TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit

Ludovic Apvrille, Jean-Pierre Courtiat, *Member*, *IEEE*,
Christophe Lohr, and Pierre de Saqui-Sannes, *Member*, *IEEE*

Abstract—This paper presents a UML 1.5 profile named TURTLE (**T**imed **U**ML and **RT**-**L**OTOS **E**nvironment) endowed with a formal semantics given in terms of RT-LOTOS. TURTLE relies on UML's extensibility mechanisms to enhance class and activity diagrams. Class diagrams are extended with specialized classes named *Tclasses*, which communicate and synchronize through gates. Also, associations between *Tclasses* are attributed by a composition operator (*Parallel*, *Synchro*, *Invocation*, *Sequence*, or *Preemption*) which provides them with a formal semantics. TURTLE extends UML activity diagrams with synchronization actions and temporal operators (deterministic delay, nondeterministic delay, time-limited offer, and time-capture). The real-time dimension of TURTLE has been further improved by the addition of two composition operators, *Periodic* and *Suspend*, as well as suspendable delay, latency, and time-limited offer operators at the activity diagram level. Core characteristics of TURLE are supported by TTool—the TURTLE toolkit—which includes a diagram editor, a RT-LOTOS code generator and a result analyzer. The toolkit reuses RTL, a RT-LOTOS validation tool offering debug-oriented simulation and exhaustive analysis. TTool hides RT-LOTOS to the end-user and allows him/her to directly check TURTLE modeling against logical errors and timing inconsistencies. Besides the foundations of the TURTLE profile, this paper also discusses its application in the context of space-based embedded software.

Index Terms—Real-time systems, UML, RT-LOTOS, formal validation.

1 Introduction

The increasing development of computer-based applications that need to answer external stimuli under timing constraints has stimulated research work on real-time system design techniques. Also, life-criticality and tremendous prototyping costs of complex real-time systems —think, e.g., of aircrafts, satellites, and nuclear plants—has created potential conditions for deploying formal methods and a priori validation techniques that enable early detection of design errors in the life cycle of a system.

Despite of the expected benefits of a priori validation based on formal modeling, industrial practitioners have been reluctant to use formal methods. By contrast, an informal modeling language has received increasing acceptance in industry: the Unified Modeling Language [29], which is now an international standard at OMG (Object Management Group). This situation has stimulated research work on coupling UML and formal modeling languages with the purpose of giving the OMG-based notation a formal semantics and to enable a priori validation of UML diagrams (see, e.g., [1], [6], [7], [8], [9], [12], [13], [14], [16], [18], [19], [20], [21], [25], [35], [38]).

- L. Apvrille is with the ENST, Institut Eurecom, BP 193, 2229 route des Cretes, 06904 Sophia-Antipolis Cedex, France. E-mail: ludovic.apvrille@enst.fr.
- J.-P. Courtiat is with LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France. E-mail: courtiat@laas.fr.
- C. Lohr is with the Electrical and Computer Engineering Department, Concordia University, 1455 de Maisonneuve W., Montreal, Canada H3G 1M8. E-mail: lohr@ece.concordia.ca.
- P. de Saqui-Sannes is with ENSICA and LAAS-CNRS, 1 place Emile Blouin, 31056 Toulouse Cedex 05, France. E-mail: desaqui@ensica.fr.

Manuscript received 9 Jan. 2004; revised 17 Mar. 2004; accepted 8 Apr. 2004. Recommended for acceptance by M. Broy.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0007-0104.

A common way to add formality to UML is to define a "profile," i.e., a customization of the OMG-based notation, in order to meet specific needs in a particular application domain. In this paper, we define a real-time UML profile named TURTLE (*Timed UML and RT-LOTOS Environment*), which extends UML with structuring capabilities, advanced logical and temporal operators, and formal validation procedures borrowed from RT-LOTOS [11]. TURTLE is compliant with UML 1.5. The profile has a formal semantics given in terms of RT-LOTOS. TURTLE should not be viewed as a graphic syntax of RT-LOTOS. Indeed, the translation of TURTLE models into RT-LOTOS is not straightforward: Most operators described in this paper have no direct counterpart in RT-LOTOS. The major advantage of backing TURTLE on RT-LOTOS lies in the possibility to reuse RTL [34], the formal validation tool developed by LAAS-CNRS for RT-LOTOS. A prototype named TTool has been developed to assist real-time system designers in diagramming TURTLE models. TTool can generate a RT-LOTOS specification from TURTLE models. Once generated, a RT-LOTOS specification can be formally validated directly from TTool using RTL, making the use of the formal language RT-LOTOS totally hidden to UML designers.

This paper is organized as follows: Sections 2 and 3 introduce the TURTLE profile. They present the native TURTLE operators introduced in [2] and advanced ones proposed in [28], respectively. Section 4 explains how TURTLE's semantics has been formalized in RT-LOTOS. Formal validation of TURTLE models is the subject of Section 5. Section 6 reports a successful experience in using TURTLE for the formal validation of space-based embedded software. Section 7 surveys related work including

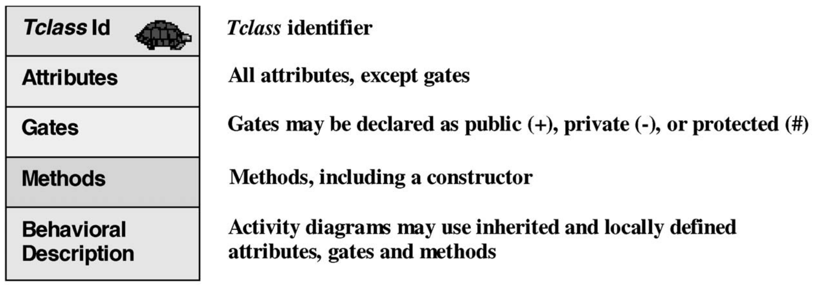

Tclass Id
Attributes
Gates
Methods
Behavioral Description
Tclass identifier
All attributes, except gates
Gates may be declared as public (+), private (-), or protected (#)
Methods, including a constructor
Activity diagrams may use inherited and locally defined attributes, gates and methods

Fig. 1. Structure of a *Tclass*.

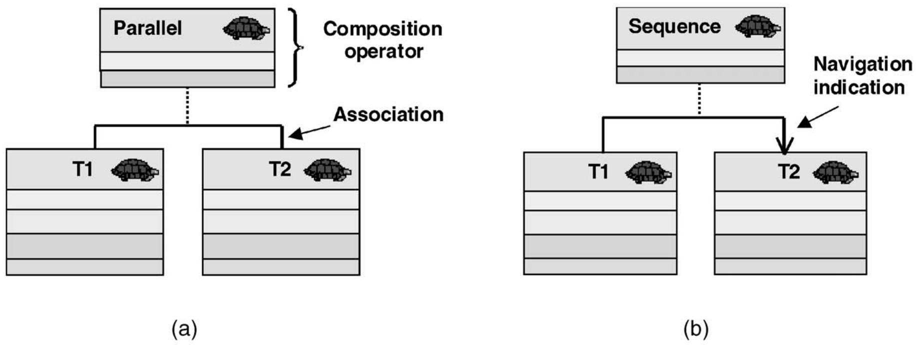


Fig. 2. Composition operators: (a) Parallel and (b) Sequence.

recently released material on UML 2.0. Section 8 concludes the paper and outlines future work.

2 Native TURTLE

2.1 A UML Profile

A UML "profile" may contain selected elements of the reference metamodel, a description of the profile semantics, additional notations, and rules for model translation, validation, and presentation. A profile definition enhances UML in a controlled way, in particular using the "stereotype" extensibility mechanism. A stereotype extends the UML metamodel, allowing one to create new kinds of building blocks derived from existing ones, but specific to a class of problems.

The TURTLE profile extends two diagrams of the UML 1.5 notation [29]: class diagrams which describe the static architecture of the system under design and activity diagrams which describe the internal behavior of the system's components.

2.2 Extended Class Diagrams

A TURTLE class diagram is made up of "normal" classes and stereotyped classes that we call *Tclasses*. Fig. 1 depicts the structure of a *Tclass*. A *Tclass* is identified by a "turtle" icon located in the upper right corner of the five-box *Tclass* symbol. Communications through public attributes or method calls are limited to communications between a *Tclass* and a normal class, or between two normal classes. Communication between two *Tclasses* uses so-called "gates." A gate is a particular *Tclass* attribute of type *Gate*. A gate can be used for synchronized communication with another *Tclass*, or for an action internal to a *Tclass*.

The internal behavior of each *Tclass* must be described using an activity diagram. The latter is the fifth element in a *Tclass* symbol (Fig. 1).

So far, we considered an isolated *Tclass* that may be used to represent a single task. In practice, real-time systems execute tasks that run in parallel and occasionally synchronize their activities. Therefore, we extend UML class diagrams with high-level operators enabling description of concurrency, sequence, synchronization, and preemption between *Tclasses*. If we refer to the vocabulary used in the process algebra community, we term these high-level operators as "composition operators."

Let us consider two *Tclasses* T1 and T2. To create a composition operator between T1 and T2, we create a relation link between T1 and T2,[1] and we decorate that relation with an associative class labeled by the relevant composition operator. The TURTLE profile originally presented in [2] supports the following composition operators: *Parallel* (Fig. 2a), *Synchro*, *Sequence* (Fig. 2b), and *Preemption*. The first three operators are used to describe tasks that execute in concurrency, tasks that synchronize on gates, and tasks that are executed in sequence. *Preemption* gives a task the possibility to interrupt another task forever.

Note: The association link in Fig. 2b is a directed one. It is indeed mandatory to explicitly state which task executes first, and which task executes in sequence (*Tclasses* T1 and T2, respectively).

2.3 Extended Activity Diagrams

The TURTLE profile extends UML activity diagrams with synchronization operators and temporal operators. The former are used to express synchronizations which are internal to *Tclasses*, or synchronizations between *Tclasses*.

1. A class may be associated to itself.

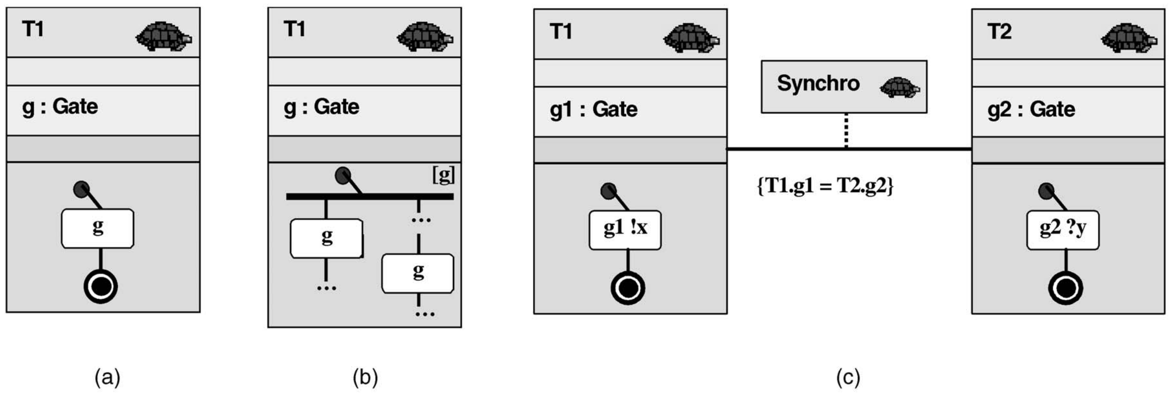


Fig. 3. (a) Internal action on g. (b) Internal synchronization on g. (c) External synchronization on g1/g2.

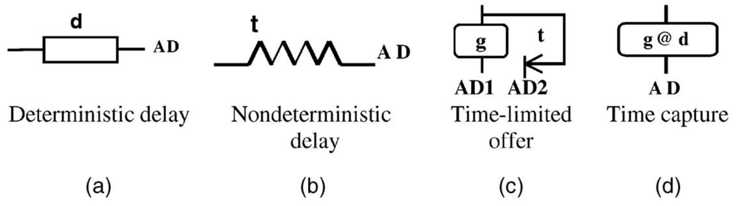


Fig. 4. Temporal operators supported by TURTLE activity diagrams.

The latter are used to describe temporal constraints that apply exclusively to the internal activities of *Tclasses*.

2.3.1 Synchronization Operators

In its activity diagram, a *Tclass* T may perform a call on a gate g. Three cases may apply:

- g is not synchronized with any other gate. In that case, a call on this gate models an internal action, see Fig. 3a.
- g is internally synchronized. For example, in Fig. 3b, g is synchronized with regards to two subactivities of T1 starting at a parallel operator whose synchronization gate list is "[g]." The synchronization occurs when both subactivities are ready to execute action "g."
- g is externally synchronized (g must be declared as public). In that case, T must be linked with a *Synchro* composition operator to another *Tclass*, and g must be specified as synchronization gate in an OCL formula attached to the *Synchro* operator. For example, in Fig. 3c, T1 synchronizes with T2. The synchronization gate of T1 is g1 and that of T2 is g2 (cf. the OCL formula). Any relation attributed by a *Synchro* operator must be decorated by an OCL formula stating which gates are connected together.

Two *Tclasses* may exchange data at synchronization time. In Fig. 3c, let us consider the two activity diagrams of *Tclasses* T1 and T2. When synchronization on gate g1/g2 occurs, the value of x is sent. This value is received by T2 at synchronization time, and stored in attribute y. Note: A synchronization action between two *Tclasses* T1 and T2 is syntactically valid if and only if T1 and T2 use two interconnected gates as well as compatible parameter lists.

Let us now focus on gate typing in TURTLE. A *Gate* abstract type is introduced and specialized in *Ingate* and *Outgate* that correspond to gates for data receiving and sending, respectively. Unlike "ports" in Rose RT [32] and ports in UML 2.0 [30], TURTLE gates are not defined by a list of authorized messages.

Note: TURTLE activity diagrams include parallel and synchronization operators which are also available at class diagram level, but do not include *Sequence* and *Preemption*. Indeed, *Suspend* and *Preemption* are dedicated to the modeling of relations between tasks at structuring level, i.e., at class diagram level. It is not possible to use a *Suspend* or *Preemption* inside a TURTLE activity diagram. To keep our profile as close as possible to UML 2.0, TURTLE activity diagrams only support standard UML parallel and synchronization operators.

2.3.2 Temporal Operators

Four temporal operators are introduced at activity diagram level.

First, a deterministic delay characterized by a fixed duration d (see the rectangle in Fig. 4a). The execution of AD is delayed by d units of time.

Second, a nondeterministic delay based on the RT-LOTOS latency operator [11]. The associated pictogram is given by Fig. 4b. Note that unless using proprietary solutions, UML 2.0 does not make it possible to work with temporal indeterminism and temporal intervals. The TURTLE profile does. In TURTLE, a temporal interval is modeled by putting a deterministic delay operator and a nondeterministic delay operator in sequence. The nondeterministic operator expresses the time latency inside the time interval.

Third, a time-limited offer which describes the possibility for an action to be performed before an amount of time has elapsed (see Fig. 4c).

Finally, TURTLE offers a time capture operator whose pictogram is depicted on Fig. 4d. The @ operator stores the

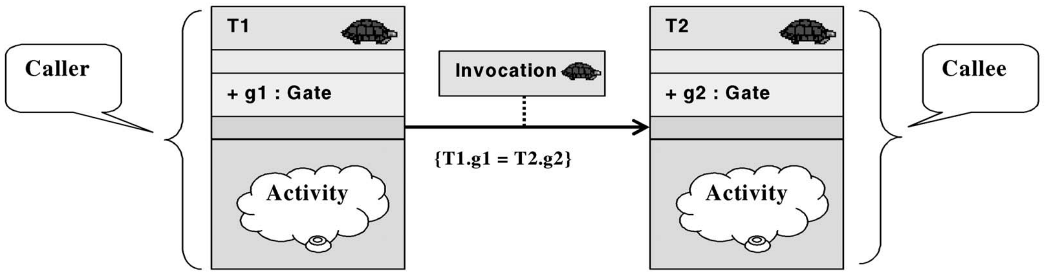


Fig. 5. Association attributed with an *Invocation* operator.

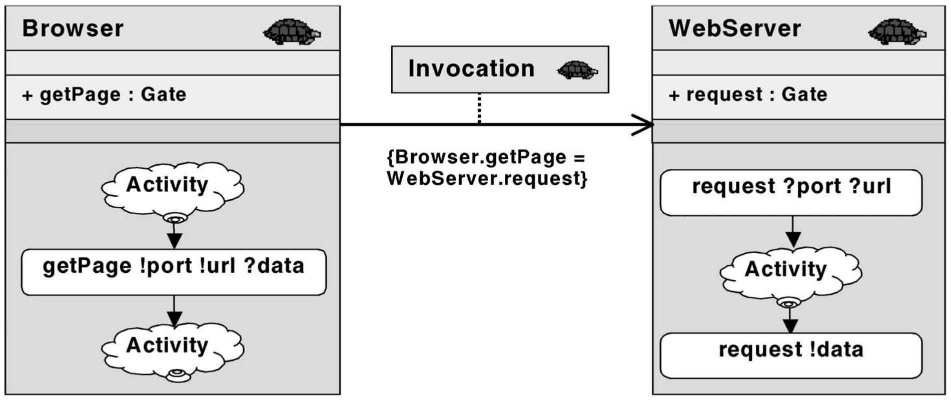


Fig. 6. Use of an *Invocation* operator in a browsing system.

amount of time elapsed between the instant when the *Tclass* offers an action (i.e., this action is ready to execute, but it cannot be performed immediately because it is synchronized with other actions which are not yet ready to execute), and the actual execution of this action.

The combination of these temporal operators makes it possible to model temporal behaviors such as timeouts, watchdogs, and many other mechanisms encountered in real-time systems.

3 ADVANCED TURTLE

The TURTLE profile originally published in [2] and outlined in previous section, was missing important features in terms of object orientation and real-time system structuring. On the object-oriented side, native TURTLE lacks a gate-based communication counterpart to method calls. A solution is proposed hereafter with the so-called *Invocation* operator. On the real-time side, native TURTLE extends activity diagrams with powerful temporal operators, but still lacks real-time operators at the class diagram level. In [28], we proposed to endow TURTLE with two composition operators: *Periodic* and *Suspend/Resume*. Definitions of these operators and examples on their use are given hereafter.

3.1 Invocation

Method call is a fundamental feature of object-oriented languages and of UML class diagrams in particular. With native TURTLE, modeling a method call requires the use of two *Synchro* operators. Moreover, the activity diagram of the two *Tclasses* involved in the two synchronizations must check for the validity of data exchange performed during that synchronization. In particular, when returning from a method call, data should be exchanged only from the callee to the caller. Such complexity leads us to introduce a novel operator—called *Invocation*—which makes it possible for a *Tclass* to insert the activity of another *Tclass* in its execution flows. *Invocation* differs from *Synchro* since the latter characterizes synchronization between two separate execution flows.

Fig. 5 depicts an *Invocation* associative class. The class is attached to an association starting at *Tclass* T1 (*Caller*) and heading to *Tclass* T2 (*Callee*). Then, we say that T2 can be invoked by T1.

Like synchronization, an invocation involves one gate in each *Tclass* it concerns. Let us call g1 and g2 two gates of T1 and T2, respectively. Let us assume the OCL formula {T1.g1 = T2.g2} is attached to the relation. Then, when T1 performs a call on g1, T1 must wait for T2 to perform a call on g2. When T2 performs the expected call, data can only be exchanged from T1 to T2, following the direction indicated by the arrow. In other words, parameters can be passed from T1 to T2. T1 is blocked until T2 performs a new call on g2. Call return values and other data can be passed from T2 to T1. Due to controlled data exchange, the *Invocation* operator is more complex than a mere sequence of two *Synchro* operators.

Fig. 6 depicts a browsing system model. An *Invocation* connects a *Browser* to a *WebServer*, playing *Caller* and *Callee* roles, respectively. The values passed by *Browser* when the invocation occurs are prefixed by "!" (*port*, *url*), whereas the returned values are prefixed by "?" (*data*).

3.2 Periodic

Native TURTLE lacks a high-level operator for the description of periodic tasks. The *Periodic* composition operator makes it possible to characterize the periodic behavior of a

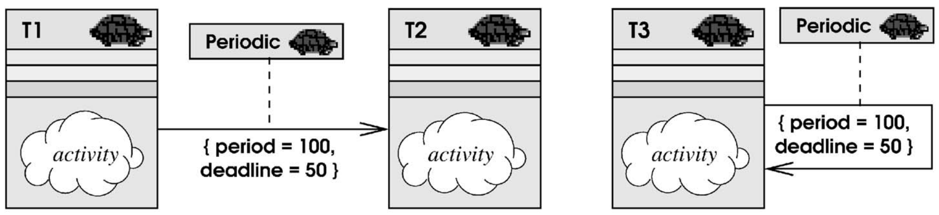


Fig. 7. Association attributed by a *Periodic* operator.

task modeled by a *Tclass*. This composition operator can attribute either an association linking two *Tclasses* or an association linking a *Tclass* to itself.

Let us first consider an association between two *Tclasses*. The *Tclass* pointed out by the association starts only after the *Tclass* at the origin of the association has completed its execution. Once started, the pointed *Tclass* executes its main activity periodically with regards to the *period* and *deadline* values specified in an OCL formula attached to the association. For example, in Fig. 7, when T1 has terminated, T2 starts and executes its activity diagram periodically, with a period of 100 time units and a deadline of 50 time units. Note that if the *Tclass* T2 is started by another *Tclass* than T1—let us assume there is a Sequence relation between a *Tclass* T4 and T2—then, when T4 completes, T2 is started in a nonperiodic mode.

Let us now consider a single *Tclass* with a looping association. That *Tclass* always executes its activity diagram periodically according to the *period* and *deadline* values specified in the OCL formula attached to the association (see T3 in Fig. 7).

3.3 Suspend/Resume

Native TURTLE includes a *Preemption* operator which allows a *Tclass* T1 to abruptly interrupt a *Tclass* T2 without any possibility for T2 to keep track of its current state and resume later on. Thus, TURTLE lacks a high-level operator to express the possibility for a *Tclass* to be suspended and subsequently resumed with its execution context being unchanged. The *Suspend*[2] operator introduced in this section answers that need.

Fig. 8 depicts a *Suspend* associative class attached to an association directed from a *Tclass* T1 (*Suspender*) to a *Tclass* T2 (*Suspended*). T2 can be suspended and reactivated by T1. Both operations require a call on gate *s* (s appears in the OCL formula associated with the relation from T1 to T2). When T1 performs a call on *s*, T2's activity is suspended. Then, the next call on s performed by T1 resumes T2. T1 can suspend and resume T2 as many times as needed.

TaskManager (Fig. 9) implements a basic scheduler which alternatively switches between *TaskA* and *TaskB*, two tasks sharing the same processor resource. *TaskManager* can suspend (or resume) *TaskA* and *TaskB* using gates *SwitchA* and *SwitchB*, respectively. When the application starts, both tasks are "normally" running. The state of these tasks is termed as "active state." But, they are immediately suspended by *TaskManager* (cf. the calls on *SwitchA* and *SwitchB* at the beginning of *TaskManager*'s activity diagram). Consequently, both tasks are in "suspended state." Then, *TaskA* and *TaskB* are activated one after the other during a quantum of time. The attribute *quantumA* (respectively, *quantumB*) denotes the quantum of time allocated to *TaskA* (respectively, *TaskB*).

3.4 Suspendable Temporal Operators

Native TURTLE temporal operators have been introduced assuming a universal time. Time continuously progresses and cannot be suspended. Further, it uniformly applies to all components of the system under design. These temporal operators can be used to model, for example, a timer but also to model the execution time of an algorithm.

With the *Suspend* composition operator, it becomes possible to interrupt the execution of a *Tclass*. When the activity of a *Tclass* is suspended, time progresses anyway and, consequently, the timers of the *Tclass* continue to run. By contrast, the algorithm implemented by the task must be stopped as soon as the *Tclass* is suspended. Nevertheless, the task should be allowed to resume all its activities and, more particularly, its algorithms at the point where they were suspended. Therefore, we introduce temporal operators supporting the concept of time suspension and resume. We call them "suspendable temporal operators."

Fig. 10 depicts the four new suspendable temporal operators: a "suspendable" deterministic delay (Fig. 10a), a "suspendable" nondeterministic delay (Fig. 10b), a "suspendable" timed-limited offer (Fig. 10c), and at last, a "suspendable" time-capture. A small hourglass added to the original symbols denotes time suspension. For the three first operators, the suspension stops the elapsing of time. For the fourth one (suspendable time capture, Fig. 10d), the semantics is a bit more complex. Indeed, in the nonsuspendable time capture, the delay d denotes the time between the offer on g, and the action on g. For the suspendable version of this operator, the delay d denotes the time spent in active state by the *Tclass* between the offer on g and the action on g.

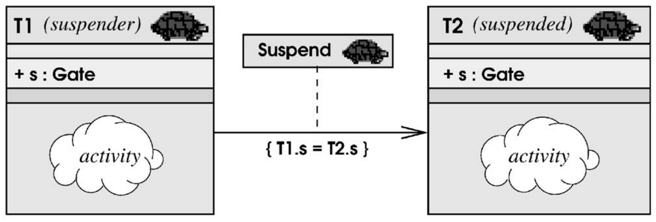


Fig. 8. Association attributed with a *Suspend* operator.

2. *Suspend* is an abbreviation for *Suspend/Resume*.

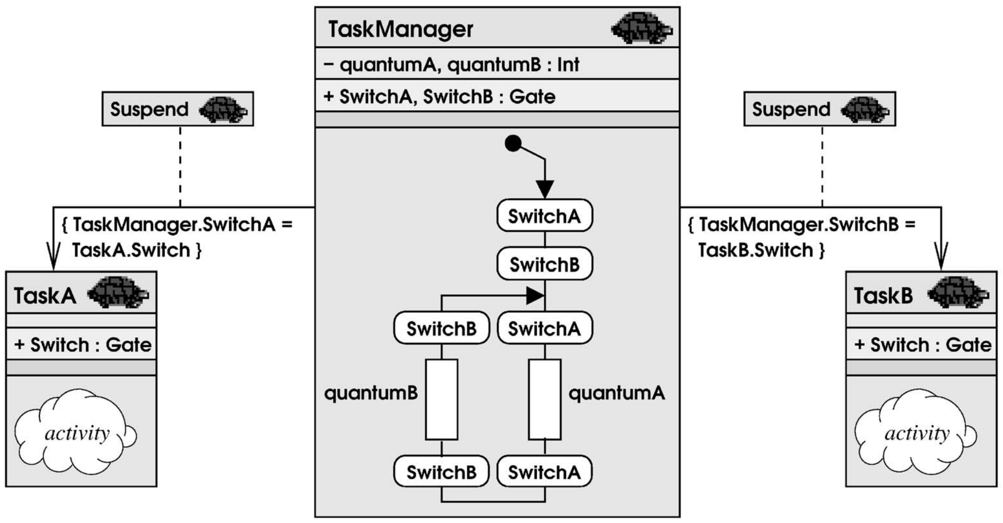


Fig. 9. Example of a basic scheduler modeled using the *Suspend* operator.

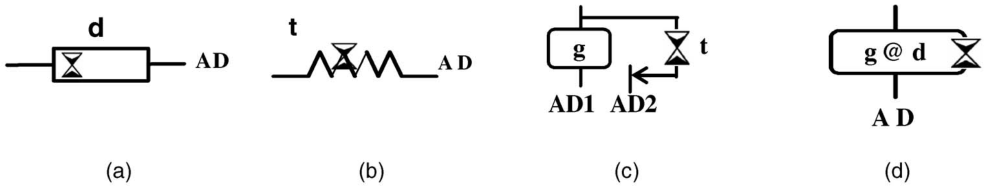


Fig. 10. Suspendable temporal operators. (a) Suspendable deterministic delay. (b) Suspendable nondeterministic delay. (c) Suspendable time-limited offer. (d) Suspendable time capture.

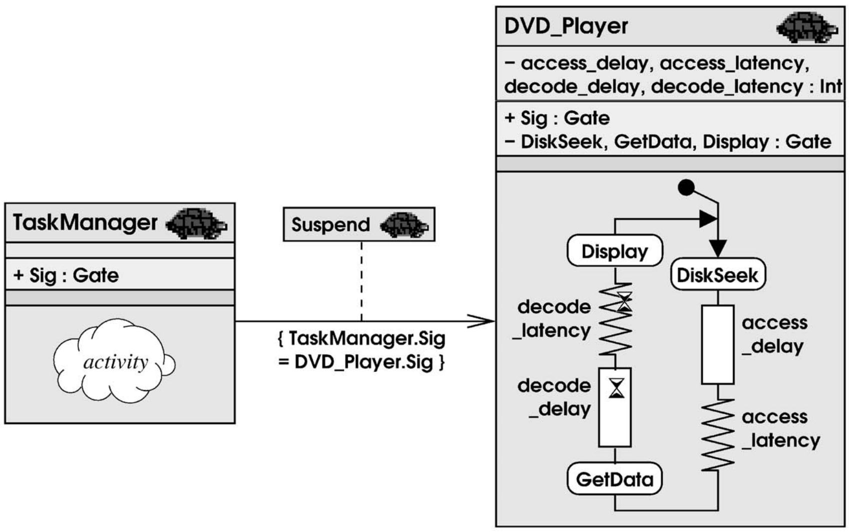


Fig. 11. A PC DVD player model using nonsuspendable and suspendable delays.

Fig. 11 illustrates the use of the deterministic and nondeterministic delay operators in their original and suspendable versions. The system under design is a DVD player for Personal Computer. First, the DVD player takes a disk and then it checks it. The access time to tracks depends on the mechanical properties of the device. Mechanical operations cannot be interrupted. Therefore, they are completed independently of any other computations on the PC. Conversely, *TaskManager* can suspend data decoding which follows synchronization on *getData*. As a consequence, we associate an hourglass with the deterministic and nondeterministic delays on the left branch of *DVD_Player*'s activity diagram.

4 Formal Semantics

Native TURTLE has a formal semantics expressed in RT-LOTOS [11]. Any TURTLE model can be translated to a RT-LOTOS specification. The semantics of advanced operators (*Invocation*, *Periodic*, *Suspend*, suspendable temporal operators) is given in native TURTLE which, in turn, can be translated to RT-LOTOS [28].

A TURTLE modeling structures a system into *Tclasses* and associates an activity diagram with each Tclass. We use a two-step TURTLE algorithm to translate TURTLE models to RT-LOTOS. At Step 1, an RT-LOTOS process is computed for each activity diagram. At Step 2, the RT-LOTOS

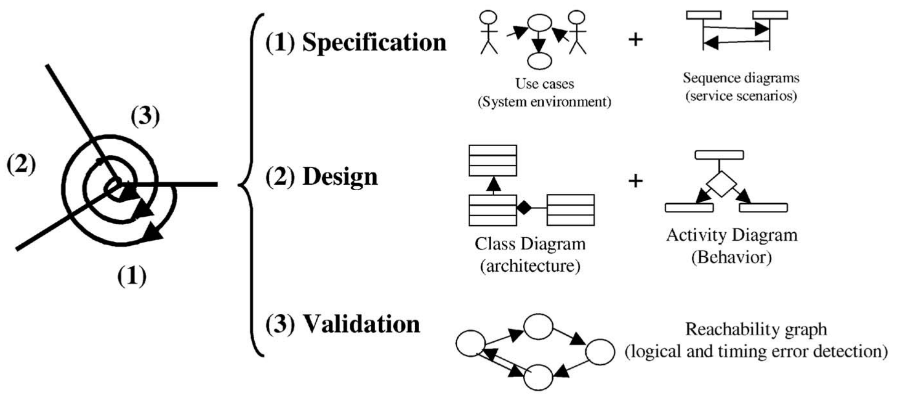


Fig. 12. Design trajectory.

processes obtained at the first step are composed using information supplied by the class diagram. This two-step algorithm can be sketched as follows:

1. **Activity diagrams**. Each *Tclass* contains an activity diagram that is translated to an RT-LOTOS process. The latter possibly encapsulates subprocesses that implement temporal operator, loop, junction, action state, attributes modification, and parallel/synchronization structures. TURTLE temporal operators (deterministic delay, nondeterministic delay, time-limited offer, and time-capture) and synchronization on gates have quite direct counterparts in RT-LOTOS. The translation of other operators is far more complex: It needs additional internal synchronizations and subprocesses. For example, the translation of the parallel operator P of UML activity diagrams is done as follows. First, all activities leading to P are synchronized together on an additional gate. Then, once synchronized, all these activities are killed, and a new RT-LOTOS process p is started. At last, the process p starts in parallel all activities starting at P.
2. **Class diagrams**. Associations between *Tclasses* can be attributed with composition operators. Let *T1* be a *Tclass* involved in a relation specified by a composition operator and *P1.1* the RT-LOTOS process obtained from the translation of *T1*'s activity diagram at Step 1. At Step 2, for each *Tclass*, a new RT-LOTOS process *P1.2* taking into account the composition operators involving *T1* is created. The body of process *P1.2* may call *P1.1* using appropriate RT-LOTOS gates (declaration or renaming). It may also refer to other processes *Pn.2* where *Tn* denotes another *Tclass* related to *T1* with a TURTLE composition operator. Given a class diagram, *Tclasses* are processed in the following order:

   a. *Tclasses* at the origin of *Preemption* operators,
   b. *Tclasses* at the origin of *Sequence* operators,
   c. *Tclasses* pointed out by *Preemption* or *Sequence* operators,
   d. *Tclasses* at the origin of both *Sequence* and *Preemption* operators, and
   e. Subsets of *Tclasses* connected together either by *Parallel* or by *Synchro* operators.

For example, let us consider three *Tclasses* T1, T2, and T3. T2 and T3 synchronize on g, and there is a *Sequence* relation between T1 and T2, and T1 and T3. The translation algorithm generates six processes, P1.1, P1.2, P2.1, P2.2, P3.1, and P3.2. P1.2, P2.2, and P3.2 contain the translation of the activity diagram of T1, T2, and T3, respectively. P2.2 and P3.2 just start P2.1 and P3.1, respectively. However, the body of P1.2 is more complex. Indeed, after completing its activity, T1 executes P2 and P3 in sequence, P2 and P3 being synchronized on g. Therefore, the body of P1.2 is *P1.1 >> (P2.2 |[g]| P3.2)*.

At least, for each instance of *Tclass Tn* at system startup, the RT-LOTOS specification instanciates a *Pn.2* process.

The TURTLE to RT-LOTOS translation algorithms give the profile a formal semantics. Their implementation enables reuse of the RTL [11], [34] validation toolkit.

5 The TURTLE Toolkit

5.1 Methodology

The methodology associated with TURTLE does not cover an entire life cycle. It is centered on model validation in early stages of the system's design trajectory. Fig. 12 depicts the incremental design trajectory suggested to TURTLE users.

- **Specification step**. The system's specification is expressed as a combination of use-cases and sequence diagrams.
- **Design step**. Extended class and activity diagrams describe the system's architecture and the classes' internal behavior, respectively.
- **Validation step**. In this paper, we address a priori validation. TURTLE diagrams of a system can be checked against design errors. For that, TURTLE class and activity diagrams are translated into RT-LOTOS. A priori validation includes simulation, which partly explores the system's space state, and verification based on exhaustive analysis. If simulation applies to all syntactically correct TURTLE models, verification can only be performed on TURTLE models with finite behavior, modulo a "reasonable" time for reachability analysis computation. Thus, verification faces the well-known state

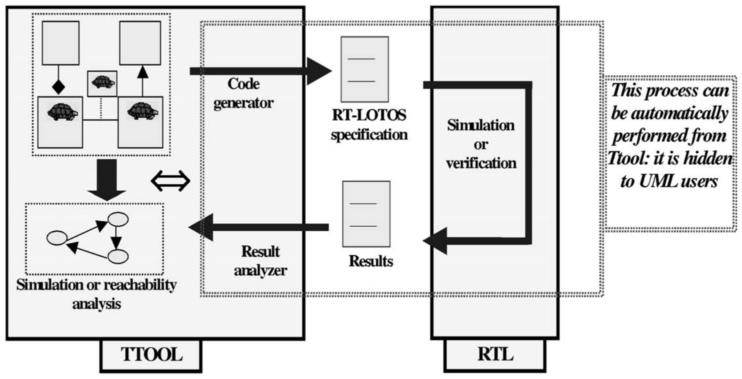

Code generator
RT-LOTOS specification
Simulation or verification
This process can be automatically performed from Ttool: it is hidden to UML users
Result analyzer
Results
Simulation or reachability analysis
TTOOL
RTL

Fig. 13. The TURTLE validation process is fully automated.

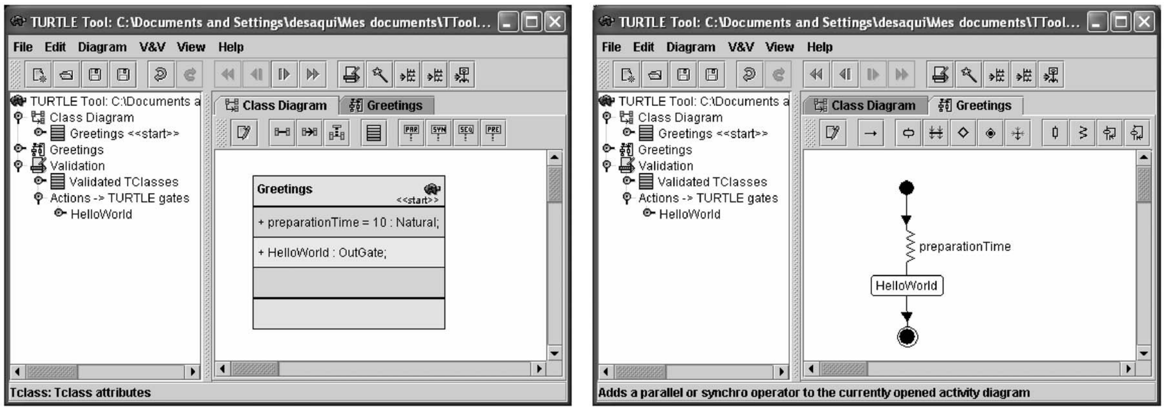

TURTLE Tool: C:\Documents and Settings\desaqui\Mes documents\TTool...
File Edit Diagram V&V View Help
TURTLE Tool: C:\Documents a
Class Diagram
Greetings <<start>>
Greetings
Validation
Validated TClasses
Actions -> TURTLE gates
HelloWorld
Class Diagram
Greetings
Greetings <<start>>
+ preparationTime = 10 : Natural;
+ HelloWorld : OutGate;
Tclass: Tclass attributes
preparationTime
HelloWorld
Adds a parallel or synchro operator to the currently opened activity diagram

Fig. 14. HelloWorld class and activity diagram edited using TTool.

space explosion problem. As a consequence, its use is usually limited to verifying core mechanisms and fundamental algorithms that deserve exhaustive analysis. Simulation is the complement to verification. It assists designers for detailed design of complex systems, in particular when the system's state space cannot be computed because of memory overflow.

5.2 Tools

The TURTLE toolkit includes TTool [39] and RTL [34], see Fig. 13.

- TTool—which stands for **T**URTLE **Tool** kit—offers a TURTLE class and activity diagram editor, a syntax checker, a RT-LOTOS code generator, and a simulation/verification graphical analyzer.
- RTL—which stands for RT-LOTOS Laboratory—takes as input a RT-LOTOS specification generated by TTool and performs either random[3] simulation for a given period of time or verification based on exhaustive analysis. For "finite" systems of "reasonable" size, RTL eventually outputs an optimized reachability graph which explicitly mentions time progression and clock constraints.

Fig. 14 illustrates the use of TTool for diagramming a HelloWorld application. A *Tclass* Greetings executes a synchronization action named *HelloWorld*. This action is delayed from 0 to *preparationTime* units of time. This example features a nondeterministic delay (see the symbol with a spring shape), a temporal operator not available with UML 2.0.

3. Dates are selected inside time intervals by applying one of the stochastic laws implemented by RTL.

5.3 RT-LOTOS Code Generation

A TURTLE class diagram and its associated activity diagrams being edited and checked for syntax, TTool is ready to generate an RT-LOTOS specification. The user has just to scroll the appropriate menu and select the "generate RT-LOTOS" option. Then, he or she obtains a RT-LOTOS specification ready to be provided as input to the RTL validation tool. The TURTLE to RT-LOTOS translation process is completely automated and there is no obligation for the designer to read the RT-LOTOS code before launching the validation.

5.4 Simulation

TTool processes simulation traces output by RTL and establishes correspondences between the RT-LOTOS code and the original TURTLE model. Fig. 15 gives an example of a simulation trace obtained for the *HelloWorld* example diagrammed in Fig. 14. The synchronization on gate *HelloWorld* occurs 7 units of time after the *Greetings Tclass* started.

5.5 Reachability Analysis and Property Verification

A reachability graph generated from a LOTOS specification contains a set of states and labeled transitions between the states. A transition between two states may involve a synchronization action between two *Tclasses*. If so, the

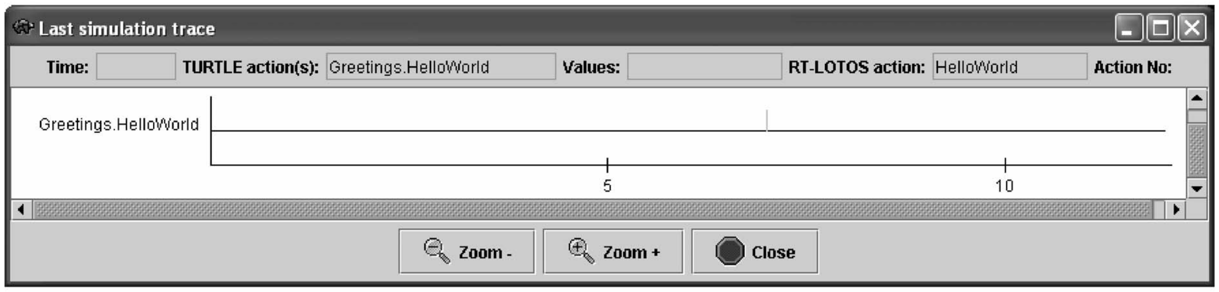


Fig. 15. *HelloWorld*: timelines displayed by TTool from a simulation trace generated by RTL.

transition is labeled by an identifier corresponding[4] to one of the gates involved in the synchronization. The purpose of generating a reachability graph is to verify one or several properties. A basic solution is to traverse the graph and look for all the transition identifiers that characterize the property.

This approach can be extended using the concept of observers [23]. An observer is a module external to the modeling of the system under design and usually expressed in the same language as the system's modeling. An observer can access the system's model components, in particular its variable. In the TURTLE context, an observer can synchronize with a *Tclass* on a dedicated gate so that the observer remains non intrusive. Any synchronization between a *Tclass* and its observer appears in the reachability graph under the form of a labeled transition. Researching *Tclass*-to-observer synchronization labels in the reachability graph makes it possible to analyze properties checked by the observer.

Formal proofs can be achieved by applying logic formula on the reachability graph. For that purpose, we used Kronos [24], a model checker which takes as input a logic formula and the reachability graph, and answers whether the property holds or not.

6 Experience with a Software-Based Satellite System

6.1 System Overview

The SAGAM project [33] has investigated new solutions for communication bandwidth optimization in the context of multibeam geostationary satellites offering bidirectional multimedia services. The SAGAM system (Fig. 16) implements a fast ATM cells switch and a temporal cell multiplexer on downlinks. User data switching and multiplexing are realized according to differentiated QoS.

Under the assumption that management functionalities are software-implemented and embedded in the satellite, the system under design implements the following functionalities:

- Users can initiate new ATM connections by sending a CAC-sig signal (CAC stands for Connection Admission Control) to the satellite.
- DAMA (Demand Assigned Multiple Access) and BAC (Block Acceptance Control) manage the ATM Variable Bit Rate and Unspecified Bit Rate traffics.
- User signals are sent by CAC-client (CAC-sig), DAMA-client (DAMA-sig), and BAC-client (BAC-sig). These services are located at User Earth Stations (Fig. 15).
- Every 50 ms, the satellite sends a Frame Allocation Report to let users know their allocated slots in the next uplink frame.

Details of the SAGAM functionalities can be found in [4], [10], and [33].

6.2 System Design in TURTLE

The SAGAM system was modeled and checked against design errors using the methodology depicted in Fig. 12. For space reasons, the user requirements captured by sequence diagrams are not reported in the paper. The class and activity diagrams designed for the SAGAM case study offer a high level of abstraction therefore enabling the use of reachability analysis techniques.

SAGAM software functionalities are executed by a set of real-time tasks that run concurrently, communicate asynchronously, and need to meet real-time constraints (deadlines, etc.). Moreover, the system works in various modes triggered by internal errors and external orders (*regular mode*, *error mode*, *recovery mode*, etc.). System tasks can be started or stopped when changing mode.

The system is abstracted as a software architecture characterized by tasks, communication between tasks, and various running modes. We also model the tasks' behavior (tasks' internal algorithms).

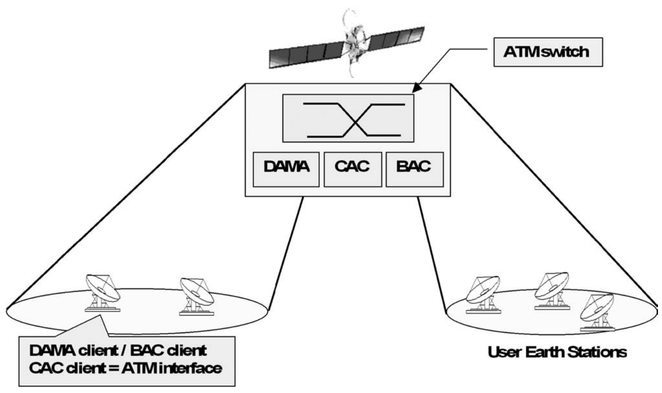


Fig. 16. The SAGAM system.

4. The TURTLE to RT-LOTOS translation renames gate identifiers. A correspondence table constructed during the translation process makes it possible to identify the TURTLE gates corresponding to a label in the reachability graph.

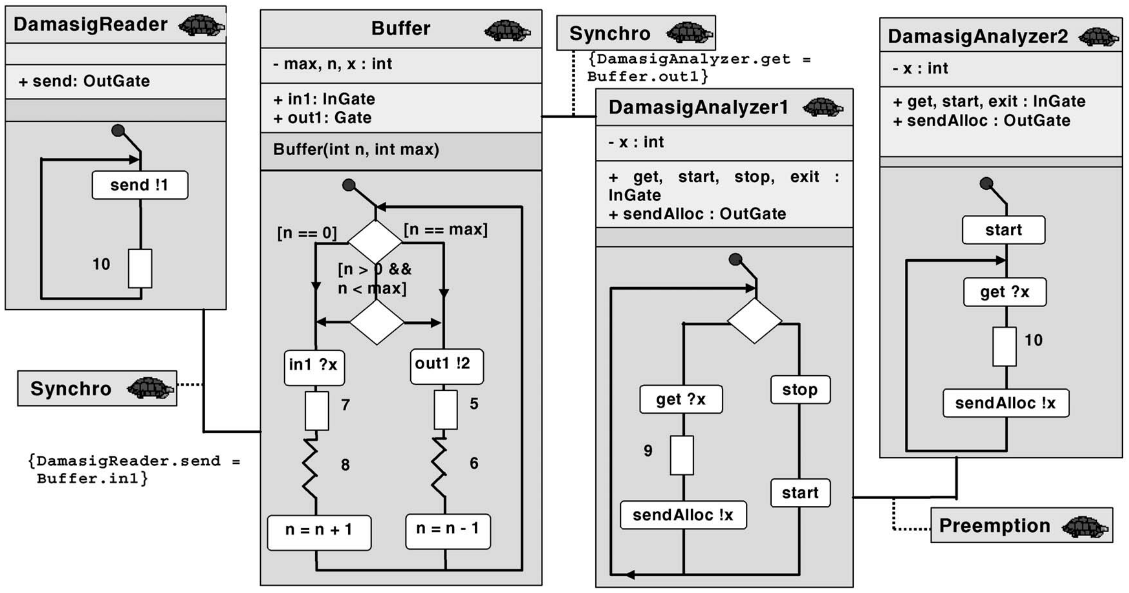


Fig. 17. Excerpt of the class diagram.

In practice, we define one *Tclass* per software real-time task, such as *DamasigReader* that receives *Dama-sig* user signals. Consequently, the TURTLE class diagram depicted in Fig. 17 contains a *Tclass* named *DamasigReader*. The *DamasigReader* task drives the hardware equipment which receives Dama-sig, formats received Dama-sig, and forwards them to *DamasigAnalyzer*. For each Dama-sig signal, *DamasigAnalyzer* decides whether the allocated bandwidth of the user who sent the signal needs to be increased or not. The result is forwarded to *FARSender*, the task that receives new allocations from other tasks (*DamasigAnalyzer*, etc.), and regularly sends a Frame Allocation Report to users.

Let us now consider relations between tasks. Tasks communicate by means of asynchronous message exchanges. Since the only mean for two *Tclasses* to exchange data is to perform a synchronization action, we introduce an intermediate *Tclass* which models an asynchronous communication link between two Tasks. In Fig. 17, a *Tclass* named *Buffer* models an asynchronous communication link between *DamasigReader* and *DamasigAnalyser*. Both *DamasigReader* and *DamasigAnalyser* synchronize with *Buffer. Send* is an *OutGate* of *DamasigReader. in1* is an *InGate* of *Buffer*. Consequently, the synchronization between *DamasigReader* and *Buffer* models a unidirectional communication link from *DamasigReader* to *Buffer*. The link between *Buffer* and *DamasigAnalyzer1* is unidirectional.

We further need to consider functioning modes where tasks are possibly stopped or started. Consider, e.g., that, in *error mode*, task *DamasigAnalyzer1* is stopped, and *DamasigAnalyzer2* is started. The model uses a *Preemption* composition operator between *DamasigAnalyser1 and DamasigAnalyzer2*. Fig. 17 depicts only a subset of the original class diagram, which includes 20 *Tclasses*. The class diagram in Fig. 17 models real-time tasks (*DamasigReader*, *DamasigAnalyzer1*, *DamasigAnalyzer2*) and communication links (*Buffer* plus two *Synchronize* composition operators). With the *Preemption* composition operator, the diagram in Fig. 17 also addresses software's evolution in terms of tasks.

Lets us now focus on the real-time tasks' behavioral modeling. *Tclasses* representing real-time tasks model algorithms at a high level of abstraction. The latter includes main input arguments, main results, and the time taken to compute these results (details of the algorithms are provided in non-Tclasses not depicted in Fig. 16). Three methods have been used to estimate algorithms' duration in the SAGAM software:

1. Algorithm simulations. For space-based embedded software, a commonly used simulator is ERC32-SIS developed by ESTEC [17].
2. Algorithm compilation and test on target.
3. Count of maximum possible operations performed and conversion to timing constraints that depend on potential targets.

For instance, consider *DamasigAnalyzer1 Tclass* on Fig. 17. It can be stopped and start again (right branch of choice). It may also get data from gate get (left branch). If so, it computes received data, and outputs an allocation on gate *sendAlloc*. The algorithm which computes the allocation is modeled with a deterministic delay between *get?x* and *sendAlloc!x*. This delay equals the maximum duration obtained during simulation [4].

6.3 Formal Validation

Reachability graph analysis was used to check the SAGAM model against logical and real-time properties. Each action in the reachability graph is stamped with data exchange and time information. This makes it possible to check for trivial properties but not for complex one. For example, suppose that an action "a" should be performed only every 100ms. One has to search on the graph for each action "a" and check that all "a" time-stamped are correct, which is a very tedious task: Reachability analysis cannot be manually

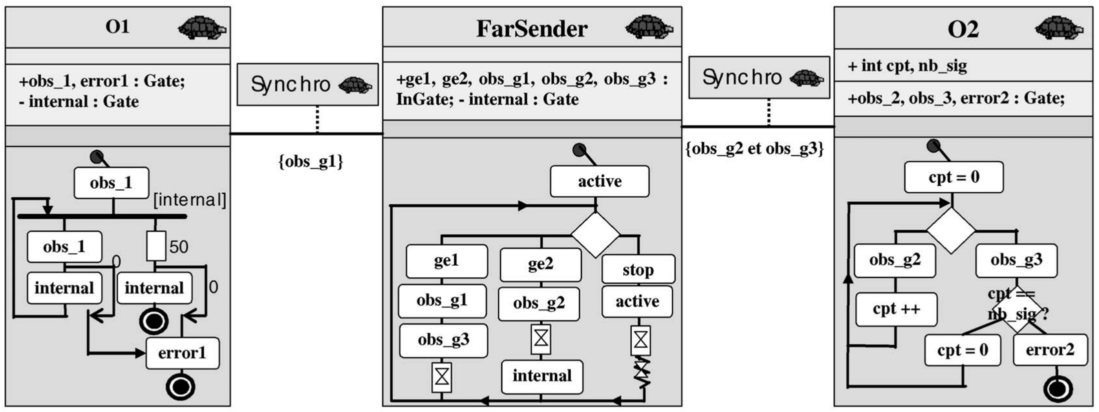


Fig. 18. Modeling two observers.

performed for checking complex properties. The observer technique helped us overcoming the problem.

Let us consider software has the following two properties:

- Property 1: The frame allocation report is sent every 50 milliseconds.
- Property 2: Every user signals must be computed before the next frame allocation report is sent.

We use two observers (see Fig. 18) *O1* and *O2* to verify *prop1* and *prop2*, respectively. O1 and O2 synchronize with *FarSender* to get information data. As long as a property checked by an observer remains valid, that observer keeps synchronized with *FarSender*. Therefore, observers are qualified as "nonintrusive" with respect to the application. For instance, gate *obs_g1* is always offered by *O1* as long as Property 1 holds. When a property becomes false, the corresponding observer performs the *error* action: The observer is stopped, and so is the observed class the next time it synchronizes with that observer.

For example, let us consider observer O1. Its purpose is to check that every frame allocation report is really emitted every 50 milliseconds. First, O1 executes action *obs_1*, which is synchronized with action *obs_g1* of *FarSender*. This synchronization corresponds to the first sending of a frame allocation report. Then, two subprocesses which synchronize on the *internal* gate of O1 are started. The first subprocess waits for synchronization on *obs_1* (next frame allocation report). Then, either it can execute action *internal* immediately (note the time-limited offer operator with the "0" value) or it is rerouted to action *error1*. At the same time, the second subprocess first waits for 50 times units (1 time unit = 1 millisecond). Then, it executes at once action *internal* or action *error1*. Thus, the first subprocess waits for the next frame allocation, whereas the second one checks for the time duration of the first one. If the property is satisfied, the first subprocess launches again two new subprocesses, and so on (loop operator). Otherwise, both subprocesses stop, and *FarSender* stops the next time it executes action *obs_g1*.

Fig. 19 depicts the reachability graph obtained for the SAGAM class diagram enriched with observers O1 and O2. Transition *error2* proves that property 2 can be violated. Indeed, all distinct noninfinite transition paths leading from the graph's initial state to the *error2* transition represent a set of traces leading to the violation of Property 2.

Reachability graphs were generated for abstract models in which algorithms were represented by their real-time profile. Indeed, with its nondeterministic real-time operator, TURTLE provides an explicit way to model lower and upper limits of algorithms' duration.[5] Such a modeling is not possible at all with UML 1.5. The reachability graph for the case study was computed in less than 5 minutes on a SUN UltraSparc. The size of the reachability can be reduced by a limited use of variables and nondeterministic delays.

7 Related Work

This section compares the TURTLE profile with other real-time UML profiles and proposals of coupling UML with a formal language. These proposals range from commercial tools, such as Rose RT which supports an extended UML with ROOM language [35] to academic propositions which either gives a precise semantics to UML [7], [18] or couple UML and a formal description technique such as Labeled Transition Systems [21], Petri Nets [13], Z [16], Esterel [1], PVS [38], B [25], and E-LOTOS [8]. Few papers have focused on verification capabilities: see, e.g., [9], [26] for UML and abstract machines and [14] for UML and UPPAAL.

7.1 Comparison with UML 1.x-Based Proposals

- **Compliance with the OMG Standard**. Unlike the Petri-net-based notation proposed by [13], TURTLE is fully compliant with UML 1.5 and exclusively uses the stereotype and tag value extension mechanisms authorized by the standard.
- **Structuring**. TURTLE inherits the notion of composition operator from the RT-LOTOS process algebra. The idea of composing behaviors is as old as the idea of composing communicating state machines, as implemented by Rose RT [32] or Tau G2 [36]. With these tools, composition is hidden and implicit. Conversely, TURTLE makes composition explicit.

5. Estimated from target platform benchmarking.

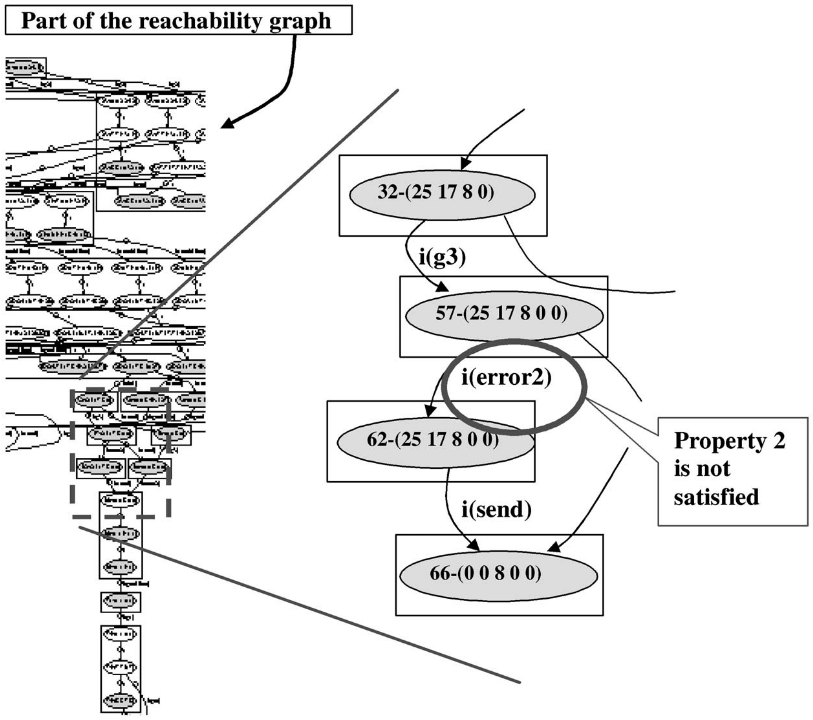


Fig. 19. Reachability graph of the example.

Associations between *Tclasses* are attributed by associative classes containing composition operators.

- **Communication**. Unlike Esterel-based Syncharts [1], TURTLE implements an asynchronous paradigm, which properly answers our objective to model distributed systems. With its LOTOS-based rendezvous mechanism, TURTLE remains independent from implementation details, such as the FIFO-queued communication implemented by Rationale's RoseRT, Telelogic's TAU, and Ilogix's Rhapsody [31]. The ACCORD/UML profile described in [20] also implements an asynchronous paradigm, including a broadcast mechanism. TURTLE does not implement such a broadcast mechanism. TURTLE-P [3], an enhanced TURTLE with component and deployment diagrams, overcomes that limitation and enables description of broadcast links with quality of service parameters.
- **Behavior**. The internal behavior of TURTLE *Tclasses* is described by means of activity diagrams, not Statecharts. We miss the hierarchical structuring facilities of Statecharts, but we make a clear distinction between structure and behavior descriptions. Nevertheless, a TURTLE activity diagram may contain parallel execution flows, which means that TURTLE allows intraclass parallelism and synchronization.
- **Real time**. TURTLE offers four temporal operators: a deterministic delay, a nondeterministic delay, a time-limited offer, and a time capture operator. Combining the deterministic and nondeterministic delay makes it possible to express time intervals. Therefore, TURTLE is more powerful than UML profile such as the Rose RT's one, which is limited to expressing fixed duration without any implementation-independent means to express temporal latency. A deterministic delay operator, which by essence expresses a fixed duration, is not sufficient for describing jitter and skew in a networked multimedia system. This is why TURTLE also has a nondeterministic delay operator. Further, TURTLE gives its temporal operators a formal semantics. The profile enables abstract description with large independence of implementation target. TURTLE's deterministic delay operator thus differs from the "tm" operator implemented by Ilogix's Rhapsody. It also differs from the timeout operator supported by Artisan Software's Real-Time Studio [5]. Last but not least, let us add that TURTLE supports suspendable timed operators which, to our knowledge, have no counterpart in commercial tools. Also, the Periodic and Suspend/Resume operators enable compact descriptions of mechanisms occurring frequently in real-time systems. Moreover, the *Suspend* operators make it possible to model mechanisms related to hardware interruptions.
- **Formal Semantics**. The TURTLE profile shares with [8], [16], and [38] the fact that its formal semantics is given by translation to a formal language. Theelen et al. [37] states that a "delay" operator suffices to model real-time systems as soon as that "delay" can be combined with an "interrupt" operator. Whether the statement holds for weak urgency semantics or not, it is no longer valid in the general case where the four TURTLE temporal operators are necessary.

- **Validation Tools**. The importance of a priori validation or, in other terms model analysis before coding, has been acknowledged by UML practitioners and real-time system designers in particular. For instance, the EU-project OMEGA has selected Rhapsody and TAU to support a development methodology adapted to real-time embedded systems [22]. The project addresses a subset of UML and gives that subset a formal semantics [12]. The OMEGA profile does not offer any temporal extension. Also, ACCORD [20] is a UML methodology for real-time development based on Softeam's Objecteering. ACCORD limits as much as possible the number of extensions to UML, and stems its originality in using design patterns and novel design rules.

Unlike UML tools geared to code debugging, TURTLE goes beyond code animation and enables formal validation at high level of abstraction. The validation process described in this paper has been fully automated. TURTLE benefits of advances functionalities provided by RTL, including optimized reachability analysis based on clock region generation and scheduling automaton generation (Timed Labeled Scheduling Automaton [27]). Again, the TURTLE toolkit makes the designer use RTL transparently thanks to the user-interface offered by TTool.

7.2 Comparison with UML 2.0

UML 2.0 [30] has been adopted as an OMG standard in August 2003, not much later after we started writing this paper. UML 2.0 has been designed with real-time systems in mind, and particularly applies to protocol modeling and communication architecture validation [15].

In UML 2.0, classes may have ports. Recently released tools such as Telelogic's TAU G2, implement queued communication. In TURTLE, *Tclasses* may have gates. The latter enable rendezvous communication which is more abstract than queuing in UML 2.0, and makes design more independent of implementation concerns.

UML 2.0 has introduced composite structure diagrams which enable description of so-called "parts" contained in classes. "Parts" own ports that can be interconnected by communication channels. The TURTLE profile may evolve in the near future in order to integrate composite structure diagrams. The designer will be advised to use the class diagram to describe *Tclasses* and composite structure diagrams to describe newly named *Tparts* with their ports and interconnection between these ports. Communication will remain based on rendezvous *à la* RT-LOTOS.

On the behavioral description side, UML 2.0 supports extended statecharts with an SDL-like, transition oriented syntax. TURTLE supersedes UML 2.0 by its temporal operators. In particular, TURTLE has a nondeterministic delay operator. UML 2.0 is limited to a deterministic delay operator which expresses a fixed duration. Therefore, UML 2.0 makes it impossible to work with temporal intervals unless using proprietary solutions not backed by any formal semantics.

TURTLE further offers formal verification functionalities that are unmatched by code animators of recently released UML 2.0 tools. The profile should evolve to integrate timing diagrams now include in the OMG standard. In addition, TTool may represent simulation traces in the form of timing diagrams. Indeed, simulation traces depict all the actions performed by the system during a simulation. By restricting these actions to the one of a particular *Tclass*, it should be possible to build all the time-stamped state transitions (actions) of this *Tclass*.

8 Conclusions and Future Work

Considering the low usage of formal methods and the increasing acceptance of the Unified Modeling Language in industry, the paper proposes a solution to take the best of the OMG-based notation in terms of diagramming facilities, and the best of the RT-LOTOS formal language in terms of structuring capacity. The result is a real-time UML profile named TURTLE, an acronym for Timed UML, and RT-LOTOS Environment. TURTLE is based on composition operators, wide-spectrum temporal operators, and formal validation functionalities. TURTLE is fully compliant with UML 1.5.

TURTLE extends UML classes with *Tclasses* endowed with synchronization gates. An association between two *Tclasses* can be attributed with a composition operator. Whereas concurrency is implicit in UML 2.0, TURTLE composition operators enable explicit expression of concurrency, synchronization, sequence, periodicity, and suspension/resume of tasks modeled by *Tclasses*.

Each *Tclass* of a TURTLE class diagram contains an activity diagram describing the internal behavior of that *Tclass*. TURTLE extends activity diagrams with a nondeterministic delay, a time-limited offer, and a time capture operator. Among TURTLE temporal operators, only the fixed delay operator has a counterpart in UML 2.0. The three other TURTLE operators do not. TURTLE enables description of time intervals and timing uncertainty, a feature of high importance for communication architecture validation.

The TURTLE profile has indeed been developed with a priori validation in mind, i.e., with the objective to offer to real-time system developers a formal support to validate their design as soon as possible in the system's design trajectory. TURTLE is supported by a toolkit made up of TTool [39] and RTL [34]. TTool integrates a TURTLE diagram editor, a TURTLE diagram syntax checker, and a RT-LOTOS code generator. The generated code can be given as input to RTL throughout the selection of debugging-oriented and exhaustive simulation options offered by the TTool interface. The press-button approach of TTool makes RT-LOTOS and RTL hidden to users.

The paper discusses the lessons learned in applying TURTLE to the design of a space-based embedded software. The case study particularly illustrates the use of reachability analysis. This technique faces the well-known state explosion problem. So far, RTL can generate graphs of several thousand states. We plan now to develop a new validation tool that might generate graphs of several millions states.

In conclusion, we plan to make the TURTLE profile evolve to describe distributed systems [3] and to take into account the latest development of UML 2.0 at OMG.

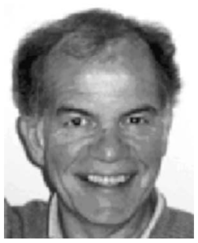
ACKNOWLEDGMENTS

The authors thank Professor Roch H. Glitho for his valuable comments on a draft version of this paper. Ludovic Apvrille was supported by an Alcatel-Space grant, and the Electrical and Computer Engineering Department of Concordia University, Montreal, Quebec, Canada.

REFERENCES

[1] C. André, M.-A. Peraldi-Frati, and J.-P. Rigault, "Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems," *Proc. Int'l Conf. Unified Modeling Language,* J.-M. Jézéquel, H. Hußmann, and S. Cook, eds., 2002.
[2] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat, "A New UML Profile for Real-Time System Formal Design and Validation," *Proc. Int'l Conf. Unified Modeling Language,* M. Gogolla and C. Kobryn, eds., pp. 287-301, 2001.
[3] L. Apvrille, P. de Saqui-Sannes, and F. Khendek, "TURTLE-P: A UML Profile for Distributed Architecture Validation," *Proc. Colloque Francophone sur l'Ingénierie des Protocoles,* Oct. 2003.
[4] L. Apvrille, P. de Saqui-Sannes, P. Sénac, and C. Lohr, "Verifying Service Continuity in a Dynamic Reconfiguration Procedure: Application to a Satellite System," *Automated Software Eng.,* vol. 11, no. 2, Apr. 2004.
[5] Artisan Software, http://www.artisansw.com/UMLCenter/UML. asp, 2003.
[6] M. Bjorkander, "Real-Time Systems in UML and SDL," *Embedded System Eng.,* Oct./Nov. 2000, http://www.telelogic.com.
[7] J.-M. Bruel, "Integrating Formal and Informal Specification Techniques. Why? How?" *Proc. IEEE Workshop Industrial-Strength Formal Specification Techniques,* 1999.
[8] R.G. Clarck and A.M.D. Moreira, "Use of E-LOTOS in Adding Formality to UML," *J. Universal Computer Science,* vol. 6, no. 11, pp. 1071-1087, 2000.
[9] K. Compton, Y. Gurevich, J. Huggins, and W. Shen, "An Automatic Verification Tool for UML," Technical Report CSE-TRE-423-00, Univ. of Michigan, 2000.
[10] S. Combes, C. Fouquet, and V. Renat, "Packet-Based DAMA Protocols for New Generation Satellite Networks," *Proc. 19th AIAA Int'l Comm. Satellite Systems Conf.,* 2001.
[11] J.-P. Courtiat, C.A.S. Santos, C. Lohr, and B. Outtaj, "Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique," *Computer Comm.,* vol. 23, no. 12, pp. 1104-1123, 2000.
[12] W. Damm, B. Josko, A. Votintseva, and A. Pnueli, "A Formal Semantics for a UML Kernel Language," OMEGA project deliverable IST/33522/WP1.1/D1.1.2, Jan. 2003.
[13] J. Delatour and M. Paludetto, "UML/PNO, a Way to Merge UML and Petri Net Objects for the Analysis of Real-Time Systems," *Proc. Workshop Object-Oriented Technology and Real Time Systems (ECOOP '98),* 1998.
[14] K. Diethers, U. Goltz, and M. Huhn, "Model Checking UML Statecharts with Time," *Proc. Workshop on Critical Systems Development with UML (UML '02),* J.-M. Jézéquel, H. Hußmann, and S. Cook, eds., 2002.
[15] L. Doldi, *UML2 illustrated: Developing Real-Time and Communications Systems.* TMSO, Oct. 2003.
[16] S. Dupuy and L. du Bouquet, "A Multi-Formalism Approach for the Validation of UML Models," *Formal Aspects of Computing,* no. 12, pp. 228-230, 2001.
[17] ESTEC, http://www.estec.esa.nl/wsmwww/erc32/erc32.html, free simulation software for ERC-32: http://www.estec.esa.nl/wsmwww/erc32/freesoft.html, 1999.
[18] A.S. Evans, S. Cook, S. Mellor, J. Warmer, and A. Wills, "Advanced Methods and Tools for a Precise UML," *Proc. Second Int'l Conf. Unified Modeling Language (UML '99),* 1999.
[19] S. Flake and W. Mueller, "A UML Profile for Real-Time Constraints with the OCL," *Proc. Int'l Conf. Unified Modeling Language,* J.-M. Jézéquel, H. Hußmann, S. Cook, eds., 2002.
[20] S. Gerard, F. Terrier, and Y. Tanguy, "Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML," *Proc. Conf. Advances in Object-Oriented Information Systems, OOIS 2002 Workshops,* J.-M. Bruel and Z. Bellahsene, eds., pp. 260-269, 2002.
[21] A. Le Guennec, "Méthodes Formelles avec UML: Modélisation, Validation et Génération de Tests," *Actes du 8éme Colloque Francophone sur l'Ingénierie des Protocoles (CFIP '2000),* pp. 151-166, 2000.
[22] J. Hooman, "Towards Formal Support for UML-Based Development of Embedded Systems," *Proc. PROGRESS Workshop Embedded Systems,* Oct. 2002.
[23] C. Jard, J.-F. Monin, and R. Groz, "Development of Véda, A Prototyping Tool for Distributed Algorithms," *IEEE Trans. Software Eng.,* vol. 14, no. 3, Mar. 1988.
[24] Kronos, http://www-verimag.imag.fr/TEMPORISE/kronos, 2002.
[25] R. Laleau and A. Mammar, "An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations," *Proc. 15th IEEE Int'l Conf. Automated Software Eng.,* 2000.
[26] J. Lilius and I.P. Paltor, "vUML: A Tool for Verifying UML Models," *Proc. 14th IEEE Int'l Conf. Automated Software Eng.,* 1999.
[27] C. Lohr and J.-P. Courtiat, "From the Specification to the Scheduling of Time-Dependent Systems," *Proc. Seventh Int'l Symp. Formal Techniques in Real-Time and Fault Tolerant Systems,* pp. 129-145, 2002.
[28] C. Lohr, L. Apvrille, P. de Saqui-Sannes, and J.-P. Courtiat, "New Operators for the TURTLE Real-Time UML Profile," *Proc. IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems,* E. Najm, U. Nestmann, and P. Stevens, eds., pp. 214-228, 2003.
[29] Object Management Group, "Unified Modeling Language Specification," Version 1.5, http://www.omg.org/docs/formal/03-03-01.pdf, Mar. 2003.
[30] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/docs/ptc/03-08-02.pdf, 2003.
[31] http://www.ilogix.com/products/rhapsody/index.cfm, 2003.
[32] http://www.rational.com, 2003.
[33] L. Roullet et al., "SAGAM Demonstrator of a G.E.O. Satellite Multimedia Access System: Architecture & Integrated Resource Manager," *Proc. European Conf. Satellite Comm.,* 1999.
[34] Real-time LOTOS, http://www.laas.fr/RT-LOTOS, 1998.
[35] B. Selic and J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems," http://www.rational.com, 1998.
[36] http://www.telelogic.com, 2003.
[37] B.D Theelen, P.H.A. van der Putten, and J.P.M. Voeten, "Using the SHE Method for UML-Based Performance Modeling," *Proc. Forum on Specification and Design Languages (FDL '02),* 2002.
[38] I. Traoré, "An Outline of PVS Semantics for UML Statecharts," *J. Universal Computer Science,* vol. 6, no. 11, pp. 1088-1108, 2000.
[39] http://www.eurecom.fr/apvrille/TURTLE/index.html, 2003.

Ludovic Apvrille received the MSc degree in computer science, network and distributed systems specialization, from ENSEIRB and ENSICA in 1997 and 1998, respectively. Then, he completed the PhD degree at LAAS-CNRS, Toulouse, France, in the research group Software and Tools for Communication and under the supervision of Michel Diaz and Patrick Sénac. This PhD work was part of a collaboration between the Department of Applied Mathematics and Computer Engineering at ENSICA, and Alcatel Space Industries. After a postdoctoral term at Concordia University (Montreal, Canada), he joined LabSoc as an assistant professor at ENST, Ecole Nationale Supérieure des Télécommunications. His research interests focus on tools and methods for the modeling of embedded systems.



Jean-Pierre Courtiat graduated in computer science from ENSEEIHT (1973) and received the PhD degree (1976) and a "Doctorat d'Etat" degree (1986) in computer science from the University of Toulouse, France. He was an appointed researcher at LAAS-CNRS from 1973 to 1976. From 1976 to 1980, he was a French technical coopérant and an appointed professor of computer science at the Federal University of Rio de Janeiro, Brazil. In 1980, he returned to LAAS, as a "Chargé de recherche au CNRS" and then a "Directeur de recherche au CNRS" (research positions on the French National Council of Scientific Research). At LAAS, he heads the OLC (Software and Tools for Communication) research group. His research interests include the design of protocols as well as the definition and application of formal methods to the specification, verification and testing of protocols, distributed systems, and multimedia applications, areas in which he has authored or coauthored more than 120 international publications. He contributes to several research projects on the expression of time-constraints in formal description techniques, and the application of these techniques to the formal design of cooperative high speed interactive multimedia distributed applications. He has also participated in standardization activities, as an expert of AFNOR and ISO. He has served on several program committees of international conferences. He was the program chair of MMM '96 (MultiMedia Modeling International Conference held in Toulouse in November 1996), CFIP '2000 (Colloque Francophone sur l'Ingénierie des Protocoles held in Toulouse in October 2000), and the cochair of Tel '04 (Technology Enhanced Learning held in Toulouse in August 2004). He is a member of IEEE and ACM.

Christophe Lohr received the Diplôme d'Ingénieur and DEA in computer science from ENSEEIHT, Toulouse, in 1997 and the PhD degree in computer science in 2002 from the Institut National Polytechnique of Toulouse (INPT). He prepared the PhD degree at Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) of the Centre National de la Recherche Scientifique (CNRS), Toulouse. He is currently an invited researcher at Concordia University, Montreal. His research interests include the design of real-time systems, as well as the definition and application of formal methods for the specification, verification, and testing of real-time and distributed systems.

Pierre de Saqui-Sannes graduated in 1985 from the University Paul Sabatier of Toulouse, France. He prepared his PhD at LAAS-CNRS (Laboratoire d'Analyse et d'Architecture des Systèmes-Centre National de la Recherche Scientifique) and received the PhD degree from Université Paul Sabatier in 1990. In 1991, he spent a postdoctoral year at the University of Montreal, Canada. Since 1992, he has been an associate professor with the Department of Applied Mathematics and Computer Engineering at ENSICA (Toulouse), and an associated researcher with the Software and Tools for Communication Group at LAAS-CNRS. His research interests include modeling techniques (UML, formal methods) and their application to real-time system design and protocol engineering. He has served as a member on program committees (IWPTS '95, CFIP '00, CFIP '02, ISSADS '04) and organizing committees (MMM '96, IDMS '99, CFIP '00, MSR '01, EDSYS '04, Tel '04). He is a member of IEEE and ACM.