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

Abstract. We present in this paper a technique and a tool for validating operational UML models by simulation and verification of dynamic properties.
With respect to language coverage, our approach takes into consideration most of the structural and behavioral characteristics of classes and their interplay. We tackle issues like the combination of *operations*, *state machines*, *inheritance* and *polymorphism*, with a particular *run-to-completion* and *concurrency* semantics. This is an important point, as many previous approaches applying model checking to UML put limiting conditions on the models.
The UML dialect considered here also includes a set of extensions for expressing timing, which were defined in detail in [18].
For writing properties about models, we introduce *UML observer objects*. Observers are both easy to use – they reuse existing concepts of UML, and powerful — they are equivalent to linear temporal logic.
Our approach is implemented by a tool built on top of an XMI repository. The tool is connected to several commercial and non-commercial UML editors, and to other model checking tools.

1 Introduction

The Unified Modeling Language makes it possible to describe operational models of a system at various levels of abstraction corresponding to the different phases of development. In a model driven development process, the ability to check and rapidly prototype these (partial) models is an important added value that a CASE tool may offer.

In this paper we present a technique and a tool that offers capabilities for simulating UML models and verifying satisfaction of complex behavioral properties.

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

In terms of language coverage, we focus on the operational part of UML: classes with structural and behavioral features, relationships (associations, inheritance), behavior descriptions through state machines and actions. The issues that we tackle, like the combination of *operations* and *state machines*, *inheritance* and *polymorphism*, *run-to-completion* and *concurrency*, go beyond the results of previous work done in this area (see section 1.1), which has only focused on the simulation and verification of statecharts.

The definition of a simulation and verification framework for UML models involves a number of design choices. They concern notably the set of *covered UML language elements*, possible *language extensions* (such as the formalism for specifying actions), and the *semantics* of a model. Our choices are outlined in section 2, and in section 3 we discuss how they are implemented by our framework.

An important issue in designing real-time systems is the ability of the modeling formalism to capture quantitative timing requirements and assumptions, as well as time dependent behavior. In this work we rely on the extensions defined for this purpose by Graf et al. in [18, 16]. We summarize these extensions and their use in analysis in section 4.

Simulation in our framework is based on a guided construction of the graph of states that may be reached by the model. This is a common technique that comes from formal design languages and is already used by some UML tools, especially those targeting real-time or critical systems development[1]. The offered functionality is that of an advanced debugger, including scenario replay, manual solving of non-determinism, control of scheduling policy and time parameters, etc.

Verification of dynamic properties is a powerful model validation technique, which is less used primarily because some forms of it are less intuitive and cannot be used by non-specialists. An important factor in verification is the formalism in which properties are expressed, which may range from temporal logics to automata-based specifications. In section 5 we introduce a simple property description language that reuses some concepts from UML (like objects, state machines) while remaining sufficiently expressive for a large class of properties (equivalent to the linear temporal logic LTL). The language of *observer objects* makes use of concepts that are familiar to any UML user, and has the potential to alleviate the cultural shock of introducing formal dynamic verification to UML models.

The entire approach presented in this paper is based on the formal model of *communicating extended timed automata* (CETA) which is presented as a preliminary in subsection 1.2. The model, as well as a system description language and a simulation/verification toolkit based on it were previously presented in [6, 8] and have been productively used in a number of research projects and case studies [7, 17].

Section 6 presents the *UML validation toolset* we created based on the approach presented here. By using the IF tools as underlying simulation and veri-

[1] Examples include Rational Rose RT, I-Logix' Rhapsody and Telelogic's TAU G2.

fication engine, the UML tools presented here benefit from a large spectrum of model reduction and analysis techniques already implemented therein. Among them we mention *static analysis* and optimizations for state-space reduction, *partial order* reductions, some forms of *symbolic* exploration, model minimization and comparison, etc [6, 8]. These techniques improve the scalability of the tools, which is essential when analyzing complex UML models.

The techniques and the tool presented in this paper are subject to experimental validation on several larger case studies within the OMEGA research project [1].

At the end of the paper, we draw conclusions and present future work plans.

1.1 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 ([24, 25, 23, 22, 21, 31, 14, 15, 33, 4] are most oftenly cited).

Like in our case, most of these authors base their work on an existing model checker (SPIN in the case of [24, 25, 23, 31], COSPAN in the case of [33], Kronos for [4], UPPAAL for [21]), 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 (e.g. [23, 24, 25]) while others make use of UML collaboration diagrams (e.g. [21, 31]).

Concerning language coverage, however, all these authors restrict themselves to *flat class structures* (no inheritance) and to behaviors specified *only by statecharts*. In this respect, many important features which make UML an object-oriented formalism (inheritance, polymorphism and dynamic binding of operations) are missed. Our approach is, to our knowledge, the first to try to fill this gap.

For our handling of UML state machines (not described in detail in this paper), the starting point was the material cited above together with previous work on Statecharts ([19, 13, 26] to mention only a few). In the definition of our concurrency model we have taken inspiration from our previous assessment of the UML concurrency model [29], and from other positions on this topic (see for example [32]).

1.2 The back-end model, techniques and tools

The validation approach proposed in this work is based on the formal model of communicating extended timed automata (CETA) and on the IF environment built around this model [6, 8]. We summarize the elements of this model in the following.

Modeling with communicating extended automata

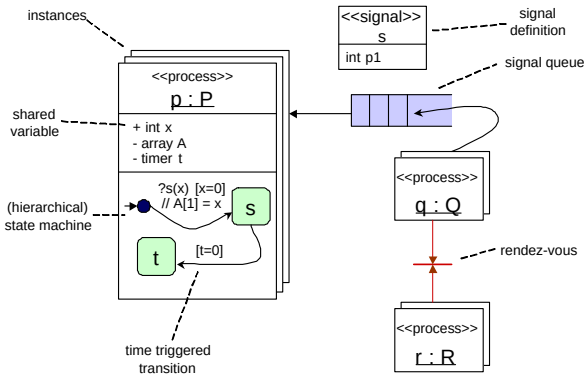

Fig. 1. Constituents of a CETA model.

The CETA model and IF were developed at VERIMAG in order to provide an instrument for modeling and validating *distributed systems* that can manipulate *complex data*, may involve *dynamic aspects* and *real time constraints*. Additionally, the model allows to describe the semantics of higher level formalisms (e.g. UML or SDL) and has been used as a format for inter-connecting validation tools.

In this model, a system is composed of a set of communicating *processes* that run in parallel (see figure 1). Processes are instances of *process types*, which have their own identity (PID), which may own complex data variables (defined through ADA-like data type definitions), and whose 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 parallels 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*. As we will see in the sequel, operations, inheritance and other notions may be layered on top of the CETA model resulting in a more modular definition of the semantics of object models.

Processes may inter-communicate via *asynchronous signals* (similar to the UML 1.4 homonym), via shared variables (similar to public attributes in UML), or via rendez-vous (synchronous handshake of two processes). 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.

As mentioned before, the CETA model and IF provide support for real time constraints, which may be expressed using special *clock* variables (having a value that increases with time) and guard conditions on them. The underlying semantics for CETA is based on *finite timed automata with urgency* [3, 5].

For more details on the model, the reader is referred to the papers presenting either the IF language [6, 8] or its semantics [9].

A framework for modeling priority

On top of the above model, we use a framework for specifying dynamic priorities via partial orders between processes. The framework was formalized in [2]. Basically, a CETA model is associated with a set of priority directives of the form: (*state condition*) $\Rightarrow p_1 \prec p_2$. They are interpreted as follows: given a system state and a directive, if the condition of the directive holds in that state, then process with ID p_1 has priority over p_2 for the next move (meaning that if p_1 has an enabled transition, then p_2 is not allowed to move).

Property description and verification with observers

In the CETA model and the IF tool, dynamic properties of systems may be expressed using *observer automata*. These are special processes that may monitor and react both to changes in the *state* of a model (variable values, contents of queues, etc.) and to *events* occurring in it (inputs/outputs, creation/destruction of processes, etc.).

Observers may have an arbitrary complex state machine and internal data variables. They are executed in parallel with CETA model, the semantics being that of a "weak synchronous composition", i.e.:

- the observer receives the control after each atomic step (transition) of the system
- depending on the events and conditions occurring in the system, the observer executes 0 or more steps, in a run-to-completion fashion

For expressing properties, states are classified (syntactically) as *ordinary*, *error* or *success*. An observer may be used to express a *safety property*, case in which the success/error states are considered as final states of a finite automaton. Alternatively, it may be used to express a *liveness property*, case in which the success states are considered as accepting states of a Büchi automaton.

Analysis techniques and the IF-2 toolbox

The IF-2 toolbox [6, 8] is the validation environment built around the model 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. Among the state of the art techniques that are implemented we mention *data flow analysis* (e.g. dead variable reduction) and *slicing*.
3. **front-ends and exporting tools** which provide source-level coupling to higher-level languages (UML, SDL) and to other verification tools (Spin, Agatha, etc.)

More details on the functionality and the scalability of the IF-2 tools, as well as the industrial case studies on which it has been applied may be found in [6, 8].

2 Ingredients of UML models

This section outlines our design choices with respect to:

- the **UML concepts covered** (outlined in section 2.1).
- the **computation** and **concurrency model**, which are among the semantic variation points of UML that have to be fixed by every particular tool (described in section 2.2).

2.1 UML concepts covered

In this work we consider an operational subset of UML, which includes the following model element types:

- *Classes* : active or passive (see explanations in 2.2).
- *Operations* : triggered/primitive (see explanations in 2.2), constructors, destructors.
- *Signals* for asynchronous communication.
- *Attributes* with *basic types* or object *reference types*.
- *Basic data types* : Integer, Boolean, Real for the moment.
- *Associations* : simple and composite, with bounded multiplicity.
- *Generalizations*. Their semantics implies polymorphism and dynamic binding of operations.
- *Statecharts* They are not presented in this paper as already tackled in many previous works like [24, 25, 23, 22, 21, 31, 15, 33, 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.

In this work we use a textual action language compatible with UML 1.4, which covers: *object creation/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 language is of minor importance in 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 [16, 18] and are discussed in section 4.

2.2 The execution model

We describe in this section some of the design choices made with respect to the computation and the concurrency model implemented by our method and tools. The purpose is to illustrate some of the particularities of the model and not to give a complete/formal semantics for UML. Actually, a precise semantics is given by our proposed mapping to CETA/IF, outlined in the next section and implemented by our tools.

The execution model presented in the following is the one proposed in [12].

Activity groups and concurrency. There are two kinds of classes: *active* and *passive*, both being described by attributes, relationships, operations and state machines.

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, which is either the activity group of the active object that owns the passive object by composition (if there is one), or the activity group of the object that created the passive object.

Activity groups execute in parallel, and objects inside an activity group execute sequentially. This means that requests (signals or operations) are sequentialized at the border of the activity group, and handled one by one when the whole group is *stable*.

The notion of *stability* is defined as follows: an *object* is stable if it has nothing to execute spontaneously and no pending operation call from inside its group. An *activity group* is stable when all objects that form it are stable.

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 the group from the moment an external request is taken by one of the group objects, and until the group becomes stable. During a step, other requests coming from outside the activity group are not handled and are enqueued.

Operations and state machines. We assume two kinds of operations (distinguished syntactically), allowed in any class: *primitive* operations and *triggered* operations. Primitive operations have the body described by a method (with an associated action), while triggered operations are handled in the state machine of a class, and their effect is described on transitions. (Triggered operations differ from *signals* in that they may have a return value)

With respect to concurrency, the two kinds of operations are handled differently: at any moment an object having the control may call a primitive operation on an object from the same activity group, and the call is stacked and handled immediately. On the contrary, triggered operations and signals go to the boundary of the active group and are queued for handling in a later run-to-completion step. Primitive operation calls that transgress the boundary of an active group are also queued and handled like signals and triggered operations.

3 Mapping UML models to CETA/IF

In this section we show that it is possible to capture the behavior of a UML model using a CETA model. We give the main lines of the mapping here. For more detail concerning the concrete translation of UML into the IF language the reader is referred to [28].

3.1 Mapping the object domain to CETA

Mapping of attributes and associations. Every class X is mapped to a process type P_X that will have a local variable of corresponding type for each attribute or association of X. As inheritance is flattened in the CETA model, all inherited attributes and associations are replicated in the processes corresponding to each heir class.

Additionally, a process type GM will implement active group management, with an instance for each active group during model execution. In each P_X there is a local variable *leader*, which for each instance will point to the process GM managing its active 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 the CETA model:

- 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. If the call is made in the same activity group, *waiting* indicates the process that waits for the completion of the call in order to continue execution. *caller* designates the process that is waiting for a return value, while callee designates the process corresponding to the object 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 2.2), a process type $P_{X::m}(waiting : pid, caller : pid, callee : pid, p_1 : t_1, p_2 : t_2, ...)$ which will describe the behavior of the operation using a CETA 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 CETA.
- if the operation is *triggered* (see 2.2), its implementation will be modeled in the state machine of P_X (see the respective section below). Transitions triggered by a $X :: m$ call event in the UML state machine will be triggered by $call_{X::m}$ in the CETA automaton.

The action of invoking an operation $X :: m$ is modeled in CETA by the sending of a signal $call_{X::m}$. The signal is sent either directly to the concerned

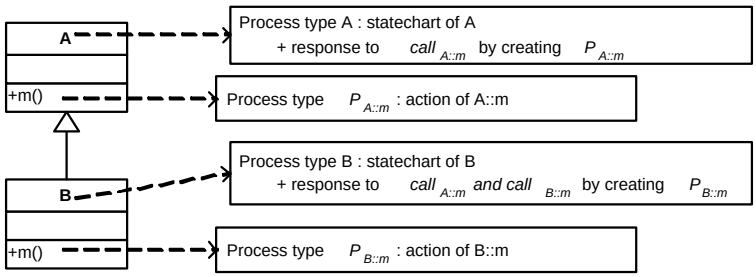

Fig. 2. Mapping of primitive operations and inheritance.

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 will enqueue the call and will forward it to the destination when the group becomes stable.

The handling of incoming calls is simply modeled by transition loops (introduced in every state[2] of the process P_X) which, upon reception of a $call_{X::m}$ will create a new instance of the automaton $P_{X::m}$ and wait for it to finish execution.

The above mapping provides a simple solution for handling *polymorphic* calls in an inheritance hierarchy: if A and B are a class and its heir, both implementing the method m, then P_A will respond to $call_{A::m}$ by creating a handler process $P_{A::m}$, while P_B will respond to both $call_{A::m}$ and $call_{B::m}$, in each case creating a handler process $P_{B::m}$ (figure 2).

This solution is similar to the one used in most object oriented programming language compilers, where a "method lookup table" (called differently from one language to another) 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 (take $X :: m$ in the following) differ from primitive operations in one respect: their binding is static. As such, they do not need the definition of the $call_{X::m}$ signal and the call (creation) action is directly the creation of the handler process $P_{X::m}$. The handler process begins by creating a P_X object and its strong aggregates, after which it continues execution like a normal operation.

Mapping of signals and state machines. UML signals are mapped to signals of the CETA model. UML state machines are mapped almost directly in CETA state machines. Certain transformations are necessary in order to support features that are not directly in the CETA model, such as entry/exit actions, fork/join nodes, history, etc.

Several prior research results tackle the problem of mapping UML statecharts to (hierarchical) automata (e.g. [26]). The method we apply is similar to such approaches.

Actions.

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

[2] This is eased by the fact that CETA/IF support hierarchical automata.

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

3.2 Modeling run to completion and the activity group concept with dynamic priorities

The main lines of the concurrency model for UML considered in this work were introduced in section 2.2. We discuss here how this model is realized in CETA using the dynamic partial priority order mechanism presented in 1.2.

As mentioned, the calls or signals coming from outside an activity group are queued at the border of the group and handled one by one in run-to-completion steps. In the CETA model, the group management objects (GM) handle the simple enqueuing and forwarding behavior.

In order to obtain the desired run-to-completion (RTC), 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 priorities than their group manager:
  $\forall x, y.\, (x.leader = y) \Rightarrow x \prec y$
  This ensures that as long as an object inside the group may move, the group manager will not initiate the next 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 following priority rule:
  $\forall x, y.\, (x = y.leader.running) \wedge (x \neq y) \Rightarrow x \prec y$
  ensures that as long as an object has something to execute (the continuation of an action, or the initiation of a new spontaneous transition), no other object in the group may run.
- Every object x with the behavior described by a statechart in UML will execute $x.leader.running := x$ at the beginning of each transition. In regard of the previous rule, such a transition is 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.

3.3 The semantic model of a UML specification

Having defined the CETA model corresponding to a UML specification, the semantic state/transition graph of the specification is defined as the product

automaton corresponding to the CETA model. This automaton is characterized by the following:

- all the runtime objects at a specific point in the execution, as well as the call stacks, pending messages, etc. are identifiable as a part of the global automaton state. This allows tracing back to the UML specification during the simulation or verification.
- UML model's intrinsic parallelism is captured by the interleaving of CETA transitions corresponding to steps in parallel components. The notion of step is very fine grained (sub-action size); steps corresponding to UML model actions and control steps (RTC and concurrency management, communication, etc.) are identifiable.

The simulation and verification tools (section 6) work by constructing the CETA model and applying different reduction and analysis techniques available with IF (section 1.2).

4 UML extensions for capturing timing

In order to build a faithful model of a *real-time* system in UML, one needs to represent several 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 as a hypothesis under which the system works. Examples are the worst case execution times of system actions, scheduler latency, etc.

In addition to that, a high-level UML model may also contain timing *requirements* to be imposed upon the system.

Different UML tools targeting real-time systems adopt different UML extensions for expressing such timing information. A standard UML Real-Time Profile, defined by the OMG [30], provides a common set of concepts for modeling timing, but their definition remains mostly syntactic. For a discussion of these approaches, the reader is referred to [18].

In this work, we are using the framework defined in [18] 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 [30] (e.g. timers, certain data types), and additionally allows expressing *duration constraints* between various events occurring in the system.

4.1 Features for modeling timing

For modeling *time-triggered behavior*, we are using a small set of concepts compatible with those of [30]:

– two data types: *Time* and *Duration*, and a global expression *now* denoting the current absolute time.
– *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*, we use a set of features which allow us to:

– identify syntactically many of the meaningful **events** occurring in a UML 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 5.2, as they also constitute an essential part of our property specification language presented in section 5.
– 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 example in figure 3 shows how these features may be used in a UML model. This model describes a typical 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, the reader is referred to [18]. That paper also gives a formal semantics to the above concepts using timed automata, which is the basis of the analysis performed by our tools.

4.2 Validation of timed specifications

The modeling concepts outlined in the previous section translate naturally to CETA. *Clock*s exist natively in the model, and timers may be simulated using a clock and a manager process. All the UML *events* enumerated before, and their associated parameters, can be identified in the CETA model. For example: the UML event of invoking an operation $X :: m$ equates to the CETA event of sending the $call_{X::m}$ signal, etc.

For testing or enforcing a timing constraint from the UML model, we are presented with two alternatives:

– the constraint is *local* to a CETA 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 3.

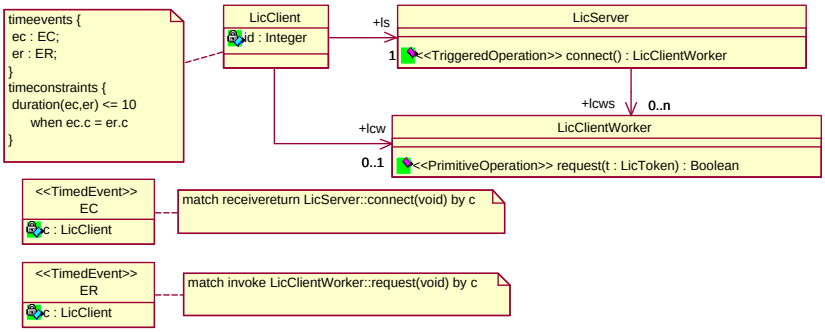

Fig. 3. Using events to describe timing constraints.

In this case, the constraint may be tested or enforced by the CETA process itself, using an additional clock for measuring the duration concerned by the constraint, as well as a CETA transition with an appropriate guard on that clock.

– or, the constraint is not local to a CETA process (we call it *global*). In that case, the constraint will be tested or enforced by a CETA observer running in parallel with the system.

The tools will 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 Dynamic properties written as UML observers

We discuss in this section a technique for specifying and verifying dynamic properties of UML models, that we call *UML observers*. UML observers, which are similar to CETA observer automata (section 1.2), may be seen as special objects which run in parallel with a UML system and monitor its *state* and the *events* that occur.

Observers are described by special classes stereotyped with $\ll observer \gg$. They may own attributes and methods, and may be created dynamically. An important part of the observer is its *state machine*, which is triggered by events occurring in the UML model, as we will see in the following.

Both UML and IF observers are rooted in the observer concept introduced by Jard, Groz et Monin in the VEDA tool [20]. This intuitive and powerful property specification formalism has been adapted over the past 15 years to other languages (LOTOS, SDL) and implemented by industrial case tools like Telelogic's ObjectGEODE. Adapting this idea to UML implies a definition of a UML-like syntax for it (which reuses the UML concepts of class and statecharts by adding several stereotypes), as well as the re-definition of the set of visible events (with specific UML event types like operation invocation, etc.).

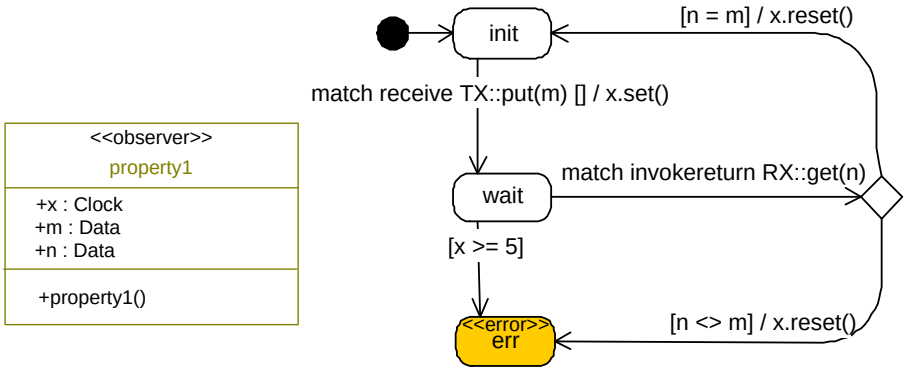

Fig. 4. Example of observer for a safety property.

The advantage of UML observers with respect to other property specification languages is that they use concepts that are known to UML designers (event driven state machines) while remaining sufficiently formal and expressive for many types of properties[3].

5.1 An example of property

Let us take a simple example: assume that we have a point-to-point communication protocol described in UML. Two interfaces TX and RX encapsulate the transmission and reception operations, and, to simplify, at runtime there exists exactly one object implementing each interface. The interface TX has one blocking operation $put(p : Data)$ (where $Data$ is the packet type) and the interface RX has one blocking operation $get()$ that returns a $Data$.

Assume that we want to express the following reliability property: *whenever put is called with some Data, within at most 5 time units the same Data is received at the other end*. This also supposes that the user at the other end has called *get* within this time frame, reception being signified by the return from *get*. This property is specified in the observer in figure 4.

5.2 Basic ingredients: the events

An important ingredient of the observer in figure 4 are the event specifications on some transitions. Here, the notion of event and the event types are the ones introduced in [18][4]:

[3] In [27], a similar property language is shown to be equivalent to the linear temporal logic LTL.

[4] Some of these event types may be irrelevant in certain types of UML models, depending on the computation/communication model considered. This list may be subject to further variations.

- 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 [16, 18] and in this work): **set**, **reset**, **occur**, **consume**.

The trigger of an observer transition may be a **match** clause, in which case the transition will be triggered by certain types of events occurring in the UML model. The clause specifies the type of event (e.g. **receive** in figure 4), some related information (e.g. the operation name $TX :: put$) and observer variables that may receive related information (e.g. m which receives the value of the $Data$ parameter of put in the concerned call).

Besides events, an observer may access any part of the state of the UML model: object attributes and state, signal queues.

5.3 Writing and verifying safety and liveness properties

To support the writing of properties, the states of an observer may be classified as **error**, **success** or **ordinary** states. The interpretation of these states depends on the kind of property expressed by the observer, and corresponds to different verification modes:

- *safety* verification, in which the observer state machine is considered a finite automaton, the accepting states of which are the **error** and **success** states. The verification tool looks for particular executions which lead to these states.
- *liveness* verification, in which the observer state machine is considered a Büchi automaton whose accepting states are the **success** states. The verification tool looks for *infinite* executions which are not accepted by this Büchi automaton (i.e. they do not pass infinitely often through a **success** state), and which correspond to lack of progress ad infinitum.

The verification problem is reduced in the *safety* case to a reachability problem in the product automaton. In the *liveness* case, verification corresponds to a search for non-progress cycles in the reachability graph, a classical verification problem that requires a nested dept-first search of the graph [10].

The tool-set described in the next section implements these techniques and is capable of generating fault scenarios leading to violation of a property written as observer.

Writing timing properties. Certain timing properties may be expressed directly in a UML model using the extensions presented in section 4. However, more complicated properties which involve several events and more arbitrary

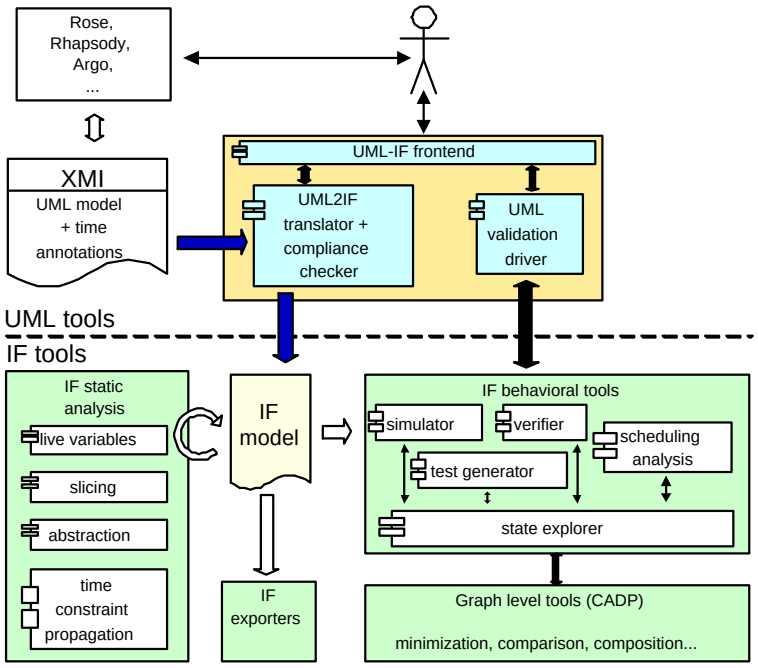

Fig. 5. Architecture of the UML-IF validation toolbox.

ordering between them should be written using observers. In order to express quantitative timing properties, observers may use the concepts available in our extension of UML, such as *clocks* and *Time* and *Duration* values.

6 The simulation and verification toolset

The principles presented in the previous sections are being implemented in the UML-IF validation toolbox, the architecture of which is shown in figure 5. With this tool, a designer may simulate and verify UML models and observers developed in third-party editors[5] and stored in XMI[6] format. The functionality offered by the tool, is that of an advanced debugger (with step-back, scenario generation, etc.) doubled by a model checker for properties expressed as observers.

In a first phase, the tool generates an IF specification and a set of IF observers corresponding to the model. In a second phase, it drives the IF simulation and verification tools so that the validation results fed back to the user may be marshaled back to level of the original model. Ultimately, the IF back-end tools shall be invisible to the UML designer.

[5] Rational Rose, I-Logix Rhapsody and Argo UML have been tested for the moment.
[6] XMI 1.0 or 1.1 for UML 1.4

As mentioned in the introduction, by using the IF tools as underlying engine, the UML tools have access to several model reduction and analysis techniques already implemented. Such techniques aim at improving the scalability of the tools, essential in a UML context. Among them, it is worth mentioning *static analysis* and optimizations for state-space reduction, *partial order* reductions, some forms of *symbolic* exploration, model minimization and comparison [6, 8].

A first version of this toolset exists and is currently being used on several case studies in the context of the OMEGA project.

6.1 The broader picture

The work presented here is part of a broader effort within the OMEGA project [1], which aims to produce a formal methods-based toolset and a methodology dedicated particularly to the development of real-time and embedded systems. The OMEGA framework will support activities like:

- static well formedness checks of UML models
- timed model checking of UML models against observers (presented here) as well as scheduler synthesis based on a timed automaton model
- untimed model cheching of UML models, including against LSC specifications (a variant of interactions with stronger structuring constructs [11])
- model 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 the OMEGA website [1].

7 Conclusions and plans for future work

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

Although this problem has been previously studied [14, 24, 23, 22, 21, 31], our approach introduces a new dimension by considering the important object-oriented features present in UML: inheritance, polymorphism and dynamic binding of operations, and their interplay with statecharts. We give a solution for modeling these concepts with automata: operations are modeled by dynamically created automata, and thus call stacks are implicitly represented by chains of communicating automata. Dynamic binding is achieved through the use of signals for operation invocation. We also give a solution for modeling run-to-completion and a chosen concurrency semantics using dynamic priorities.

Our experiments on small case studies show that the simulation and model checking overhead introduced by modeling these object-oriented aspects remains low, thus not hampering the scalability of the approach.

For writing and verifying dynamic properties, we propose a formalism that remains within the framework of UML: observer objects. We believe this is an

important issue 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).

The plans for future work include the following main directions:

- assessment of the applicability of our technique to larger models: the tool is beginning to be applied to a set of four case studies provided by industrial partners in the OMEGA project.
- extension of the language scope covered by the tool: we plan to integrate the component and architecture specifications.
- improvement of the ergonomics and integration of the toolset (e.g. the presentation of validation results in terms of the UML model).

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, 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.

[7] 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.

[8] 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.

[9] Marius Bozga and Yassine Lakhnech. If-2.0 common language operational semantics. Technical report, 2002. Deliverable of the IST Advance project, available with the authors.

[10] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] 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.

[17] Susanne Graf and Guoping Jia. Verification experiments on the MASCARA protocol. In *Proceedings of SPIN Workshop '01 (Toronto, Canada)*, January 2001.

[18] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations with UML. Submitted to SVERTS'2003, 2003.

[19] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Iulian Ober. *Specification and Validation of Timed Systems with Formal Description Languages*. PhD thesis, Institut National Polytechnique de Toulouse, September 2001.

[28] Iulian Ober and Marius Bozga. Translation from omega uml to if. Technical report, VERIMAG, 2003.

[29] Iulian Ober and Ileana Stan. On the concurrent object model of UML. In *Proceedings of EUROPAR'99*, LNCS. Springer Verlag, 1999.

[30] OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG ducument ad/2002-03-04, March 2002.

[31] 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.

[32] WOODDES. Workshop on concurrency issues in UML. Satelite workshop of UML'2001. See http://wooddes.intranet.gr/uml2001/Home.htm.

[33] 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.