Validation of UML models via a mapping to communicating extended timed automata[⋆]

Iulian Ober, Susanne Graf, Ileana Ober

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

Abstract. We present a technique and a tool for model-checking operational UML models based on a mapping of object oriented UML models into a framework of communicating extended timed automata - in the IF format - and the use of the existing model-checking and simulation tools for this format.
We take into account most of the structural and behavioral characteristics of classes and their interplay and tackle issues like the combination of operations, state machines, inheritance and polymorphism, with a particular semantic profile for communication and concurrency. The UML dialect considered here, also includes a set of extensions for expressing timing.
Our approach is implemented by a tool importing UML models via an XMI repository, and thus supporting several commercial and non-commercial UML editors. For user friendly interactive simulation, an interface has been built, presenting feedback to the user in terms of the original UML model. Model-checking and model exploration can be done by reusing the existing IF state-of-the-art validation environment.

1 Introduction

We present in this paper a technique and a tool for validating UML models by simulation and property verification. The reason why we focus on UML is that we feel some of the techniques which 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 (e.g. with the SDL language [8]) 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 supported by the OMEGA European Project (IST-33522). See also http://www-omega.imag.fr

This work is part of the OMEGA IST project, whose aim is building a basis for a UML based development environment for real-time and embedded systems, including a set of notations for different aspects with common semantic foundations, tool supported verification methods for large systems, including real-time related aspects [11].

1.1 Basic assumptions

Before going into more detail, in this work we made the following *fundamental assumptions*:

UML is broader than what we need or can handle in automatic validation. In UML 1.4 [33] there are 9 types of diagrams and about 150 language concepts (metaclasses). Some of them are too informal to be useful in validation (e.g. use cases) while for others the coherence and relationships with the rest of the UML model are not clearly (or uniquely) defined (e.g. collaborations, activity diagrams, deployment).

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. The choices, which are not fully explained in this paper, are not made ad-hoc. This work is part of a broader project (IST-OMEGA [1]) which aims to define a consistent subset of UML (*kernel language*) to be used in safety critical, real-time applications. See also [12, 11].

UML has neither a standard nor a broadly accepted dynamic semantics. As a consequence, one facet of the OMEGA project is a quest for a suitable semantics for UML to be used in complex, safety critical, real-time, possibly distributed applications. Effort is put into: *finding* the right concepts (e.g. communication mechanisms between objects, concurrency model, timing specification features, see [12]), *defining* them formally (a formalization in PVS is available [23]) and *implementing and testing* these concepts in tools.

In this paper *we discuss only the problems of implementing and testing the semantics*, while the definition and formalization are tackled in [12] and [23]. We describe a translation to an automata-based formalism implemented in the IF tool [6, 9]. This results in a flexible implementation of the semantics, in which we can easily test the choices of the OMEGA formal semantics and propose changes.

To produce powerful tools we have to build upon the existing. This motivates our choice to do a translation to the IF language [6, 9], for which a rich set of tools (for static analysis, model checking with various reduction techniques, model construction and manipulation, test generation, etc.) already exist.

Our claim is that most of this tools work on UML-generated models with only minor updates[1]

[1] At least model checking, model construction and manipulation were already tested.

Moreover, in order to be usable a validation tool has to accept UML models edited with widely used CASE tools. Our choice to work on the standard XML representation for UML (XMI) is a step into this direction.

1.2 Our approach in more detail

In terms of **language coverage**, in our semantics and in our tool 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 we tackle, like the combination of *operations* and *state machines*, *inheritance* and *polymorphism*, *run-to-completion* and *concurrency*, go beyond the previous work done in this area (see section 1.3), which has mainly focused on verification of statecharts. Our choices are outlined in section 2.

Our implementation of the operational semantics of UML models is based on a mapping from UML into an intermediate formal representation IF[5] based on *communicating extended timed automata* (CETA). This choice is motivated by the existence a verification toolset based on this semantic model [6, 9] which has been productively used in a number of research projects and case studies, e.g. in [7, 17]. The main features of the IF language are presented in section 1.4, and in section 3 we discuss a mapping from UML into this model which respects the semantics given in [12, 23].

An important issue in designing real-time systems is the ability to capture quantitative timing requirements and assumptions, as well as time dependent behavior. We rely on the **timing extensions** defined in the context of the Omega project [18, 16]. We summarize these extensions and their mapping into IF in section 4.

Another important issue is the formalism in which **properties** of models are expressed. In section 5 we introduce a simple property description language (*observer objects*) that reuses some concepts from UML (like objects, state machines) while remaining sufficiently expressive for a large class of linear properties. The use of concepts that are familiar to most UML users has the potential to alleviate the cultural shock of introducing formal dynamic verification to UML models.

Finally, section 6 presents the *UML validation toolset*. By using the IF tools as underlying simulation and verification engine, the UML tools presented here benefit from a large spectrum of model reduction and analysis techniques already implemented therein, such as *static analysis* and optimizations for state-space reduction, *partial order* reductions, some forms of *symbolic* exploration, model minimization and comparison, etc [6, 9].

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

1.3 Related work

The application of formal analysis techniques (and particularly model checking) to UML has been a very active field of study in recent years, as witnessed by the number of papers on this subject ([29, 30, 28, 27, 26, 35, 14, 15, 37, 3] are most oftenly cited).

Like ourselves, most of these authors base their work on an existing model checker (SPIN[22] in the case of [29, 30, 28, 35], COSPAN[21] in the case of [37], Kronos[38] for [3] and UPPAAL[25] for [26]), 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. [28, 29, 30]). Others use UML collaboration diagrams (e.g. [26, 35]) which are too weak to express all relevant properties. We propose to use a variant of UML state machines to express properties in terms of observers.

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 not dealt with. Our approach is, to our knowledge, the first to try to fill this gap.

Our starting point for handling of UML state machines (not described in detail in this paper) was the material cited above together with previous work on Statecharts ([20, 13, 31] 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 [32], and from other positions on this topic (see for example [36]) and we respected the operational semantics defined in the OMEGA project [12].

1.4 The back-end model and 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 [6, 9, 10]. We summarize the elements of this model in the following.

Modeling with communicating extended automata IF was 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*. 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*.

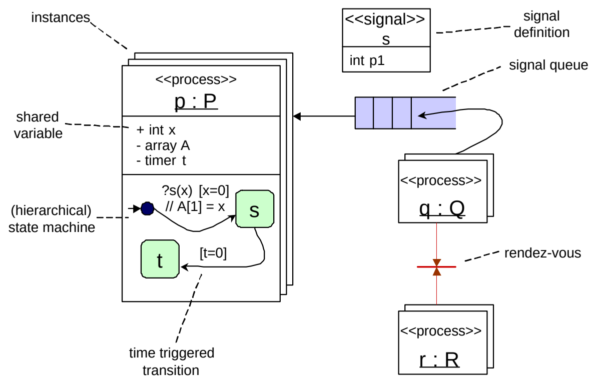

Fig. 1. Constituents of a communicating extended automata model in IF.

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

Processes may inter-communicate via *asynchronous signals*, via *shared variables* or via *rendez-vous*. Parallel processes are composed asynchronously (i.e. by interleaving). The model also allows *dynamic creation* of processes, which is an essential feature for modeling object systems that are by definition dynamic.

The link between system execution and time progress may be described in a precise manner, and thus offers support for modeling real time constraints. We use the concepts from *timed automata with urgency* [4]: there are special variables called *clocks* which measure time progress and which can be used in transition guards. A special attribute of each transition, called *urgency*, specifies if time may progress when the transition is enabled, and by how much (up to infinity or only as long as the time-guard of the transition remains true).

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 system description is associated with a set of priority directives of the form: $(\textit{state condition}) \Rightarrow p_1 \prec p_2$. 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 specification with observers Dynamic properties of IF models may be expressed using *observer automata*. These are special processes that may monitor[2] the changes in the *state* of a model (variable values, contents

[2] The semantics is that observer transitions synchronize with the transitions of the system.

of queues, etc.) and the *events* occurring in it (inputs, outputs, creation and destruction of processes, etc.).

For expressing properties, the states of an observer may be classified (syntactically) as *ordinary* or *error*. 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 rooted in the observer concept introduced by Jard, Groz and Monin in the VEDA tool [24]. 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.

Analysis techniques and the IF-2 toolbox The IF-2 toolbox [6, 9] is the validation environment built around the formalism presented before. It is composed of three categories of tools:

1. **behavioral tools** for simulation, verification of properties, automatic test generation, model manipulation (minimization, comparison). The tools implement techniques such as *partial order reductions* and *symbolic* simulation of time, 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), *slicing* and simple forms of *abstraction.*
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.).

The toolbox has already been used in a series of industrial-size case studies [6, 9].

2 Ingredients of UML models

This section outlines the semantic– and design–related choices with respect to *the UML concepts covered* and *the computation* and to the *execution model* adopted.

2.1 UML concepts covered

In this work we consider an operational subset of UML, which includes the following UML concepts: active and passive *classes* - with their *operations* and *attributes*, *associations*, *generalizations* - including polymorphism and dynamic binding of operations, *basic data types*, *signals*, and *state machines.* State machines are not discussed in this paper as they are already tackled in many previous works like [29, 30, 28, 27, 26, 35, 15, 37, 3].

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 semantic 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, which may be found in [12, 23].

The execution model chosen in OMEGA and presented here is an extension of the execution model of the Rhapsody UML tool (see [19] for an overview), which is already 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*, 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.

Different activity groups execute concurrently, and objects inside the same an activity group execute sequentially. Groups are sequential on purpose, in order to have some default protection against concurrent access to shared data (passive objects) in the group. 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*.

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 the 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.

Operations, signals and state machines. In the UML model we distinguish syntactically between two kinds of operations: *triggered* operations and *primitive* operations. Reaction to *triggered operation calls* 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 have the body described by a method, with an associated action. Their handling is more delicate since they are dynamically bound like in all object-oriented models. This means that, when such an operation call is sent to an object, the most appropriate operation implementation with respect to the actual type of the called object and to the inheritance hierarchy has to be executed.

With respect to call initiation, an object having the control may call a primitive operation on an object from the same activity group at any time, and the call is stacked and handled immediately. However, in case of triggered operation calls, the dynamic call graph between objects should be acyclic, since an object that has already called a triggered operation is necessarily in an unstable state and may not handle any more calls. This type of condition may be verified using the IF mapping.

Signals sent inside an activity group are always put in the group queue for handling in a later run-to-completion step. This choice is made so that there is no intra-group concurrency created by sending signals.

We note that the model described here corresponds to that of concurrent, internally-sequential *components* (activity groups), 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 by several synchronous languages.

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 idea is to obtain a system that has the same operational semantics as the initial UML model (i.e. the same labeled transition system up to bisimulation). The intermediate layer of IF helps us tackle with 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 a way that all runtime UML entities (objects, call stacks, pending messages, etc.) are identifiable as a part of the IF model's 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 will have a local variable of corresponding type for 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 process of a type called GM. This process sequentializes the calls coming from outside the activity group, and helps to ensure the run-to-completion policy. In each P_X there is 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. 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 an automaton. The parameters have the same meaning as in the $call_{X::m}$ signal. The *callee* PID is used to access local attributes of the called object, via the shared variable mechanism of IF.
- 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 IF automaton.

The action of invoking an operation $X :: m$ is mapped to the sending of 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 will queue 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 (in every state[3] 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 (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 for extensions to various types of calls other than the ones currently supported in the OMEGA semantics (e.g. 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 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 3).

[3] This is eased by the fact that IF supports hierarchical automata.

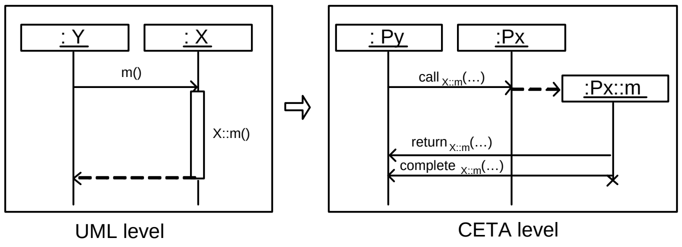

Fig. 2. Handling primitive operation calls using dynamic creation.

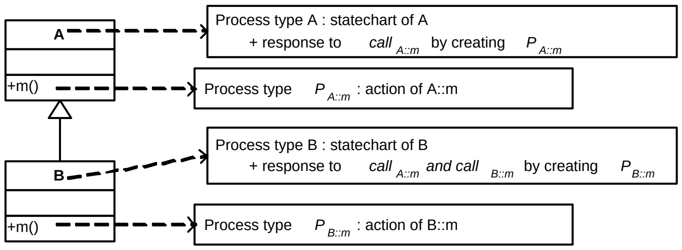
A
+m()
B
+m()
Process type A : statechart of A
+ response to call A::m by creating P A::m
Process type P A::m : action of A::m
Process type B : statechart of B
+ response to call A::m and call B::m by creating P B::m
Process type P B::m : action of B::m

Fig. 3. Mapping of primitive operations and inheritance.

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 (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 state machines. UML state machines are mapped almost syntactically in IF. Certain transformations, not detailed here, are necessary in order to support features that are not directly in IF: entry/exit actions, fork/join nodes, history, etc. Several prior research papers tackle the problem of mapping statecharts to (hierarchical) automata (e.g. [31]). The method we apply is similar to such approaches.

Actions. The action types supported in the original UML model are *assignments*, *signal output*, *control structure actions*, *object creation*, *method call* and *return*.

Some are directly mapped to their IF counterparts, while the others are mapped as mentioned above to special signal emissions (*call*, *return*) or process creations.

3.2 Modeling run-to-completion using dynamic priorities

We discuss here how the concurrency model introduced in section 2.2 is realized using the dynamic partial priority order mechanism presented in 1.4.

As 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 objects (GM) handle this simple queuing 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 enforces the following property:
  As long as an object inside the group may move, 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:
  $\forall x, y. (x = y.leader.running) \wedge (x \neq y) \Rightarrow x \prec y$
  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 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.
  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

In order to build a faithful model of a *real-time* system in UML, one needs to represent two types of timing information:

Time-triggered behavior (*prescriptive modeling*): this corresponds, for example, to the 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*): information taken as a *assumption* (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 (*assertions*) 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 [34], provides a common set of concepts for modeling timing, but their definition remains mostly syntactic.

We base our work on the framework defined in [18] for modeling timed systems. The framework reuses some of the concepts of the standard real-time profile [34] (e.g. timers, certain data types), and additionally allows expressing *duration constraints* between various events occurring in the system.

4.1 Validation of timed specifications

In this section we present the main concepts taken from [18], that we use in our framework, and we give the principles of their mapping to IF.

For modeling *time-triggered behavior*, we are using *timer* and *clock* objects compatible with those of [34], which are mapped in a straightforward manner to IF.

The modeling of the *descriptive timing information* makes intensive use of the **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). All these UML *events* have a corresponding event in IF. For example: the UML event of invoking an operation $X :: m$ corresponds to the event of sending the $call_{X::m}$ signal, etc.

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.

Between the events identified as above, we may define **duration constraints**. The constraints may be either *assumptions* (hypotheses to be enforced upon the system runs) or *assertions* (properties to be tested on system runs).

The class diagram example in figure 4 shows how these events and duration constraints 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 testing or enforcing a timing constraint from the UML model, we are presented with two alternatives:

– if the constraint is *local* to an object, i.e. all involved events are directly observed by the object, the constraint may be tested or enforced by the IF

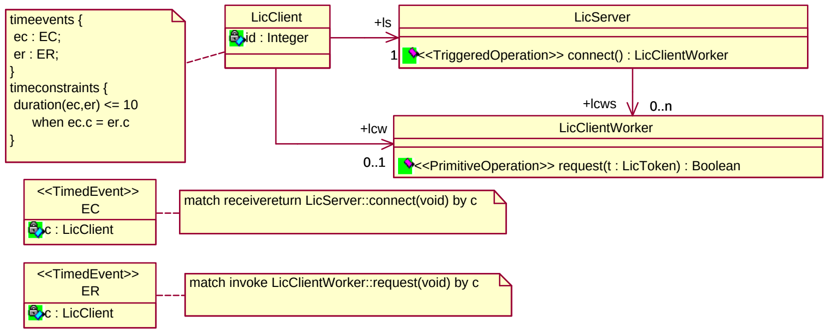

Fig. 4. Using events to describe timing constraints.

process implementing the object[4]. It will use an additional clock for measuring the duration concerned by the constraint, and a transition to an error state (in case of an assertion) or to an invalid state (in case of an assumption) with an appropriate guard on that clock.

– if the constraint is *not local* to an object (we call it *global*), the constraint will be tested or enforced by an 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*. Similarly to IF observer automata (section 1.4), UML observers are special objects which run in parallel with a UML system and monitor its *state* and the *events* that occur.

Syntactically, observers are described by special UML classes stereotyped with ≪*observer*≫. 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. The main issue in defining UML observers is the choice of visible event types (which include specific UML event types like operation invocation, etc.).

For UML users, the advantage of UML observers compared 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.

[4] This is the case in figure 4. In general, outputs and inputs of a process are directly observed by itself, but they are not visible to other processes.

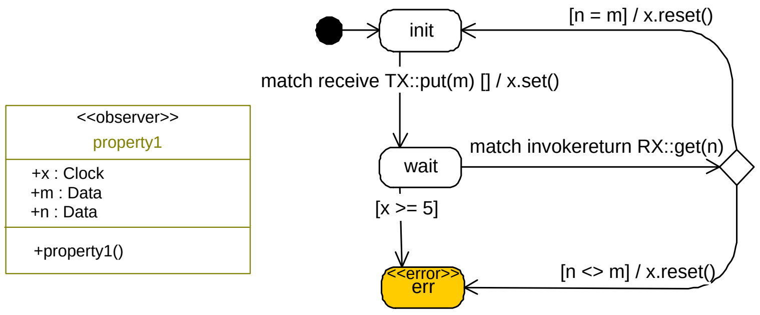

Fig. 5. Example of observer for a safety property.

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 5.

5.2 Basic observer ingredients

An important ingredient of the observer in figure 5 are the event specifications on some transitions. Here, the notion of event and the event types are the ones introduced in [18]:

– 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 5), 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.

As in IF observers, properties are expressed by classifying observer states as *error* or *ordinary.* Note that an observer may be used also to formalize a hypothesis on system executions, in which case the observer *error* states mark the system states that should be considered *invalid* with respect to the assumptions.

Expressing 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 ordering between them may 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.*

6 The simulation and verification toolset

The principles presented in the previous sections are being implemented in the UML-IF validation toolbox[5], the architecture of which is shown in figure 6. With this tool, a designer may simulate and verify UML models and observers developed in third-party editors[6] and stored in XMI[7] 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 will be invisible to the UML designer.

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, 9].

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

[5] See http://www-verimag.imag.fr/PEOPLE/ober/IFx.
[6] Rational Rose, I-Logix Rhapsody and Argo UML have been tested for the moment.
[7] XMI 1.0 or 1.1 for UML 1.4

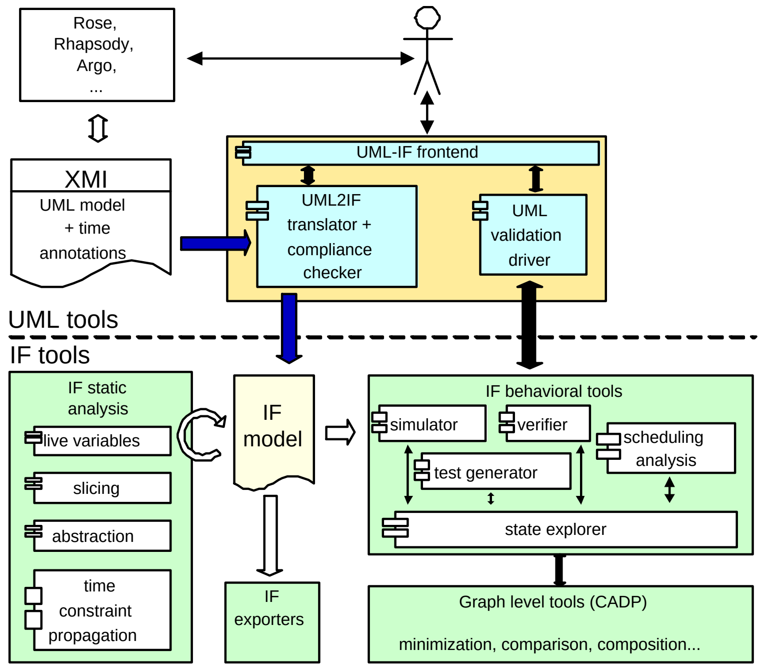

Fig. 6. Architecture of the UML-IF validation toolbox.

7 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 [14, 29, 28, 27, 26, 35], 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).

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 in the OMEGA project.
- Extend the language scope covered by the tool. We plan to integrate the component and architecture specification framework defined in OMEGA.
- Improve the ergonomics and integration of the toolset (e.g. the presentation of validation results in terms of the UML model).
- Study the possibility of using the additional structure available in the object-oriented UML models for improving verification, static analysis, etc.

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] Vieri Del Bianco, Luigi Lavazza, and Marco Mauri. Model checking UML specifications of real time software. In *Proceedings of 8th International Conference on Engineering of Complex Computer Systems*. IEEE, 2002.
[4] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.
[5] M. Bozga, J.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.
[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, 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.
[9] 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.

[10] 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.

[11] The Omega consortium (writers: Susanne Graf and Jozef Hooman). The Omega vision and workplan. Technical report, Omega Project deliverable, 2003.

[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. In *Proceedings of SVERTS'2003 (Satellite workshop of UML'2003). Available at http://www-verimag.imag.fr/EVENTS/2003/SVERTS*, San-Francisco, 2003.

[19] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

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

[21] Z. Har'El and R. P. Kurshan. Software for Analysis of Coordination. In *Conference on System Science Engineering*. Pergamon Press, 1988.

[22] G. J. Holzmann. The model-checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 1999.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] OMG. Unified Modeling Language Specification (Action Semantics). OMG Adopted Specification, December 2001.

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

[35] 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.

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

[37] 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.

[38] S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.