

# Validation and Test Generation for Object-Oriented Distributed Software

Thierry Jéron, Jean-Marc Jézéquel, Alain Le Guennec  
Irisa/CNRS  
Campus de Beaulieu  
F-35042 Rennes Cedex, FRANCE  
jeron/jezequel/aleguenn@irisa.fr

## Abstract

*The development of correct OO distributed software is a daunting task as soon as the distributed interactions are not trivial. This is due to the inherent complexity of distributed systems (latency, error recovery, etc.), leading to numerous problems such as deadlocks, race conditions, and many difficulties in trying to detect and reproduce such error conditions and debug them. The OO technology is ill-equipped to deal with this dimension of the problem. On the other hand, the willingness of mastering this complexity in the context of telecommunication protocols gave birth to specific formal verification and validation tools. The aim of this paper is to explore how the underlying technology of these tools could be made available to the designer of OO distributed software. We propose a framework allowing the integration of formal verification and validation technology in a seamless OO life-cycle based on UML, the Unified Modeling Language. From a UML model, this framework would allow to conduct model checking activities as well as random simulation and automatic test generation.*

## 1. Introduction

It is now widely admitted [14] that only system development based on “real-world” modeling is able to deal with the complexity and the versatility of large software systems. Once the idea of analyzing a system through modeling has been accepted, there is little surprise that the object-oriented (OO) approach is brought in, because its roots lie in Simula-67, a language for simulation designed in the late 1960s, and simulation basically relies on modeling. This is the underlying rationale of the numerous object-oriented analysis and design (OOAD) methods that have been documented in the literature [22]. OOAD methods allow the same conceptual framework (based on objects) to be used during the whole software life cycle. This seamlessness should yield considerable benefits in terms of flexibility and traceability.

These properties would translate to better quality software systems (fewer defects and delays) that are much easier to maintain because a requirement shift usually may be traced easily down to the (object-oriented) code.

But today many such large software systems have acquired a distributed nature. This distributed nature may be either a constraint from the problem statement, or may be introduced as the consequence of a design decision to handle performance problems and/or fault tolerance. Frameworks such as CORBA help in deploying distributed solutions, but any experienced software engineer recognizes that the design, implementation and maintenance of correct distributed software is still a very difficult exercise. Distributed systems have indeed an *inherent* complexity resulting from fundamental challenges such as latency of asynchronous communications, error recovery, service partitioning and load balancing. Furthermore, being intrinsically concurrent, distributed software faces race conditions, deadlock, starvation problems, etc. This complexity is quite orthogonal to the programming-in-the-large problems addressed by OO technology, including CORBA. There are currently no approaches to deal with this aspect of the problem in an OO context (see [20, 23] for a good overview on current approaches at V&V for OO systems).

The nature of the complexity of distributed systems has been widely explored in many academic (and other) circles for several years. In the context of telecommunication protocols, the willingness of mastering this complexity gave birth to the development of standardized formal description techniques (FDT) and to a set of associated formal verification and validation tools. Unfortunately, for several reasons that we explore later in this paper, these tools usually cannot be easily used in an integrated OO life-cycle.

The aim of this paper is to explore a way by which the underlying technology of these formal verification and validation tools could be made available to the designer of OO distributed software. We start in section 2 by recalling what the principles of formal verification and validation tools are, and how they address the inherent complexity of distributed

systems. We then try to analyze why they are still seldom used. Section 3 presents the most important aspects of the Unified Modeling Language (UML) and explain how it relates to formal description techniques. In section 4, building on this analysis, we outline a tentative OO framework based on the UML making possible the use of formal verification and validation technology. We then show in section 5 how the various formal verification and validation activities can be conducted within our framework. Finally, we conclude on the applicability of our approach for real size cases, and on the perspectives of the integration of formal verification and validation technology in the OO life-cycle.

## 2. Validation and Verification of Distributed Software

### 2.1. A set of complementary formal techniques

Validation techniques vary widely in their forms and their abilities, but they always need a formal description of the distributed software system. They output data on properties of the system under consideration that can be viewed with some confidence level. Basically, the designer may attack his/her software by three complementary techniques. We list here their advantages and major drawbacks:

- *formal verification of properties*: it gives a definite answer about validity by formally checking that all possible executions of the specification of the distributed software respect some properties (e.g. no deadlock). But existing methods, such as *model-checking*, which often imply the construction of the graph of all the states the distributed system could reach, can only be easily applied to the analysis of very simplified models of the considered problem [11]. Otherwise there is a combinatory explosion of the number of states that forbids such a brute force verification. This forces the distributed software to be described at a high abstraction level, so its formal verification lets the problem of property preservation during its refinement course widely open. An alternative is to use on-the-fly model-checking [5] or local model-checking [25] which may avoid the construction of the complete graph.
- *intensive simulation*, using a simulated (and centralized) environment: it can deal with more refined models of the problem and can efficiently detect errors (even tricky or unexpected ones) on a reasonable subset of the possible system behaviors. Formally, it consists in randomly walking the reachability graph of the distributed software. The main difficulty is to formally describe and simulate the execution environment. This is generally simplified, because it would not be realis-

tic (nor interesting) to take into account all the parameters of a real system, such as the influence of message size on transmission delays, or the exact operation durations (which are not computable without execution).

- *observation and test* of an implementation: here, the execution environment is a real one. But since there is a lack of tools to observe a distributed system as a whole, it will be difficult to actually validate the software. Anyway, producing the test cases for the distributed system is a costly task that can be alleviated only if one is able to automatically generate the tests from a formal specification of the system and a set of test purposes.

It appears that these approaches are more complementary than in competition, and that an advised project manager would try to use them all. However this is hard in practice because the formalisms used in these various stages differ widely. Most of these techniques have been developed in the context of the Formal Description Techniques (FDTs) for protocols, where they have been successfully applied to various toy and real problems.

### 2.2. An example of test generation tools: TGV

TGV is a prototype tool developed in collaboration by the Pampa project of IRISA/INRIA Rennes and Verimag Grenoble [7, 8, 17]. Its aim is to automatically generate test cases for conformance testing of reactive systems, starting from a formal specification of the system and test purposes allowing to select test cases. These test cases are composed of interaction sequences. An interaction is either an output of the tester which is proposed to the implementation, or an input which is an expected answer of the implementation according to its specification. Test cases also contain timers which insure the finiteness of the test execution and verdicts that are produced according to the conformance or non-conformance of the implementation with respect to the specification.

The principle of TGV is to compute a test case from a specification of the system and a test purpose. A test purpose characterizes an abstract property that the system should have and that one wants to test. It is formalized by a finite automaton labelled with some interactions of the specification and it is used to select a test case from all possible behaviours of the specification. The specification must be given in a language whose operational semantics allows the set of possible behaviors of the specification to be represented by a transition system. This transition system is either explicit or implicit. If it is explicit, it is previously computed by a simulation tool which takes as an entry the specification and computes its possible behaviors. The generated graph is then translated into a format accepted by

TGV. Since testing only considers traces of observable interactions, internal actions are discarded and the graph is determinized. The resulting graph represents the observable behaviours of the specification on which the main algorithm of TGV can be applied.

**On-the-fly generation** allows TGV to handle very large and even infinite state graphs. The principle (see Figure 1) is to compute a test case while constructing, in a lazy strategy, a part of the graph which is necessary for the test case computation. In order to be applicable, TGV must be provided with some basic functions for the graph construction, to respectively compute the initial global state, the fireable transitions, the new global state reached from a previous global state by firing a transition, and functions to compare global states and store them in memory.

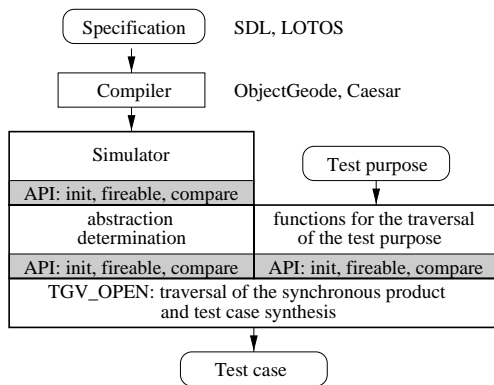


Figure 1. Structure of TGV

**Taking into account the test architecture** is an important capability of TGV. Testers often face the problem that they do not directly communicate with the implementation under test because it might be embedded into a complex system. This is the case for usual testing architectures in the context of communication protocols. An implementation is often accessible only through a dedicated service, as in the so-called “remote test method” where the tester communicates asynchronously with the implementation.

The consequence is a loss of control and observation on the implementation which should be taken into account for correct test generation. Our approach is to include a model of the architecture in the specification by adding one or several processes and to apply TGV on this modified specification. This produces tests for conformance of the implementation in its environment against the specification in a model of the environment. The drawback could be twofold: the state explosion problem, and the fact that tests would concern the complete modified specification. But this is partially reduced in TGV by on-the-fly test generation and by the fact that TGV allows the user to select tests using test purposes focussed on the sole specification under test.

## 2.3. Difficulties in using FDTs

It is very disappointing to see that formal validation based on standard FDTs (such as SDL [4], Estelle [13] and Lotos [12]) never acceded to a widespread use in the industry, despite excellent results on most of the pilot projects where it has been used [18]. While the interest of formal techniques is widely acknowledged (at least in the context of mission-critical distributed software), their use is still deferred for various reasons:

- their learning curve is steep, because they rely on non-trivial formalisms and unusual syntaxes and semantics,
- they require the analysis to be much more accurate in the early stages (which is not necessarily a bad thing, but it is a matter of facts that few projects are prepared to pay the additional cost early),
- and there is a lack of integration of this promising technology in widely used software development methods and life-cycles.

In our experience, this last point is probably the most important one. Because standard FDTs lack basic support for modern software engineering principles, it is extremely clumsy to try to use them as implementation languages for real, large scale distributed applications. Furthermore, being fully formal implies that FDTs are based on a close world assumption, making them awkward to deal with the open nature of many distributed softwares: specifiers become prisoners of the FDTs underlying semantics choices. For example, all FDTs force a given communication semantics (multi rendez-vous for Lotos, FIFO for Estelle) upon the user, who has to painfully reconstruct the set of communication semantics needed for a given distributed system starting from the FDTs one, sometimes with a high performance cost (Estelle FIFO between protocol layers are difficult to circumvent for instance).

Using FDTs validation technology thus imposes a model rupture in the usual life-cycle: the formal model for the validation has to be built and maintained separately from the analysis and design model (expressed in e.g., OMT or UML). For example, this implies that formal validation technology may be used during the maintenance phase of a system only after a costly reverse engineering effort. Each time you make a modification in your distributed software, you have to propagate it to the separate model described with your formal description technique, and start all over again your formal validation, which is quite impracticable in the real world. Since the maintenance phase cost for large, long-live systems can represent up to 3 or 4 times its initial development cost, this is not a good point for FDTs. As a consequence, formal validation rarely passes the stage of an annex (and more or less toy) task which gets low priority and low budget.

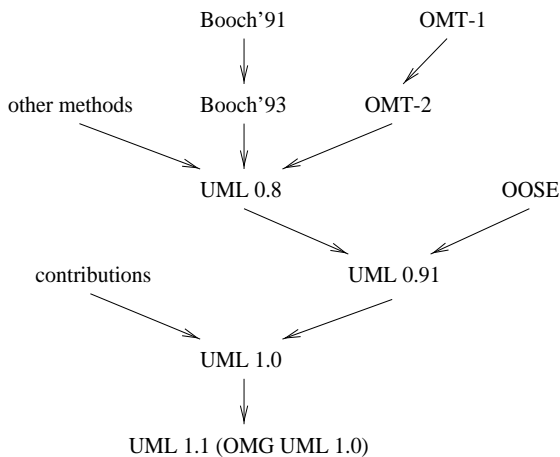
### 3. The Unified Modeling Language

#### 3.1. UML: a step towards formal OO notation

On the one hand, FDTs often are not really adequate for expressing all parts of a design. On the other hand, the various notations used by OO modeling methods have a stronger expressive power, but their semantics are not so well defined, making direct application of formal verifications impossible. Moreover methods such as the Booch [3] method, OMT [24] or OOSE [15], which were the most influential during the design of UML, each have their own notation and process (the process is the way a project is conducted following a method.) Although the corresponding notations do share many concepts, the way those common concepts are represented slightly varies depending on which notation is used, which can be a real problem for communication between people trained to different notations.

UML addresses both issues: first, it is a standard notation that can be the support for effective communication of designs. It is also a formally defined OO modeling language: indeed, UML relies upon a *meta-model* [1] which is formally defined (at least partially), while still offering the flexibility needed to model real, large scale systems.

The UML is the result of the convergence of several notations used by popular object-oriented methods. The Booch method, OMT and OOSE were the most influential during the design of UML. Initially based on a merge of the Booch and OMT notations, UML was progressively enriched with ideas coming from many other contributors (see Figure 2) and is now an OMG standard.



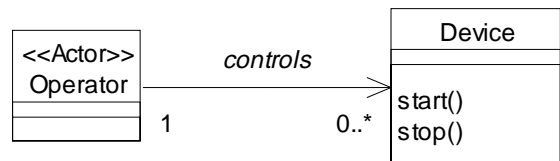
**Figure 2. History of the UML**

The UML defines several kinds of diagrams that provide a particular view of a system being modeled. The following sections explain some important aspects of the UML notation. A more complete introduction can be found in [9].

#### 3.2. Static structure diagrams

**Class diagrams** show the type of objects present in the system and the static relationships among them. The most important static relationships are:

- associations, which represent relationships between instances of classes. For example in Figure 3, an operator *controls* a device. Association ends have a cardinality (e.g. one operator controls zero or more devices), and may be decorated by an arrowhead to express navigability in a given direction (e.g. the operator knows about devices through the controls association.)
- generalizations, which represent the “is-a” relationships between classes.

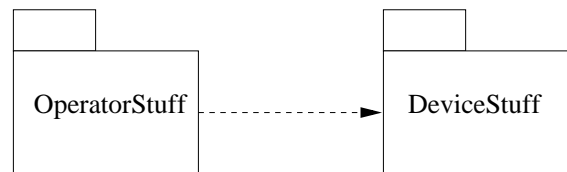


**Figure 3. UML class diagram**

Classes are represented as boxes divided into three parts:

- the name compartment
- the attribute compartment
- the operation compartment

**Package diagrams** can be used to structure the model of a big project into smaller parts, to improve the modularity of the design. Packages can contain any kind of modeling elements, even diagrams. In the context of our simple example, we may have a first package which contains the description of devices (in terms of classes, state diagrams and so on) and a second one which contains the description of operators. The fact that an operator knows about devices through its *controls* association is directly translated into a dependency between the two packages on Figure 4.



**Figure 4. UML package diagram**

### 3.3. Behavior diagrams

**Sequence diagrams** describe an interaction between a set of objects collaborating to achieve an operation. The messages exchanged during the interaction explicitly appear on the diagram in Figure 5. Objects participating in the collaboration are laid out along the horizontal axis, while the vertical axis represents time.

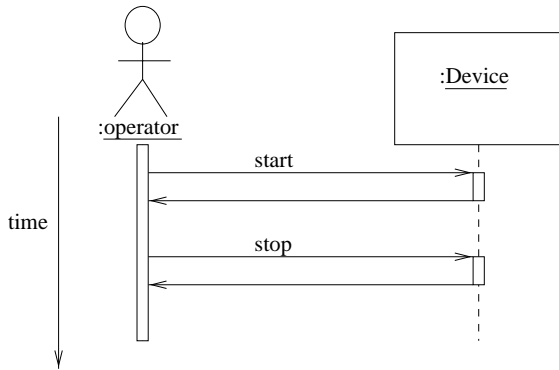


Figure 5. UML sequence diagram

**Collaboration diagrams** provide another view of object interactions. Contrary to sequence diagrams, time is not a represented by a separate axis. Instead, the collaborating objects are shown with the relationships that play a role in the collaboration (which means collaboration diagrams look like class diagrams.) Messages are placed on the relevant relationships, and are decorated with a sequence number to replace the missing time axis.

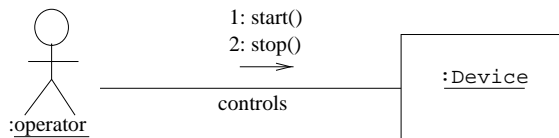


Figure 6. UML collaboration diagram

**Statechart diagrams** describe the evolution of an object over its life time. Figure 7 represents the statechart corresponding to the Device class. Figure 7 indicates that a device enters the *idle* state when it is initialized (the solid filled circle is the default entry), and then can be toggled from *idle* to *active* using the *start* and *stop* methods respectively.

### 3.4. Component and deployment diagrams

A component in UML represents a physical module representing the implementation of a part of the system. Objects contained in a component provide its realization.

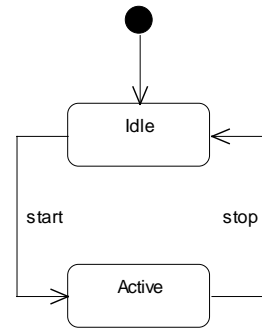


Figure 7. UML statechart diagram of the Device class

**Deployment diagrams** show how components of a system are distributed at run-time. The run-time environment is composed of a set of *nodes* connected by *links* representing the physical connection between nodes. Components are then assigned to nodes according to the configuration (deployment) chosen by the system developer.

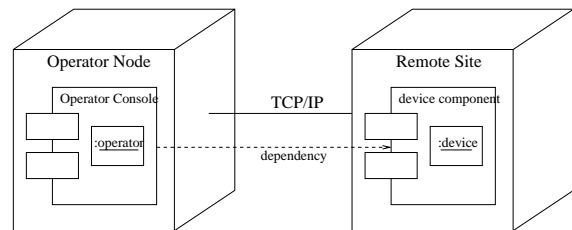


Figure 8. UML deployment diagram

## 4. Bringing validation in the OO life-cycle

### 4.1. An integrated OO life-cycle

OOAD methods along with an OO implementation allow the same conceptual framework (based on objects) to be used during the whole software life cycle.

It should be stressed that the boundaries between analysis, design and implementation are not rigid. We advocate for extending this seamless OO development process to also encompass validation, not as a post facto task (as promoted in the classical vision of the waterfall or the V-model of the life-cycle), but as an *integrated* activity *within* the OO development process, as shown in Figure 9 where dashed arrows represent feedback from validation results. The key point in implementing this idea is to rely on the sound technological basis that has been developed in the context of formal validation based on FDTs, and to make it available

to the OO designer through a dedicated framework. Our proposal is based on *UMLAUT*, a tool that can manipulate the UML representation of the system being designed.

Formal validation usually takes place on a separate simulation model of the system. This different model must be updated (and revalidated) each time the model is changed, which is both costly and error prone. Using *UMLAUT*, the properties of the system that are relevant to the validation are automatically exposed by directly processing its UML representation. This is possible because the UML notation has a well defined semantics, contrary to its predecessors. An equivalent UML model can be automatically produced that explicitly shows the protocol entities involved in asynchronous communications and the new system states that result from those communications.

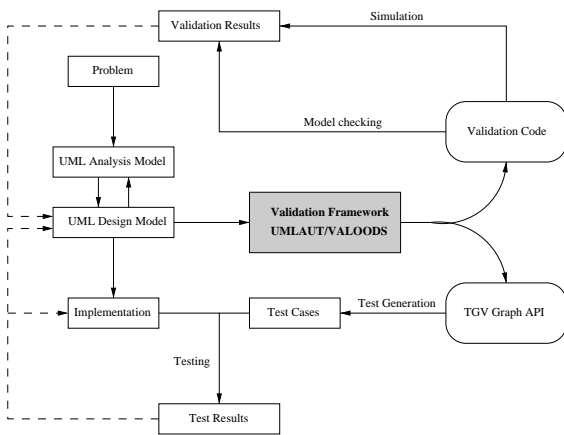


Figure 9. OO Life Cycle

Once the UML model has been appropriately modified so as to fit in our validation framework called VALOODS (VALidation of Object Oriented Distributed Software), *UMLAUT* can proceed to the validation of the UML design. The VALOODS framework offers the possibility to walk through the graph of accessible states of the distributed system (accessibility graph). Concretely, it defines an abstract interface for walking through the accessibility graph. We can then conduct model checking, intensive simulation, or test cases generation by plugging the appropriate “validation engine” in the framework.

*UMLAUT* then generates the validation code corresponding to the modified UML model. This code can be compiled to a model checking executable, or to an intensive simulation executable. It can also be interfaced with sophisticated validation toolbox such as *CADP* [6]. The information available through the reactive objects makes it possible to build a transition graph of the system suitable for *CADP*. The API that *CADP* uses to build such a graph can easily be derived from the interface to reactive objects within our

framework, and hence is provided by the generated code.

*UMLAUT* uses CDIF (CASE Data Interchange Format) as its exchange format when communicating with other parts of the development environment, which ensures interoperability and independence from CASE tool vendors. Therefore *UMLAUT* can become a part of the development environment while preserving the investment represented by the other tools already used in the project. As a side effect, *UMLAUT* can output its modifications as a CDIF file that can be imported in any CASE tool supporting this format, to see how the original UML model was transformed.

#### 4.2. The VALOODS Validation framework for OO Distributed Software

We now outline the principle of the VALOODS framework. Its purpose is to be a testbed for OO designs of distributed software. The *UMLAUT* tool described in the following section is dedicated to putting UML models into a form suitable for the application of formal validation technology within the VALOODS framework.

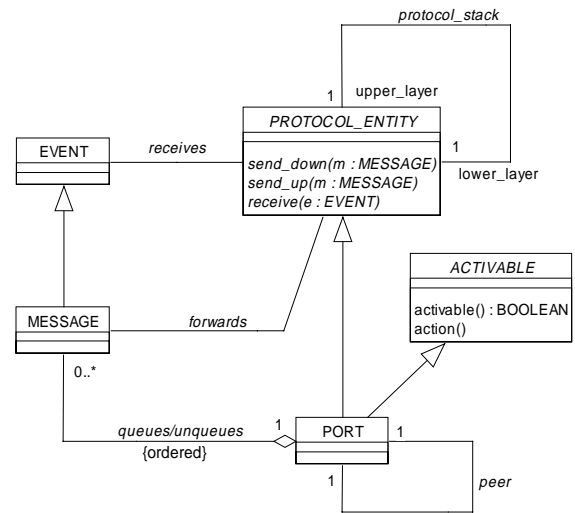


Figure 10. UML class diagram of VALOODS

The framework consists of a collection of classes (see Figure 10) together with many patterns of collaboration among instances of these classes. It provides a model of interaction among several objects that belong to classes defined by the framework. The basic abstractions in VALOODS are:

- Pro-active objects, that inherit from the class *ACTIVABLE* and must define the methods *activable* and *action*. Pro-active objects would be run in parallel, using an interleaving semantics for their actions (the method *action* being atomic).

- Protocol entities, that inherit from the class `PROTOCOL_ENTITY` and represent any object that may communicate through the network. Protocol entities must define the method `receive (e : EVENT)` to handle events, which can be either asynchronous messages, signals, or notifications of a timer expiration. Messages can be forwarded to the upper or lower layer of the protocol stack using the method `send_up` or `send_down`, respectively.
- The network interfaces (modeled through the class `PORT`), which are a special kind of `PROTOCOL_ENTITY` (hence the generalization relationship between the two classes) that plays the role of the bottom layer of a protocol stack. `PORTS` are also pro-active object: the `action` method can be called to enqueue messages received from its peer to pass them to the upper layer. By calling `action` at arbitrary moments, it becomes possible to test the effect of network latency.

The idea of VALOODS is that any class that interacts with a remote site in the distributed system must be a subclass of `PROTOCOL_ENTITY`, and will use a subclass of `PORT` for its remote communications. `PORTS` are coming in several flavors (that is, subclasses) in the VALOODS library. This is to model the various addressing schemes and quality of services (e.g. reliable, unreliable, etc.) available to the designer of a distributed software.

Once the complete OO distributed software design has been implemented in this framework, we get an accurate formal representation of the behavior of the distributed software as a whole. Furthermore we get the reversibility for free: if the design needs to be changed, it is easy to validate it again in the VALOODS framework. We no longer have to separately maintain a model of the distributed application for formal validation purposes and the application itself.

### 4.3. Making a UML model fit into the framework: the UMLAUT tool

Now let us see how the companion tool of VALOODS, called UMLAUT, transforms the original UML model into a new one suitable for validation, where pro-active objects, protocol entities, and network interfaces appear explicitly.

The starting point of the transformation is to determine which entities may interact with another one on a remote site. This information is provided by the deployment diagram. The deployment diagram of the UML model indeed shows the physical location of each component in the delivered distributed system and the relationships among them.

Based on this information, the transformations are carried out for both the static and dynamic views of the original UML model.

#### 4.3.1 Static model transformations

The first step is to make the VALOODS framework available within the model to be modified. This can be done by importing all VALOODS definition in the model (in a specific package for example.)

Then each class whose instances may communicate through the network is considered as the top-level layer of a protocol stack, and modifications are made according to the following rules:

- UMLAUT first adds a generalization (inheritance) relationship between each class that can have asynchronous communications with remote sites and the `PROTOCOL_ENTITY` class, making them explicit heirs of `PROTOCOL_ENTITY` (see Figure 11.)
- classes stereotyped as `<<actor>>` are also made heirs of `ACTIVABLE`, so that the behavior of the actor can be activated on demand through the `action` method.
- Since network communication are handled by instances of the `PORT` class, a `protocol_stack` link is established between each instance of the class representing the upper layer and an instance of `PORT` (playing the role of the lower layer). Related `PORT` are connected by a `peer` link along which messages are sent.

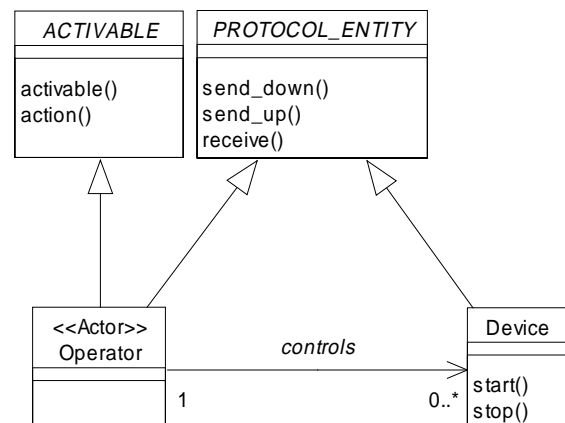


Figure 11. Transformed UML class diagram

Objects of the `ACTIVABLE` class provide a set of stimuli to exercise the dynamic properties of the system. An activable object is just an heir of the abstract class `ACTIVABLE`, which features an entry point called `action` that may be called from time to time by, e.g., a scheduler or a transition-graph builder, provided that the method `activable` returns true. This way, a validation engine can call the `action` operation of any of the relevant objects in order to arbitrary test the system, simulating users' actions or network interfaces' behavior.

### 4.3.2 Dynamic model transformations

The first step is to find all occurrences of method invocation between objects on different nodes. Since the caller and the target objects now both inherit from the ProtocolEntity class, we will redirect method invocations by sending an appropriate message through the *send\_down* operation. Method invocations are to be found on state transition diagrams, where they appear as the result of firing a transition.

For each call-site, we replace the direct invocation by the construction of an appropriate message which is then sent to the lower part of the protocol stack using *send\_down*. Figure 12 shows how a simple call to *device.start()* initiated by the operator is actually transformed.

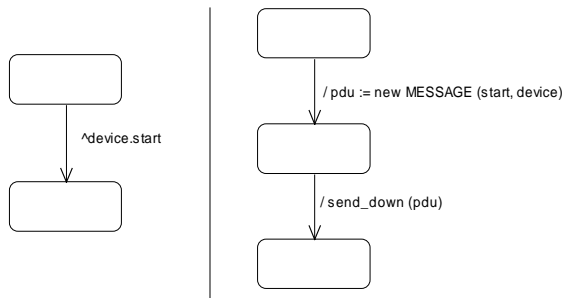


Figure 12. A call to *device.start*

The second step is to produce the state transition diagram corresponding to the *receive* method of *Reactive* objects, which is the “engine” of the automaton associated to protocol entities. It fires transitions (i.e. calls the relevant method) depending on both the received event type and the state in which the object is when the *receive* method is invoked. The implementation of such a method thus needs a double dispatch operation that has several well-known implementation methods (see e.g. the *State* design pattern from [10]).

### 4.4. Generation of validation code

To be able to apply validation techniques to the transformed UML model, this model has to be made executable. UMLAUT is also in charge of generating the code once all the necessary transformations have been realized. This is done by walking through the connected graph stored in UMLAUT in the form of instances of UML Meta-Model classes. The principle is very similar to the generation of implementation code as can be performed by various CASE tools, except that the code is produced specifically for validation purposes.

UMLAUT is written in the Eiffel [21] language. We also have chosen Eiffel as the output produced by UMLAUT when it generates code. An important feature of

Eiffel is its builtin support for assertions. Hence the validity constraints defined on the model and the pre- and post-conditions directly map to Eiffel assertions whose violation can be trapped by the Eiffel execution environment.

The mapping between transitions of the UML dynamic model and methods of an OO language such as Eiffel is obtained according to the following rules:

- the events triggering the transitions become the names of the methods.
- the starting state, plus optional conditions on the event parameters or other conditions on local variables, can be specified in a precondition attached to the method definition.
- the arrival state can be specified in a postcondition.
- the method body must be implemented in such a way that it guarantees the post-condition. If the body only consists of simple actions, these actions can be directly written on the transition labelled by the method name. When the body is more complicated, it can be described by a sub-machine that is executed when the transition is fired (execution of a sub-machine is shown by entering a state representing this sub-machine.)

Below is the Eiffel code implementing the *Device* class whose statechart was given in Figure 7. Since there were only two states, a simple boolean attribute was used.

Pre-conditions and post-conditions are expressed with the *require* and *ensure* keywords respectively.

```
class DEVICE
creation
  make
feature
  is_active : BOOLEAN -- state

make is
  is_active := FALSE
  -- default transition
end -- make

start is
  require
    device_is_idle : not is_active
  do
    is_active := TRUE
  ensure
    device_is_active : is_active
  end -- start

stop is
  require
    device_is_active : is_active
  do
    is_active := FALSE
  ensure
    device_is_idle : not is_active
  end -- stop
end -- DEVICE
```

## 5. Applying validation techniques to the UML model

Within the VALOODS framework, a validation process may be carried on a seamless way. Since our system can now be compiled to a reactive program offering a set of transitions (guarded by activation conditions) located in the activable objects, we have many opportunities to apply the basic technologies that have been developed in the context of FDT based formal validation.

### 5.1. Model-checking

If we want to try the model-checking road, we can use a driver setting the system in its initial state and then constructing its reachability graph by exploring all the possible paths allowed by activable transitions. The only problem is to be able to externalize the relevant global state (made of the states of the various objects in the system, plus the state of the communication queues). We basically solve this problem by leveraging the Memento pattern [10].

The main drawback of this approach is that global state manipulations (comparison, insertion in the table, etc.) are then very costly operations that could compromise large scale model-checking.

### 5.2. Intensive Simulation

For larger systems, an intensive simulation (randomly following paths in the reachability graph) would probably be a more fruitful avenue. Running such a simulation involves the use of a scheduler object implementing a redefinable scheduling policy among the activable transitions (e.g., random selection).

It is also possible to observe the system, using an observer, as in Veda [16]. An observer is a program which permits to catch and analyze informations about execution. It can see every interaction exchanged in the system, and also every internal state of a module.

A protocol sequencing error is detected as a precondition violation on the observer (such an error is detected e.g. when the PORT corrupts data). The execution environment then allows the user to precisely locate and delimit the responsibility of the error, by providing him with an *exception history trace* including a readable dump of the call stack.

Ideally, when the scheduler has driven the system into such a faulty state, it should be possible to transpose the trace (which may not be easy to read) into an equivalent UML interaction diagram (a sequence diagram or a collaboration diagram) representing the critical scenario. This interaction diagram could even be integrated in the original

UML model of the system for documentation purposes, providing the designer with a diagnosis of the problem in the notation that they are familiar with. This feedback allows for correction of the UML design so as to solve the problem, as outlined in Figure 9 where dashed arrows represent feedback.

### 5.3. Test generation with TGV

TGV was first developed in the context of conformance testing of telecommunication protocols. So it is based on standard languages of the domain and thus is applicable to specifications written in SDL [4] or LOTOS [12] and can produce test cases in the TTCN language. Nevertheless, it is relatively independent of any language because it manipulates common models like automata and "transition systems" which are used to represent the possible behaviours of specifications, test purposes and test cases.

The output of TGV is a test case which is given by a graph in an ad hoc format. We can translate this test case into TTCN. In the context of telecommunication protocols, it is important to make this translation as TTCN is the de facto standard for writing test cases. Translating our test cases in another format is quite easy. For example, we can envisage to translate test cases into C code.

On-the-fly generation has already been applied successfully in the context of LOTOS specifications using the CADP toolbox from Verimag [6]. In the context of SDL specifications we have also applied on-the-fly generation using an open version of the ObjectGEODE simulator from Verilog [2] which offers an API with state graph construction functions as described above. In this case some libraries of CADP are also used for graph storage. VALOODS is currently being improved to provide the API required by TGV.

The VALOODS framework should also allow automatic generation of tests to catch improper behavior of an implementation with respect to its specification requirements. Indeed, all the necessary information is accessible from within the framework, as mentioned in section 4. The role of VALOODS/UMLAUT is then that of a bridge between TGV and the UML model of the system being designed.

TGV is very efficient because its algorithms are based on efficient algorithms coming from the model-based verification domain. In this domain, there are algorithms whose purpose is to check that a specification satisfies a property given by a logic formula or by an automaton. Some algorithms are based on traversal of the state graph. If the property is not satisfied, some tools provide a diagnosis sequence. The algorithm of TGV adapts this principle. Searching a sequence of the specification which satisfies the test purpose can be seen as producing a sequence that characterizes the non satisfaction of the negation of this test pur-

pose. In fact, it is more complicated as we don't produce a single sequence but a set of sequences i.e. a sub-graph. Very efficient algorithms exist for doing this, and in particular those which perform "on-the-fly" verification are well adapted for "on-the-fly" generation of test cases.

## 6. Conclusion and future work

We have shown the interest and feasibility of integrating formal verification and validation techniques in an established OO life-cycle for the construction of correct OO distributed software systems. We have described how a continuous validation framework can be set up to go smoothly from the OO analysis to the OO implementation of a validated distributed system. This approach is not limited to simple problems: the intensive simulation techniques have already been used on real OO systems, e.g., the implementation of a parallel SMDS server where it allowed to detect non trivial problems at early stages of the life-cycle [19].

Future work will consist in consolidating and extending VALOODS to deal with higher level interactions between distributed objects (e.g. in the context of CORBA.) Once UMLAUT/VALOODS is a truly usable validation framework, we will make it widely available (see <http://www.irisa.fr/pampa/UMLAUT/>).

## References

- [1] *UML Semantics*. Rational Software Corporation, September 1997.
- [2] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON Project: A VALidatiON environment for SDL/MSD Descriptions. In *SDL 93 Forum*, 1993.
- [3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd edition, 1994.
- [4] CCITT. *SDL, Recommendation Z.100*, 1987.
- [5] J. Fernandez, C. Jard, T. Jéron, and L. Mounier. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1993.
- [6] J.-C. Fernandez, H. Garavel, L. Mounier, C. R. A. Rasse, and J. Sifakis. A toolbox for the verification of programs. In *International Conference on Software Engineering, ICSE'14, Melbourne, Australia*, pages 246–259, May 1992.
- [7] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96)*, New Brunswick, New Jersey, USA, LNCS 1102. Springer, July 1996.
- [8] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, (29):123–146, 1997.
- [9] M. Fowler. *UML Distilled : Applying the Standard Object Modeling Language*. Addison-Wesley Object Technology series, 1997.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [11] S. Graf, J. Richier, C. Rodriguez, and J. Voiron. What are the limits of model checking methods for the verification of real life protocols? In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989. Springer-Verlag, LNCS #407, pages 189–196.
- [12] ISO. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO/ DP 8807, March 1985.
- [13] ISO. *Estelle: a Formal Description Technique based on an Extended State Transition Model*. ISO 9074 TC97/SC21/WG6.1, 1989.
- [14] M. Jackson. *System Development*. Prentice-Hall International, Series in Computer Science, 1985.
- [15] I. Jacobson. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1994.
- [16] C. Jard, R. Groz, and J. Monin. Development of VEDA: a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, volume 14,3, pages 339–352, March 1988.
- [17] T. Jéron and P. Morel. Abstraction,  $\tau$ -réduction et détermination à la volée : application à la génération de test. In G. Leduc, editor, *CFIP'97 : Ingénierie des Protocoles*. Hermes, September 1997.
- [18] J.-M. Jézéquel. Experience in validating protocol integration using Estelle. In *Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain*, November 1990.
- [19] J.-M. Jézéquel, X. Desmaison, and F. Guerber. Performance issues in implementing a portable SMDS server. In IFIP, editor, *6th International IFIP Conference On High Performance Networking*, pages 267–278. Chapman & Hall, London, September 1995.
- [20] S. Kirani. *Specification and Verification of Object-Oriented Programs*. Phd thesis, University of Minnesota, 1994.
- [21] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
- [22] D. E. Monarchi and G. I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 9(35):35–47, September 1992.
- [23] R. M. Poston. Automated testing from object models. *Communications of the ACM*, 37(9):48–58, Sept. 1994.
- [24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.
- [25] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. In J. Diaz and F. Orejas, editors, *TAPSOFT'89 : Theory and Practice of Software Development*. Springer-Verlag, March 1989.