TURTLE-P: a UML Profile for the Formal Validation of Critical and Distributed Systems

Ludovic Apvrille[1], Pierre de Saqui-Sannes[2], Ferhat Khendek[3]

[1] GET / ENST, Laboratoire Labsoc, Institut Eurécom, 2229 route des Crêtes, B.P. 193
06904 Sophia-Antipolis Cedex, France
`Ludovic.Apvrille@telecom-paris.fr`
[2] ENSICA, Department DMI, 1 place Emile Blouin
31056 Toulouse Cedex 05, France
`desaqui@ensica.fr`
[3] Concordia University, Electrical and Computer Engineering Department,
1455 de Maisonneuve W.,
Montreal, QC, H3G 1M8, Canada
`khendek@ece.concordia.ca`

Abstract. The Timed UML and RT-LOTOS Environment, or TURTLE for short, extends UML class and activity diagrams with composition and temporal operators. TURTLE is a real-time UML profile with a formal semantics expressed in RT-LOTOS. Further, it is supported by a formal validation toolkit. This paper introduces TURTLE-P, an extended profile no longer restricted to the abstract modeling of distributed systems. Indeed, TURTLE-P addresses the concrete descriptions of communication architectures, including quality of service parameters (delay, jitter, etc.). This new profile enables co-design of hardware and software components with extended UML component and deployment diagrams. Properties of these diagrams can be evaluated and/or validated thanks to the formal semantics given in RT-LOTOS. The application of TURTLE-P is illustrated with a telecommunication satellite system.

1 Introduction

Distributed real-time and critical systems represent a real challenge for software theoreticians and practitioners. Despite their precision, and the associated validation and automatic code generation techniques, formal methods have not been widely deployed in industry. By contrast, less formal notations have rapidly gained wide acceptance and became de-facto standards. UML [20] is an example of such notations.

UML 2.0 [21], the latest OMG release of UML, does not meet all the expectations of real-time system designers. UML 2.0 has indeed a unique temporal operator (fixed delay) unable to express temporal indeterminism. UML 2.0 further lacks a formal semantics; therefore, the formal validation of UML models becomes difficult if not impossible.

Real-time UML profiles based on formal methods have been proposed in order to overcome these limitations. For instance, [2] introduces a profile named TURTLE, an acronym for '*Timed UML and RT-LOTOS Environment*'. With its formal semantics given in RT-LOTOS [5] [18] and its toolkit [27], TURTLE enables a priori detection of design errors through a combination of simulation and verification/validation techniques. By "simulation" we mean a partial exploration of the system state space. It is often used for debugging purposes and to quickly increase confidence in a design. For finite state space systems, exhaustive analysis is also possible. Verification relies on the exploration of the whole system state space in order to prove absence of deadlocks for instance and other general properties that should be satisfied by any system. Validation also relies on exhaustive analysis to demonstrate that a model meets specific requirements, or exhibits a certain behavior. We distinguish between verification and validation; we use the term "verification" for checking general properties any system should exhibit, and term "validation" for system's specific properties such as the validation of a design against the requirements.

So far, the TURTLE profile has improved UML in terms of structuring and behavioral description by revisiting both class and activity diagrams. However, this is not sufficient to make TURTLE very appropriate for the design of distributed systems. Indeed, the modeling of the low level architecture of a distributed system remains an open issue. Further, TURTLE does not bring any specific solution to express Quality of Service parameters and communications constraints in general.

This paper proposes TURTLE-P, an extended TURTLE profile with formally defined component and deployment diagrams. More particularly, some features of TURTLE, such as temporal operators and TURTLE observers are mapped to these new diagrams. Also, the paper proposes an associated methodology for distributed system design and validation.

The paper is organized as follows. Section 2 summarizes the main features of the TURTLE profile and briefly presents its toolkit. Section 3 introduces TURTLE-P and its formal semantics. The methodology associated with TURTLE-P is described in Section 4. The application of TURTLE P is illustrated in Section 5. Section 6 surveys related work, while Section 7 concludes the paper.

2 The TURTLE Profile

A UML "profile" may contain selected elements of the reference meta-model, 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, using the "stereotype" extensibility mechanism in particular. A stereotype extends the vocabulary of the UML, allowing one to create new kinds of building blocks. These blocks are derived from existing ones but are specific to a class of problems.

2.1 Overview

The TURTLE profile enables modeling and formal validation of complex real-time systems [2]. It is not intended to cover the entire life cycle of a complex system, but rather to address the high-level design and its formal analysis.

A TURTLE model includes the description of the system architecture and a set of behavioral descriptions for each entity in the architecture. Thus, the TURTLE profile extends three UML diagrams: class/object diagrams for architectural description, and activity diagrams for behavioral description. Architectural description includes the composition of entities. The concept of composition operator[1] is fundamental for TURTLE class/object diagrams (Section 2.2). It is used to represent parallelism and synchronization between objects. The dynamic aspect of the system is expressed using activity diagrams, which have been extended with synchronization actions and temporal operators. Unlike UML 2.0, TURTLE offers solutions to express temporal indeterminism and to work with temporal intervals.

Besides its syntax expressed in UML, The TURTLE profile has a formal semantics expressed by means of TURTLE to RT-LOTOS [5] translation algorithms [18]. These algorithms are implemented by TTool [27]. An RT-LOTOS specification generated by TTool can be formally validated using the RTL tool [23]. Unlike real-time UML tools, the TURTLE toolkit made of TTool and RTL is not limited to model animation: when the system is bounded and of reasonable size, RTL may generate a reachability graph. The latter may be exploited to check a TURTLE model against logical and real-time properties.

2.2 Class Diagrams

TURTLE uses UML extension capabilities to introduce three main concepts.
1. A stereotyped class named *Tclass*. A *Tclass* declares its attribute of type *Gate* separately from the other ones (see Fig. 1). An instance of a *Tclass* (called a *Tinstance*) uses exclusively gates to communicate with another *Tinstance*. The behavior of a *Tclass* is described using a TURTLE activity diagram (see next section).

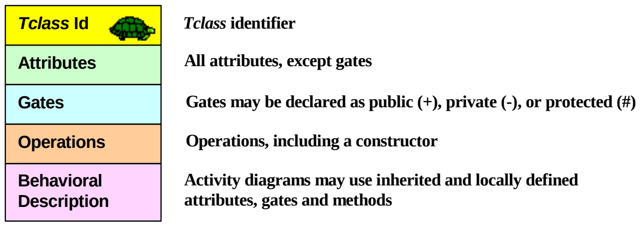
Tclass Id
Attributes
Gates
Operations
Behavioral Description
Tclass identifier
All attributes, except gates
Gates may be declared as public (+), private (-), or protected (#)
Operations, including a constructor
Activity diagrams may use inherited and locally defined attributes, gates and methods

Fig. 1. The *Tclass* stereotype. An instance of a *Tclass* is called a *Tinstance*.

2. A *Gate* abstract type specialized into *InGate* and *OutGate*. Two *Tinstances* use gates to synchronize with each other and to exchange data.

[1] Composition operators are not extensions of the diamond symbol used in UML class diagrams. We use 'composition operator' in the process algebra manner.

3. A stereotype named "*Composition Operator*". As suggested by its name, such an operator is used to compose two instances of *Tclasses* in the sense that it gives a precise meaning to a joint action between the two instances in terms of parallelism, synchronization, execution in sequence or preemption. To be rigorous, we say that a composition operator applies to a link between two *Tinstances*. In practice, TURTLE allows one to mix *Tclasses* and instances of *Tclasses* in the one diagram. Accordingly, a composition operator may also be applied to an association between two *Tclasses* if none of these *Tclasses* is instantiated more than once.
For an illustration purpose, let us consider two *Tinstances* T1 and T2 executing in parallel. An association is created between T1 and T2 and attributed with a *Parallel* composition operator (see Fig. 2 (a)). Notice that directed composition operators should attribute associations with navigation indication. In Fig. 2b, the execution in sequence of two *Tinstances* T1 and T2 requires the use of an association directed from T1 to T2, and attributed with a *Sequence*.

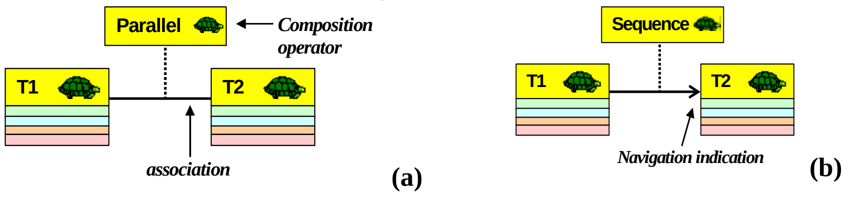

Fig. 2. Use of composition operators. (a) Parallel. (b) Sequence

Two *Tinstances* may exchange data at synchronization time. An OCL formula of the form *{Tinstance1.g1 = Tinstance2.g2}* should be attached to the association between synchronized *Tinstances*. Fig. 3, for instance, illustrates the interconnection of g1 in T1 and g2 in T2. Whenever one of the *Tinstances* performs a call on its gate, it must wait for the other *Tinstance* to make a call on the corresponding gate. Notice that data exchange is allowed during synchronization. For example, *g1!x* means that T1 will provide a value of type Natural at synchronization. T2 will synchronize with this using *g2?y* and receive the value into *y*.

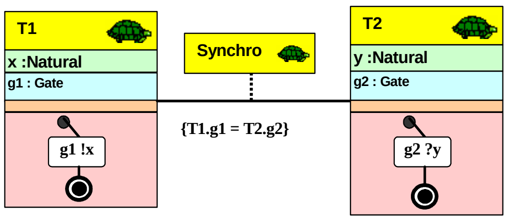

Fig. 3. Synchronization and data exchange between two Tclasses instances

2.3 Activity Diagrams

TURTLE enhances UML 1.x activity diagrams with logical and temporal operators. First, it makes it possible to express in regular UML action states synchronizations on gates with data exchange in input and/or output direction. An example is given by the pictogram on the left in Fig. 4. Other pictograms represent the temporal operators of TURTLE. Unlike many real-time extensions of UML (including UML 2.0), TURTLE enables description of temporal indeterminism using a non deterministic delay. The latter may be combined with a fixed duration to specify a time interval. TURTLE also makes it possible to limit the amount of time a *Tinstance* spends waiting for another *Tinstance* to synchronize with it. The corresponding operator is the time-limited offer operator. Last but not least, the "@" operator makes it possible to use a variable (d on Fig. 4) to store the amount of time elapsed between the offering on gate g and the effective synchronization on g.

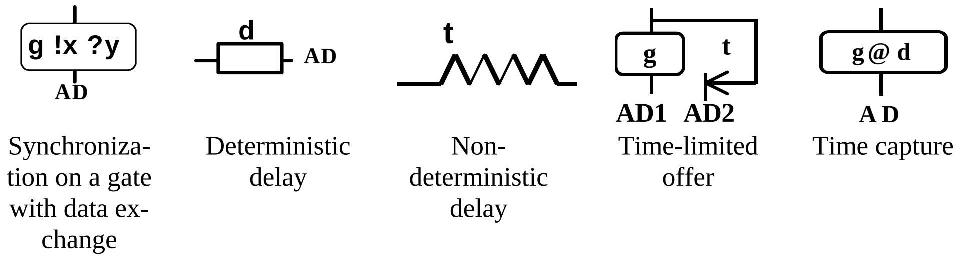

Fig. 4. Activity diagram symbols: synchronization and temporal operators

2.3 The TURTLE toolkit

The TURTLE toolkit includes TTool [27] and RTL [23].

- ***TTool***, – which stands for TURTLE Toolkit - offers a TURTLE class and activity diagram editor, a syntax checker, an RT-LOTOS code generator, and a graphical analyzer of simulation/verification results;
- ***RTL***, – which stands for RT-LOTOS Laboratory - takes as input an RT-LOTOS specification generated by TTool and performs either random[2] 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.

A system designer uses TTool to define the structural architecture of *Tinstances* and to associate activity diagrams with these *Tinstances*. First, the designer uses the simulation capabilities of the toolkit to debug the model. Then, reachability analysis for an exhaustive exploration of the model can be used. Section 4.2 discusses further the usage of the TURTLE toolkit in a complex system design methodology.

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

3 The TURTLE-P Profile

High-level designs such as TURTLE models have to be refined into more concrete designs before reaching the implementation phase. One of the major steps in this process is the identification of the components and their deployment. We are interested in defining components, deploy them, and study the properties of such deployments as early as possible in the development life cycle. Since UML deployment diagrams have no formal semantics, a formal investigation of properties of potential deployments is impossible. This section proposes to extend the TURTLE profile with formally defined component and deployment diagrams, and discusses the applications of these diagrams.

3.1 Modeling the Deployment of Software Components

UML deployment diagrams depict the “actual physical” configuration of a distributed system. Thus, a UML deployment diagram can be seen as a low-level design of a system, i.e. a design closer to the real system implementation. Software methodologies scarcely use deployment diagrams, and when they do, their role is limited to documentation. As deployment diagrams are obviously suitable for the description of concrete distributed architectures [9], we propose to extend the TURTLE profile in order to take these diagrams into account. We give them a formal semantics with the purpose to perform formal validation of low-level UML designs.

UML deployment diagram operators are:

- ***Nodes***. These nodes represent the various physical locations of the system under consideration;
- ***Software components***. According to the UML 1.5 standard, software components are deployable components that encapsulate functions and that offer interfaces to their environment. An instance of a software component can execute at a time on only one node;
- ***Communication links***. A communication link connects a node to another node, and is often stereotyped to indicate its type (e.g. <<Ethernet>>);
- ***Dependency links***. A dependency link connects two software components.

Fig. 5 depicts an UML deployment diagram where a client node is connected to a server node.

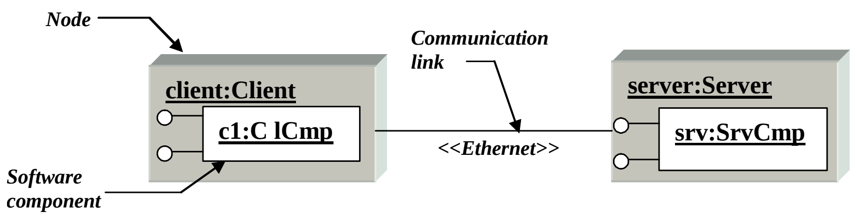

Fig. 5. Example of a UML deployment diagram

UML deployment diagrams suffer several drawbacks. First, communication links, if stereotyped, remain imprecise because no clue can be given whether, when several links are modeled between two nodes n1 and n2, which link is used for sending a message from a component on n1 to a component on n2. Second, if UML deployment diagrams have been introduced with distributed systems in mind, they do not offer any features for modeling large distributed systems, i.e. systems with a high number of nodes. At last, UML deployment diagrams are used for documentation purpose. We do think this is an important drawback since the deployment of components among nodes may introduce new errors inherent to distribution characteristics.

If UML 2.0 has introduced a composite structure diagram to address some of the drawbacks listed above, this diagram does not support communication links with various characteristics (FIFO, delay, jitter, and so on). Also, it is not possible to model physical nodes on composite structure diagrams.

Therefore, we propose the following enhancements for UML deployment diagrams:

- ***Software components*** may contain UML classes, TURTLE classes and TURTLE composition operators. They can be formally validated;
- ***A multiplicity*** can be used at node level. This makes it possible to use a single graphical node to describe several physical nodes on which different instances of the same software components run;
- ***Communication links*** connect component interfaces and not nodes. Also they can be characterized with *Quality of Service* parameters, such as transmission delay and jitter.

3.2 Software Components Used in TURTLE-P Deployment Diagrams

Definitions. A TURTLE-P software component consists of classes (regular UML classes and *Tclasses*) that execute software functionalities on the same physical location. These classes must be first defined at class diagram level. Therefore, we view TURTLE-P components as a subset of classes defined in class diagrams. Also, nodes of TURTLE-P deployment diagrams may be composed exclusively of TURTLE-P components.

At deployment diagram level, UML restricts relations between nodes to communication links. Therefore, we do not wish to allow composition operators between TURTLE-P components. Thus, TURTLE-P components are built as follows. If we see a class diagram as a 3-uple <R, T, C> with R a set of regular classes, T a set of *Tclasses* and C a set of compositions operators connecting these Tclasses, a TURTLE-P component is a 3-uple <r, t, c> with:

- r a subset of R;
- t a subset of T;
- c a subset of C such that c contains all composition operators connecting two *Tclasses* of t.

For illustration purpose, let us consider the class diagram shown in Fig. 6.

1. C_a = {C1, C3} is a TURTLE-P software component with no composition operator.
2. C_b = {C1, C2, C4} is a TURTLE-P software component with a *Synchro* composition operator between C1 and C2. C4 executes in parallel with regards to C1 and C2.

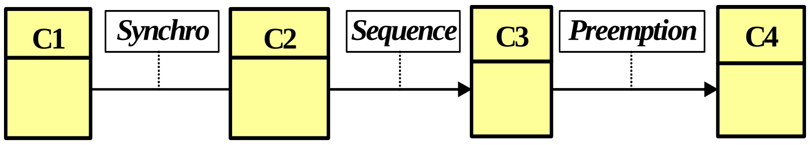

Fig. 6. Example of a TURTLE class diagram

Only *Tclasses* (and not regular classes) are taken into account at validation step. Because this paper focuses on validation issues, we assume in the rest of the paper that TURTLE-P components are exclusively composed of *Tclasses*.

TURTLE-P Component Interface. TURTLE-P software components consist of *Tclasses (or Tinstances).* A *Tclass* communicates with its environment using gates. Therefore, we view the interface of a software component as a subset of the union of the gates of the *Tclasses* it contains.

Some gates of a component may be involved in synchronization internal to the component, *i.e.* synchronization between the *Tclasses* of the component. Also, some gates can be declared as *private* or *protected*. Consequently, the interface of a software component is defined by all its public gates not involved in internal synchronizations.

We further need to distinguish between input and output interfaces. *InGate* (resp. *OutGate*) gates can be used as input (resp. output) interfaces. Conversely, attributes of type *Gate* can be considered either as input interface or output interface (but not both in the same component).

We propose to add the possibility to represent input and output interfaces on UML deployment diagrams. As an example of icon used for such interfaces, in Fig. 8, *g1_c2* is an input interface of component C_b whereas *g1_c1* is an output interface of component C_a. When possible, we also suggest putting input interfaces on the left part of the component and output interfaces on the right (not compulsory).

Modeling TURTLE-P Software Components. Component diagrams have been introduced in the UML standard with the purpose to describe software components. In TURTLE-P, any software component used in a deployment diagram must be first described in a component diagram.

At deployment diagram level, interfaces of software components connected to a communication link must be depicted. Interfaces not connected to any link may not be depicted to avoid overloading of the diagram. The interface of the gate *g* of a *Tclass c* should be named *g_c*. Also, an optional multiplicity given in a node describes the number of times all software components of this node should be deployed (*i.e.* instantiated) on distinct nodes.

As an example, let us consider the component diagram given in Fig. 7. A deployment diagram using software components previously declared in the component diagram is depicted in Fig. 8. The component diagram allows the description of the inner classes of a component, without providing semantic links between these classes.

The graphical representation of interfaces is optional since the latter are implicitly modeled at class diagram level. Indeed, interfaces are all the public *InGate* or *Outgate* gates not involved in synchronization relation (see the definition of *Component Interface*). Therefore, we suggest to limit the graphical representation of interfaces to the one connected by links. Thus, the interface *g1_C2* (gate *g1* of *TClass C2*) is depicted because it is connected to a link.

Notice that the *Client* node in Fig. 8 has a multiplicity of *n*. Therefore, the component C_a is deployed on *n* nodes of type *Client*.

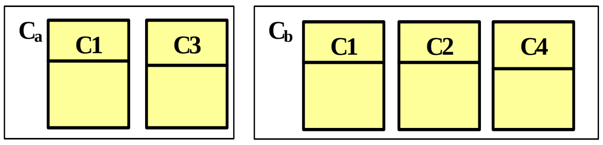

Fig. 7. Example of a TURTLE-P component diagram. Composition operators internal to TURTLE-P components are ignored (they are implicit) for space-reason.

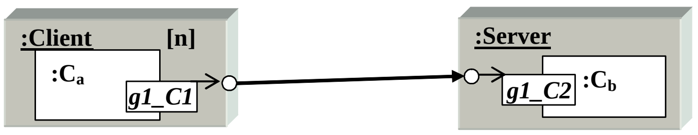

Fig. 8. Example of a TURTLE-P deployment diagram

3.3 Links in Deployment Diagrams

UML has no formal semantics, and therefore any diagram element might be considered as given for documentation purpose. This remark particularly applies to links in deployment diagrams. Since our objective is to take communication constraints between components into account during formal validation, this section provides a formal semantics to links between components.

Links are supposed to be unidirectional and asynchronous. A link is graphically represented by an association - with navigability - between the two interconnected interfaces. We attribute these associations with four additional and optional parameters given in an OCL formula:

1. A minimal transmission delay (*min_delay*),
2. A maximal transmission delay (*max_delay*),
3. A bandwidth indicated by the maximal number of messages carried at the same time on the link (*max_msg*), and
4. An average loss rate given as a percentage (*average_loss_rate*).

Fig. 9 shows a link connecting *Client1* to *Server*. The minimal transmission delay of this link is 10 units of time, and its maximal delay is 15 units of time. Also, at most 1000 messages can be carried at the same time on the link. The average message loss rate is 0.001%.

The multiplicity of 10 (*Client1*) indicates that at a maximum of 10 clients can be executed at the same time in the distributed system under design. The multiplicity at node level raises several problems to address. Let us consider a message *m* sent from an instance of *Client1* to *Server*: is *m* conveyed on the same link as that used by a message going from another instance of *Client1* to the (same) *Server*? The default behavior is duplicated link. It means that messages going from different instances to the same destination location are conveyed on different links. Such a link is qualified of "personal" link. Fig. 9 depicts a personal link from *Client1* to *Server*. This link is identified at it starting point by an unfilled circle. On the contrary, a link common to all messages sent from the same origin execution sites and leading to the same destination site is represented by a filled circle at the origin of the link, for instance the link from Client2 to Server in Fig. 9. Such a link is called a "common" link.

Consider the example in Fig. 9, the multiplicity of the node Client1 is 10, which means there is at most 10 distinct sites executing component C_a. Each C_a component communicates with Server with its own link. Conversely, all C_b components of the five *Client2* nodes use the same link to send their message to *Server*.

Moreover, a link connecting a node with a multiplicity of *n* (n > 1) to a node with a multiplicity of *m* (m>1) means that all messages sent trough this link are received by the *m* receivers nodes. At receiver side, if the interface is graphically represented with a fill-in circle, then the message is considered as duplicated at receiver side. Otherwise, it is duplicated at sending side.

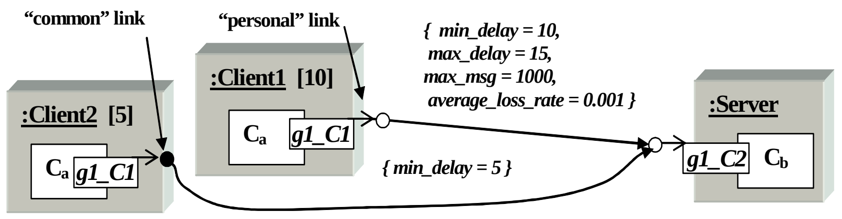

Fig. 9. Modeling a multi-client link with delay, jitter, bandwidth and loss rate

3.4 Formal Semantics of Deployment Diagrams

TURTLE has a formal semantics given in terms of RT-LOTOS [18]. The formal semantics of TURTLE-P is given by translation to TURTLE, which makes it possible to reuse algorithms in [18] after translating a TURTLE-P model into TURTLE (Fig. 10).

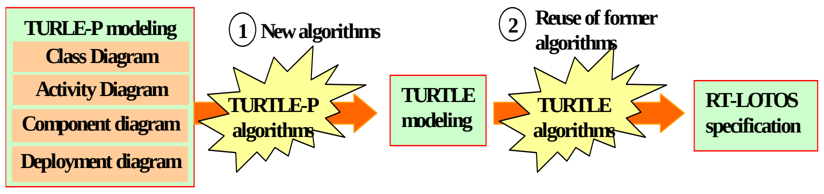

Fig. 10. TURTLE-P translation process

The principles of TURTLE-P to TURTLE translation algorithms are listed hereafter.

A TURTLE-P deployment diagram is seen as a 2-uple <*N*, *L*> with *N*, a list of nodes = {n_i, i ∊ 1..n}, and *L* a list of links = {l_i, i ∊ 1..m}, where:

1. A node is a 2-uple <*C*, *mult*> with *C* a set of software components = {c_j, j ∊ 1..n_{ij}} and *mult* the multiplicity of the node.
2. A link is a 10-uple <n_o, n_d, $g_1\_Co_1\_C_1$, $g_2\_C0_2\_C_2$, *dmin*, *dmax*, *max_msg*, *loss_rate*, *type_o*, *type_d*> where:
   - n_o and n_d denote the node at the origin and at the destination of the link, respectively,
   - $g_1\_Co_1\_C_1$ and $g_2\_Co_2\_C_2$ denote the origin and destination interfaces of the link ($g_1\_Co_1\_C_1$ = gate g_1 of component Co_1, C_1 being a *Tclass* of Co_1),
   - *dmin* and *dmax* the minimal and maximal transmission delay of the link,
   - *max_msg* and *loss_rate* the maximum number of messages at a moment on the link and the average percentage of lost messages on the link, respectively,
3. *type_o* ∊ {*personal_link*, *common_link*} and *type_d* ∊ {*dup_at_destination*, *dup_at_origin*}.

The translation algorithm is given in Fig.11. *Dc* denotes the TURTLE class diagram obtained from translation of the TURTLE-P model.

```
// Generating Tclasses of Dc
For each node $n_i$ of N, for each component $Co_j$ of $n_i$
    For k ranging from 1 to the multiplicity of $n_i$
        Classes of $Co_j$ are renamed from initial_name to ni_k_Coj_initial_name
        Classes of $Co_j$ are added to Dc
        For each composition relation cr between each classes initial_name1 and initial_name 2 of $Co_j$,
            Cr is added between ni_k_Coj_initial_name1 and ni_k_Coj_initial_name2
End For, End For, End For

// Building links
For each link $l_s$ = <$n_o$, $n_d$, $g_1\_Co_1\_C_1$, $g_2\_Co_2\_C_2$, $dmin_p$ $dmax_p$, $max\_msg_p$, $loss\_rate_p$, $type\_o_p$, $type\_d_p$> of L
    If $type\_o_p$ = personal_link then
        For k ranging from 1 to the multiplicity of $n_o$
            A Tclass $Ln_o\_g_1\_k\_c_p$ is added to Dc. The behavior of $Ln_o\_g_1\_k\_Co_1\_C_1$ models the behavior of the considered link including its transmission delay, jitter, bandwidth, loss_rate and destinations nodes (more information are provided after the algorithm)
            A Synchro relation attributes a new association between $Ln_o\_g_1\_k\_Co_1\_C_1$ and the Tclass $no\_k\_Co_1\_C_1$. The OCL formula {$g_1\_Co_1\_C_1$} is added to the association
        End For
        For k ranging from 1 to the multiplicity of $n_d$
            The gate $g_2\_Co_2\_C_2$ of $no\_k\_Co_2\_C_2$ is rename $g_2\_k\_Co_2\_C_2$
            A Synchro relation attributes a new association between and the Tclass $Lno\_g_x\_k\_Co_1\_C_1$ and the Tclass $no\_k\_Co_2\_C_2$. The OCL formula {$g_2\_k\_Co_2\_C_2$} is added to the association
        End For

    Else //common link
        A Tclass $Ln_1\_g_1\_Co_1\_C_1$ is added to Dc. The behavior of $Ln_o\_g_1\_Co_1\_C_1$ models the behavior of the considered link.
        For k ranging from 1 to the multiplicity of $n_o$
            A Synchro relation attributes a new association between $Ln_1\_g_1\_Co_1\_C_1$ and the Tclass $no\_k\_Co_1\_C_1$. The OCL formula {$g_1\_Co_1\_C_1$} is added to the association
        End For
        For k ranging from 1 to the multiplicity of $n_d$
            The gate $g_2\_Co_2\_C_2$ of $no\_k\_Co_2\_C_2$ is rename $g_2\_k\_Co_2\_C_2$
            A Synchro relation attributes a new association between and the Tclass $Ln_o\_g_1\_Co_1\_C_1$ and the Tclass $no\_k\_Co_2\_C_2$. The OCL formula {$g_2\_Co_2\_C_2$} is added to the association
End For; End If; End For
```

Fig. 11. TURTLE-P main translation algorithm

In summary, this algorithm builds a class diagram made of the following *Tclasses*:

- *Tclasses issued from a software component*. Tclasses of each instance of software components are renamed and added to the class diagram. For example, let us assume that a software component C contains two Tclasses T1 and T2, and that C runs on two different nodes n1 and n2. T1 is duplicated and renamed to *n1_1_C_T1* and *n2_1_C_T1*. The same is done with T2 with identifiers *n1_1_C_T2* and *n2_1_C_T2*. The four *Tclasses* are added to the class diagram. Moreover, if there is a composition relation between T1 and T2, then, the same composition relation is added between *n1_1_C_T1* and *n1_1_C_T2*, and between *n2_1_C_T1* and *n2_1_C_T2*.
- *Tclasses issued from link translation*. Each link of the deployment diagram is translated as a new *Tclass* T synchronizing with *Tclasses* connected to this link. The behavior of T models the characteristics of the link it represents. The following of this section addresses the automatic translation of such characteristics into TURTLE. For this, let us consider the link depicted in Fig. 12. It connects gate g1 of *Tclass* C1 to gate g2 of *Tclass* C2. *Tclass* C1 belongs to the component Co1 running on the node n1. Similarly, *Tclass* C2 belongs to component Co2 running on node n2. Our objective is to illustrate the modeling of this link for various values of *nb1*, *nb2*, and of the OCL expression *expr*.

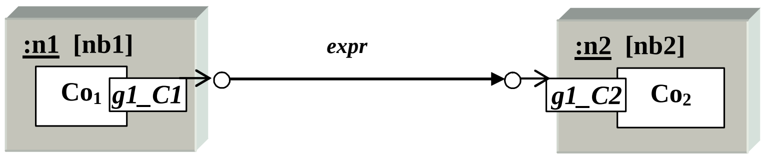

Fig. 12. Graphical modeling of a link

Minimum and maximum transmission delay

Let us assume that *expr = {min_delay = x, max_delay = y}*, and nb1 = nb2 =1. To model the transmission delay and jitter, we use a deterministic delay whose value is *x*, and a non-deterministic delay whose value is *y –x*. Fig. 13 models this link. The minimum and maximal transmission delays of the link are modeled at *Ln1_g1_Co1_C1* activity diagram (notice the two temporal operators: the deterministic delay models the minimum transmission delay; the non-deterministic delay models the time interval between the minimum delay and the maximum delays).

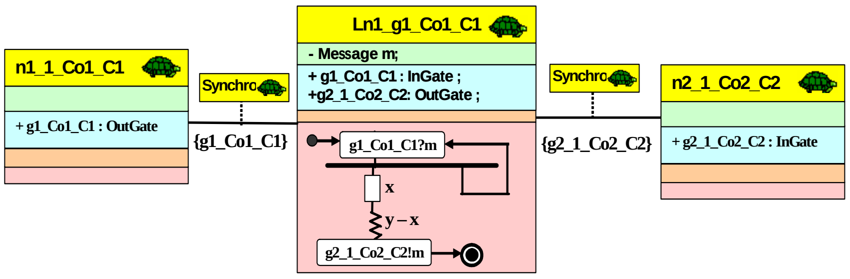

Fig. 13. Modeling the transmission delay and jitter of a link

Bandwidth

The bandwidth of a link is given in terms of a maximal number of messages carried at the same time over the link. Let us now assume that *expr = {min_delay = x, max_delay = y, max_msg = z}*. The maximal bandwidth is limited by a second activity running in parallel with regards to the activity managing message forwarding (see Fig. 14), where only the *Tclass Lg1_Co1_C1* is depicted): the message sending synchronization (synchronization on *g1_Co1_C1*) on the link can be performed only when the current number of messages being carried on the link is less than the maximal number (current number is modeled by n). When a message enters the link (additional synchronization on *msg1*), n is incremented. Conversely, n is decremented when a message exits the link (synchronization on *msg2*).

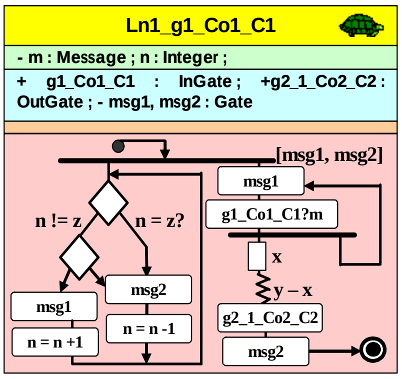

Fig. 14. Modeling the bandwidth of a link

Loss rate

The loss rate of a link is expressed as a percentage. Let us consider the example of a link with a loss rate *l*. The corresponding OCL formula is *{average_loss_rate = l}*. The main idea of the modeling (not provided here for space reason) is to drop in a non-deterministic way one message for each sequence of *100 * l* messages.

Multiplicity

We do not provide translation schemes for all possible multiplicities.

The multiplicity of a node at the origin of a link is taken into account in the general algorithm described previously. For example, consider nb1 = 2. If the link is a *common* link, then, the algorithm generates the class diagram depicted in Fig. 15. Both *Tclasses n1_1_C01_C1* and *n1_2_C01_C1* synchronize with *Ln1_g1_Co1_C1*. The latter makes no difference between messages sent by one or the other. Each message is forwarded to *Tclass n1_1_Co2_C2*.

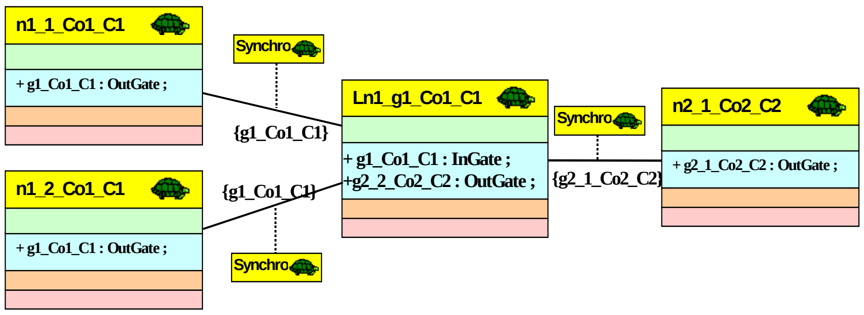

Fig. 15. Translation of a common link

Conversely, if the link is a *personal* link, two *Ln1_g1 Tclasses* are generated (one for each personal link). They both synchronize with *n2_1_Co2_C2*.

For destination nodes whose multiplicity is greater than 1: we consider nb2 = 2 (and nb1 = 0) and expr = <<*min_delay* = *x*, *max_delay* = *y*>>. A message sent on such a link can be duplicated either at the origin of the link (*type_d* = *dup_at_origin)*, or at the destination of the link (*type_d* = *dup_at_destination)*.

In the case of duplication at the origin, the message is duplicated right after the *Tclass* Ln1_g1_Co1_C1 has performed the synchronization on *g1_Co1_C1* (see Fig. 14): when *Ln1_g1_Co1_C1* gets a message from class *n1_Co1_C1* (synchronization on *g1_Co1_C1*), *Ln1_g1_Co1_C1* duplicates the message and executes two different activities that model the transmission of the message on two different links.

If duplication occurs at destination, then, *Ln1_g1_Co1_C1* executes only one activity after receiving a message from *n1_Co1_C1*, but executes in parallel two activities: the first one synchronizes on gate *n2_1_Co2_C2* and the second one on gate *n2_2_Co2_C2*.

4. Methodology

The modeling and profile presented in the previous sections are used as a formal framework for a stepwise design and validation methodology for real-time distributed systems. This methodology consists of three phases as shown in Fig.16, starting from user requirements captured and validated in terms of scenarios, and ending with components and deployment diagrams and their properties evaluation and validation.

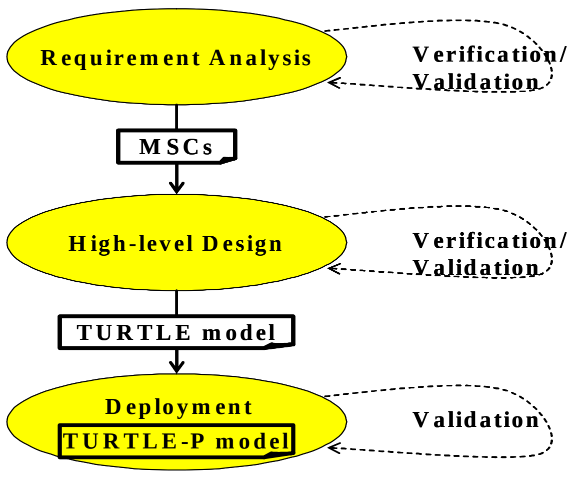

Fig. 16. Overall Methodology

4.1 Requirement Analysis

As most of today’s software methodologies we start by capturing functional requirements in terms of behavioral scenarios using Message Sequence Charts (MSC-2000) [19]. Basic MSC (bMSC) allows for the description of simple scenarios like MSC M1 and MSC M2 in Fig. 17. These bMSCs can be combined into a more complex specification using high-level MSC (HMSC) construct for sequential, alternative and parallel composition as shown in the HMSC in Fig. 17.a.
The MSC specification is used to capture the interactions between the logical components of the system. Real-time constraints can be specified as absolute or relative time constraints between events (cf. the MSC M1 in Fig. 17.b). The time constraints can also be used to limit the execution time of a given scenario as in Fig. 17.c. The MSC specifications can be checked for logical and time consistency. For instance, they can be checked for potential deadlocks, or inconsistencies between the various time constraints. In [30], we propose a set of algorithms for verifying the consistency of MSCs. In general this problem is NP-Complete. However, we have characterized a set of useful MSCs where these algorithms can perform efficiently.

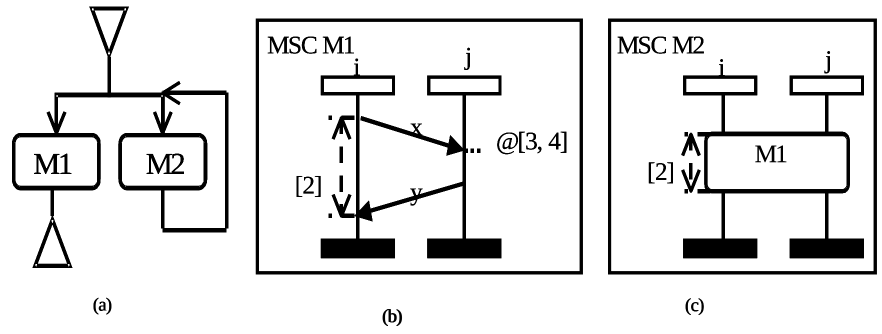

Fig. 17. MSC Examples

MSC are verified and also validated by the user as they will be used as a reference specification in the following phases. Notice that our work so far has focused on using MSC-2000, however, the interaction and sequence diagrams of UML 2.0 together have the same expression power and can be used as well. Indeed the development of these UML 2.0 sequence and interaction diagrams has been influenced by the MSC-2000 language [10], which on the other hand has a formal semantics.

4.2 High Level Design

Once requirements have been captured with MSCs, verified and validated, the high-level design starts with the construction of class and activity diagrams in TURTLE, which as mentioned previously has a formal semantics expressed in RT-LOTOS [5]. The MSC specification of Phase 1 is used as input for the design of the TURTLE model, which are translated into an RT-LOTOS specification [18], then verified and validated.

Fig. 18 depicts the complete translation and verification process for TURTLE models. The translation process takes as input TURTLE classes, and generates an RT-LOTOS specification. The non TURTLE classes are ignored during this translation. They are used for documentation purpose only.

For formal verification/validation, we reuse RTL, the Real-Time LOTOS Laboratory developed by LAAS-CNRS. RTL offers two possibilities: (1) Simulation, which partially explores a possibly non-deterministic system and outputs traces of events occurring on process gates; (2) Exhaustive analysis, which generates a reachability graph when the state space is finite.

A reachability graph generated from a LOTOS specification contains a set of states connected with a set of labeled transitions. A transition between two states may involve a synchronization action between two *Tclasses*. In this case, the transition is labeled by an identifier corresponding[3] to one of the gates involved in the synchronization.

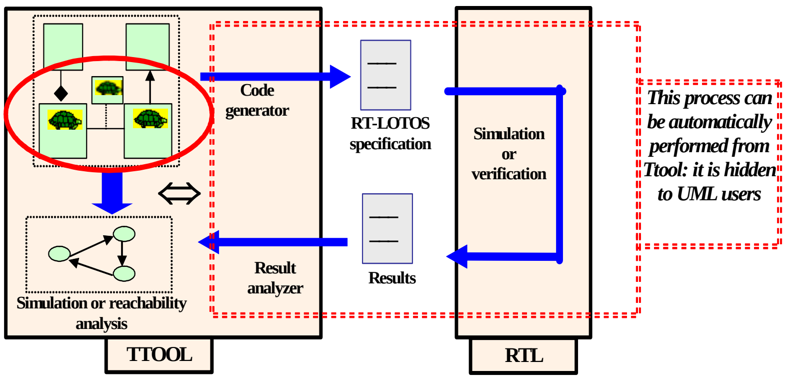
Code generator
RT-LOTOS specification
Simulation or verification
This process can be automatically performed from Ttool: it is hidden to UML users
Result analyzer
Results
Simulation or reachability analysis
TTOOL
RTL

Fig. 18. The TURTLE formal verification/validation process

The purpose of generating a reachability graph is to verify general properties, such as the absence of deadlocks, but also to validate properties specific to the system under consideration. Two complementary techniques can be used: model checking using logical formulas or the observers technique.

We have so far used the model checker Kronos [16], which takes as input a logical formula and the reachability graph, and outputs a verdict of yes/no depending on whether the property is satisfied or not by the model.

On the other hand, an observer is a module external to the system under consideration. It is described in the same language as the system itself. An observer can access the system components, such as its variables or message queues when applicable. In the TURTLE context, an observer can synchronize with a *Tclass* on a dedicated gate so that the observer remains non intrusive. In our methodology, we propose the usage of observers, as they can be part of the TURTLE-P design. For illustration purpose, let us consider a *Tclass* O (Observer) that has to obtain data from an observed *Tclass* T. To model data retrieval, we use a *Synchronization* composition operator between O and T. O can always perform this synchronization when M is ready to do it (non-intrusiveness property).

Observers should also report on property violation during the formal validation process. For each observer, we introduce *error*, a special synchronization action that is executed each time a property is violated. The *error* action is afterwards easy to identify in the reachability graph. For easier property identification, the validation process can be stopped whenever such an *error* action occurs.

Fig. 19 describes an observer analyzing logical and real-time constraints of another class. The two observed properties are:

[3] 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.

1. Property 1 (logical constraint): the 2k+1 and 2(k+1) integer values received by T on gate *ge1* should be identical.
2. Property 2 (temporal constraint): no more than t time units should elapse before two synchronizations on gate *ge2*.

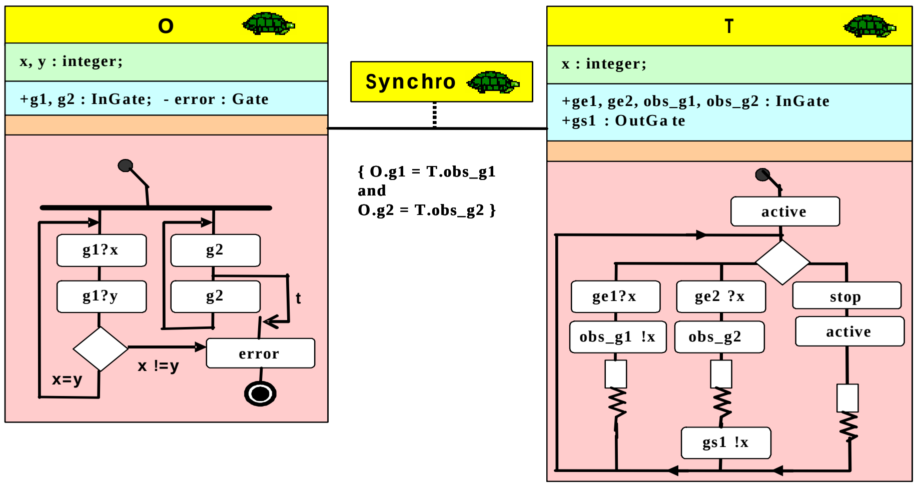

Fig. 19. Observing logical and temporal properties of a *Tclass* T

Observer O analyzes the two properties by getting data from module T. Therefore, there is a *Synchro* operator between O and T. The gates involved in the synchronization are listed in an OCL formula: *obs_g1* and *obs_g2* are connected to *g1* and *g2*, respectively. Gate *obs_g1* is used for checking property 1 and gate *obs_g2* is used for checking property 2. The two properties are checked in parallel (parallel behavior operator in O). As long as both properties remain true, actions *g1* and *g2* are always offered, which means that the observer is not intrusive. As soon as one property becomes false, action *error* is performed and the observer is stopped (its process is terminated). Once O is stopped, T will be stopped (but not terminated) the next time it executes action *obs_g1* or *obs_g2*.

More advanced behavioral observers can be derived from the MSC specification obtained from the requirement analysis phase. As a future work, we are planning to use the MSCs obtained from Phase 1 directly as observers to validate the TURTLE model in the same way as it is done today in the ObjectGeode tool [28].

The proposed methodology does not include an automatic translation from Phase 1 to Phase 2. However, techniques for translating scenarios into design specifications modeled as finite state machines [1] can be adapted very easily and this is another objective of our work in the future. This automatic translation will eliminate the need for validating the TURTLE model against the MSCs.

4.3 Lower Level Design (Deployment)

The low level design in our methodology consists of identifying nodes, specifying components as well as communication links and their characteristics. Once component and deployment diagrams have been identified, we can verify and validate some low level properties of the system at a still an early stage of the design. Our goal in this phase is to obtain very early a potential deployment model of the system and check some properties. The behaviors of the Tclasses obtained from Phase 2 are not refined further.

The translation process given in Fig. 10 makes it possible to obtain a reachability graph from all TURTLE-P diagrams, including component and deployment diagrams.

It is the task of the designer to build the software components of the deployment diagram, and validate properties of the model using observers. Moreover, a low-level design contains not only software components, but also links that also have to be observed. While the observation of a software component can be done in the same way as a *Tclass*, *i.e.* with non-intrusive observers (see previous subsection) included in software components, observing a link is not that easy as discussed hereafter.

Observing links. In TURTLE, Observers are modeled at the class diagram level. TURTLE-P takes into account deployment diagrams including nodes, links and software components. It is possible to observe a given link with usual TURTLE observers. The link to observe connects two TURTLE components: in these two components, it is possible to include *Tclasses* that plays the role of observers. To observe the link, these *Tclasses* gather information in their respective software components (sending and receiving of message on the link), and then exchange data by a no-delay, infinite bandwidth and loss-free link (modeled at deployment diagram level) between them.

Thus, even if the observer technique can be used to observe properties related to links, it is quite complex and overloads the diagrams (additional links, etc.). Therefore, we propose to introduce the concept of probes.

Modeling Probes in TURTLE-P. A *probe* usually refers to a component in charge of observing, in a non-intrusive way, a communication medium. In TURTLE-P, we introduce *probes* to observe properties related to links. A *probe* is defined as a stereotyped *Tclass (*called *Probe).* Each particular probe of the system must be defined as a *Tclass,* which inherits from the *Probe Tclass* (a probe may be identified with the <<probe stereotype>>). All *Tclasses* inheriting from *Probe* must be declared and modeled at class diagram level. If declared, they can be used at deployment diagram as follows. Let us assume that P is a <<Probe>> *Tclass*. P can be used as a probe for as many links as desired at deployment diagram level: the class must simply be used as an associative class for the considered link (an example is provided later on in the section). Note that only one probe can be attached to a link.

Several gates are declared *protected* in the *Probe Tclass*: *send*, *receive*, and *lost.* They can be used as desired by probes to observe a link. Gates used for observation must be listed in an OCL formula along with the observed link.

For example, let us model a probe, which computes the number of messages being carried on a free-loss link. If this number exceeds *max_msg*, then, the probe executes an *error* action. The class diagram with the probe is given in Fig. 20. The probe *myProbe* is attached to the link between C1 and C2 (see Fig. 20). Also, the OCL formula {send, receive} has been attached to the link. It lists the gates used by *myProbe* to observe the link. Note O1 and O2 have been totally removed from the class and component diagrams. Also, the *Tclass myProbe* doe not belong to any component because it is not executed on a particular node.

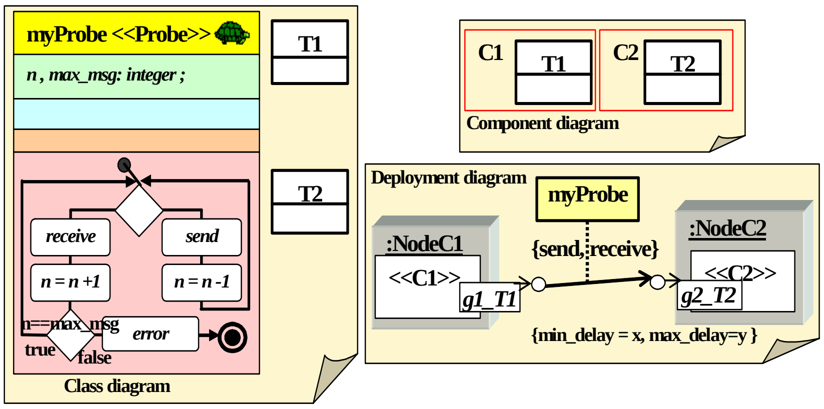

Fig. 20. Modeling of probes at class, component and deployment diagram level

Finally, the use of probes instead of observers makes TURTLE-P diagrams simpler because only one *Tclass* is used to observe a link while two *Tclasses* are requested with the Observer technique. Diagrams become more readable because the probe can be attached to the observed link on the deployment diagram (observers observing links were given at class diagram level only).

Formal Semantics of Probes. To observe links, probes use observation gates in the same way as TURTLE observers. As a consequence, the translation of probes into TURTLE is as follows (we consider a link *l* on which is attached a probe p.)

1. First, as explained in section 3, a Tclass – that we call here *Tl* – is generated to represent the behavior of the link;
2. The probe p modeled at deployment diagram level is translated as a new *Tclass Tp* in the generated TURTLE class diagram;
3. On the TURTLE class diagram, an association attributed with a synchronization composition operator connects *Tl* and *Tc*;
4. An OCL formula listing the gates is attached to the synchronization relations between probes and links; If the OCL formula attached to the probe at deployment diagram level is {gi}, the OCL formula at class diagram level is {*Tl*.gi = *Tc*.gi};
5. Finally, all the synchronization actions are modified at *Tl* behavior diagram as follows. If the *send* gate $\in$ gi, then, every action on an *OutGate* is followed by

send. Similarly, actions on InGate are followed by *receive*. Also, lost actions (modeled by internal actions *i*) are followed by an action on the gate *lost*.

Let us come back to the example in Fig. 20. The TURTLE class diagram generated from this TURTLE-P model is given in Fig. 21. The probe is modeled as a class (*P_LnodeC1_g1_C1_T1*), which synchronizes with the class representing the link. Notice the *receive* and *send* actions that have been added to the regular behavior diagram of a link with delay and jitter of Fig. 13.

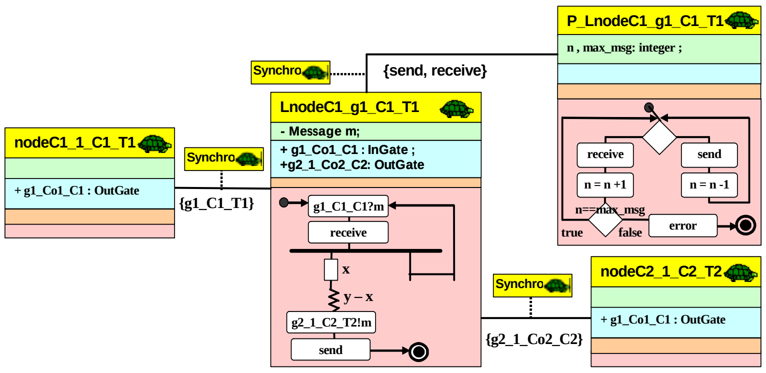

Fig. 21. Translation of a probe: generated TURTLE class diagram

5 Case Study

This case study illustrates the use of TURTLE-P, and particularly the use of deployment diagrams, for the modeling and validation of a space-based telecommunication system.

5.1 System Description

SAGAM [22] addresses the evaluation of an ATM switch embedded in a telecommunication satellite. This switch allows for routing ATM cells from a spot to a different one.

A user connection can be established as follows. When a user requires a new connection to be set up, a CAC algorithm evaluates if the system can meet required connection parameters, for example in terms of bandwidth. If so, the connection is accepted. Once its connection is established, a user can periodically insert its ATM cell in the slots of the uplink of its spot. To allocate to each user the exact amount of required slots (we address Variable Bit Rate traffic, that is, a variable bandwidth is required), users can send Dama-sig signals to the system. A Dama-sig signal informs the system that a user wants to emit at its Peak Cell Rate (ATM connection called VBR-PCR). The system is responsible for managing all Dama-sig signals: it must allocate the uplink bandwidth among all users according to all Dama-sig requests with fairness. A frame allocation report, regularly sent to users, makes a relation between slots and users. In this case study, we propose to model the whole allocation of spots in this telecommunication system.

The allocation is performed by three distinct entities: at User Earth Station (UES), in the satellite, and at last, in the satellite Network Center (NC, see Fig. 22).

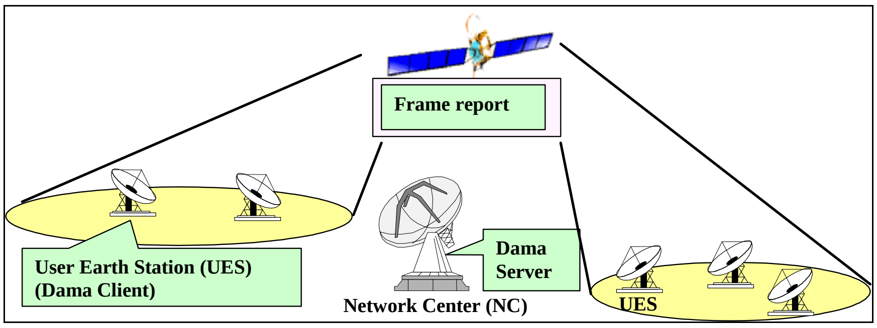

Fig. 22. System overview

At UES, a software entity (Dama-client) sends Dama-sig and listens for the allocated slots in the frame allocation report. Dama-sigs are conveyed through the satellite system to the software entity called Dama-server located at the NC. The dama-server evaluates all user requests, and computes new allocations. At last, a software entity embedded in the satellite receives allocation modifications from the NC, and sends a frame allocation report in each slot every 50 ms.

5.2 Modeling with TURTLE-P

The system under consideration has been modeled and validated using the methodology described in Section 4. We have developed:

1. MSCs representing various scenarios of the SAGAM system. This includes the scenario for sending a Dama-sig to the system, and the one for listening to the frame allocation report.
2. The system class diagram [2]; this includes the modeling of all *Tclasses* of the system, and also the modeling of observers dedicated to the observation of properties regarding the *Tclasses*. More particularly, we have modeled classes in charge of DAMA algorithms performed at user and server level, and the classes responsible for sending and receiving frame allocation reports.
3. The low-level design *i.e.* we have first extracted software components from the class diagram, and described them using a component diagram. In the following, we focus on the modeling and validation performed at this step.

   The components extracted from the class diagram are:

- Damaclient made out of two *Tclasses* Dc and *Traffic* in charge of DAMA algorithms and of the traffic generated at UES, respectively;
- DamaServer made out of three *Tclasses* Dsr, DamaAlgo, Dse which respectively receive Dama-sigs, compute them, and answer to them;
- FAR made out of the Tclass Farm (for Frame Allocation Report Manager) which computes new allocations sent by DamaServer, and of the Tclass Fars which periodically sends a frame allocation report to users.

Once developed, software components have been distributed over nodes with a TURTLE-P deployment diagram (see Fig. 23). Three links are represented in this deployment diagram: they connect together the three software components deployed over three different nodes. UES has a multiplicity of *n* that will be discussed in the following sub-section. Notice that the link between UES and NC has a delay equal to 250 ms, because the signal goes up to the satellite and comes down to the NC.

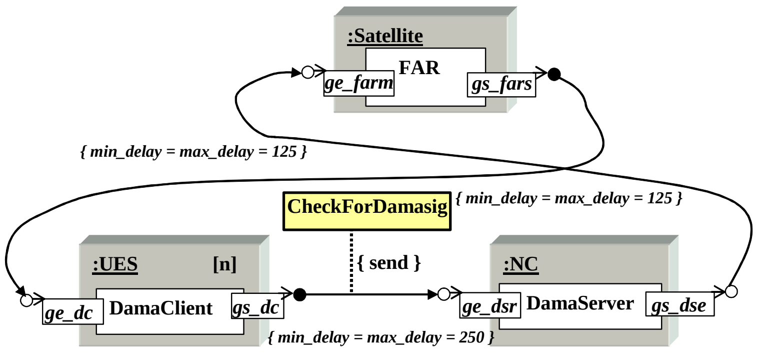

Fig. 23. Deployment diagram for the system

5.3 Deployment and Evaluation of Properties

Before investigating the properties for the TURTLE-P model, we have added probes to the model. For example, we have modeled a probe *CheckForDamasig* for checking that for a given period of 50 ms, the number of computed Dama-sig is equal to the number of users. The class declaration and the behavioral diagram of this probe are given in Fig. 24. The *CheckForDamasig* probe is designed and then added to the TURTLE class diagram, before it can be used at deployment diagram level: the link between *UES* and *NC* is attributed with this probe.

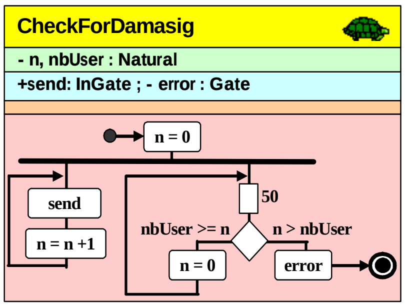

Fig. 24. The *CheckForDamasig* probe modeled at class diagram level

Using the process depicted in Fig. 10, we generated an RT-LOTOS specification and provided it as input to RTL. In the case of SAGAM, we performed reachability analysis for various values of the multiplicity n (the higher is n, the more representative the system is, the real system having potentially thousands of clients). For low values (1 to 5), we generated a reachability graph with a reasonable number of states (from 195 states for n=1 to almost 50000 states for n=3). For n greater than 5, we switched to simulation.

The property checked by the probe appeared to be true for all values of n experimented (no *error* transition in the reachability graph). Also, for values of n greater than 50, the system started to work less efficiently i.e. all dama-sig could not necessarily be computed before the next frame allocation report (*i.e.* the transition demonstrating the receive of their allocation sometimes happened after the receiving of the frame allocation report). As a consequence, all users could not be given the requested bandwidth as quickly as it is for low values of n. Therefore, the simulation made it possible to evaluate the impact of the number of nodes on the performance of the system in terms of response time to users. To evaluate this response time, we added an observer, synchronized with the *Tclass Dc*. This observer calculates the time between the sending of a dama-sig, and the receiving of the requested bandwidth. A specific transition *response_time* performed by this observer made it possible to obtain information about the performance of the system. Simulations traces performed with different values of the time of computation of dama algorithms demonstrated the negative impact of this algorithm on the performance of the response time to users when n increases.

5.4 Lessons Learned

This section summarizes results about the use of component and deployment diagrams for the modeling and validation of an industrial embedded and distributed real-time system.

First, in terms of distributed modeling capabilities, TURTLE-P provides a framework for the modeling of all system's execution locations. Also, communication constraints between these locations can be explicitly modeled, contrarily to UML links which lack semantics. Also, because deployed components are composed of

TURTLE classes, the system can be modeled using TURTLE composition and behavior operators, which were previously successfully experimented for the modeling of critical systems. Thus, TURTLE-P is well suited for the modeling of both distributed and critical systems. SAGAM is an example of a critical and distributed system, which has been successfully modeled and validated using TURTLE-P.

Second, in terms of validation, the number of nodes has a great impact on the size of the reachability graph. To reduce possible combinatory explosion, two techniques were used. The first one consists of increasing the abstraction of the system at class diagram level: we limited as much as possible the use of variables and non-deterministic delays. The second one consists of making abstraction at deployment diagram level: decrease the number of nodes (suppression of nodes of lower importance) and weakening of node constraints.

Finally, the probe technique we have used was a simple way to allow for properties validation. Probes are modeled using TURTLE classes, which are distinct from system tasks. Thus, the system software architecture remains unmodified. The use of the synchronization TURTLE operator at translation stage makes it possible to model non-intrusive probes. The use of unique action identifier (*error* labels) makes it simple to identify property violation in the reachability graph.

6 Related Work

[2] compares the TURTLE profile with UML 2.0 and various real-time UML profiles based on formal methods. Because of lack of space, this section will only focus on the contributions introduced by the TURTLE-P extension, in particular its applicability in the context of critical and distributed systems.

Critical Systems. For the modeling of critical systems, two types of operators have been explored [4] [6] [8] [24] [26]: logical operators (mostly operators for composing UML classes), and temporal operators (inner behavior of UML classes).

Surveyed papers and tool documentations indicate that class composition usually relies on asynchronous communications [4] [6] [8] [24]. Further, composition is in most cases hidden and implicit (cf., e.g. the Rose RT [24] and Tau G2 [26] tools). Conversely, TURTLE-P makes composition explicit inside components via attributed associations between *Tclasses*, and outside components via formally defined links in deployment diagrams.

It is further essential that modeling languages dedicated to critical systems may offer temporal operators enabling the description of time intervals. To address this issue, TURTLE-P extends TURTLE with real-time based quality of service parameters making it more suitable for the modeling of time intervals, variable delays, jitter and other features common to timed constrained systems, such as networked multimedia systems.

Distributed Systems. A distributed architecture can be validated once the protocols it relies on have been designed. Protocol modeling with UML has been studied at the very beginning of the UML standardization process at the OMG. Since, several studied have been published, first, with academic examples [12] [13] and then, with protocols subjacent to systems such as cooperative systems [7], electronic commerce systems [25], multi-agents systems [15] [17], and ODP-based systems [3] [11].

Contributions on UML for distributed systems generally focuses on the enhancement of UML for the description of interactions between the components, and the communication constraints between these components [14] [29]. For example, AUML [29] introduces *Protocol diagrams*, which extend sequence diagrams with logical operators to explicitly express causality, synchronization and multicasting. For describing security constraints in distributed systems, [14] extends UML deployment diagrams with stereotypes associated with links. TURTLE-P follows the two previous approaches; indeed, it uses sequence diagrams for the specification of interactions, and proposes an enhanced low-level design of the distributed system by means of valued links. TURTLE-P also offers formal validation at each step of the development process.

UML 2.0. UML 2.0 has been designed with real-time systems in mind, and particularly applies to protocol modeling and communication architecture validation.

UML 2.0 introduces a new diagram called composite structure diagram. This diagram is particularly well suited to protocol modeling because classes may have ports that can be interconnected by communication channels. Unfortunately, these diagrams do not model physical nodes explicitly. Conversely, the TURTLE-P profile gathers on the same diagrams the interconnection of software components, and the deployment of these components.

TURTLE-P further offers formal verification and validation functionalities that are unmatched by code animators available with recently released UML 2.0 tools.

7 Conclusion

The TURTLE profile enables modeling and formal validation of critical software applications [2]. TURTLE extends UML 1.5 in terms of structuring and behavioral description. Class diagrams include composition operators, and activity diagrams include synchronization actions as well as temporal operators (deterministic delay, non-deterministic delay and time-limited offer). The profile has a formal semantics given in RT-LOTOS. It is supported by a toolkit including a diagram editor [27], an RT-LOTOS code generator [27], and a formal validation tool [23].
Whether the benefits of using TURTLE have been demonstrated on various time-critical systems, its application to distributed systems has been limited. For instance, the profile did not offer the possibility to take concrete distribution of software components into account. The TURTLE-P profile proposed in this paper intends to overcome this limitation. This profile definition includes a three phases methodology: (1) the requirement analysis phase uses Message Sequence Charts, which can be checked

for consistency and validated; (2) the high-level design reuses TURTLE; (3) the low-level design phase develops UML component and deployment diagrams, and makes it possible to check deployment properties and constraints of critical distributed systems at a still an early stage of development. Indeed, the strength of TURTLE-P resides in the formal definition of its diagrams and operators. The formal semantics is given in RT-LOTOS. The use of TURTLE-P and its associated methodology is illustrated using a satellite system.

Future work will focus on two different points. One of our objectives is to construct *Tclass* activity diagrams automatically from MSCs defined during the first step, as it is already done for the Specification and Description Language (SDL) behaviors in [1]. MSCs can also be used more explicitly during the validation phase of the second and the third step of the methodology, in the same way as in ObjectGeode tool [28]. The second objective is to enhance the characteristics of the communication model at the deployment diagrams level. In particular, more advanced traffic and loss-rate schema will be defined.

8 Acknowledgments

The TURTLE profile has been defined in a joint effort involving Patrick Sénac from ENSICA, Jean-Pierre Courtiat and Christophe Lohr from LAAS/CNRS, Ludovic Apvrille and Pierre de Saqui-Sannes.

Also, we would like to thank Michel Diaz (LAAS-CNRS) for the fruitful ideas he has suggested us throughout the TURTLE project.

The work on TURTLE-P has been partially supported by the Concordia Research Chair in Telecommunications Software Engineering

9 References

1. M. Abdalla, F. Khendek and G. Butler, "New Results on Deriving SDL Specification from MSCs", Proceedings of SDL Forum'99, Montreal, Canada, June 1999
2. L. Apvrille, J.-P. Courtiat, C. Lohr, P. de Saqui-Sannes, "TURTLE : A Real-Time UML Profile Supported by a Formal Validation Toolkit", IEEE Transactions on Software Engineering, Vol. 30, No. 7, July 2004
3. M. Born, E. Holz, O. Kath, "A Method for the Design and Development of Distributed Applications Using UML", TOOLS-Pacific 2000, Sydney, Australia
4. R.G. Clarck, A.M.D. Moreira, "Use of E-LOTOS in Adding Formality to UML", *Journal of Universal Computer Science*, Vol.6, No 11, p. 1071-1087, 2000.
5. J.-P. Courtiat, C.A.S. Santos, C. Lohr, B., Outtaj "Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique", Computer Communications, vol. 23, n. 12, 2000, p. 1104-1123
6. B.P. Douglass, "Real-Time UML Tutorial", OMG Real-Time and Embedded Distributed Object Computing Workshop, Arlington, VA, USA, July 2002
7. J. M. Espinosa, O. Nabuco, K. Drira, "A UML Model for Session Management in Collaborative Design for Space Activities", 8th European Concurrent Engineering Conference (ECEC'2001), Valence (Espagne), pp.170-174, April 2001

8. S. Gerard, F. Terrier, Y. Tanguy, “Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML” In J.-M. Bruel, Z. Bellahsene (Eds.): Proceedings of the Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, LNCS 2426, pp. 260-269
9. H. Gomaa, “Designing Concurrent, Distributed and Real-Time Systems with UML”, Addison Wesley, ISBN 0201657937, 2000
10. O. Haugen, “Comparing UML 2.0 Interactions and MSC-2000”, SDL and MSC Workshop, (SAM’04), Ottawa, Canada, June 1-4, 2004
11. M.-P. Huget, “Extending Agent UML Protocol Diagrams”, Agent Oriented Software Engineering (AOSE-02), Fausto Giunchiglia and James Odell and Gerhard Weiss (eds.), Bologna, Italy, July 2002.
12. C. Jard, J.-M. Jézéquel, F. Pennaneach, “Towards using protocol validation tools in UML”, Techniques et Sciences Informatiques, Vol 15, No 11, pp. 1-15, 1998 (in French)
13. M. Jaragh, K.A. Saleh, “Modeling Communications protocols using the Unified Modeling Language”, TENCON’2000, Intelligent Systems and Technologies for the New Millenium, Kuala Lumpur, Malaysia, September 2000.
14. J. Jürjens, “UMLsec: Extending UML for Secure Systems Development”, UML 2002, Dresden, Sept. 30 - Oct. 4, 2002, LNCS 2460
15. K. Kavi, D.C. Kung, H. Bhaambhani, G. Pancholi, M. Kanikarla, R. Sah, “Extending UML to Modeling and Design of Multi-Agent Systems”; Workshop on Software Engineering for Large Multi-Agents Systems, associated with ICSE’2003.
16. Kronos. http://www-verimag.imag.fr/TEMPORISE/kronos
17. J. Lind, “Specifying Agent Interaction Protocols with Standard UML”, Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), LNCS 2222
18. C. Lohr, “Contribution to real-time system specification relying on the RT-LOTOS formal description technique”, PhD Thesis, Institut National Polytechnique de Toulouse, December 2002 (in French)
19. ITU-T, “Message Sequence Charts: MSC-2000”, Z.120, Geneva, Nov. 1999
20. Object Management Group, “Unified Modeling Language Specification”, Version 1.5, http://www.omg.org/docs/formal/03-03-01.pdf, March 2003
21. Object Management Group, “UML 2.0 Superstructure Specification”, http://www.omg.org/docs/ptc/03-08-02.pdf
22. L. Roullet, “SAGAM Demonstrator of a G.E.O. Satellite Multimedia Access System: Architecture & Integrated Resource Manager”, European Conference on Satellite Communication, Toulouse, France, November 1999.
23. http://www.laas.fr/RT-LOTOS
24. B. Selic, « A UML Profile for Modeling Complex Real-Time Architectures », http://www.omg.org/news/meetings/workshops/presentations/realtime2001/6-3%20Selic.presentation.pdf.
25. I.W. Siu, Z .S. Guo, “The Secure Communication Protocol for Electronic Ticket Management System”, University of Macau, June 2001
26. http://www.telelogic.com
27. http://www.eurecom.fr/~apvrille/TURTLE/index.html
28. Verilog, “ObjectGeode”, Toulouse, France, 1999.
29. J. Wei, S.C. Cheung, X. Wang, “Exploiting Automatic Analysis of E-Commerce Protocols”, 25th Annual Computer Software and Application Conference, Chicago, USA, October 2001.
30. T. Zheng, F. Khendek, “Time Consistency of MSC-2000 Specifications”, Computer Networks, Elsevier, Vol. 42, No. 3, pp. 303-322, June 2003.