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: 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 behavioral 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 (defined in [22]) 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 stronger limiting conditions on the input models.

For expressing properties about models, a formalism called *UML observes* is defined in this paper. Observers reuse existing concepts like classes and state machines, and they may express 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.

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

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 [10]) show that this integration can only work if validation takes into account widely used modeling languages. Currently, UML based model driven development encounters a big success with 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 [1]) 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 UML profile for real-time applications [15, 13], and a formal semantics of it [27]. 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* are the starting point for this work :

- *UML is broader than what we need or can handle in automatic validation.* In UML 1.4 [39] there are 9 types of diagrams and about 150 language concepts (metaclasses). Some of them are too informal to be useful in validation (for example use cases) while for others the relationships and the coherence with the rest of the UML model are not clearly (nor uniquely) defined (for example collaborations, system-level activity diagrams, deployment diagrams).
  In consequence, in this work we focused on a subset of UML concepts that define an operational view of the modeled system: objects, their structure and their behavior.
- *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 [8,11], 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 theses tools function on 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 using 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 *behavior* 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.

This semantics is implemented by translating UML models to IF descriptions. IF [7] is a formal language based on *communicating extended timed automata* (CETA), for which there exist powerful validation tools[8,11], and which has been productively used in a number of research projects and case studies [9,21]. The main features of the IF model are presented in section 1.4. The translation is explained in section 3.

An important issue in designing real-time systems is the ability to capture quantitative timing requirements and assumptions, as well as time dependent behavior. A set of *timing extensions* for UML are defined in the OMEGA profile [22,20], and are summarized in section 4 together with their mapping to IF.

Section 5 presents a lightweight extension of UML (*observers classes*) which is used as a *property* description language. Instances of observer classes may express a large class of linear temporal properties, by using a specific semantics for their state machines. Experience shows that the use of such familiar concepts alleviates the cultural shock of introducing formal verification to UML users.

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

1.3 Related work

The application of formal analysis techniques (and particularly model checking) to UML is a very active field of study in recent years, as witnessed by the number of papers on this subject ([34,35,33,31,30,41,18,19,44,4] are most oftenly cited).

Like ourselves, many of these authors base their work on existing model checkers (SPIN[26] in the case of [34, 35,33,41], COSPAN[25] in the case of [44], Kronos[45] for [4] and UPPAAL[29] for [30]), and on the mapping of UML to the input language of the respective tool.

For specifying properties, some authors opt for the property language of the model checker itself, e.g., [33–35]. Other authors [30,41] use UML collaboration or sequence diagrams, which specify required or forbidden sequences of messages between objects, but are too weak to express more complex properties. We propose the use of a variant of UML classes and state machines to express properties.

Concerning language coverage, all previous approaches are restricted to *flat class structures* (no inheritance) and to behaviors, specified *exclusively by statecharts*. In this respect, many important features which make UML an object-oriented formalism (inheritance, polymorphism and dynamic binding of operations) are missed. The approach presented in this paper is, to our knowledge, the first to fill this gap. However, the material cited above, together with previous work on Statecharts ([24,16,37] to mention a few), provided us inspiration for handling of UML state machines.

The concurrency model of the OMEGA profile is inspired by the concurrency model of the Rhapsody tool [23]. The improvements are the formalisation of its semantics, and a more relaxed interpretation of nondeterminism which allows a higher level of abstraction and opening to different implementations (Rhapsody models have a predefined scheduling scheme). In the definition of the profile we also took inspiration from our previous assessment of the UML concurrency model [38], and from other positions on this topic (see for example [43]).

Finally, the work presented in this paper is part of a broader effort [1] 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 well formedness checks
– timed model checking models against observers (presented here) as well as scheduler analysis based on a timed automata model

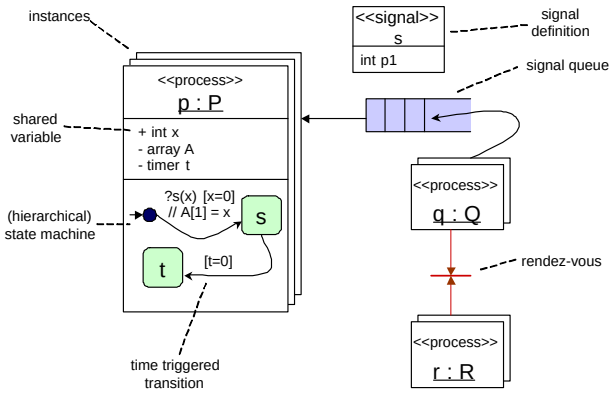

Fig. 1. Constituents of a communicating extended automata model in IF.

- non-timed model checking against LTL formulas or LSC specifications (a variant of interaction diagrams with stronger structuring constructs [14])
- state diagram synthesis from LSC specifications
- deductive verification using the interactive theorem prover PVS: compositional verification, consistency checks and reasoning with OCL specifications

For more detail, the reader is referred to [1].

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 environment built around this model [8,11,12]. We summarize the elements of this model in the following..

Modeling with communicating extended timed automata

The IF language and the associated toolset developed at VERIMAG are conceived for modeling 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 higher level formalisms such as UML or SDL, and is also used as a format for inter-connecting model-based tools.

An IF description defines the structure of a system and the behavior 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 behavior is defined by 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*, which makes it easier to describe their formal semantics in terms of *finite automata*. 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 inter-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 modeling object systems.

IF provides support for real time constraints expressed using *clock* variables and guard conditions on them. The values of such variables increases with time. The underlying semantics is based on *finite timed automata with urgency* [3,5].

For more details on the IF model and its semantics, the reader is referred to [8,11,12].

A framework for modeling 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 formalized in [2].

Property description and verification with observers

Temporal properties may be expressed in IF using *observers*. These are special processes which execute synchronously with the system, and which may 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* states. Thus, observers may be used to express *safety properties*. A re-interpretation of success states as accepting states of a Büchi automaton could also allow observers to express liveness properties.

IF observers are inspired by the observer concept introduced by Jard, Groz and Monin in the VEDA tool [28]. This intuitive and powerful property specification formalism has been adapted over the past 15 years to other modeling languages (LOTOS, SDL) and implemented in industrial case tools like ObjectGEODE.

Analysis techniques and the IF-2 toolbox

The IF toolbox [8,11] is the validation environment built around the language presented before. It is composed of three categories of tools:

1. **behavioral 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** which provide source-level optimizations that help reducing furthermore the state space of the models, and thus improve the chance of obtaining results from the behavioral tools. The implemented *data* and *control* flow analysis techniques 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 [26], Agatha [36], etc.)

The toolbox has already been used in a series of industrial-size case studies [8,11].

2 The OMEGA UML profile

This section outlines the main features of the OMEGA UML profile [32,17,15] 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.
- *Generalizations*. 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 like [34,35,33,31,30,41,19,44,4].

In order to describe a meaningful behavior 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. Begining 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, which thus varies from one tool to another.

The OMEGA profile [32] 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 details concerning the concrete syntax of this action language are left out of the scope of this paper.

Additionally to the elements mentioned above, a number of UML extensions for describing timing constraints and assumptions are supported. They were introduced in [20,22] and are discussed in section 4.

2.2 The execution model

The purpose of this section is to illustrate some of the particularities of the OMEGA model and not to give its complete formal semantics, which may be found in [17,15,27]. The execution model chosen in OMEGA and presented here is an extension of the execution model of the Rhapsody UML tool (see [23] 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*. 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, more precisely to the group 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 execute concurrently, and objects inside the same an activity group execute sequentially. The consequence is that requests (asynchronous signals or operation calls) coming from other groups (or even from the same in case asynchronous signals) are placed in a queue belonging to the activity group. They are handled one by one when the whole group is *stable*.

The motivation for making activity groups sequential is to have some level of protection against concurrent access to shared data in the group. Moreover, this hypothesis implies that every activity group has (or may be seen as) a single control thread, which simplifies subsequent analysis (like scheduling analisys) and implementation.

An *activity group* is stable when all its objects are stable. An *object* is stable if it has nothing to execute spontaneously and no pending operation call from inside its group. Note that an object is not necessarily stable when it reaches a stable state in the state machine, as

there may be transitions that can be taken simply upon satisfaction of a Boolean condition.

The above notion of stability defines a notion of *run-to-completion* step for activity groups: a step is the sequence of actions executed by the objects of a group from the moment an external request is taken from the activity group's queue by one of the objects, and until the whole group becomes stable. During a step, other requests coming from outside the activity group are not handled and are queued.

The semantics of activity groups described here corresponds to that of concurrent, internally-sequential *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.

In the UML model we distinguish syntactically between two kinds of operations: *triggered* operations and *primitive* operations.

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, besides asynchronous signals. Triggered operations differ from *asynchronous signals* in that they may have a return value.

Primitive operations are more close 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 stacked and handled immediately (i.e. on the same control thread), like in usual programming languages. However, in case of triggered operation calls, the dynamic call graph between objects should be acyclic, since an object that has already made a call from inside a triggered operation is necessarily in an unstable state of the state machine and may not handle any more calls. (This type of condition may be verified using the IF mapping.)

Calls to primitive or triggered methods *from other activity groups* are queued and handled by the target of the call in a subsequent run-to-completion step. The caller (and its group) are blocked until the return of the call.

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 so that there is no intra-group concurrency created by sending 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 runtime 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. As inheritance is flattened, all inherited attributes and associations are replicated in the processes corresponding to each heir class.

Activity group management. Each *activity group* is managed at runtime by a special *group manager* process (of type GM). This process sequentializes the calls and the signals coming from objects in other activity groups, and helps to ensure the run-to-completion policy. Each 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. *waiting* indicates the process that waits for the completion of the call in order to continue execution (either the *caller* if it is in the same group as the *callee*, or the group manager of *callee* otherwise). *caller* designates the process that is 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 it.
- 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 $call_{X::m}$).
- if the operation is *primitive* (see section 2.2), a process type
  $P_{X::m}(waiting : pid, caller : pid, callee : pid, p_1 : t_1, p_2 : t_2, ...)$
  which describes the behavior of the operation using an automaton. The parameters have the same meaning as in the $call_{X::m}$ signal. The *callee* PID is used

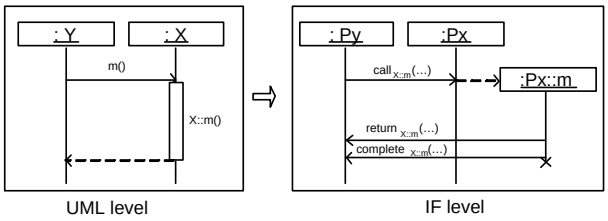

Fig. 2. Handling primitive operation calls using dynamic creation.

to access local attributes of the called object, via the shared variable mechanism of IF.
– if the operation is *triggered* (see section 2.2), its implementation is modeled in the state machine of P_X. Transitions triggered by a $X :: m$ call event in the UML state machine will be 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* (if the caller is in a different group). The group manager queues the call and forwards it to the destination when the group becomes stable.

The handling of incoming calls is modeled by transition loops in every state[1] of the process P_X, which, upon reception of a $call_{X::m}$ create a new instance of $P_{X::m}$ and wait for it to finish execution (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 "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.

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 like a normal operation.

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

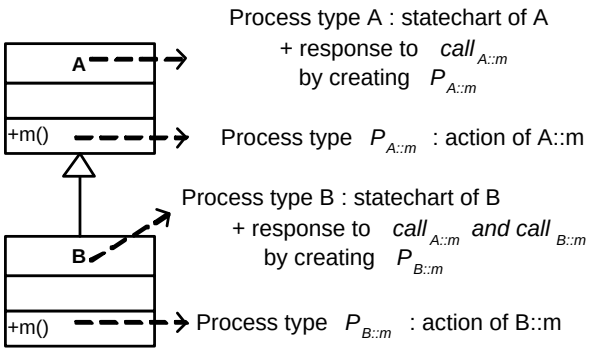

Fig. 3. Mapping of primitive operations and inheritance.

Mapping of state machines. UML state machines are mapped almost syntactically in IF. Several prior research papers tackle the problem of mapping statecharts to (hierarchical) automata (e.g., [37]). The method we apply is similar.

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

– *object creation* is modeled by the creation of the constructor's handler process
– *method call* is modeled 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 modeled by the sending of a *return* signal.
– *control structure actions* are directly supported in IF.

3.2 Modeling 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 simple queuing and forwarding behavior.

In order to obtain the desired run-to-completion (RTC) semantics, the following priority protocol is 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 or most recently running 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 behavior 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.

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 behavior (*prescriptive modeling*). For example, it is common practice in real-time programming environments to link the execution of an action to the expiration of a delay (represented sometimes by a *timer* object).
- knowledge about the timing of events (*descriptive modeling*). 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., end-to-end response time).

Different UML tools targeting real-time systems adopt different extensions for expressing such timing information. A standard Real-Time Profile, defined by the OMG [40], provides a common set of concepts for modeling timing, but their definition remains mostly syntactic.

In this work, we are using the framework defined in [22] for modeling timed systems. Its main ideas are given in the following. The framework reuses some of the concepts of the standard real-time profile (timers, certain data types), and additionally allows expressing *duration constraints* between various events occurring in the system.

4.1 Features for modeling timing

The following concepts, compatible with those of the standard real-time UML profile [40], are used for modeling *time-triggered behavior*:

- two data types: *Time* and *Duration*, and a global operator *now* for retrieving the current absolute time (since system start).
- *timer* objects, which measure time. They may be set for a deadline, reset, and they send an asynchronous signal upon expiry.
- *clock* objects, which measure time and their relative value may be consulted by other objects.

For modeling *descriptive timing information*, the extensions defined in [22] allow to:

- identify syntactically many of the meaningful **events** occurring in 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 tested 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.

The class diagram in figure 4 contains an example using these features. This model describes a client-server architecture in which worker objects on the server are supposed to expire after a fixed delay of 10 seconds. A timing assumption attached to the client says that: *"whenever a client connects to the server, it will make a request before its worker object expires, that is before 10 seconds".*

For additional details on this framework for modeling timing and its semantics, the reader is referred to [22].

4.2 Validation of timed specifications

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 manager process sending timeout signals. *Events* and their associated parameters can be identified in the IF model: for example, the event of invoking an operation $X :: m$ equates to the IF event of sending the $call_{X::m}$ signal, etc.

For testing or enforcing a timing constraint, there are two alternatives:

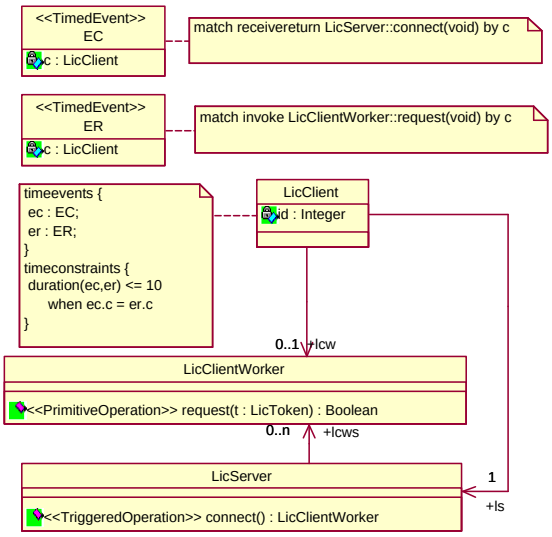

Fig. 4. Using events to describe timing constraints.

– the constraint is *local* to an IF process, in the sense that all involved events are directly observed by the process. (For example, the outputs and inputs of a process are directly observed by itself, but they are not visible to other processes.) This is the case in figure 4. In this case, the constraint may be tested or enforced by the IF process itself, using an additional clock for measuring the duration concerned by the constraint.
– the constraint is not local to a process (it is *global*). In that case, the constraint will be tested or enforced by an observer running in parallel with the system.

The semantics implemented by tools ensures 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 use an operational formalism: *UML observers*. Similarly to IF observers (section 1.4), these are special objects executing synchronously with a system and monitoring run-time *state* and *events*.

Observers are described in UML by classes stereotyped with ≪ *observer* ≫. They may have a local memory (*attributes*) and their behavior is described by a state machine.

Properties are expressed by classifying some observer states as ≪ *error* ≫ states. A system execution which leads the observer to an error state is a violation of the property. An observer may also formalize a hypothesis under which the system works, by marking some of the states as ≪ *invalid* ≫ with respect to the hypothesis.

Several examples of properties specified with 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 sufficiently expressive for a large class of linear temporal properties.

5.1 Observations

The main issue in defining observers is the choice of event types which trigger their transitions, and which must include specific UML event types. We are using the following event types defined in [22]:

– 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 *signal exchange*: **send**, **receive**, **consume**.
– Events related to *actions or transitions*: **start**, **end** (of execution).
– Events related to *states*: **entry**, **exit**.
– Events related to *timers* (this notion is specific to the model considered in [20,22] and in this work): **set**, **reset**, **occur**, **consume**.

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 parametersP).

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 The simulation and verification toolset

The translation of UML models to IF and the validation techniques presented in the previous sections are implemented in an extended version of the IF toolset - IFx[2]. The architecture of the toolset is shown in figure 5. With it, a designer may simulate and verify UML models and observers developed in third-party editors[3] and stored in XMI[4] 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

[2] http://www-verimag.imag.fr/~ober/IFx.
[3] Rational Rose, I-Logix Rhapsody and Argo UML have been tested for compatibility.
[4] XMI 1.0 or 1.1 for UML 1.4

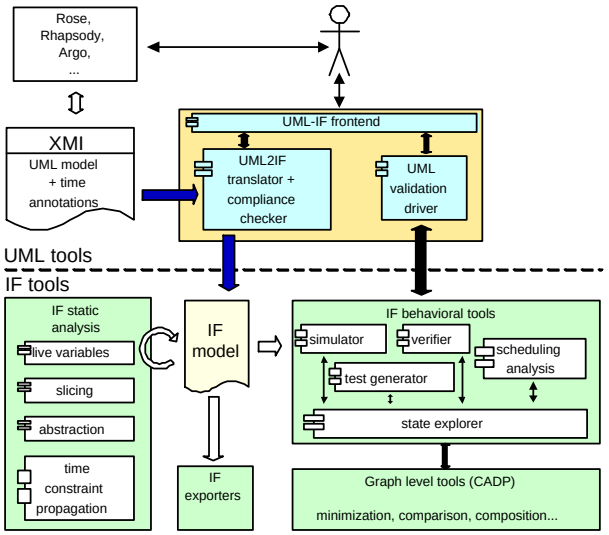

Fig. 5. Architecture of the IFx validation toolbox.

model and results are provided in the form of compile 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 level of the original model. The idea is to make the back-end 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 tools as underlying engine gives access to several existing state space reduction and analysis techniques: *static analysis* and *partial order* optimizations for state-space reduction, *symbolic* model exploration, model minimization and comparison [8,11]. 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 section 7.

7 The Ariane-5 case study [5]

This case study has been performed in collaboration with EADS Launch Vehicles in the IST OMEGA project, in order to evaluate the applicability of both the description language and the validation tools. The study consisted 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. 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.

In the following we summarize the interesting results of the experiment, and give the main lines of a *verification methodology* that may be used in connection with the IFx toolbox. Indeed, even if recent research has considerably improved the efficiency of validation tools, one is still unlikely to be able to apply them on large examples in a strict push-button manner, and some form of iterative methodology is necessary.

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 about 7000 lines of IF code.

The launcher flight.

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

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

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

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

The flight program.

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

The main functions of the flight program are the following:

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

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

The UML description models only the *regulation* and *configuration* parts in detail. The *flight control* part is abstracted as a set of empty control functions, since this part is a relatively independent synchronous reactive control system.

The environment.

In order to obtain a realistic functional model of the flight program, the environment of the launcher software must also be modeled. We have considered an abstract and deterministic version of the environment, in which the flight control part sends the right flight commands at specific moments in time which correspond to a particular flight of the Ariane-5 launcher. The ground part abstracts the nominal behavior 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) are modeled 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 behavior. For example, it is required that the firing and the extinction (in case of anomaly) sequences 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.

7.2 UML modeling

The Ariane-5 flight program is modeled as a collection of objects communicating mostly through asynchronous signals, and the behavior of which is described by state machines. Operations (with an abstract body) are used to model the guidance, navigation and control tasks. For modeling timed-dependent behavior, 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's stages);
- a model of the regulated equipment (e.g., *Valves*, *Pyros*);

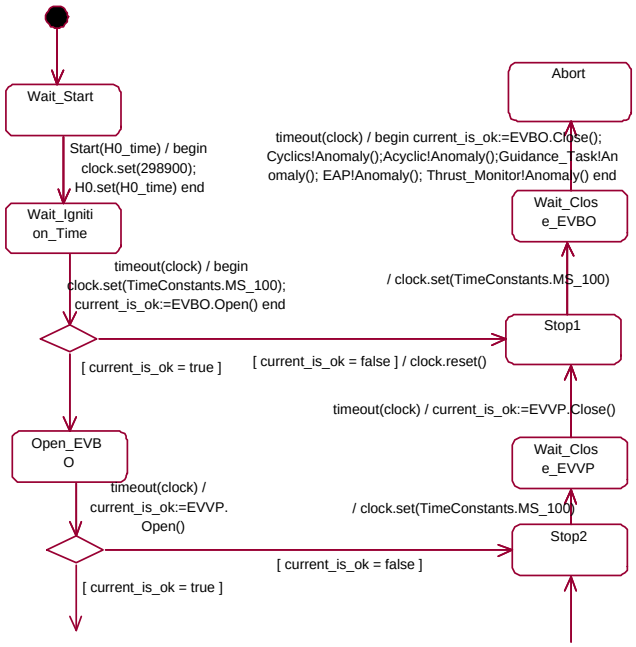

Fig. 7. Behavior of the EPC regulation process (part).

- 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 modeling the communication with synchronous GNC tasks).

The behavior 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 behavior in figure 7). The sequence is time-driven, with the possibility of safe abortion in case of anomaly.

The flight configuration part implements several tasks: EAP separation, EPC separation, payload separation, etc. 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.

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 (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 guided or random manner. Simu-

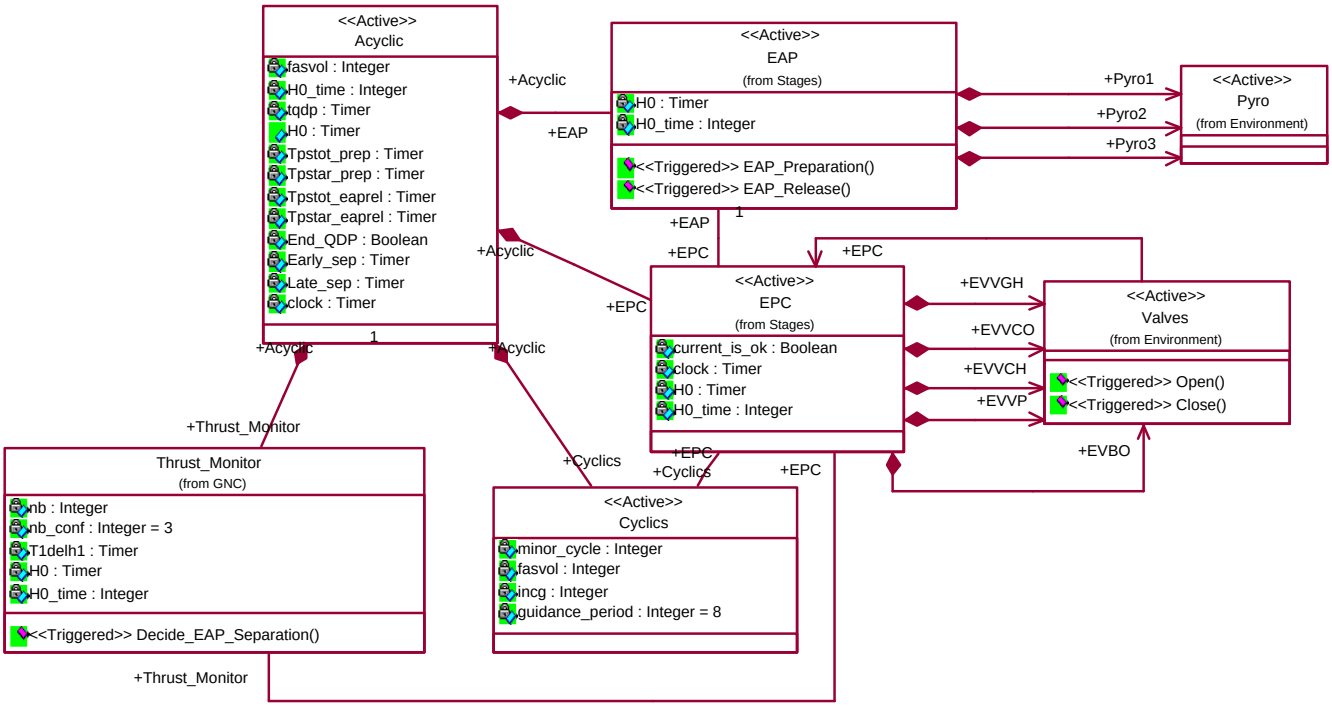

Fig. 6. Structure of the UML specification (part).

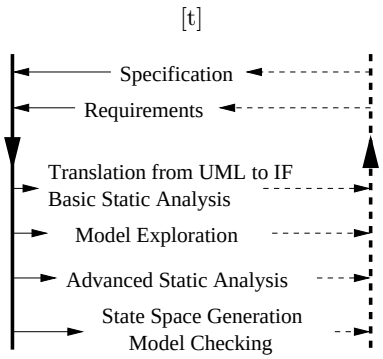

Fig. 8. Verification methodology in IF.

lation 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. Secondly, the user can *test* simple safety properties, which might hold on all execution paths. Such properties might 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. Optimization 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 behavior.

For example, one possible optimization introduces systematic resets for variables which are dead in a certain control state of the specification. In this way, it prevents the simulator to distinguish between simulation states which differ only by values of dead variables. This technique is very effective given that it can be applied locally at control-state level, and may collapse large (bisimulation equivalent) parts of the state graph. For this case study, however, the live reduction was not impressive due to the relatively reduced number 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 model checking, work on the fly without the need of generating the state space beforehand. Others, like model abstraction and minimization, work on a generated state space.

In the context of UML models, the most intuitive verification techniques presented in the following are *model minimisation* and *observer 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 observation criteria) of the overall behavior of the specification. Such a model can be then visualised and possible incorrect behaviors can be detected. These abstract models are computed by Aldebaran (a tool connected to IFx [6]) and, depending on

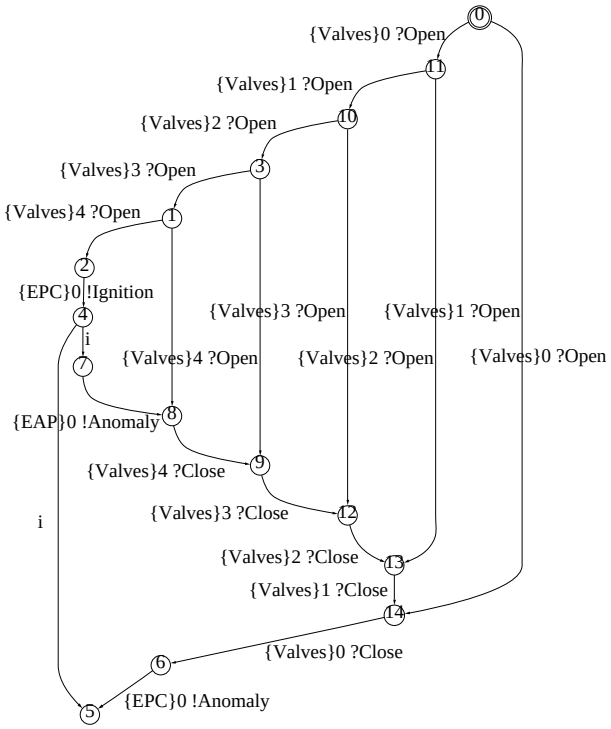

Fig. 9. A minimal model generated with ALDEBARAN.

the (bi)-simulation relation used, they preserve different classes of properties.

In order to obtain an abstract model, the state space is first generated by exhaustive simulation. In order to cope with 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 partial order reductions, scheduling policies, etc.

Example 1. For Ariane-5, the use of partial order reduction has been necessary to construct tractable models. We applied a simple *static* partial order reduction which eliminates spurious interleaving between internal steps occurring in different processes at the same time. Internal steps are those which do not perform visible communication actions, neither signal emission nor 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.
By using partial order reduction on 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 finish, 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 model checker.

After the state space is generated, it may be abstracted with ALDEBARAN. Abstraction takes into account both the observation criteria which are relevant for the property being verified (i.e. the actions that have to remain visible), and the type of property that has to be preserved (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 [42] and keeping observable only some of the actions. In this case we are interested in the actions corresponding to the opening/closing of the EPC valves, the ignition of the EPC stage and the detection of anomalies
The branching structure and all safety properties involving these actions are preserved on the graph from figure 9. It is easy to check on this abstract model that if an EAP anomaly occurs, then all the valves are closed and afterwards an EPC anomaly is signaled. Also, it is easy to check that the EPC sends the *Ignition* signal only after all valves have been (correctly) opened.

Observer model-checking may be used 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, but the different state representations and exploration strategies presented in the previous paragraph may be applied.

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 $TimeConstants.MN\_5 * 2 + Tpstar\_prep$.

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 [18,34,33,31,30,41], our approach introduces a new

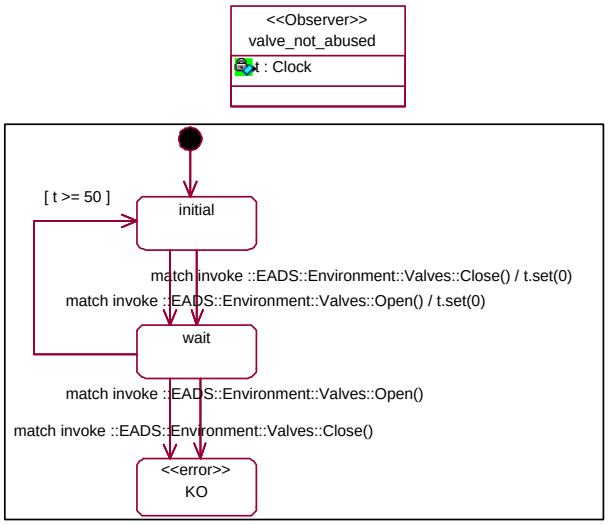

Fig. 10. A timed safety property of the Ariane-5 model.

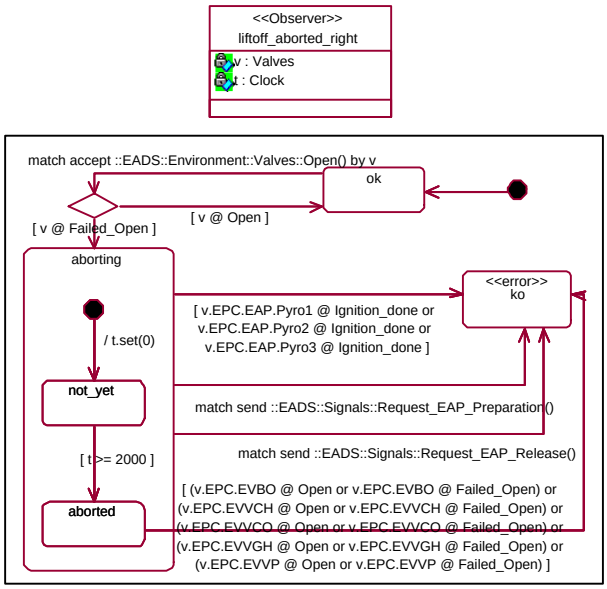

Fig. 11. A timed safety property of the Ariane-5 model.

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 modeling 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

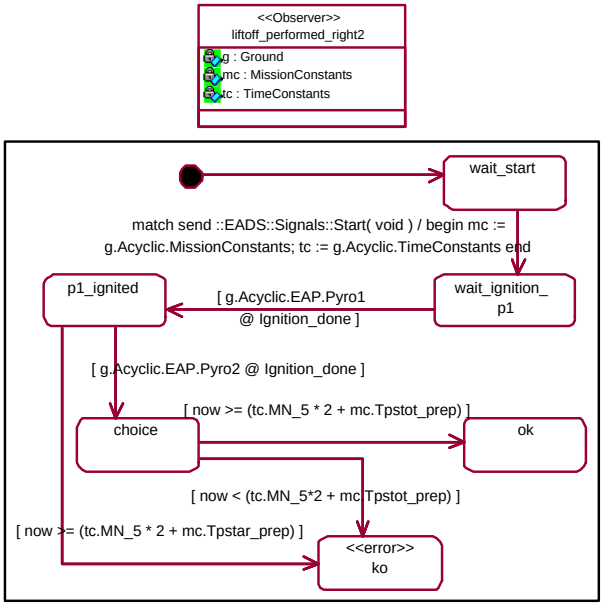

Fig. 12. A timed safety property of the Ariane-5 model.

a large class of properties (linear properties with quantitative time).

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

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

References

1. http://www-omega.imag.fr - website of the IST OMEGA project.
2. 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.
3. R. Alur and D.L. Dill. A theory of timed automata. In *TCS94*, 1994.
4. 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.
5. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.
6. 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.
7. M. Bozga, J.Cl. 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 Symposium on Formal Methods 99, Toulouse*, number 1708 in LNCS. Springer Verlag, September 1999.

8. M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen*, LNCS. Springer Verlag, June 2002.
9. M. Bozga, D. Lesens, and L. Mounier. Model-Checking Ariane-5 Flight Program. In *Proceedings of FMICS'01 (Paris, France)*, pages 211–227. INRIA, 2001.
10. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. IF: An Intermediate Representation for SDL and its Applications. In R. Dssouli, G. Bochmann, and Y. Lahav, editors, *Proceedings of SDL FORUM'99 (Montreal, Canada)*, pages 423–440. Elsevier, June 1999.
11. Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the IF environment. In *2001 IEEE International Symposium on Network Computing and Applications (NCA 2001)*. IEEE, October 2001.
12. 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.
13. The Omega consortium (writers: Susanne Graf and Jozef Hooman). The Omega vision and workplan. Technical report, Omega Project deliverable, 2003.
14. 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.
15. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Proceedings of FMCO'02*, LNCS. Springer Verlag, November 2002.
16. 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.
17. 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.
18. 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.
19. 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).
20. S. Graf and I. Ober. A real-time profile for UML and how to adapt it to SDL. In *Proceedings of SDL Forum 2003 (to appear)*, LNCS, 2003.
21. Susanne Graf and Guoping Jia. Verification experiments on the MASCARA protocol. In *Proceedings of SPIN Workshop '01 (Toronto, Canada)*, January 2001.
22. Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations with UML. Proceedings of SVERTS'2003, satellite workshop of ≪UML'2003≫. VERIMAG, 2003.
23. David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.
24. David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
25. Z. Har'El and R. P. Kurshan. Software for Analysis of Coordination. In *Conference on System Science Engineering*. Pergamon Press, 1988.
26. G. J. Holzmann. The model-checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 1999.
27. J. Hooman and M.B. van der Zwaag. A semantics of communicating reactive objects with timing. In *Proceedings of SVERTS'03 (Specification and Validation of UML models for Real Time and Embedded Systems)*, San Francisco, October 2003.
28. 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.
29. H. Jensen, K.G. Larsen, and A. Skou. Scaling up UPPAAL: Automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT 2000*, 2000.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.

38. Iulian Ober and Ileana Stan. On the concurrent object model of UML. In *Proceedings of EUROPAR'99*, LNCS. Springer Verlag, 1999.
39. OMG. Unified Modeling Language Specification (Action Semantics). OMG Adopted Specification, December 2001.
40. OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG ducument ad/2002-03-04, March 2002.
41. 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.
42. 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.
43. WOODDES. Workshop on concurrency issues in UML. Satelite workshop of UML'2001. See http://wooddes.intranet.gr/uml2001/Home.htm.
44. 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.
45. S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.