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

*C. Jard**, *J.-M. Jézéquel†*, *L. Nédelka‡*

*I.R.I.S.A. Campus de Beaulieu
F-35042 RENNES CEDEX, FRANCE*

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.

Keywords

Object-Oriented modeling, validation and verification, protocols, software engineering, Alternating-Bit-Protocol

*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

1 INTRODUCTION

It is now widely admitted (Jackson, 1985) 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 (Monarchi, 1992). 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 (Meyer, 1993),

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* (Matsuoka, 1993) and is not easy to circumvent (Meseguer, 1993).

Another approach would be to tightly integrate a given parallelism semantics within a language as in Parallel Eiffel (Meyer, 1993), 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 exemplified by the fuzzy semantics of most popular methods like Booch (Booch, 1994) or OMT (Rumbaugh, 1991) (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 (Coleman, 1994)), some aspects of a real system become very hard (and thus very costly) to model formally (Etique, 1995).

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 (Meyer, 1992), 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 (Attali, 1993).

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 description¹ of the distributed software system. They output data on properties of the system under consideration that can be viewed with some confidence level. Basically,

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

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 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 (Jézéquel, 1990). 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 fifteen years ago. It is thus well-known among the FDT practitioners, and it will be easy to compare our modelization with previous solutions.

²See B. W. Boehm and W. Humphrey's works (Boehm, 1975)(Humphrey, 1989) for more thoughts on this topic.

Figure 1: Architecture of the Alternating Bit Protocol.

```

body sender_body for Sender;
  const retrans_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 (retrans_time) name timeout:
      begin output psap.mess(data_buffer, bit_buffer) end;
  end;
end;

```

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

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 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 (Jard, 1988), 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 (Rumbaugh, 1991). 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: The OMT Object Model of the Alternating Bit Protocol.

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 PROTOCOL_ENTITY). 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 4: The OMT Dynamic model of the Sender Object.

figure 2. The OMT functional model is omitted here because it is mostly irrelevant to our discussion.

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 (Coplien, 1994). 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 in the OSI style"
deferred class PROTOCOL_ENTITY
feature
  initialize is
    -- Setup the initial state
    lower_layer, upper_layer: PROTOCOL_ENTITY
    -- lower & upper protocol layers
    connect_lower_layer(new_lower: like lower_layer) is
      -- set a new lower_layer
      ensure lower_layer = new_lower
    connect_upper_layer(new_upper: like upper_layer) is
      -- set a new upper_layer
      ensure upper_layer = new_upper
    insert_on_top_of(other: PROTOCOL_ENTITY) is
      -- self insert on top of the other
      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)
    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
      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 implementation
invariant
  symmetric: upper_layer /= Void implies
    (upper_layer.lower_layer = Current)
end -- PROTOCOL_ENTITY

```

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, invoked by extern objects, is the engine of the protocol layer (objects derived from class `PROTOCOLENTITY`). It processes actions

Example 3.2

```

indexing
  description: "Abstract specification of
               a sender protocol entity"
deferred class SENDER_SPEC
inherit
  PROTOCOL_ENTITY
feature
  retrans_time : DOUBLE is 1.0
  idle, wait_ack : INTEGER is unique
  state : INTEGER; watch_dog: timer
  altbit : BOOLEAN
  initialize is
    -- initialize the sender state
    deferred -- no implementation provided
    ensure then -- initialize leads to idle state
      idle: state = idle
    end -- initialize
  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
    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
    ensure
      idle: state = idle
      alternated_bit: altbit /= old (altbit)
    end -- confirm_indication
  retransmit (ack : BITALT_ACK) is
    -- retransmission in case of a NACK
    require
      wait_ack: state = wait_ack
      reception_not_ok: ack.altbit /= altbit
    deferred -- no implementation provided
    ensure same state: state = old (state)
    end -- retransmit
  timeout is
    -- retransmission in case of a timeout
    require
      wait_ack: state = wait_ack
      ringing: watch_dog.is_ringing
    deferred -- no implementation provided
    ensure same state: state = old (state)
    end -- timeout
invariant
  watch_dog_active_when_waiting_ack:
    watch_dog.is_running = (state = wait_ack)
end -- SENDER_SPEC

```

depending on both the received event type and the state in which the object is when the method is invoked. The implementation of such a method needs a double dispatch operation that has several well-known implementation methods (Guidéc, 1995).

From this design framework and the OMT analysis diagrams, it is possible to generate the specification of PROTOLENTITY subclasses. Example 3.2 shows the definition of class SENDER_SPEC, 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 BITALT_OBSERVER, see Example 4.1) implementing the test of the global property introduced in Section 3.1.

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 4.1

```

indexing
  description: "To observe properties on
               the Alternate Bit Protocol"
class BITALT_OBSERVER
inherit
  OBSERVER
creation
  make
feature
  waiting, transiting_message,
  transmitted_message : INTEGER is unique
  state : INTEGER
  make is
    -- initialize the observer
    do
      state := waiting
    ensure state = waiting
    end -- make
  data_sent is
    -- data has been sent by the sender
    require
      waiting: state = waiting
    do
      state := transiting_message
      print("sender: data_sent%N")
    end -- data_sent
  data_received is
    -- data has been received by the receiver
    require
      transiting: state = transiting_message
    do
      state := transmitted_message
      print("receiver: data_received%N")
    end -- data_received
  confirm_received is
    -- confirmation received by the sender
    require
      transmitted: state = transmitted_message
    do
      state := waiting
      print("sender: confirm_received%N")
    end -- confirm_received
end -- BITALT_OBSERVER

```

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

³This 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
creation
  initialize
feature
  initialize is
    -- initialize the sender state
    do
      state := idle
    end -- initialize
  data_request(new_data : USER_DATA) is
    -- transmit a new user data
    local pdu : BITALT_MESS
    do
      data := new_data -- buffer the data
      !!pdu.make(altbit,data)
      send_down(pdu)
      watch_dog.set(retran_time)
      state := wait_ack
    end -- data_request
  confirm_indication(ack : BITALT_ACK) is
    -- transmit a confirmation SDU
    local conf : CONFIRM
    do
      !!conf; send_up(conf)

      watch_dog.reset
      state := idle
      altbit := not altbit -- alternate bit
    end -- confirm_indication
  retransmit(ack : BITALT_ACK) is
    -- retransmission of the buffered data
    local pdu : BITALT_MESS
    do
      !!pdu.make(altbit,data)
      send_down(pdu)
      watch_dog.set(retran_time)
    end -- retransmit
  timeout is
    -- retransmission in case of a timeout
    local pdu : BITALT_MESS
    do
      !!pdu.make(altbit,data)
      send_down(pdu)
      watch_dog.set(retran_time)
    end -- timeout
  receive(e : EVENT) is
    -- process an input event
    do -- body omitted for the sake of brevity
    end -- receive
feature {NONE}
  data : expanded USER_DATA
    -- buffer used for retransmissions
end -- SENDER

```

Example 4.3

```

indexing
  description: "A protocol entity sending
    USER_DATA and waiting for CONFIRM"
class SENDING_USER
inherit
  PROTOCOL_ENTITY
  ACTIVABLE
  OBSERVABLE
  redefine my_observer end
creation
  initialize
feature
  activable : BOOLEAN
    -- are we ready to send data?
  action is
    -- send a USER_DATA
    local data : USER_DATA
    do
      !!data; send_down(data)
      activable := False
      debug("OBSERVATION")
      my_observer.data_sent
    end -- debug
    ensure then

    waiting_confirm: not activable
    end -- action
  receive_confirmation(sdu : CONFIRM) is
    -- receive a confirmation SDU
    require waiting_confirm: not activable
    do
      activable := True
      debug("OBSERVATION")
      my_observer.confirm_received
    end -- debug
    ensure activable: activable
    end -- receive_confirmation
feature -- on event reception
  receive(e : EVENT) is
    -- process an input event
    local confirm_sdu : CONFIRM
    do
      confirm_sdu ?= e
      receive_confirmation(confirm_sdu)
    end -- receive
feature {OBSERVER}
  my_observer : BITALT_OBSERVER
end -- SENDING_USER

```

its initial state and then constructing its reachability graph by exploring all the possible paths allowed by activable transitions.

Example 4.4

```

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

```

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 (Jard, 1988). 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 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 *transmitted_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 (Guerber, 1994).

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 (Fernandez, 1992).

6 REFERENCES

- Attali I., Caromel D., Oudshoorn M. (1993) A Formal Definition of the Dynamic Semantics of the Eiffel Language. *In: Proc. of the Sixteenth Australian Computer Science Conference (ACSC-16)*. Brisbane, Australia.
- Boehm, B. W. (1975) The High Cost of Software. *In: Horowitz, Ellis (ed), Practical Strategies for Developing Large Software Systems*. Addison-Wesley.
- Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*. 2nd edn. Benjamin Cummings.
- Coplien, J.O. (1994) Generative Pattern Languages: An emerging direction of software design. *C++ Report*, 6(6).
- Coleman, D. et al. (1994) *Object-Oriented Development - The Fusion Method*. Prentice-Hall Object-Oriented Series.

- Etique, P.A., Hubaux, J.P., and Saydam, T. (1995 June) Vérification et validation de services de télécommunications spécifiés par une méthode orientée objets. *Pages 469–481 of: Colloque Francophone pour l'Ingénierie des Protocoles, CFIP'95, Hermès.*
- Fernandez, J.-C., Garavel, H., Mounier, L., Rasse, A., Rodriguez, C., and Sifakis, J. (1992 May) A Toolbox for the Verification of Programs. *Pages 246–259 of: International Conference on Software Engineering, ICSE'14, Melbourne, Australia.*
- Guerber, F., Jézéquel, J.-M., and André, F. (1994 November) *Conception et implantation d'un serveur SMDS sur architectures modulaires.* Tech. rept. 885. IRISA.
- Guidéc, F. (1995 June) *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.
- Humphrey, W. (1989) *Managing the Software Process.* Addison Wesley.
- Jackson, M.A. (1985) *System Development.* Prentice-Hall International, Series in Computer Science.
- Jard, C., Groz, R., and Monin, J.F. (1988 March). Development of VEDA: a prototyping tool for distributed algorithms. *Pages 339–352 of: IEEE Trans. on Software Engin., vol. 14.*
- Jézéquel, J.-M. (1990 November) Experience in Validating Protocol Integration Using Estelle. *In: Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain.*
- Matsuoka, S. and Yonezawa, A. (1993) Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. *In: Agha, G., Wegner, P. and Yonezawa, A. (eds), Research Directions in Concurrent Object Oriented Programming.* MIT Press.
- Meseguer, J. (1993) Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. *Pages 220–246 of: Nierstrasz, O. (ed), Proceedings ECOOP'93. LNCS 707. Kaiserslautern, Germany: Springer-Verlag.*
- Meyer, B. (1992) *Eiffel: The Language.* Prentice-Hall.
- Meyer, B. (1993) Systematic Concurrent Object-Oriented Programming. *Communications of the ACM, 36(9).*
- Monarchi, D. E., and Puhr, G. I. (1992) A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM, 9(35), 35–47.*
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design.* New Jersey: Prentice Hall.

7 BIOGRAPHY

Claude JARD is graduated from the Ecole Nationale Supérieure des Télécommunications de Bretagne in 1981. He received the Doctorat degree in engineering from the University of Rennes, France, in 1984, and the Habilitation in 1994. He is now a Research Supervisor in the Centre National de la Recherche Scientifique. From 1982 to 1986, he led the “Protocol Specification and Validation” team in the Centre National d’Etudes des Télécommunications. Since 1986, he has been working at the IRISA, Rennes, on the design, verification, and implementation of distributed algorithms. He is now at the head of the Pampa team, on programming environments for distributed computers. He has written numerous articles on these subjects, notably on dynamic verification of distributed systems, and teaches postgraduate students in universities and schools of engineering. He also launched development projects like Veda, Echidna or TGV, which have been used by industries.

Jean-Marc JÉZÉQUEL received an engineering degree in Telecommunications from the ENSTB in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989. He is a research manager in the Irisa Lab for the Centre National de la Recherche Scientifique. His research interests include software engineering and object oriented technology for telecommunications and distributed computers. He is the author of the book *Object Oriented Software Engineering with Eiffel*, Addison-Wesley, Feb. 1996.

Laurence NÉDELKA received a DEA (Diplome d’Études Approfondies) in computer science from university of Rennes I in 1995. She is actually preparing a Ph.D. degree at Irisa in the Pampa project. Her research interests concern utilisation of formal methods especially to do verification in open object based distributed systems.