L'ingénierie des protocoles avec UML

Claude Jard, Jean-Marc Jézéquel, Alain Le Guennec, Benoit Caillaud

Protocol Engineering using UML

Claude Jard Jean-Marc Jézéquel Alain Le Guennec Benoit Caillaud
and the UML Group*

IRISA, Campus de Beaulieu
F-35042 Rennes Cedex
Email : First name.Last name@irisa.fr
Web : www.irisa.fr/pampa

Abstract

Despite the irresistible growth of interest in formal methods and related validation and verification tools, the development of distributed systems seldom relies on them. We claim this is mainly due to formal methods lack of support for modern software life-cycles. The construction and maintenance of open distributed systems are mostly based on object-oriented software development. We investigate how frameworks may help to embed formal validation techniques in an object-oriented process based on the UML notation. We show how standard model-checking techniques can be used right now on UML models by exploiting informations available in class and deployment diagrams, and using an operational semantics of statecharts. We also present how the behavioural views of UML, including sequence or collaboration diagrams could be consistently managed using a common semantics model called BDL (standing for "Behavioural Description Language") with which the various behavioural views of UML can be translated into one another. BDL is a reactive synchronous language with a true concurrency semantics. Basic object interactions are represented in BDL by partially ordered sets of events and the behaviour of a complete (or incomplete) system is expressed by composition of basic interactions. BDL offers new perspectives for a flexible verification of systems by modeling them as globally asynchronous networks of locally synchronous software components.

Key words: protocol engineering, object-oriented modeling, validation and verification, software engineering, UML, BDL

1 Introduction

The techniques currently used in the development of distributed software on communication networks most often leads to flimsy and poorly maintainable software. Numerous software engineering methods are proposed to tidy up this practice. They have to cover the different phases starting from preliminary requirement specifications, then design phases and down to the use of software tools to perform verification, validation, test and code generation. A true protocol engineering craft based on mathematically founded methods has been consolidated

*This paper presents a synthesis of a collective work at IRISA with many contributors. This group is composed of A. Benveniste, B. Caillaud, H. Canon, WM. Ho, C. Jard, JM. Jézéquel, A. Le Guennec, F. Pennaneach, JP. Talpin. This research is partially funded by Alcatel.

during these past ten years. It relies on languages with formally defined semantics allowing for a whole set of provably correct transformations. In parallel to this consolidation, the development of general, non distributed software promoted the use of object-oriented design and programming. In that context, the interest for the seamless OO development process has somewhat eclipsed formality. This can be seen as a retreat in the development of complex critical systems. The challenge is thus to introduce formal methods as soon as possible in the development process and to keep using them all along the process. Most formal description techniques (FDT for short) designed for protocol engineering cannot be easily used in an integrated OO life-cycle. They all have specific semantic models, suited for a given particular step of development. Their use does not prevent "model ruptures" (breaches in the continuity of models) which are detrimental to the soundness and efficiency of the development process. This explains the growing interest for designing "frameworks", specialized for telecommunication software, and providing a set of formal engineering tools. The present paper adheres to this timely attitude.

Object design methodologies are now gathered in the UML notation ("Unified Modeling Language" [4]), which is becoming a standard and diffuses at high speed in industrial circles.

The objective of this paper is to cast a method and present tools which extensively re-use several formal validation techniques developed during the few past years in the context of protocol engineering and to apply them to an OO design method based on UML.

The plan of the paper is the following: We start by briefly recalling the techniques of validation based on simulation and their situation with respect to the necessary integration in a software life-cycle centered on object-oriented design. Then we present the current state of UML. The third part shows how to generate a simulation code by refinement. We then discuss the way of connecting the validation tools. The last section is devoted to the presentation of BDL and the prospects it opens.

2 Simulation Techniques for Validation and Verification of Distributed Software

2.1 A set of complementary formal techniques

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 (abstracted) models of the considered problem [15].
- *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 behaviours. 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.
- *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.

All these approaches are complementary. 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 real problems. They are now disseminating in small industrial niches, mainly in the context of development of hardware or critical systems.

2.2 Difficulties in using FDTs

It is very disappointing to see that formal validation based on standard FDTs (such as SDL [9], Estelle [17] and Lotos [16]) never acceded to a widespread use in the industry, despite excellent results on most of the pilot projects where it has been used [20]. 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. 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 [8] method, OMT [22] or OOSE [18], which were the most influential during the design of UML, 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 well-defined OO modeling language: indeed, UML relies upon a *meta-model* [6] which is formally defined (at least in its syntactic aspects, and only partially for the dynamical behavioural aspects), 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 and is now an OMG standard.

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 [13].

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 1, 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.

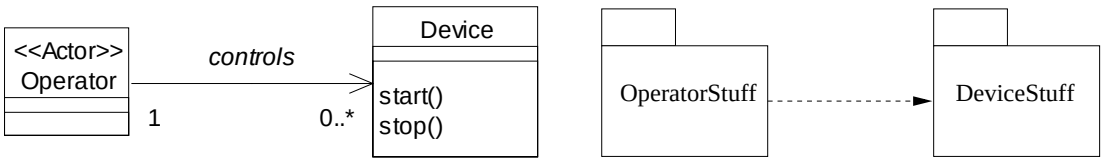

Figure 1: UML class and package diagrams

Classes are represented as boxes divided into three parts: the name, attribute and operation compartments.

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 1.

3.3 Behaviour 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 2. Objects participating in the collaboration are laid out along the horizontal axis, while the vertical axis represents time.

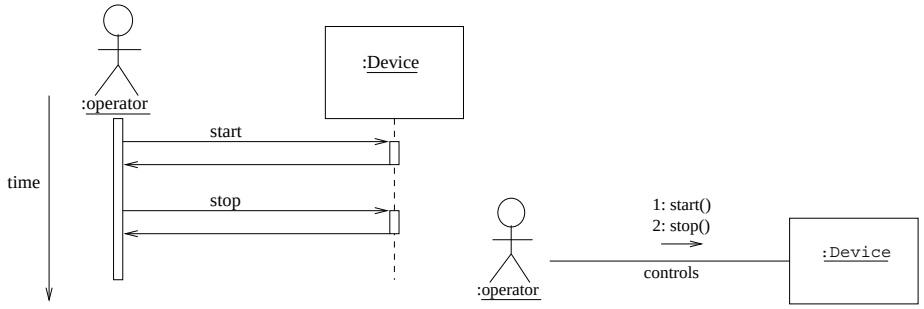

Figure 2: UML sequence and collaboration diagrams

Collaboration diagrams provide another view of object interactions. Contrary to sequence diagrams, time is not 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.

Statechart diagrams describe the evolution of an object over its life time. Figure 3 represents the statechart corresponding to the Device class. Figure 3 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.

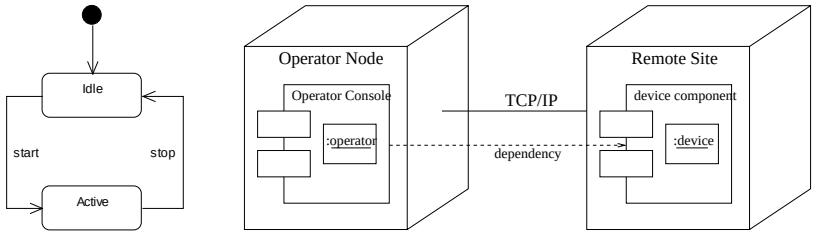

Figure 3: UML statechart diagram of the Device class and the deployment diagram

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.

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.

4 Bringing validation in the OO life-cycle

4.1 An integrated OO 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 4 where dashed arrows represent feedback from validation results. The key point in implementing this idea is to rely on the sound technological basis (mainly based on a simulation principle) 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.

UMLAUT exposes the properties relevant to the validation by automatically building a more detailed, but equivalent, UML model that explicitly shows the new states and transitions resulting from the actual way (e.g. asynchronous) objects do communicate within a framework modeling the underlying middle-ware. This framework, called VALOODS (VALidation of Object Oriented Distributed Software) is presented in Section 4.2, and the transformation process is described in Section 4.3.

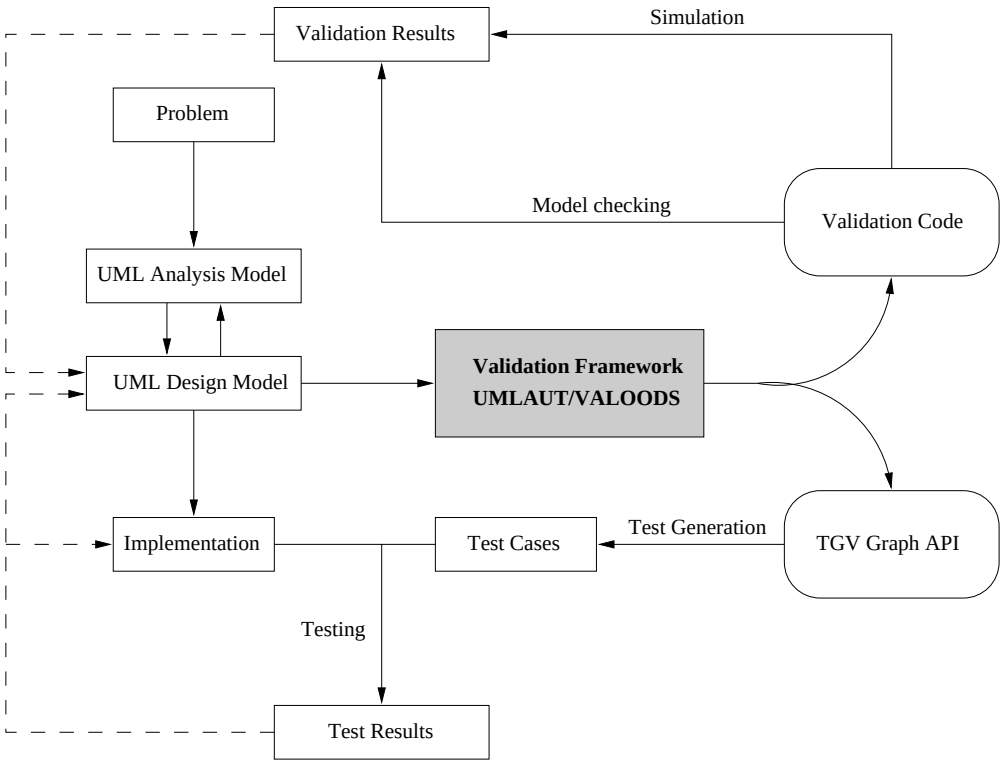

Figure 4: OO Life Cycle

The VALOODS framework offers the possibility to walk through the graph of reachable states of the distributed system. Actually, it defines an abstract interface for walking through the state graph. We can then conduct model checking, intensive simulation, or test cases generation by plugging the appropriate "validation engine" (for instance CADP [11]) into

the framework (see Section 5).

UMLAUT currently uses CDIF (CASE Data Interchange Format) as its exchange format[1] 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 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

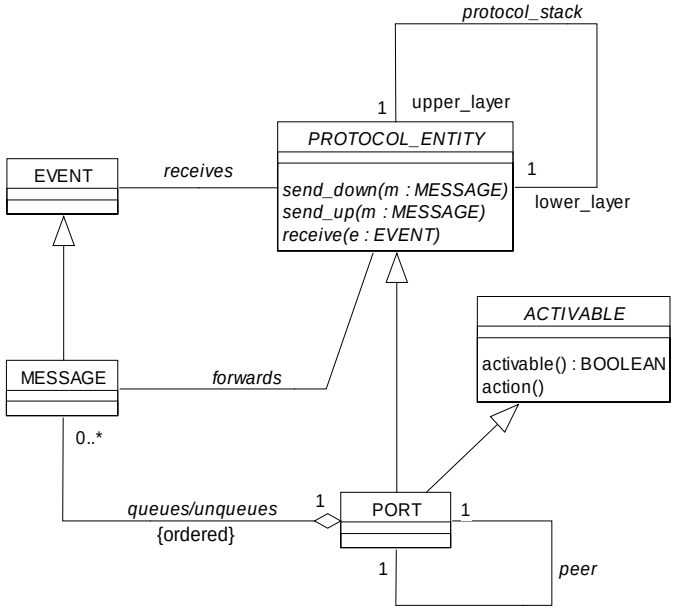

Figure 5: UML class diagram of VALOODS

The framework consists of a collection of classes (see Figure 5) together with 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.

[1] An XMI support is also planned in the near future.

- 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 flavours (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.

4.3 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.

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 6.)
- classes stereotyped as <<actor>> are also made heirs of ACTIVABLE, so that the behaviour 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.

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' behaviour.

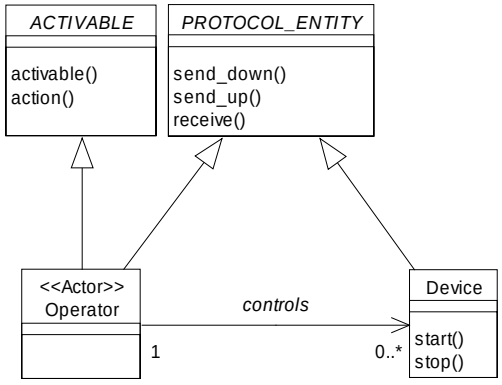

Figure 6: Transformed UML class diagram

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 7 shows how a simple call to `device.start()` initiated by the operator is actually transformed.

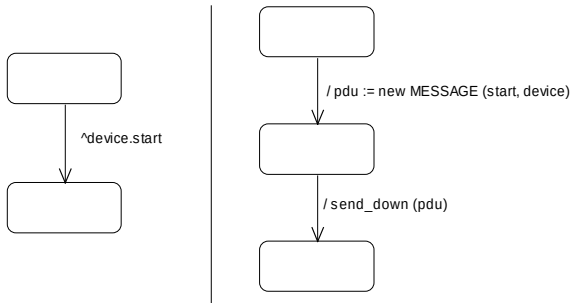

Figure 7: A call to device.start

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 invocated. 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 [14]).

4.3.3 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 3. 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                                  stop is
      require                                   require
         device_is_idle   : not is_active          device_is_active : is_active
      do                                        do
         is_active := TRUE                         is_active := FALSE
      ensure                                    ensure
         device_is_active : is_active              device_is_idle  : not is_active
      end  --  start                            end -- stop
end  --  DEVICE
```

5 Applying validation techniques

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 [10]. 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 [14].

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. Looking for more clever ways of representing the global state (since we don't forbid the dynamic creation of objects, its size is not even a constant) of a UML model is one of our research goals.

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 [19]. 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. 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 designers 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 4 where dashed arrows represent feedback.

5.3 Test synthesis and generation with TGV

TGV [12] 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 [9] or LOTOS [16] 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 in an ad hoc format. From this format, test cases can be generated as either TTCN tables, C programs or even UML message sequence charts.

On-the-fly generation has already been applied successfully in the context of LOTOS specifications using the CADP tool-box from Verimag [11]. In the context of SDL specifications we have also applied on-the-fly generation using an open version of the ObjectGEODE simulator from Verilog [7] 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 behaviour 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.

6 Towards a Semantic Pivot for UML

The gap between the declarative views (classes and sequences) and the behavioural view (state-charts) is still too large to allow a consistent management of these two aspects of design. Our approach is to use a common semantics model (BDL) in which the views can be translated one into the others.

BDL is an enrichment of the class diagram notation with behavioural concepts. Each class or object arises as a directed graph whose nodes are valued events. These graphs specify scheduling constraints between their event nodes. A class or object comprises a disjunction of such graphs, possibly guarded by conditions on values. The execution of a class proceeds by successive reactions; each of which transforms the state of objects in reaction to stimuli from the environment and emits in return some events to the environment. Two objects are composed by unification of their respective reactions.

BDL recovers in fact the technology developed for synchronous languages (Esterel, Lustre, and in fact more particularly Signal [5]). The new point, which relies on recent fundamental results, is the possibility to interpret indifferently BDL objects according to a synchronous or an asynchronous interaction mode. An effective characterization of those systems of objects with equivalent synchronous and asynchronous behaviours has also been established [1].

A first version of a denotational semantics of BDL was published in [2]. In this section we present for the first time an operational semantics allowing to define a concept of execution. The declarative character of the language raises difficulties (the analogue with the problem of the translation from scenarios to automata). A prototype of simulator is under development by using the Open-Caesar environment (CADP) [3]. In the short run, this environment will give access to various functions of validation (model-checking, generation of tests...) which we will try to specialize.

6.1 Presentation of BDL

BDL is intended to enrich mainly the dynamic views of UML. The idea is that one can associate with each box of static diagrams (classes or objects) a behavioural part described by a BDL graph. In BDL diagrams, boxes (classes or objects) will thus have generally three parts: a name (C), declarations (D) and a graph of actions (A).

The actions (Figure 8) of the objects constitute the behavioural part, expressed in BDL. The basic nodes are formed by the *ports*. A port has a name and possibly a value being used for the communication. Then, the graphs are combined together using parallel composition, the choice operator and causality arrows, in order to describe the (partial) scheduling of actions. A special operator ! denotes a synchronization of actions.

A denotational semantics of BDL is given by a family of graphs. This family is built by considering the graphs associated with the objects, then by considering their parallel composition. These graphs summarize the possible dependences between actions in the course of time. The exact configuration depends on the values of variables at the moment being

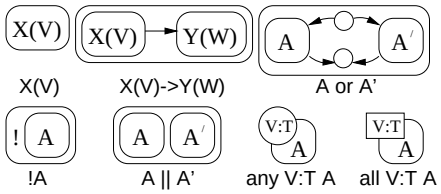

Figure 8: Graphical syntax for the BDL actions

considered. Thus "running" a BDL graph consists in "unfolding" this graph and studying its evolution along the successive instants. Concerning the meaning of an instant, various points of view are desirable, according to whether the system is placed in a synchronous, asynchronous or mixed environment. These subtle variations are the subject of the section 6.2.

The behaviour of an object consists in a labelled graph, whose nodes are either ports, or special labels (*pins*) forming the borders between instants and being used for the concatenation of the graphs. Values of ports are represented by either a constant value, or a variable (which name is preceded by a star *) intended for objects internal communications. In this last case, it is an implicit choice on the value of the variable, chosen in a set of possible values (its type). A pin has also an optional value, which represents the value of an object's attribute. They define the global state of the system at the borders between the instants. An example is given in Figure 9.

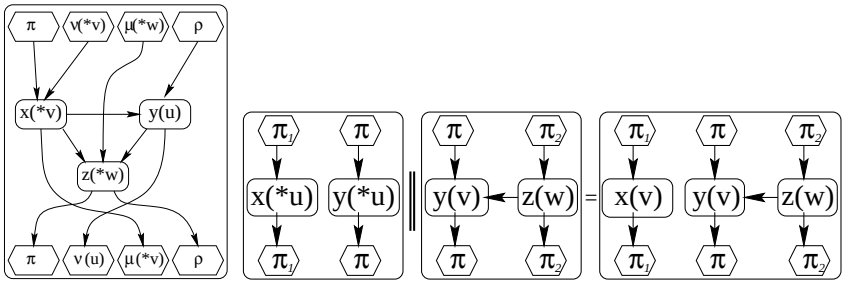

Figure 9: Parallel composition of BDL graphs

These elementary graphs can then be merged together (Figure 9) in order to give the behaviour of a set of objects as a whole.

6.2 Operational Semantics

The execution of graphs is defined by a labelled transitions system where a transition represents the selection of an action to be carried out. It mainly consists in the choice of a port and of a value of the corresponding variable.

Choosing a particular execution semantics among the four operational semantics (synchronous with small or great steps, asynchronous, or asynchronous-synchronous) depends on the point of view adopted. Synchronous semantics with small steps is the most detailed. It makes explicit every action (choice of port and value) and the passage through borders of instants. Great step synchronous semantics takes into account only the state between two atomic instants. Great step semantics amounts to masking intermediate configurations within an instant. The asynchronous semantics uses again the same principles as in the synchronous semantics with small steps. However synchronization on global states is lost. And finally asynchronous-synchronous semantics combines the two aspects, by considering synchronous *capsules*, where the execution is performed in a synchronous way and which interact with one another in an asynchronous way.

One considers a BDL *term*. It is either a graph of partial order, or the parallel composition ($\|$) of two BDL terms or a disjunction ($\vee$). The *signature* of a term is the union of the minimal labels of the graphs. The signature of the graph of Figure 9 is $\pi, \nu(v), \mu(*w), \rho$. An asynchronous term is a BDL term is formed with the asynchronous parallel composition of two terms ($\|_a$).

Indeed, a BDL term corresponds to the execution of the program during one instant. There is a strong synchronization at the end of each instant (border of instant). During the asynchronous execution, one removes these borders which remain only in a logical state. One obtains this by *concatenation* on graphs of both sides of the border (Figure 10). A complete execution comes to concatenate an unspecified number of graphs of the family of the term.

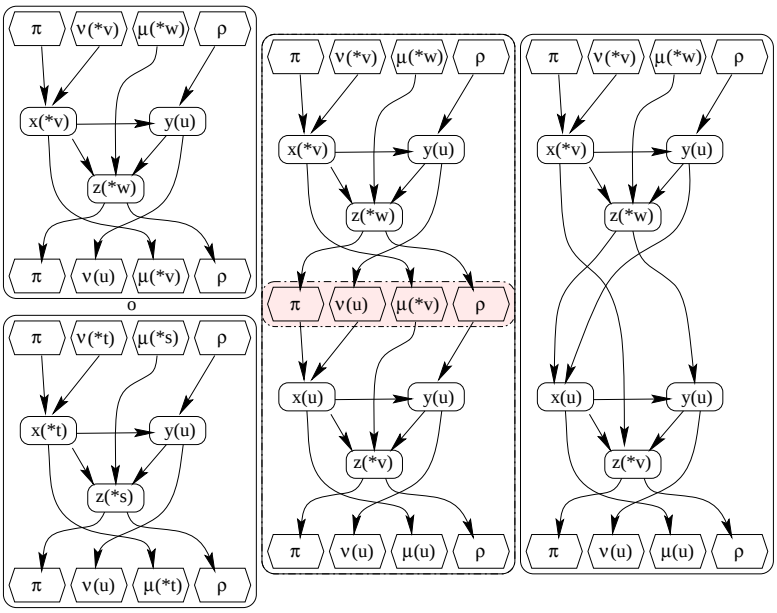

Figure 10: BDL graph concatenation

With each node, one associates a value. This value must be in conformity during the various operations of composition and concatenation. This value can be either a value of a certain type, or the set of the values of the type. It acts in this last case of an existential quantification. The exact value of the port will be given only after unification with the remainder of the program. One can also add a particular value $\perp$ (the "absent" value) who means that the node does not take part in the execution in the considered instant.

6.2.1 Synchronous semantics

Let us consider the execution of a BDL term over one instant. The starting data are B_0 the term which one considers, and M_0 the state of the memories at the initial instant. The current state is composed of a term B partially executed, and of an environment E containing the values of the variables of the term B. One notes it (b, e). The initial state is (b_i, e_i) where B_i is the term B_0 in which the pins took the value of the memories at the initial instant (given by M_0), and E_i the empty set (no variable is defined at the beginning of the instant) to which one added the variables defined by the memories M_0.

The transitions are labelled by $x(v)$ where x is the label which one considers and v the value or a set of values for the label x.

Simple graph If B is a simple graph G, one starts a transition taking one from the labels of the signature. We then remove it from the graph. ($G \backslash x$ represents the graph G in which

one withdrew the port x and all edges connected to this port).

$$\frac{x(v) \in min(G) \quad G' = G \backslash x}{G, E \xrightarrow{x(v)} G', E}$$

Parallel composition In the case of parallel composition, one starts the transition if the two terms start the same transition with compatible values, or if the fireable label of one is absent from the other.

$$\frac{\begin{array}{ll} B_1, E \xrightarrow{x(v_1)} B_1', E_1' & v = unif(v_1, v_2) \\ B_2, E \xrightarrow{x(v_2)} B_2', E_2' & E' = E_1'[v_1 \leftarrow v] \cup E_2'[v_2 \leftarrow v] \end{array}}{B_1 \parallel B_2, E \xrightarrow{x(v)} B_1' \parallel B_2', E'}$$

The unification $unif(v_1, v_2)$ amounts taking the intersection of the sets of values v_1 and v_2. If the result is the empty set, then the unification fails. In this case the rule is not applicable and one cannot start the transition. The graphs are not composable.

$$\frac{B_1, E \xrightarrow{x(v)} B_1', E' \quad x \notin B_2}{B_1 \parallel B_2, E \xrightarrow{x(v)} B_1' \parallel B_2, E'} \qquad \frac{B_2, E \xrightarrow{x(v)} B_2', E' \quad x \notin B_1}{B_1 \parallel B_2, E \xrightarrow{x(v)} B_1' \parallel B_2, E'}$$

$x \notin B_2$ means that there is no label of B_2 containing x. One of both graph continues its execution independently of the other. It is different from the case where the value of the variable is $\perp$. In this case, the absent value must be on a port common to both graphs.

Disjunction Disjunction corresponds to the choice. One takes only one of the two terms.

$$\frac{B_1, E \xrightarrow{x(v)} B_1', E'}{B_1 \vee B_2, E \xrightarrow{x(v)} B_1', E'} \qquad \frac{B_2, E \xrightarrow{x(v)} B_2', E'}{B_1 \vee B_2, E \xrightarrow{x(v)} B_2', E'}$$

If one of two rules leads to impossibility (a unification which fails), one returns (backtracking) to take the other rule. If both fail, one goes up higher to test a possible other choice.

End of instant The instant is finished when all the non-maximum labels were consumed. Once that there remain only maximum labels, one updates the memories (which contain the values associated with these labels), then one passes at the following instant by taking again the whole term.

$$(exit_{x_1}(v_1), ..., exit_{x_i}(v_i)), E \xrightarrow{\tau} G', E_i$$

where $G' = G_0$ in which the minimal pins take the values in the memories at the preceding instant, i.e. $x_1(v_1), ..., x_i(v_i)$.

Great step semantics One considers only the borders of instants. The state is thus the value of the memories M. The transitions are labelled by the words λ from semantics with small steps.

$$\frac{B, E_i \xrightarrow{\lambda\tau}{}^{*} B', E_i \quad M = B|_{min(B)} \quad M' = B'|_{min(B')}}{M \Rightarrow^{\lambda} M'}$$

This semantics is particularly adapted in the synchronous case. One reasons in logical terms of instants. The way in which was carried out intermediate computations are less significant than the value of the memories at the clock signal.

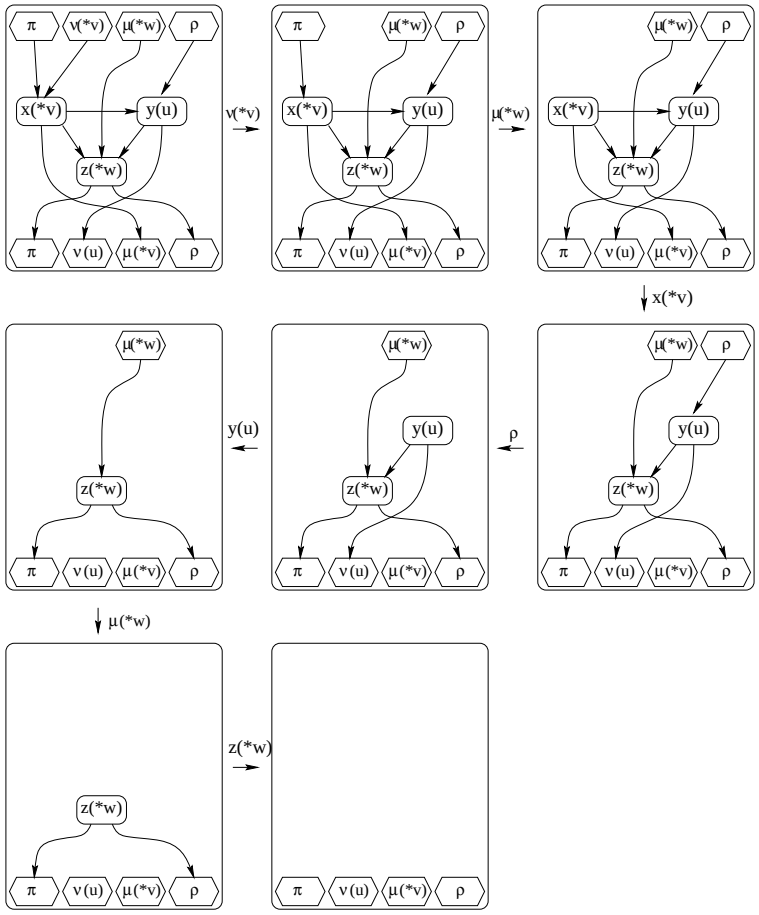

Figure 11: Possible execution of a BDL graph

6.2.2 Asynchronous semantics

One considers asynchronous terms here, but without synchronous parallel composition inside. That amounts considering that the objects communicate only in an asynchronous way. It is no more possible to test the presence of $\perp$.

The principle is the same as for semantics with small steps (synchronous), except that instead of stopping with the maximum labels (at the border of the instant), one concatenates the term corresponding to the following instant in order to continue the execution. One is obliged to take into account the instant (its sequence number) in which the current execution is: one also preserves the values of the variables at every instant, and the values of the memories at the borders of instants. It is also necessary to keep the complete graph for concatenation with the passage of instant.

Current state $(B^0, B, \overline{E}, \overline{M})$ is made of B^0, the considered term; G the partially evaluated term in which each node is labelled by the instant number; $\overline{E} = (E_1, E_2, ...)$ the sequence of contexts (valuations of variables for every instants) and $\overline{M} = (M_1, M_2, ...)$ the sequence of valuations of pins at the encountered borders.

Simple graph In the case of the simple graph, one proceeds in the same way as previously.

$$\frac{(x(v), i) \in min(G) \quad G' = G \backslash x}{G^0, G, \overline{E}, \overline{M} \xrightarrow{x_i(v)} G^0, G', \overline{E}, \overline{M}}$$

Parallel composition The parallel composition spreads too. Here, a purely asynchronous composition was chosen. It is simply necessary to take care with which instant to refer.

$$\frac{\begin{array}{ll} B_1^0, B_1, \overline{E}_1, \overline{M}_1 \xrightarrow{x(v_1)} B'^0_1, B'_1, \overline{E}'_1, \overline{M}'_1 & v = unif(v_1, v_2) \\ B_2^0, B_2, \overline{E}_2, \overline{M}_2 \xrightarrow{x(v_2)} B'^0_2, B'_2, \overline{E}'_2, \overline{M}'_2 & \overline{E}' = \overline{E}'_1[v_1 \leftarrow v] \cup \overline{E}'_2[v_2 \leftarrow v] \\ & \overline{M}' = \overline{M}'_1[v_1 \leftarrow v] \cup \overline{M}'_2[v_2 \leftarrow v] \end{array}}{B_1^0 \parallel_a B_2^0, B_1 \parallel_a B_2, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1 \parallel_a B'^0_2, B'_1 \parallel_a B'_2, \overline{E}', \overline{M}'}$$

$$\frac{B_1^0, B_1, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1, B'_1, \overline{E}'_1, \overline{M}'_1 \quad x \notin B_2 \vee x(\bot) \in B_2}{B_1^0 \parallel_a B_2^0, B_1 \parallel_a B_2, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1 \parallel_a B'^0_2, B'_1 \parallel_a B'_2, \overline{E}', \overline{M}'}$$

Moreover, the absent value allows to advance one of the two graphs compared to the other. It may be thus that it occurs several instants before a communication can take place on a port.

Disjunction The choice is treated in the same manner.

$$\frac{B_1^0, B_1, \overline{E}_1, \overline{M}_1 \xrightarrow{x(v)} B'^0_1, B'_1, \overline{E}', \overline{M}'}{B_1^0 \vee B_2^0, B_1 \vee B_2, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1, B'_1, \overline{E}', \overline{M}'}$$

Border of instant When one arrives on a maximum pin, one concatenates again the term with the current one in order to be able to continue. An example is given in Figure 12.

$$\frac{(x(v), i) \in max(G) \cap min(G) \quad G' = G \backslash x \quad \overline{M}' \mathrm{tq} M'_i(x) = v}{G^0, G, \overline{E}, \overline{M} \xrightarrow{x(v)} G^0, G' \oplus G^0, \overline{E}, \overline{M}'}$$

6.2.3 Synchronous-asynchronous semantics

Mixing at the same time synchronous and asynchronous semantics is required when certain systems are made of objects joined together in a synchronous capsules (and communicate in a synchronous way), while the capsules communicate in an asynchronous way.

One takes again the synchronous semantics to which one adds the rules for the synchronous composition.

$$\frac{\begin{array}{ll} B_1^0, B_1, \overline{E}_1, \overline{M}_1 \xrightarrow{x(v_1)} B'^0_1, B'_1, \overline{E}'_1, \overline{M}'_1 & v = unif(v_1, v_2) \\ B_2^0, B_2, \overline{E}_2, \overline{M}_2 \xrightarrow{x(v_2)} B'^0_2, B'_2, \overline{E}'_2, \overline{M}'_2 & \overline{E}' = \overline{E}'_1[v_1 \leftarrow v] \cup \overline{E}'_2[v_2 \leftarrow v] \\ & \overline{M}' = \overline{M}'_1[v_1 \leftarrow v] \cup \overline{M}'_2[v_2 \leftarrow v] \end{array}}{B_1^0 \parallel B_2^0, B_1 \parallel B_2, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1 \parallel B'^0_2, B'_1 \parallel B'_2, \overline{E}', \overline{M}'}$$

$$\frac{B_1^0, B_1, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1, B'_1, \overline{E}'_1, \overline{M}'_1 \quad x \notin B_2}{B_1^0 \parallel B_2^0, B_1 \parallel B_2, \overline{E}, \overline{M} \xrightarrow{x(v)} B'^0_1 \parallel B'^0_2, B'_1 \parallel B'_2, \overline{E}', \overline{M}'}$$

In this case, the absent value does not make possible to advance the other graph. To advance the graph it is necessary that both have the same value for the same pin (normal value or absent value).

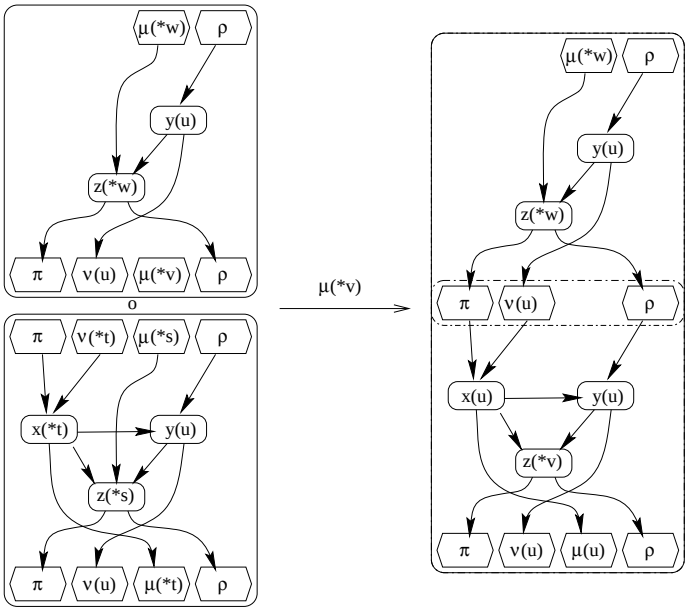

Figure 12: Asynchronous execution of a BDL graph at the border of instant

6.3 Implementation in CADP

To use OPEN/CAESAR and the software which is associated to him, it is enough to define functions handling the input language. In particular, a function which, being given a state of the execution gives the list of the possible transitions starting from this state. The tool then explore the transition system obtained.

Thus, starting from the rules of the asynchronous semantics, one builds this function. Being given the modular structure of the rules (according to that of BDL), one has a series of functions treating each case.

6.4 Optimization by structuring BDL graphs

Building a hierarchical BDL graph is the role of a so-called clock calculus. The clock calculus consists in determining before execution which are the instants when given ports are present. Indeed if a port is present in one term of a parallel composition, but not in the other, it is pointless to carry on the simulation, since the unification is due to fail. A clock is a special boolean port saying at every instant if a port of the specification is present (active) or not. The clock calculus consists in finding the values of these clocks for all the variables of the program. Once these clocks known, one does keep in the family of graphs only those resulting from successful unifications.

7 Conclusion

The approach which has been presented can be characterized by the following features:

- It focuses on well-tried techniques of model-checking which have the advantage of being easily implemented from the definition of operational semantics of the formal languages in use. Actually, the generation of an exhaustive simulator gives access to formal validation like checking properties, intensive simulation or generation of tests.

- It proposes a set of transformations of UML models, specialized in the production of protocol-oriented simulation code. Only the statecharts view of system dynamics is used for simulation. It is the purpose of the UMLaut tool we develop in our group.
- It includes a prototype of a validation framework relying on the CADP and Open-Caesar packages, developed at INRIA in Grenoble and offering a whole set of validation functions which are interfaced to the API of the simulator generated by UMLaut.
- We also prepare for the future by implementing a pivot semantic model in which the behavioural aspects of the various UML views (sequence diagrams, collaboration diagrams, statecharts...) can be expressed. This should allow for joint and coherent simulations of a multi-view UML models. We developed our own language, called BDL (Behavioural Description Language). It currently enables us to experiment with various operational semantics based on synchronous or asynchronous semantics of computation. They can deal with specifications mixing at the same time statecharts and High-level Message Sequence Charts (HMSCs)[2].

References

[1] A. Benveniste, B. Caillaud and P. Le Guernic. From Synchrony to Asynchrony. In *CONCUR'99, 10th International Conference on Concurrency Theory*, 1999.

[2] Jean-Pierre Talpin, Albert Benveniste, Benoit Caillaud, Claude Jard, Zakaria Bouziane, Hubert Canon. *BDL, a specification language for distributed object-oriented real-time systems.* Proceedings of ISORC'98 : International Symposium on Object-oriented Real-time distributed Computing, 1998.

[3] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighineanu. In Computer Aided Verification. *CADP: a protocol validation and verification toolbox*, 1996.

[4] S.W. Ambler. *The Unified modeling language and beyond: the techniques of object-oriented modeling*, (http://www.ambysoft.com), August 1997.

[5] A. Benveniste, P. Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, 1991.

[6] *UML Semantics.* Rational Software Corporation, September 1997.

[7] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON Project: A VALidatiON environment for SDL/MSC Descriptions. In *SDL 93 Forum*, 1993.

[8] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 2nd edition, 1994.

[9] CCITT. *SDL, Recommendation Z.100*, 1987.

[10] 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.

[11] 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.

[2]HMSCs are a superset of the current UML sequence diagrams and will probably be integrated in UML 2.0.

[12] 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.

[13] M. Fowler. *UML Distilled : Applying the Standard Object Modeling Language*. Addison-Wesley Object Technology series, 1997.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[15] 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.

[16] ISO. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO/ DP 8807, March 1985.

[17] ISO. *Estelle: a Formal Description Technique based on an Extented State Transition Model*. ISO 9074 TC97/SC21/WG6.1, 1989.

[18] I. Jacobson. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1994.

[19] 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.

[20] 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.

[21] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.