HAL
archives-ouvertes.fr

An Approach to Integrate Formal Validation in an OO Life-cycle of Protocols

Claude Jard, Jean-Marc Jézéquel, Laurence Nedelka

▶ **To cite this version:**

Claude Jard, Jean-Marc Jézéquel, Laurence Nedelka. An Approach to Integrate Formal Validation in an OO Life-cycle of Protocols. [Research Report] RR-2808, INRIA. 1996. <inria-00073884>

HAL Id: inria-00073884
https://hal.inria.fr/inria-00073884

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Approach to Integrate Formal Validation in an OO Life-cycle of Protocols

Claude Jard, Jean-Marc Jézéquel, Laurence Nédelka

N° 2808

Février 1996

PROGRAMME 1

ISSN 0249-6399

An Approach to Integrate Formal Validation in an OO Life-cycle of Protocols

Claude Jard*, Jean-Marc Jézéquel**, Laurence Nédelka***

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet PAMPA

Rapport de recherche n°2808 — Février 1996 — 23 pages

Abstract: Despite excellent results on pilot projects, formal validation based on standard Formal Description Techniques (FDTs) never really catch up in the industry. We claim that this is mainly due to standard FDTs lack of support for the modern software development methods and life-cycles needed in the construction and maintenance of open distributed systems. We propose to go the other way round, that is to integrate formal validation technology within well established object-oriented (OO) development methods. Building on the intuition that a universal language taking into account all the possible semantics aspects of parallelism and communication is a holy grail, we propose to rely on an open (but simpler) OO language to build dedicated *frameworks*. Such frameworks can be specialized toward classes of distributed applications, and integrate formal validation tools. We illustrate our approach using the famous alternating bit protocol example. We investigate on this example how a continuous validation framework could be set up to go smoothly from the OO analysis to the OO implementation of a validated distributed system.

Key-words: Object-Oriented modeling, validation and verification, protocols, software engineering, Alternating-Bit-Protocol.

(Résumé : tsvp)

*Irisa/CNRS, Tel: +33–99847193 — Fax: +33–99847171 — E-mail: jard@irisa.fr
**Irisa/CNRS, Tel: +33–99847192 — Fax: +33–99847171 — E-mail: jezequel@irisa.fr
***Irisa/Inria, Tel: +33–99847191 — Fax: +33–99847171 — E-mail: lnedelka@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Une approche pour intégrer la validation formelle dans un cycle de vie objet des protocoles

Résumé : Bien qu'ayant déjà apporté d'excellents résultats sur des projets pilotes, les techniques de descriptions formelles ne sont pratiquement pas utilisées dans l'industrie pour la validation de protocoles. Ceci est essentiellement dû à un manque de support pour le formalisme dans les outils modernes de développement de logiciels. Nous proposons une solution qui consiste à intégrer des techniques de formalisation dans le cylce de développement d'un logiciel orienté objet. Plutôt que d'étendre un langage à tous les aspects possibles du parallélisme, ce qui semble irréalisable, nous proposons de construire des frameworks dans un langage orienté objet existant, le langage Eiffel. Chaque framework ainsi construit est spécialisé pour une classe d'applications distribuées et intègre des outils de validation de protocoles. Nous illustrons notre démarche par l'exemple du protocole du bit alterné.

Mots-clé: Modélisation orientée objet, validation et vérification, protocoles, ingénierie logicielle, protocole du bit alterné.

1 Introduction

It is now widely admitted [11] that only system development based on "real-world" modeling is able to deal with the complexity and the versatility of open distributed 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 [18]. 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.

However the design, implementation and maintenance of correct distributed software is still a very difficult exercise, and people in both research laboratories and in software companies agree about the necessity of validating the software system at different stages of its life cycle, with as few "model-ruptures" as possible. To meet the challenge of a continuously validated development process for open object-based distributed systems, the following two problems have to be solved:

- an OO approach must correctly deal with parallelism and distribution issues, without hindering good implementation performances,
- formal validation tools that can be used continuously during the OO life cycle of the distributed software must be made available.

As B. Meyer writes in [17],

> *To judge by the looks of the two parties, the marriage between concurrent computation and object-oriented programming appears an easy enough affair to arrange. This appearance is deceptive: the problem is a hard one.*

The introduction of parallelism in an object-oriented language may be considered with two perspectives, whether it is loosely or tightly coupled with the rest of the language. In the first case, it is more or less orthogonal to the structuration brought in by the object-oriented paradigm: it basically relies on the model

of communicating sequential processes with one or more ad hoc communication semantics (e.g., the various CORBA bindings existing for most object-oriented and object-based languages). This simple parallelism model does not melt well with inheritance, however, because synchronization constraints are hard to inherit in a context in which subtype substitutability is to be preserved. This problem is known as the *inheritance anomaly* [14] and is not easy to circumvent [15].

Another approach would be to tightly integrate a given parallelism semantics within a language as in Parallel Eiffel [17], where there is no distinction between "active" and "passive" objects. However, Parallel Eiffel relies on the availability of a (distributed) multi rendez-vous communication scheme, which is powerful enough to foster simple, concise, and elegant solutions for classic concurrency problems (such as the *dining philosophers*), but which limits drastically the domains of interest of this method.

Building on the intuition that a universal language taking into account all the possible semantics aspects of parallelism and communication is a holy grail, we propose to rely on an open (but simpler) language to build dedicated *frameworks*. Such a framework is specialized toward a class of distributed applications, and integrates formal validation tools.

It is common sense to remark that formal validation needs formal descriptions. But, as examplified by the fuzzy semantics of most popular methods like Booch [3] or OMT [19] (Object Modeling Technique), it is a matter of facts that most people (outside the academic world) don't bother to be fully formal in early stages of analysis. Trying to understand why this is so falls out of the scope of this paper. Let us just note that when a method is a bit more formal (e.g., Fusion [5]), some aspects of a real system become very hard (and thus very costly) to model formaly [6].

Still we cannot do formal validation of a distributed system without a formal description of its communication structure. Our approach is thus to attack the problem at the output of the analysis stage, where we use an OO language that is both abstract and formal enough to let us specify crucial parts precisely (thus allowing the specifications to be validated), and efficient enough for a direct implementation (thus providing the continuous framework allowing the validation of the implementation). Our favorite language is Eiffel [16], which is a pure object-oriented language featuring multiple inheritance, polymorphism, static typing and dynamic binding, genericity, garbage collection, a disciplined exception mechanism, and systematic use of assertions to improve software correctness in the context of *programming by contract*. The semantics of Eiffel is defined formally [1].

The presentation will be organized as follows. First, we will argue for an openness allowing the formal descriptions of distributed systems to be validated at the various steps of the software development, even as far as the maintenance phase. As a first step in this direction, we then use the famous alternating bit protocol example to illustrate how such an openness helps in pushing FDTs part way into the implementation phase. Finally, we will discuss how to actually integrate validation in a seamless OO life-cycle.

2 Validating Open Distributed Software with Formal Description Techniques

2.1 A set of complementary techniques

Validation techniques vary widely in their forms and their abilities, but they always need a formal description[1] 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, but existing methods can only easily be applied to analyzing simplified models of the considered problem. This forces the distributed algorithm to be described at a high abstraction level, so its formal verification lets the problem of property preservation during its refinement course widely open.

- *protocol simulation*, using a simulated (and centralized) environment: it can deal with more refined models of the problem and can efficiently detect errors on a subset of the possible system behaviors. The main difficulty is to formally describe and simulate the execution environment. This is generally very simplified, because it would not be realistic (nor interesting) to take into account all the parameters of a real system, as for example, the exact influence of message size on transmission delays, or the action durations (which are not computable without execution).

- *observation and test* of an implementation: here, the execution environment is a real one. But as there is a lack of tools to observe a distributed system

[1] Note that an executable program *is* a formal description.

as a whole, it will be difficult to actually validate the software. It will also be difficult to generalize the possible behaviors from the observation.

It appears that these approaches are more complementary than in competition, and that an advised project manager would try to use all of them.

However this is hard in practice because the formalisms used in these various stages differ widely. Until now, there is a need for a formal specification to validate something against. Most of these techniques have been developed in the context of the Formal Description Techniques (FDTs) for protocols.

2.2 Difficulties in using FDTs

It is very deceptive to see that formal validation based on standard FDTs (such as SDL, Estelle and Lotos) never acceded to a widespread use in the industry, despite excellent results on most of the pilot projects where it has been used [13]. In our experience, this is mainly due to the lack of integration of this promising technology in widely used software development methods and life-cycles.

Because of the standard FDTs lack of 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 open distributed systems: 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 laboriously 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 layers are difficult to circumvent for instance).

Using FDTs validation technology thus imposes a model rupture in the usual life-cycle. This implies that formal validation technology may be used during the maintenance phase of a system only after a costly reverse engineering effort. 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.

2.3 Alternative: Integrate Validation in an 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. The first

step toward an object-oriented analysis is concerned with devising a precise, relevant, concise, understandable, and correct model of the real world. The purpose of object-oriented analysis is to model the problem domain so that it can be understood and serve as a stable basis in preparing the design step. The *design* phase starts with the output of the analysis phase and gradually shifts its emphasis from the application domain to the computation domain: the implementation strategy is defined, and trade-offs are made accordingly. Auxiliary classes may be introduced at this stage to deal with complex relationships or implementation-related matters. The output of the object-oriented design phase is a blueprint for the implementation in an object-oriented language, which is basically an extension of the design process.

The boundaries between analysis, design and implementation are not rigid. This seamlessness of the object-oriented approach may upset the old-time programmers who favor the well-established structured methods that feature strong frontiers between phases. A reality check might be necessary here: how often does a final product match its initial requirements? What is the situation 5 or 10 years later? [2]

We advocate for extending this seamless OO development process to encompass validation, not as a post facto task (as promoted in the classical vision of the V-model of the life-cycle), but as an *integrated* activity *within* the OO development process. 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. We illustrate our vision of an integrated validation process on a classical example in the next section.

3 An Example: the Alternating Bit Protocol

3.1 Introduction to the Alternating Bit Protocol

We could have chosen a multimedia application full of bells and whistles to illustrate our approach. Since it would only have made the discussion more obfuscated, we come back to the famous example of the Alternating Bit Protocol, because it served as a common *cas d'école* at the time of the foundation of the protocol engineering community fifthteen years ago. It is thus well-known among the FDT practitioners, and it will be easy to compare our modelization with previous solutions.

We consider a system architecture made of several modules connected through interaction points. We distinguish the SENDER and RECEIVER modules, interacting through unreliable communication media (messages can be lost or corrupted). The

[2] See B. W. Boehm and W. Humphrey's works [2, 9] for more thoughts on this topic.

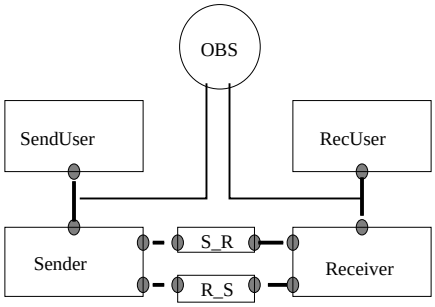

Figure 1: Architecture of the Alternating Bit Protocol

```
body sender_body for Sender;
   const retran_time = 1;
   var data_buffer : data_segment; bit_buffer : tbit;
   state idle, wait_ack;
   initialize to idle begin bit_buffer := 0 end;
   trans
     from idle to wait_ack when lsap.data_req(data) name data_request:
           begin data_buffer := data; output psap.mess(data, bit_buffer) end;
     from wait_ack when psap.ack(bit)
         provided (bit_buffer = bit) to idle name confirm_indication:
            begin alternate_bit; output lsap.confirm end;
         provided (bit_buffer <>bit) name retransmit:
            begin output psap.mess(data_buffer, bit_buffer) end;
     from wait_ack delay (retran_time) name timeout:
            begin output psap.mess(data_buffer, bit_buffer) end;
end;
```

Figure 2: An Estelle description of the Sender body for the Alternating Bit Protocol

protocol users are represented by two other distinct modules (SENDUSER and RECUSER). We have also considered that the system behavior is tested with respect to a global property of sequencing service primitives (Data Request, Data Indication and Data Confirm). In a validation system such as Veda [12], this is modeled by the presence of an observer having probes on the protocol modules.

The architecture of this system and the description of the sender module written in the Estelle FDT are given in figures 1 and 2. It is assumed that the reader is familiar with such descriptions.

3.2 The OMT analysis

To formally describe the architecture of our Alternating Bit Protocol system, we choose to use the OMT method [19]. The OMT analysis model is made of three components:

- the object model (based on classes, associations, and grouping constructs) shows the static structure of the real-world system through abstract or physical classes and their relationships.
- the dynamic model (based on events and states) shows the temporal behavior of the objects in the system.
- the functional model shows the constraints between the objects in the system (and between inputs and outputs).

Figure 3 represents the OMT object diagram of our system. Protocol entities (each layer) appear on the left side of the figure, and protocol data units (the events) on the right side (greyed boxes correspond to classes independent of the Alternating Bit Protocol).

Classes are represented as rectangles labeled with the class name. A single line represents simple relation between two classes (like data sending between SENDUSER and SENDER). Relation between more than two classes is marked by a diamond. A triangle symbolizes the inheritance relationship (for example SENDUSER inherits from PROTOCOLENTITY). A black circle at the end of a relation line marks the multiplicity of the relationship.

Using the graphical syntax of the OMT dynamic model, figure 4 shows us the automaton associated with the Sender protocol entity, which corresponds to the Estelle code of figure 2. The OMT functional model is omitted here because it is mostly irrelevant to our discussion.

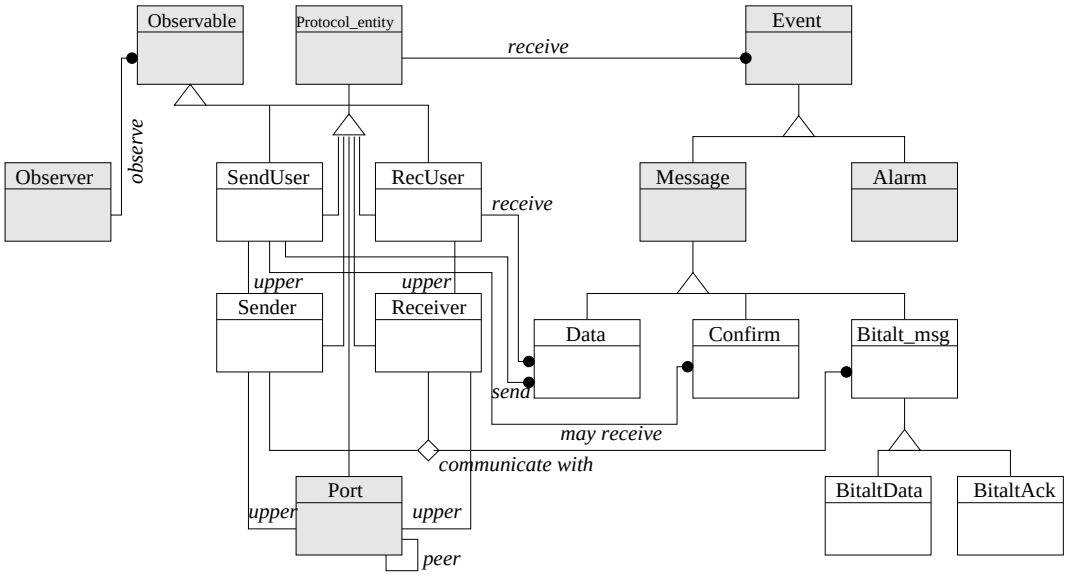

Figure 3: The OMT Object Model of the Alternating Bit Protocol

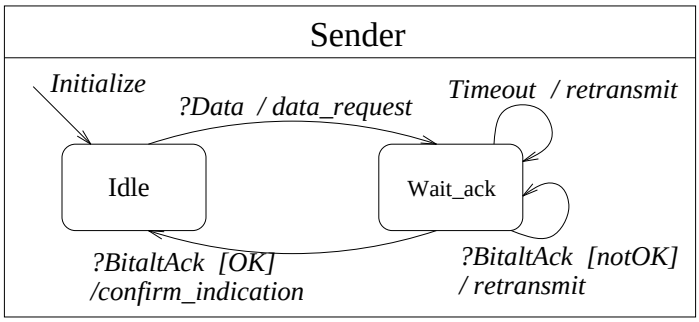

Figure 4: The OMT Dynamic model of the Sender Object

3.3 A Design Framework Integrating Validation Technology

A framework consists of a collection of classes 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. As abstractions, patterns often cut across other common software abstractions like procedures and objects, or combine more common abstractions in powerful ways [4]. The term *pattern* applies both to the thing (e.g., a collection class and its associated iterator) and the directions for making a thing. In this sense, software patterns can be likened to a dress pattern: the general shape is in the pattern itself, though each pattern must be tailored to its context.

We describe here the example of the VLOOP framework (Validation Library for Object Oriented Protocols), whose purpose is to allow protocol designers to keep the underlying model provided in a classic FDT such as Estelle and to use related formal validation technology, while benefiting from the OO seamless life-cycle and openness. For example, although Estelle communications are only possible using FIFO, the flexibility of the OO framework allows us to offer many more possibilities for communication semantics: e.g., communication by procedure call between adjacent layers of a protocol stack.

As an Estelle *module* is naturally associated to each layer of the Alternating Bit Protocol system, a class can be associated to each layer. Moreover classes and Estelle *modules* use the common notions of dynamic creation and deletion of instances (module variables in Estelle), and dynamic connection to other objects/modules. Variables of an Estelle *module* correspond to class instance variables (attributes). Transitions between states can be represented as object methods, driven by a protocol engine (see below). An Estelle *module* state is represented as an instance variable which can take as many predefined values as there are states in the automaton.

The passage to the OO technology adds the possibility to structure a system through inheritance. Thus, similar characteristics of classes are abstracted away and are grouped in a super class. The layered structure of our system allows us to illustrate this phenomena. Since each layer (PORT, SENDER, SENDUSER, etc.) may have a lower layer and an upper layer and be able to send informations to these layers, this common behavior can be factorized into the abstract class PROTOCOLENTITY (see example 3.1). Some actions, having a specific behavior depending on the actual layer level, are not described in the class PROTOCOLENTITY but in each subclass corresponding to a different layer level (for example the method *receive* in Example 3.1).

Example 3.1

```
indexing
   description: "Abstract notion of a Protocol Entity"
deferred class PROTOCOL_ENTITY
feature
   initialize is                                                              5
         -- Setup the initial step of this Protocol Entity
   lower_layer, upper_layer : PROTOCOL_ENTITY
         -- protocol layers under and above the current one
   connect_lower_layer(new_lower : PROTOCOL_ENTITY) is
      -- set a new protocol entity under the current one                     10
      ensure lower_layer = new_lower
   connect_upper_layer(new_upper : PROTOCOL_ENTITY) is
      -- set a new protocol entity above the current one
      ensure upper_layer = new_upper
   insert_on_top_of(other : PROTOCOL_ENTITY) is                               15
      -- insert this protocol entity on top of the other one
      require not_void: other /= Void
      ensure
         new_lower_is_other: lower_layer = other
         new_upper_is_old_other_upper: upper_layer = old (other.upper_layer)  20
   send_down (m: MESSAGE) is
         -- send the message to the lower layer
      require low_connected: lower_layer /= Void; m_not_void: m /= Void
   send_up (m: MESSAGE) is
         -- send the message to the upper layer                              25
      require up_connected: upper_layer /= Void; m_not_void: m /= Void
   receive (e : EVENT) is
         -- process an input event
      require e_not_void: e /= Void
      deferred -- heirs must provide an implementation                        30
invariant
   symmetric: upper_layer /= Void implies (upper_layer.lower_layer = Current)
end -- PROTOCOL_ENTITY
```

Example 3.2

```
indexing
   description: "The abstract specification of sender protocol entity"
deferred class SENDER_SPEC
inherit
   PROTOCOL_ENTITY                                                          5
      undefine initialize end
feature
   idle, wait_ack : INTEGER is unique
   state : INTEGER
   altbit : BOOLEAN                                                        10
   initialize is
      -- initialize the sender state
      deferred -- no implementation provided
      ensure then idle: state = idle -- initialize must lead to idle state
      end -- initialize                                                    15
   data_request(new_data : USER_DATA) is
      -- transmit a new user data
      require idle: state = idle
      deferred -- no implementation provided
      ensure wait_ack: state = wait_ack                                    20
      end -- data_request
   confirm_indication (ack : BITALT_ACK) is
      -- transmit a confirmation SDU
      require wait_ack: state = wait_ack; reception_ok: ack.altbit = altbit
      deferred -- no implementation provided                               25
      ensure idle: state = idle ; alternated_bit: altbit /= old (altbit)
      end -- confirm_indication
    retransmit (ack : BITALT_ACK) is
      -- retransmission of the buffered data
      require wait_ack: state = wait_ack; reception_not_ok: ack.altbit /= altbit  30
      deferred -- no implementation provided
      ensure same_state: state = old (state)
      end -- retransmit
end -- SENDER_SPEC
```

In this framework, the transitions of the OMT dynamic model (corresponding to the Estelle automaton) are translated to methods of an OO language according to the following design rules:

- the triggering event of the transition (if it exists) is transformed in a method parameter,
- the starting state, plus optional conditions on the event parameters or other conditions on local variables, are specified in the precondition (in Eiffel, this consists in a set of assertions introduced with the keyword *require*),
- the arrival state is specified in the postcondition (assertions following the keyword *ensure*),
- the method body can be either implemented (in this case, it is similar to an Estelle transition body) or left abstract (specified in Eiffel with the keyword *deferred*, equivalent to a pure virtual function in C++).

After all the specifications of the features, we can add an invariant clause (keyword *invariant*) which must always be satisfied (except during the invocation of a feature). Figure 3.1 shows us the specification for the class PROTOCOLENTITY. For each feature, only the precondition and the postcondition are displayed.

In our system example, the *receive* method, invocated by extern objects, is the engine of the protocol layer (objects derived from class PROTOCOLENTITY). It processes actions depending on both the received event type and the state in which the object is when the method is invocated. The implementation of such a method needs a double dispatch operation that has several well-known implementation methods [8].

From this design framework and the OMT analysis diagrams, it is possible to generate the specification of PROTOCOLENTITY subclasses. Example 3.2 shows the definition of class SENDERSPEC, which is the class describing the specification of the Sender layer of the Alternate Bit Protocol.

4 Validation in the VLOOP Framework

Within this framework, a validation process may be carried on a seamless way. The idea is to validate successive refinements of the system according to a set of abstract properties, until the actual implementation has been obtained.

4.1 Specializing the VLOOP framework for the Alternating Bit Protocol

First, we have to design a specialized version of observer (called BITALTOBSERVER, see Example 4.1) implementing the test of the global property introduced in Section 3.1.

Example 4.1

```
indexing
   description: "To observe properties on the Alternate Bit Protocol"
class BITALT_OBSERVER
inherit
   OBSERVER                                                                    5
creation
   make
feature
   attente, message_en_transit, message_transmis : INTEGER is unique
   state : INTEGER                                                            10
   make is
      -- initialize the observer
      do
         state := attente
      ensure state = attente                                                  15
      end -- make
   data_sent is
         -- data has been sent by the sender
      require attente: state = attente
      do                                                                      20
         state := message_en_transit
         print("sender: data_sent%N")
      end -- data_sent
   data_received is
         -- data has been received by the receiver                           25
      require message_en_transit: state = message_en_transit
      do
         state := message_transmis
         print("receiver: data_received%N")
      end -- data_received                                                    30
   confirm_received is
         -- a confirmation has been received by the sender
      require message_transmis: state = message_transmis
         do
            state := attente                                                  35
            print("sender: confirm_received%N")
         end -- confirm_received
end -- BITALT_OBSERVER
```

Then we have to provide some implementation details to the protocol entity specifications obtained as output of the OMT analysis stage. For example, the class Sender (see Example 4.2) is a suitable implementation of the Alternating Bit Protocol Sender specification presented as Example 3.2. Note that the Eiffel language constrains the methods of a subclass (here SENDER) to respect the preconditions, postconditions and invariants of their specifications in the ancestor class (here SENDERSPEC).

At this stage, we still have to provide the system with a suitable environment, which is basically made of a set of drivers and stub classes modeling the upper and lower layers of the Sender and Receiver modules. Stub classes are easy enough to derive from their OMT specification, whereas we use the VLOOP notion of activable objects to implement the drivers (i.e. the traffic generator and the network interfaces). An activable object (e.g., a SENDINGUSER in Example 4.3) is just an heir of the abstract class ACTIVABLE, which features an entry point called *action*[3] that may be called from time to time by, e.g., a scheduler, provided the method *activable* returns true. The network interfaces (modeled through the class PORT) come in several flavors (that is, subclasses) in the VLOOP library. This is to model both perfect media and subclasses loosing or corrupting data.

4.2 Applying various 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.

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.

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 (see Example 4.4 taken from the VLOOP library) 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 [12]. An observer is a program which permits to catch and analyze informations about execution. It can see every interactions exchanged in the system, and also every

[3] This abstraction would be modeled with a spontaneous transition in an Estelle description.

Example 4.2

```
indexing
   description: "A sender protocol entity implementing the SENDER_SPEC"
class SENDER
inherit
   SENDER_SPEC                                                              5
creation
   initialize
feature
   initialize is
      -- initialize the sender state                                        10
      do
         state := idle
      end -- initialize
   data_request(new_data : USER_DATA) is
      -- transmit a new user data                                          15
      local pdu : BITALT_MESS
      do
         data := new_data -- buffer the data in case of retransmission
         !!pdu.make(altbit,data); send_down(pdu)
         state := wait_ack                                                  20
      end -- data_request
   confirm_indication (ack : BITALT_ACK) is
      -- transmit a confirmation SDU
      local conf : CONFIRM
      do                                                                    25
         !!conf; send_up(conf)
         state := idle; altbit := not altbit -- alternate bit
      end -- confirm_indication
    retransmit (ack : BITALT_ACK) is
      -- retransmission of the buffered data                               30
      local pdu : BITALT_MESS
      do
         !!pdu.make(altbit,data); send_down(pdu)
      end -- retransmit
feature -- on event reception                                               35
   receive (e : EVENT) is
         -- process an input event
      do -- body omitted for the sake of brievity
      end -- receive
feature {NONE}                                                              40
   data : expanded USER_DATA -- sending buffer used in case of retransmission
end -- SENDER
```

Example 4.3

```
indexing
   description: "A protocol entity sending USER_DATA and waiting for CONFIRM"
class SENDING_USER
inherit
   PROTOCOL_ENTITY                                                          5
   ACTIVABLE
   OBSERVABLE
      redefine my_observer end
creation
   initialize                                                              10
feature
   activable : BOOLEAN
      -- are we ready to send data?
   action is
      -- send a USER_DATA                                                  15
      local data : USER_DATA
      do
         !!data; send_down(data)
         activable := False
         debug("OBSERVATION") my_observer.data_sent end -- debug           20
      ensure then waiting_confirm: not activable
      end -- action
   receive_confirmation (sdu : CONFIRM) is
      -- receive a confirmation SDU
      require waiting_confirm: not activable                               25
      do
         activable := True
         debug("OBSERVATION") my_observer.confirm_received end -- debug
      ensure activable: activable
      end -- receive_confirmation                                          30
feature -- on event reception
   receive (e : EVENT) is
         -- process an input event
      local confirm_sdu : CONFIRM
      do                                                                   35
         confirm_sdu ?= e; receive_confirmation(confirm_sdu)
      end -- receive
feature {OBSERVER}
   my_observer : BITALT_OBSERVER
end -- SENDING_USER                                                        40
```

Example 4.4

```
class SCHEDULER
creation
   make
feature
   make is                                                              5
      -- Initialization
      do
         !!process_table.make(1,0)
      end -- make
   register (new_process: ACTIVABLE) is                                 10
         -- register a new process to manage
      require not_void: new_process /= Void
      do
         process_table.force(new_process,
                                 process_table.count+1)                 15
      end -- register
   random_run(cycles, seed : INTEGER) is
         -- Randomly call registered processes
      local
         n, p : INTEGER                                                 20
         random_generator : RANDOM
      do
         !!random_generator.set_seed(seed)
         from n := 1; random_generator.start
         until n > cycles                                               25
         loop
            p := (random_generator.item
                  \\ process_table.count) + 1
            if process_table.item(p).activable then
               process_table.item(p).action                             30
            end -- if
            random_generator.forth
            n := n + 1
         end -- loop
      end -- random_run                                                 35
feature {NONE}
   process_table : ARRAY[ACTIVABLE]
invariant
end -- SCHEDULER
```

internal states of a module. In our example, the class SENDINGUSER is observable and its observer is from the class BITALTOBSERVER. SENDINGUSER (example 4.3), inheriting from the class OBSERVABLE, may invoke features from its observer after each action we want to observe. An object from the class BITALTOBSERVER (example 4.1) can take three states: *waiting*, *transiting_message* and *transmited_message*. The reception of a message (invocation of feature) from the class SENDINGUSER makes it pass from a state to another, and display indications about the evolution of the system. If a message m is received when the observer is in a state x and no m-transition is available from the state x, this expresses a system fail (for example if it receives a *data_sent* message when it is in the state *transiting_message*).

A protocol sequencing error is thus detected as a precondition violation on the observer. The Eiffel execution environment then allows the user to precisely locate and delimit the responsibility of the error.

For more abstract or complicated properties to be checked on real systems (e.g., that a service behaves like a FIFO and it is live), the observer object could be derived automatically from higher level specification languages (e.g., temporal logic specifications). The Eiffel runtime system would then only serve to trap and identify the error.

4.3 Towards the Implementation

Going from the validated system to the implementation just involves replacing the simulation scheduler with another one specialized for an efficient implementation, and the Port class with an heir implementing the actual network interface. The service offered by the classes SENDER and RECEIVER may then be made available to real users in the VLOOP framework, which has transparently been transformed to an efficient implementation environment featuring among others procedure calls between layers without copying of data, and up-calls on the arrival of messages.

The code that is used in the implementation (SENDER and RECEIVER classes) could thus be actually validated, helping towards the goal of a seamless software development life-cycle producing validated software.

5 Conclusion

We have shown the interest and feasibility of integrating formal validation techniques in an established OO life-cycle for the construction of open distributed software systems. We choose the OMT method, but our approach would have been similar

with other popular OO methods. On the well known Alternating Bit Protocol toy example, 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. But this approach is not limited to toy problems: it has also been used on real systems, e.g., the implementation of a parallel SMDS server where it has allowed us to detect non trivial problems at early stages of the life-cycle [7].

Future work will concentrate on the design of a comprehensive OO library to help the construction of VLOOP like integrated validation frameworks for OO distributed software systems (probably in the context of CORBA), and the interfacing of the framework with open validation tools such as the CADP environment [10].

References

[1] Oudshoorn M. Attali I., Caromel D. – A formal definition of the dynamic semantics of the eiffel language. – In *Proc. of the Sixteenth Australian Computer Science Conference (ACSC-16)*. Brisbane, Australia, 1993.

[2] B. W. Boehm. – The high cost of software. – In Ellis Horowitz, editor, *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, 1975.

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

[4] J.O. Coplien. – Generative pattern languages: An emerging direction of software design. – *C++ Report*, 6(6), July-August 1994.

[5] Derek Coleman et. al. – *Object-Oriented Development - The Fusion Method.* – Prentice-Hall Object-Oriented Series, 1994.

[6] P.A. Etique, J.P. Hubaux, and T. Saydam. – Vérification et validation de services de télécommunications spécifiés par une méthode orientée objets. – In *Colloque Francophone pour l'Ingénierie des Protocoles, CFIP'95, Hermès.*, pages 469–481, June 1995.

[7] F. Guerber, J.-M. Jézéquel, and F. André. – Conception et implantation d'un serveur SMDS sur architectures modulaires. – Technical Report 885, IRISA, Novembre 1994.

[8] F. Guidec. – *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire.* – Thèse de doctorat, IFSIC / Université de Rennes 1, juin 1995.

[9] Watts Humphrey. – *Managing the Software Process.* – Addison Wesley, 1989.

[10] L. Mounier A. Rasse C. Rodriguez J.-C. Fernandez, H. Garavel 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.

[11] M.A. Jackson. – *System Development.* – Prentice-Hall International, Series in Computer Science, 1985.

[12] C. Jard, R. Groz, and J.F. Monin. – Development of VEDA: a prototyping tool for distributed algorithms. – In *IEEE Trans. on Software Engin.*, volume 14, pages 339–352, March 1988.

[13] 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.

[14] Satoshi Matsuoka and Akinori Yonezawa. – Analysis of inheritance anomaly in object-oriented concurrent programming languages. – In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming.* MIT Press, 1993.

[15] José Meseguer. – Solving the inheritance anomaly in concurrent object-oriented programming. – In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 220–246, Kaiserslautern, Germany, July 1993. Springer-Verlag.

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

[17] B. Meyer. – Systematic concurrent object-oriented programming. – *Communications of the ACM*, 36(9), September 1993.

[18] 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.

[19] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. – *Object-Oriented Modeling and Design.* – Prentice Hall, New Jersey, 1991.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399