t.set(0)
match invoke ::EADS::Environment::Valves::Open() / t.set(0)
wait
match invoke ::EADS::Environment::Valves::Open()
match invoke ::EADS::Environment::Valves::Close()
<<error>>
KO

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

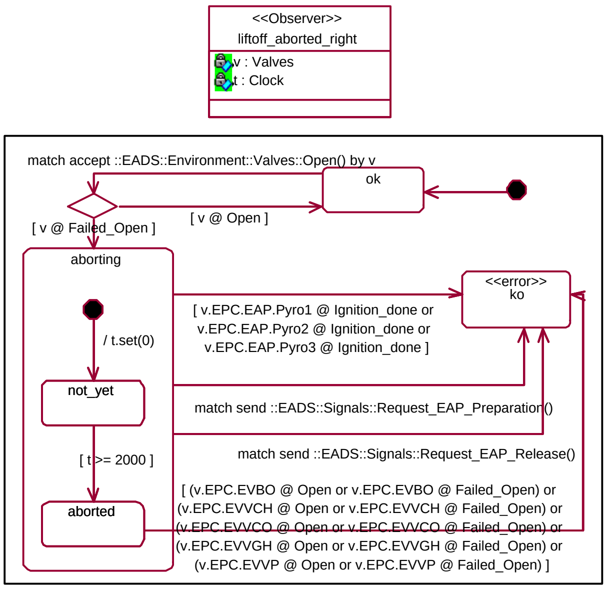
<<Observer>>
liftoff_aborted_right
v : Valves
t : Clock
match accept ::EADS::Environment::Valves::Open() by v
ok
[v @ Failed_Open]
[v @ Open]
aborting
/ t.set(0)
not_yet
[t >= 2000]
aborted
<<error>>
ko
[v.EPC.EAP.Pyro1 @ Ignition_done or
v.EPC.EAP.Pyro2 @ Ignition_done or
v.EPC.EAP.Pyro3 @ Ignition_done]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

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

set of tasks (with arbitrarily complex activation patterns and execution times) on a set of computation resources, according to a predefined scheduling policy.

In order perform an analysis of this type, the following elements have to be modelled:

– *the computation resources (CPUs).* A CPU is a special type of object, upon which other objects request *execution time* necessary for their computations. A request for execution time may be accompanied by a set of parameters (such as priority, or a time deadline). The CPU allocates execution time according to these parameters and to the other requests it is currently processing (in a way determined by its *scheduling policy*), and notifies the requesting object when the execution time has elapsed.

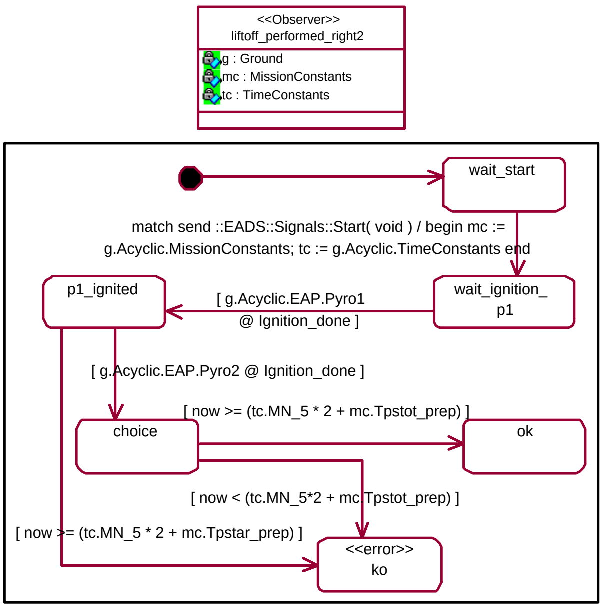
<<Observer>>
liftoff_performed_right2
g : Ground
mc : MissionConstants
tc : TimeConstants
wait_start
match send ::EADS::Signals::Start(void) / begin mc :=
g.Acyclic.MissionConstants; tc := g.Acyclic.TimeConstants end
p1_ignited
[g.Acyclic.EAP.Pyro1
@ Ignition_done]
wait_ignition_
p1
[g.Acyclic.EAP.Pyro2 @ Ignition_done]
[now >= (tc.MN_5 * 2 + mc.Tpstot_prep)]
choice
ok
[now < (tc.MN_5*2 + mc.Tpstot_prep)]
[now >= (tc.MN_5 * 2 + mc.Tpstar_prep)]
<<error>>
ko

Fig. 12. A timed safety property of the Ariane-5 model.

Depending on the complexity of the scheduling policy, CPUs can in most cases be modelled using the features of the OMEGA UML profile described in this paper. For example, a quite general model for a CPU, which uses *dynamic fixed priority preemptive scheduling*, can be modelled using the technique proposed in [?]. In this model each request for execution time comes with a *priority*, which can be computed dynamically by the functional model but is fixed once the request is made. A UML package containing this CPU model is available together with our tools and can be imported and used directly in any system model.

– *the execution requests* made by the different system objects.
– *the scheduling objectives* which are usually safety properties which can be expressed by observers.

Once these elements are modelled, scheduling analysis consists in verifying (model-checking) that the observers encoding the scheduling objectives are not violated.

Example 4. In the Ariane-5 model, tasks performed by the *regulation* components and by the *guidance-navigation-control* components are executed on the same CPU, using a fixed priority preemptive scheduling policy:

– Time consuming tasks of the Regulation components have the highest priority and are sporadic (they appear at certain moments during the 40 minutes flight, according to the application logic, and have small execution times of 2-5ms).
– Time consuming tasks of the Navigation and Control components have medium priority and execute cycli-

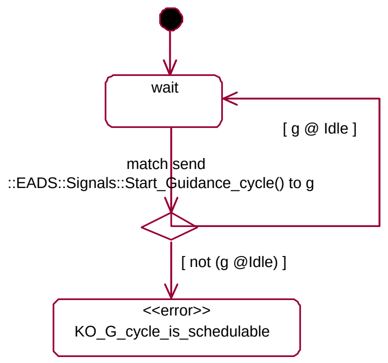
wait
[g @ Idle]
match send
::EADS::Signals::Start_Guidance_cycle() to g
[not (g @Idle)]
<<error>>
KO_G_cycle_is_schedulable

Fig. 13. A scheduling objective expressed as observer.

cally every 72ms. They take 30 to 60ms every cycle depending on the application logic.
– Time consuming tasks of the Guidance component have low priority and execute cyclically every 576ms. They take about 200ms every cycle.

There are several scheduling objectives. We mention here the most basic one, which is that the Navigation-Control computation and the Guidance computation finish in their respective cycle time (72ms, respectively 576ms). The objective for the Guidance task, formalised by an observer, is shown in Figure 13. It describes the fact that, when the Guidance object receives the signal *Start_Guidance_cycle*, it should be in state *Idle*, i.e. it should have finished the computation from the previous cycle.

Verification has been performed on a version of the EADS model in which the cyclic part is fully described, while the acyclic *regulation* part is over-approximated (its logic is preserved, but its timing is modelled as non-deterministic). Verification can provide a yes/no answer to the question of whether the system is schedulable under the described policy. In case of a negative answer, it can also provide hints on which executions overflow and by what amount of CPU time. For example, it has been determined that the the system is safe if *Guidance* computation takes not more than 230ms, but above this value it may sometimes overflow the 576ms cycle.

7.4 Assessment and lessons learned

The EADS case study has shown the feasibility of our approach, but also some of its weaknesses. On the positive side:

– Verification of the functional properties of the acyclic part have been performed on a quite detailed specification of the Regulation components, combined with an abstract specification of the cyclic Guidance-Navigation-Control part. The state space has about 1000 states and is generated in less than 1s, after *static analysis* and using *partial order reduction.*
  These techniques prove to be essential: without static analysis, the state space is infinite due to some counters and clocks which continue to grow after they stop being used. Without partial order reduction, the state space is necessarily finite, but we could not generate it (over 10^6 states, with a state vector of about 10KB).
– Verification of scheduling properties has been performed on a detailed description of the Guidance-Navigation-Control part, combined with a specification of Regulation components from which time has been abstracted away. The state space has around 66000 states and is generated in about 1m10s on a dual Pentium-II system with 2GB of memory.
– We have assessed the impact of translation on the state space size. For this we have compared the size of the state space generated by a hand-written IF description with the size of the space generated by an equivalent model translated from UML. (The comparison considers UML models which only use the structure and communication mechanisms available directly in IF. Inheritance or behaviour described through operations is not considered, as this would require an encoding in IF similar to that implemented by the UML translator.)
  We found that the translation induces a linear growth in space size, by a factor of around 4. This is mostly due to the processes which manage activity groups and the run-to-completion policy of the OMEGA semantics.

On the other hand, the case study has pointed out the necessity of using abstractions. When trying to verify both acyclic and cyclic parts of Ariane-5 without abstraction, the result is an intractable state space explosion.

The IFx tool provides several abstractions which can be applied automatically: removing or resetting dead variables, slicing away irrelevant variables for a given property, partial order reduction, symbolic representation of clock values, queue abstraction. These abstractions are "*exact*" in the sense that they preserve the reachability of observable states and preserve both the satisfaction and the non-satisfaction of safety properties.

However, such abstractions are generally not sufficient for complex problems. To exploit compositionality, one often needs to verify the properties of a component in conjunction with an abstract (*over-approximated*) version of the other components of the system. The state space of the whole system is in this case over-approximated. Such abstractions preserve the satisfaction of safety properties, but do not preserve their non-satisfaction (i.e. may lead to false negative answers).

In the Ariane-5 case study, this technique was exploited: safety properties of the regulation and configuration components were verified using an exact model for the *acyclic* part and an over-approximated behaviour of the *cyclic* part. Likewise, scheduling properties were verified using an exact model of the *cyclic* part and a time-nondeterministic model of the *acyclic* part.

This form of abstraction has to be handled mostly manually: using several versions of the model, checking manually compliance between the abstract and the concrete model of a component. A part of this management burden could be better supported by tools.

Finally, another conclusion of the case study is that static analysis is less effective on models generated from UML. This is caused by the heavy use of dynamic process creation and of shared variables in the IF counterpart. For models where architecture is mostly static, like in the case of Ariane-5, describing the architecture with a diagram (as can be done in UML 2.0) instead of describing the system creation phase with class constructors could largely improve performance of static analysis tools. Also, using operation inlining instead of our compilation scheme (section 3.1) when possible, will improve the impact of static analysis.

8 Conclusions and plans for future work

We have presented a method and a tool for validating UML models by simulation and model checking, based on a mapping to an automata-based model (communicating extended timed automata).

Although this problem has been previously studied [?,?,?,?,?,?], our approach introduces a new dimension by considering the object-oriented features present in UML: inheritance, polymorphism and dynamic binding of operations, and their interplay with statecharts and the concurrency semantics. A solution is given for modelling these concepts with timed automata extended with variables and dynamic creation.

Our experiments show that the overhead introduced by handling these object-oriented aspects during simulation and model checking remains low, thus not hampering the scalability of the approach.

For expressing and verifying dynamic properties, we propose a formalism that remains within the framework of UML: observer objects. We believe this is an important facility for the adoption of formal techniques by the UML community. Observers are a natural way of writing a large class of properties (linear properties with quantitative time).

8.1 Handling semantic variations

Several features of the IF language, such as dynamic addressing, the default atomicity of transitions or the dynamic priority mechanism, make it a satisfactory compromise between expressiveness and level of abstraction for describing different communication and synchronisation schemes. Consequently, our approach of defining the semantics of UML models by translation to IF proves to be flexible and open to semantic variations.

In the future, we plan to exploit the openness of the translation and explore variations in the:

- *communication paradigm.* Currently our model supports communication via asynchronous signal passing, synchronous (blocking) method calls and shared (public) object attributes.
  Extensions may include :
  - Communications which are not point-to-point, such as asynchronous signal multicast or broadcast. They may be mapped to IF by using dynamic addressing which allows processes to communicate without a pre-established link.
  - Asynchronous calls. They may be mapped to IF using an exchange of asynchronous signals, by loosening the constraints of our implementation of blocking calls.
  - Data flow communication between functional modules. This form of communication can be achieved using protected (atomic) access to shared variables. Dynamic priorities can be used to describe generically the activation order of functional modules in a network.
  - Rendez-vous communication is also a powerful communication mechanism used in certain types of systems. It can be implemented in IF by means of a (relatively complex) protocol. Nevertheless, rendez-vous is interesting as a primitive concept, and for this reason, we plan to extend the IF language with a rendez-vous-like primitive [?].
- *concurrency model.* In our model, activity groups are executed concurrently, and requests to a group are treated in run-to-completion steps. Other execution models may be useful in different applications, and can be accommodated by changing the translation to IF:
  - models which loosen some of the hypotheses of the OMEGA semantics. For example, relaxing the hypothesis that a passive object is part of one activity group only, and that calls are sequenced by the activity group, can yield a model closer to that of Java or C++/Posix (in which threads are orthogonal to objects).
  - models which strengthen the hypotheses of the OMEGA semantics, for example by introducing a notion of synchronous step (during which all activity groups execute a run-to-completion step, all communication being taken into account only in the next step).
- *step granularity.* The current semantics supposes that only basic actions (assignments, signal output, etc.) are atomic by construction. Different scales of granularity, up to forcing entire run-to-completion steps as atomic, are possible depending on the considered applications.

8.2 UML 2.0 and other future plans

The present work focuses on UML 1.4 as this is the most recent version of UML implemented by mature and open (in the sense of XMI export) CASE tools. In the future we plan to adapt this work to UML 2.0.

On the side of the OMEGA UML profile, the extensions proposed here are compatible with UML 2.0: the concurrency and communication model is a specialisation of that of UML 2.0, the action language syntax is compatible as there were no major changes versions 1.4 and 2.0. Observers are defined by means of standard extension mechanisms which have been preserved. Finally, the declarative time constraints defined in OMEGA have a formal counterpart in UML 2.0 (*Duration*, *DurationInterval*, *DurationConstraint*, etc. from *Common Behaviors*) but the precise identification of event types and occurrences, and the different kinds of event pair matching defined in OMEGA are still missing.

We also plan to integrate the component and architecture specification frameworks of UML 2.0 and to study the possibility of using these additional structures for improving verification, static analysis and abstractions.

On the side of translation tools, upgrading to UML 2.0 will bring major changes in the XMI format and the tool's internal repository structure which is an image of the metamodel. However, both XMI de-serialisation and UML-to-IF translation are built based on the reflectivity capabilities of Java and are loosely coupled with the repository. Also, a new repository for UML 2.0 can be generated automatically from an XMI representation of the metamodel, which we expect to be available from the OMG as it was the case with UML 1.4.

Finally, our plans include assessment of the applicability of our technique to larger models. The tool is already being applied to a set of case studies provided by industrial partners within the OMEGA project.

Acknowledgements. The authors wish to thank Marius Bozga and Yassine Lakhnech who contributed with ideas and help throughout this work.

References

1. 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.
2. R. Alur and D.L. Dill. A theory of timed automata. In *TCS94*, 1994.
3. Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. Model checking UML specifications of real time software. In *Proceedings of 8th International Conference on Engineering of Complex Computer Systems*. IEEE, 2002.
4. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.
5. M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the ALDEBARAN toolset. *Software Tools for Technology Transfer*, 1:166–183, 1997.
6. M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of Formal Methods'99, Toulouse, France*, June 1999.
7. Marius Bozga, Susanne Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen*, number 2404 in LNCS. Springer Verlag, June 2002.
8. Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM-04:RT 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, number 3185 in LNCS, June 2004.
9. Marius Bozga and Yassine Lakhnech. IF-2.0 common language operational semantics. Technical report, 2002. Deliverable of the IST Advance project, available from the authors.
10. Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97 - 11th European Conference on Object-Oriented Programming*, number 1241 in LNCS. Springer Verlag, 1997.
11. OMEGA consortium. http://www-omega.imag.fr - website of the IST OMEGA project.
12. 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*. Kluwer Academic Publishers, 1999.
13. 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*, volume 2852 of *LNCS Tutorials*. Springer Verlag, November 2002.
14. Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. *Lecture Notes in Computer Science*, 1536:186–238, 1998.
15. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. OMEGA Project Deliverable D.1.1.1 : A Formal Semantics for a UML Kernel Language. Technical report, 2002. Available at http://www-omega.imag.fr.
16. Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE'2002)*, volume 2306 of *LNCS*, pages 218–232. Springer-Verlag, April 2002.
17. Maria del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, August 2002. (http://www.jot.fm/issues/issue 2002 07/article1).
18. Elena Fersman, Leonid Mokrushin, Paul Pettersson, , and Wang Yi. Schedulability analysis using two clocks. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.

19. R.B. France, A.S. Evans, K.C. Lano, and B. Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, 1998.
20. Susanne Graf and Jozef 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 3047, pages 241–249. Springer-Verlag, May 2004.
21. Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations in UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS 2003), a satellite event of UML 2003, San Francisco, October 2003*, October 2003. downloadable through http://www-verimag.imag.fr/EVENTS/SVERTS/.
22. Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations in UML. *STTT, Int. Journal on Software Tools for Technology Transfer*, 2005. accepted for publication.
23. Gregor Gössler and Joseph Sifakis. Component-based construction of deadlock-free systems. In *proceedings of FSTTCS 2003, Mumbai, India*, LNCS 2914, pages 420–433, 2003. downloadable through http://www-verimag.imag.fr/ sifakis/.
24. Gregor Gössler and Joseph Sifakis. Priority systems. In *proceedings of FMCO'03*, LNCS 3188, 2004.
25. D. Harel and H. Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *Lect. Notes in Comp. Sci.*, pages 325–354. Springer-Verlag, 2004.
26. D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenarios-Based Requirements. In *Formal Methods in Software and System Modeling*, volume 3393 of *Lect. Notes in Comp. Sci.*, pages 309–324. Springer-Verlag, 2005. To appear.
27. D. Harel, H. Kugler, and G. Weiss. Some Methodological Observations Resulting from Experience Using LSCs and the Play-In/Play-Out Approach. In *Proc. Scenarios: Models, Algorithms and Tools*, Lect. Notes in Comp. Sci. Springer-Verlag, 2005. To appear.
28. David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.
29. David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
30. David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, Weizmann Institute of Science, Rehovot, Israel, 2000.
31. Z. Har'El and R. P. Kurshan. Software for Analysis of Coordination. In *Conference on System Science Engineering*. Pergamon Press, 1988.
32. G. J. Holzmann. The model-checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 1999.
33. C. Jard, R. Groz, and J.F. Monin. Development of VEDA, a prototyping tool for distributed algorithms. *IEEE Transactions on Software Engineering*, 14(3):339–352, March 1988.
34. Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.
35. Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Bran Selic Andy Evans, Stuart Kent, editor, *Proceedings of UML'2000*, volume 1939 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
36. Marcel Kyas and Frank S. de Boer. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Compositional Verification in UML*, volume 101 of *entcs*, pages 73–93. elsevier, 2004.
37. Marcel Kyas, Harald Fecher, Frank S. de Boer, Mark van der Zwaag, Jozef Hooman, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. In *Workshop on Semantic Foundations of Engineering Design Languages*, Electronic Notes in Computer Science. Elsevier, 2004.
38. Marcel Kyas, Joost Jacob, Ileana Ober, Iulian Ober, and Angelika Votintseva. OMEGA Project Deliverable D.2.2.2 Annex 1 : OMEGA Syntax for Users. Technical report, 2004. Available at http://www-omega.imag.fr.
39. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.
40. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioral subset of UML statechart diagrams using the SPiN model-checker. *Formal Aspects of Computing*, (11), 1999.
41. J. Lilius and I.P. Paltor. Formalizing UML state machines for model checking. In Rumpe France, editor, *Proceedings of UML'1999*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
42. Johan Lilius and Ivan Porres Paltor. vUML: A tool for verifying UML models. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering*. IEEE, 1999.
43. 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.
44. Erich Mikk, Yassine Lakhnech, and Michael Siegel. Hierarchical automata as a model for statecharts. In *Proceedings of Asian Computer Science Conference*, volume 1345 of *LNCS*. Springer Verlag, 1997.
45. Iulian Ober and Ileana Stan. On the concurrent object model of UML. In *Proceedings of EUROPAR'99*, LNCS. Springer Verlag, 1999.
46. OMG. Unified Modeling Language Specification (Action Semantics). OMG Adopted Specification, document ptc/02-01-09, January 2002.
47. OMG. UML Profile for Schedulability, Performance, and Time Specification. OMG ducument formal/03-09-01, September 2003.
48. Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.
49. M. van der Zwaag and J. Hooman. A semantics of communicating reactive objects with timing. *STTT, Int. Journal on Software Tools for Technology Transfer*, 2005. accepted for publication.

50. Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.
51. WOODDES. Workshop on concurrency issues in UML. Satelite workshop of UML'2001. See http://wooddes.intranet.gr/uml2001/Home.htm.
52. Fei Xie, Vladimir Levin, and James C. Browne. Model checking for an executable subset of UML. In *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01)*. IEEE, 2001.
53. S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.