

Verifying Service Continuity in a Dynamic Reconfiguration Procedure: Application to a Satellite System

L. APVRILLE

apvrille@ece.concordia.ca

GET/ENST/COMELEC/Lab SoC, Institut Eurecom BP 193, 2229 route des crêtes, 06904 Sophia, Antipolis Cedex, France

P. de SAQUI-SANNES

desaqui@ensica.fr

P. SÉNAC

senac@ensica.fr

ENSICA, 1 place Emile Blouin, 31056 Toulouse Cedex 05, France; LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France

C. LOHR

lohr@laas.fr

LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France; Concordia University, Electrical and Computer Engineering Department, 1455 de Maisonneuve W., Montreal, QC, H3G 1M8, Canada

Abstract. The paper discusses the use of the TURTLE UML profile to model and verify service continuity during dynamic reconfiguration of embedded software, and space-based telecommunication software in particular. TURTLE extends UML class diagrams with composition operators, and activity diagrams with temporal operators. Translating TURTLE to the formal description technique RT-LOTOS gives the profile a formal semantics and makes it possible to reuse verification techniques implemented by the RTL, the RT-LOTOS toolkit developed at LAAS-CNRS. The paper proposes a modeling and formal validation methodology based on TURTLE and RTL, and discusses its application to a payload software application in charge of an embedded packet switch. The paper demonstrates the benefits of using TURTLE to prove service continuity for dynamic reconfiguration of embedded software.

Keywords: dynamic reconfiguration, real-time UML, RT-LOTOS, formal validation, satellite

1. Introduction

Formerly limited to signal processing, satellite payloads nowadays perform cell switching and dynamic multiplexing. Consequently, they request heavier network signaling and more complex software support. The complexity in building and maintaining such systems is increased by the fact that multimedia data streams handled by payloads evolve in nature throughout satellite's lifetime (a fifteen year average). Two avenues have been explored to answer this problem. The first solution corresponds to the active networking paradigm (Chen, 2000): a programming code embedded in data streams implements a per-user or per stream network customization. In the second solution, a satellite operation center performs regular dynamic reconfiguration on the embedded software (Boutry, 2000).

The paper addresses the second solution, in particular the dynamic reconfiguration of embedded and software-implemented network functions. The problem to be solved is service

continuity. It can be phrased as follows. On the one hand, the upgrade of a satellite software service should guarantee to end-users an improved quality of service without degrading previously active functions. On the other hand, services not modified by the upgrade should not be interrupted.

Today, satellite software upgrades are exclusively implemented using the patch technique. This causes service interruptions (Stevens, 2000), and therefore the upgrade does not meet service continuity requirements. Given the cost of testing software on a satellite prototype, it would be interesting to analyze and predict the consequences of upgrading software before performing it to the satellite. This is where a priori validation can play an important role. What we name as “a priori validation” is the possibility to check a model of the system under design against its expected properties before the system is coded and tested. The paper proposes a novel approach that consists in adding a priori validation to Kramer (1990) and Purtilo (1991)’s techniques for the dynamic reconfiguration of applications. Even if applying a priori validation in the context of dynamic reconfiguration has been considered as a promising avenue (Gupta, 1996), little work has been published in this area.

In the paper, the purpose of applying a priori validation is to demonstrate service continuity in situations where embedded software is upgraded, and to prove that portions of software that are modified by the upgrade should go on running in conformance with their specifications. The proposed methodology relies on TURTLE (*Timed UML and RT-LOTOS Environment* (Apvrille, 2001b)), a real-time UML profile with a formal semantics given in terms of the Formal Description Technique RT-LOTOS (Courtat, 2000). RT-LOTOS code derived from TURTLE models is validated using RTL, the Real-Time Lotos Laboratory developed at LAAS-CNRS.

The paper is organized as follows. Section 2 surveys solutions for software dynamic reconfiguration and demonstrates the need for intrinsically reconfigurable software architectures and formal validation techniques. Section 3 introduces the TURTLE profile, which is dedicated to real-time system modeling and validation. Section 4 discusses how a priori validation of TURTLE models makes it possible to prove that software properties remain true during software dynamic reconfiguration. The case study in Section 5 addresses a dynamic reconfiguration performed in the context of a telecommunication protocol embedded onboard a satellite. Finally, Section 6 concludes the paper.

2. Related work

A major concern in applying software dynamic reconfiguration is to check whether the system’s intrinsic and extrinsic properties are altered. According to Kramer (1985), so-called “intrinsic” properties deal with the application’s internal logical consistency. These intrinsic properties include resources used by the application (for example, memory resources). Also, they include the interconnection logical consistency, which in procedure-based applications, refers to the logical consistency of procedure interconnection. We propose to extend the list of intrinsic properties with correct internal behavior properties (no deadlock, no deadline violation, etc.). Most reconfiguration environments are limited to the management of intrinsic properties. Nevertheless, our objective is also to prove that the service offered by an application, if not modified, remains unaltered during and after reconfiguration. This service

is defined as a set of extrinsic properties characterizing logical and real-time properties due by an application to its environment.

Three main proposals have been made to ensure that application properties are satisfied during and after dynamic reconfigurations.

In the first proposal, the reconfiguration generates a brand-new application, and then switches between the old application and the new one. In telecommunication systems, switching from a calculator to another one has been implemented by hardware redundancy (Rey, 1986). Obvious reasons of weight and power consumption make this solution impossible to adapt to space environments.

The second proposal assumes the underlying operating system handles application constraints. This approach is mostly used for active networking. New user code contained in packets is integrated into the application using a plug-in software mechanism. The operating system must provide the code with appropriate physical and software resources, such as bandwidth and resources scheduling (Yan, 2001). A drawback is that reconfigurations are limited to pre-defined functionalities customization and cannot upgrade routing and other advanced functions. Transposition to space embedded calculators is impossible since the functions to reconfigure are major ones.

The third family of solutions addresses dynamic reconfiguration at the application layer level. For telecommunication software developed with a functional approach, Frieder (1989), Segal (1993) and Okamoto (1994) have proposed techniques to dynamically replace one software procedure by another, and proved their solutions preserve software's logical consistency. Yet, the procedure replacement mechanism is too complex because of the difficulty (1) to implement runtime analysis of the software heap and (2) to detect logical and semantic relations between procedures (Shrivastava, 1998). Another approach implemented at the application layer relies on component-based software architectures based on weakly coupled components that asynchronously communicate through gates (Kramer, 1985; Hofmeister, 1993; Oreizy, 1998). Component-based architectures are described using an ADL (Application Description Language) (Medvidovic, 2000). If we compare component-based software and procedure-based application in the context of dynamic reconfiguration, the former is preferred over the latter. The reasons are twofold. The first reason lies in the diversity of reconfiguration operations supported by such architectures (Kramer, 1990; Purtilo, 1991, 1994) (component addition or withdrawal, connector modification, etc.). The second reason is that reconfiguration points are easier to identify (Kramer, 1990; Purtilo, 1991). But, a major drawback of this approach is that it incorrectly handles intrinsic and extrinsic real-time constraints of the system (Gupta, 1996).

As a solution, Gupta (1996) suggests to model the system and its properties, and to perform a priori validation in early stages of the system's life cycle, i.e. before the system is implemented and tested. Several approaches have been proposed in the literature. Allen (1998) introduced *wright*, an ADL that enables joint description of the architecture, configuration, and intrinsic constraints of an application. *Wright's* semantics is given in terms of translation to CSP (Hoare, 1985), which enables formal proof of system liveness properties. However, *wright* does not handle the intrinsic real-time and extrinsic constraints of applications. Conversely, the environments introduced in Feiler (1998) and *PBO* (Stewart, 1997) do handle some real-time constraints. Unfortunately, environments of Feiler (1998)

and *PBOs* have no mechanism for checking system liveness: only the consistency of the system is taken into consideration. Neither do they have a formal semantics. Besides using an ADL, other solutions have been proposed based on informal simulations performed on top of a real-time operating system (Cailliau, 2001). The disadvantage of that approach is that it cannot offer any kind of formal guarantees regarding reconfiguration procedure.

This paper defines a formal framework for a software dynamic reconfiguration methodology. Applications are developed according to a component-based architecture. Software architecture is described using the UML real-time profile TURTLE, which serves as an ADL. Following Allen (1997)'s approach, our objective is to model the component-based architecture together with its different configurations. Furthermore, our approach enables explicit modeling of intrinsic and extrinsic logical and real-time software constraints, and validates them against a dynamic reconfiguration script.

3. TURTLE

3.1. UML extensibility mechanisms

The Unified Modeling Language is defined by an international standard at OMG (OMG, 2003). UML 1.5, the latest release of the standard at the time of writing this paper, enables language profiling for a specific application domain, such as real-time systems. 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 in particular the “stereotype” extensibility mechanism. A stereotype extends the vocabulary of UML, allowing one to create new kinds of modular blocks. These blocks are derived from existing ones but are specific to a category of problems.

3.2. The TURTLE profile

UML 1.5 defines nine types of diagrams which provide complementary views of the system to be designed. The masterpiece of a UML design is definitely the class diagram which describes a structured view of the system's architecture. The internal behavior of these entities can be described in a state machine fashion using statecharts or activity diagrams.

The purpose of TURTLE, the *Timed UML and RT-LOTOS Environment* introduced in Apvrille (2001b), is to remain compliant with UML 1.5, but also to reinforce UML's expressive power in two directions. On the architecture description side, TURTLE extends class diagrams with stereotyped classes names *Tclasses* and composition operators which unambiguously define interactions between *Tclasses*. On the behavioral description side, TURTLE extends activity diagrams with synchronized actions including data exchange, and three temporal operators, namely a deterministic delay, a non-deterministic delay and a time-limited offer.

The *Tclass* stereotype represents a new type of UML *class* (see figure 1). Communications through public attributes or method calls are limited to communications between a *Tclass* and a normal class, or between two normal classes. *Tclasses* communicate with each other


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

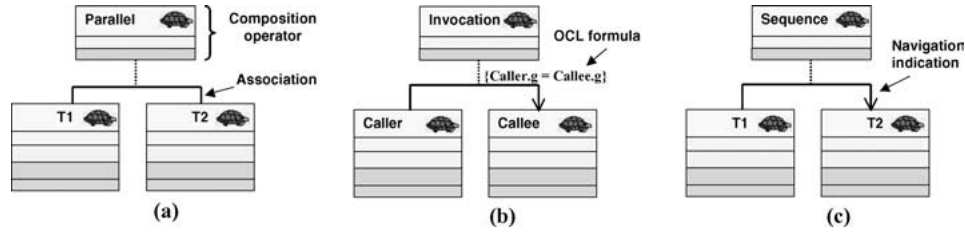
 Figure 1. Structure of a *Tclass*.


Figure 2. Using a composition operator inherited from composer.

exclusively using so-called “gates”. A *gate* is a particular *Tclass* attribute of type *Gate*. A gate can be used for synchronized communication two *Tclasses*, or for an internal *Tclass* action. A *Gate* abstract type is introduced. Its specialization as *Ingate* and *Outgate* makes it possible to describe gates dedicated to receiving and sending, respectively. Unlike “ports” in Rose RT, TURTLE gates are not defined by a list of authorized messages. A synchronization action between two *Tclasses* T1 and T2 is syntactically valid if and only if T1 and T2 use two interconnected gates and compatible parameter lists.

A *Tclass* behavior must be described with an activity diagram.

In UML 1.5, parallelism between objects is implicit. In TURTLE, parallelism and synchronization between *Tclasses* are made explicit and given a formal semantics: an association between two *Tclasses* can be attributed with a composition operator (figure 2). Five composition operators inherit from the *Composer* abstract type: *Parallel*, *Synchro*, *Invocation*, *Sequence*, and *Preemption*. In figure 2(a), the *Parallel* operator indicates that the two *Tclasses* execute in parallel without any means to synchronize each other.

A *Synchro* operator on an association between two *Tclasses* enables synchronization with value passing. It requires an OCL formula such as $\{T1.g1 = T2.g2\}$ to indicate which synchronization gates are paired.

An *Invocation* operator symbolizes an object oriented method call (see figure 2(b)). First, a caller and a callee synchronize on a gate *g*. Then, the caller’s activity, which has made the synchronization on *g*, is blocked until the callee executes action *g* again (it symbolizes the return of a object-oriented method call).

The *Sequence* operator is used to model a configuration where a task runs after another task has completed its execution (see figure 2(c)).


<i>TURTLE symbol</i>	<i>Description</i>
	Call on gate <i>g</i> , <i>x</i> being sent and <i>y</i> received.

Figure 3. Synchronization operators.

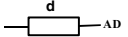


<i>TURTLE symbols</i>	<i>Description</i>
	Deterministic delay. Sub-diagram <i>AD</i> is interpreted after <i>d</i> units of time.
	Non-deterministic delay. Sub-diagram <i>AD</i> is interpreted at most after <i>t</i> units of time.
	Time limited offer. Offer on gate <i>a</i> is valid during a period of time which is lower or equal to <i>t</i> . If the offer is performed, then <i>AD1</i> is interpreted. Otherwise, <i>AD2</i> is interpreted.

Figure 4. Temporal operators.

Finally, the *Preemption* operator allows a *Tclass* to interrupt another *Tclass* once for all and at any time.

Again, each *Tclass* contains an activity diagram which models the *Tclass*'s behavior. UML constructs listed in (OMG 2003) are extended with two groups of pictograms dedicated to synchronization and temporal operators, as depicted in figures 3 and 4, respectively.

3.3. A profile with a formal semantics

TURTLE has a formal semantics expressed in RT-LOTOS (Courtat 2000), a real-time extension of the ISO-based Formal Description Technique LOTOS (ISO, 1988). Any TURTLE model can be translated to a RT-LOTOS specification (Lohr, 2002).

A TURTLE model structures a system into *Tclasses* and associates an activity diagram with each *Tclass*. (Lohr, 2002) introduced a two-step TURTLE to RT-LOTOS translation algorithm. Step 1: an RT-LOTOS process is computed for each activity diagram. Step 2: RT-LOTOS processes obtained at Step 1 are composed using information from the class diagram. More precisely, 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 sub-processes that implement loop or junction structures. TURTLE temporal operators (deterministic delay, non-deterministic delay, and time limited offer of synchronization on *Gates*) have direct counterparts in RT-LOTOS.
2. Class diagram. Associations between *Tclasses* may be attributed with Composition operators. Let *TI* be a *Tclass* involved in a relation specified by a Composition operator and *PI.1* the RT-LOTOS process obtained from the translation of *TI*'s activity diagram at step 1. At step 2, for each *Tclass*, a new RT-LOTOS process *PI.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), and may also make references 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:

- $Tclasses$ at the origin of *Preemption operators*,
- $Tclasses$ at the origin of *Sequence operators*,
- $Tclasses$ pointed out by *Preemption* or *Sequence operators*,
- $Tclasses$ at the origin of both *Sequence* and *Preemption operators*, and
- Subsets of $Tclasses$ connected either by *Parallel* or by *Synchro operators*.

For each $Tclass$ Tn active when the system starts, the RT-LOTOS specification instantiates a $Pn.2$ process.

Finally, The TURTLE to RT-LOTOS translation algorithms give the profile a formal semantics.

3.4. Formal validation tool

As we do not intend to develop new tools for the formal validation of TURTLE designs, we propose to reuse RTL, the Real-Time Lotos Laboratory developed by LAAS-CNRS. RTL implements efficient simulation strategies. When the system has a finite and reasonable number of states, RTL can also generate a “reachability graph”. By essence, reachability analysis explores all the stable states that the system possibly reaches starting from its initial state. In that sense, reachability analysis is a form of formal verification. Thus, RTL offers simulation and verification capabilities, both with a formal basis. Therefore, we say that RTL is a “formal validation” tool.

Let us now consider a reachability graph generated from a RT-LOTOS specification. A transition between two states may involve a synchronization action between two $Tclasses$. If so, the transition is labeled by an identifier corresponding¹ to one of the gates involved in the synchronization. Figure 5 depicts the complete validation process applied to TURTLE models.

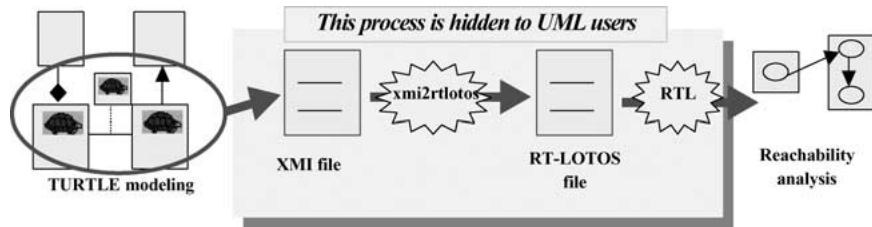


Figure 5. The TURTLE validation process.

3.5. Comparison with other real-time UMLs

3.5.1. Commercial tools. Rose RT (RoseRT, 2003) implements a real-time UML profile that basically makes class diagram and UML Statecharts evolve towards a language which is close to SDL, the protocol modeling language supported by Telelogic's TAU suite (TAU, 2003) and now partly included in UML 2.0, as confirmed by the recent release of Tau Generation 2. A common point between Rose RT and TAU is to support a formal modeling language based on extended communicating finite state machine composition. In both Rose RT and TAU, that composition is implicit. It takes the form of "connectors" in Rose RT and of "channels" and "routes" in TAU. Conversely, TURTLE inherits from the RT-LOTOS process algebra a concept of composition operator which makes composition a native construct of the profile. In other words, software component composition is explicit in TURTLE. Further, the composition operators supported by TURTLE are not limited to enable communication between *Tclasses*. They also handle "pure" parallelism, sequencing and preemption. In Lohr (2003), we have also demonstrated the power of TURTLE's native operator by extending the profile with high-level operators, such as "Periodic" and "Suspend" that we use to describe a periodic task and to suspend/resume a task, respectively. These high-level operators have a formal semantics given in terms of native TURTLE operators. With its native and explicit composition operators, TURTLE has therefore a great advantage over UML profiles that handle class composition implicitly.

Another difference between TURTLE and its counterpart implemented by TAU or Rose RT lies in the communication mechanism between software components. Communication between TURTLE *Tclasses* is based on rendezvous synchronization. By contrast, TAU Generation 2 and Rose RT associate message queues with the interface of their respective stereotyped classes. The ACCORD/UML profile (Gérard, 2002) also implements an asynchronous communication paradigm, including a broadcast mechanism. Rose RT, TAU, ACCORD and TURTLE share in common the support of asynchronous communication. By contrast, the profile defined in André (2002) defines "Synccharts" based on the Esterel synchronous language. TURTLE's rendezvous synchronization is more abstract than queued communication in Rose RT and TAU. TURTLE does not implement a communication mechanism specific to a target operating system. This is not surprising. TURTLE is intended to provide system designers with a modeling technique and a model analysis tool that remain implementation independent in terms of operating system and programming language.

Last but not least, an important difference between TURTLE and its counterpart in TAU or Rose RT lies in the set of temporal operators that it offers. TAU and Rose RT support a fixed delay operator that enable description of timeouts and other basic protocol mechanisms. Their limitations appear when the problem is to model time intervals, variable delays, jitter and other features common to timed constrained systems, such as networked multimedia systems.

3.5.2. Academic research work. Important differences between TURTLE and related research work are the following. Unlike the Petri Net based extension proposed by Delatour

(1998), the TURTLE profile remains UML 1.5 compliant in the way it integrates RT-LOTOS features to UML class and activity diagrams. TURTLE preserves the asynchronous paradigm of RT-LOTOS, and thus differs from André (2002)’s work on joint use of UML and the synchronous language Esterel. A common point with Dupuy (2001), Traoré (2000) and Clark (2000) is that our profile is given a formal semantics via a translation to a formal language. An advantage of our proposal is that RT-LOTOS is supported by a validation tool (Section 4).

Coupling UML and a process algebra was already discussed in the literature. Clark (2000) considered E-LOTOS, which misses the latency operator of RT-LOTOS. Theelen (2002) considers coupling UML and CCS for performance evaluation purposes. So far, performance issues have not been investigated in the context of UML and RT-LOTOS.

Finally, a comparison between real-time UMLs must address the question of temporal operators and their expression power. It is commonly admitted that characterizing temporal constraints requires to express temporal intervals. Theelen (2002) states that a “delay” operator suffices to model real-time systems as soon as that “delay” can be combined with an “interrupt” operator. It is indeed possible to handle a temporal interval using two instances of “delay” operator in parallel: one instance expresses delaying and the other expresses a deadline. An interrupt operator is requested so that the latter delay can interrupt the former. Therefore, one might conclude that a fixed delay operator is sufficient to model real-time systems. In practice, the situation is not so simple. Indeed, one can distinguish between two types of time interval depending whether the concern is on “uncertainty” or “opportunity”. Uncertainty refers to a situation where a system waits for an external event inside a time interval, and therefore cannot be certain about the date of occurrence of that event. In that case, we use a time limited offer. Opportunity refers to the possibility for a system to generate an event inside a time interval, at a date selected by this system. In that case, we use a combination of deterministic and non-deterministic delays. As a conclusion, a major concern for real-time modeling languages is to support temporal operators to describe the two types of time intervals.

4. Using TURTLE to verify service continuity during dynamic reconfiguration

4.1. Definitions

A continuous service offered to users by an application can be characterized by a set of properties to be verified by the application at any moment, including dynamic reconfiguration. Hereafter, we formally define this set of properties.

We denote by t_1 the date at which the dynamic reconfiguration starts, and t_2 its completion date.

We denote by P_1 and P_2 the set of properties valid before and after reconfiguration, respectively.

We also denote by P the set of properties that must be verified during dynamic reconfiguration (see figure 6).

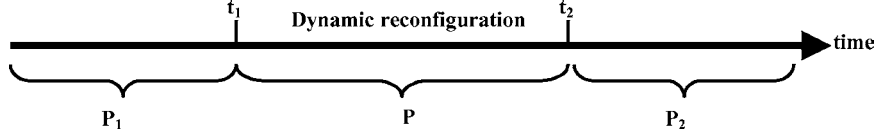


Figure 6. Chronogram of dynamic reconfiguration service continuity.

A dynamic reconfiguration is said to offer no-service continuity with regards to P_1 and P_2 if and only if the three following assertions are true:

1. $\forall t < t_1, \forall p \in P_1, p = \text{true}$
2. $\forall t > t_2, \forall p \in P_2, p = \text{true}$
3. $P = \emptyset$

Indeed, all properties of P_1 are true before dynamic reconfiguration and therefore, before t_1 . All properties of P_2 are true after reconfiguration i.e. after t_2 . Because no service is offered between t_1 and t_2 , $P = \emptyset$.

A dynamic reconfiguration ensures partial service continuity if some of the services offered by the application before dynamic reconfiguration, i.e. before t_1 , are still offered after t_1 . This definition can be formally termed as follows.

We note $P_2 = \{p_{21}, p_{22}, \dots, p_{2n}\}$.

A dynamic reconfiguration ensures partial service continuity if and only if the three following assertions are true:

1. $\forall t < t_1, \forall p \in P_1, p = \text{true}$
2. $\forall t > t_2, \forall p \in P_2, p = \text{true}$
3. $\forall t, \exists p \in P_1, (p_{21} \wedge p_{22} \wedge \dots \wedge p_{2n} \Rightarrow p) \Rightarrow p \in P$ (The set of predicates p that satisfy this assertion define a set of services that are continuously delivered).

Indeed, all properties of P_1 that are valid after reconfiguration must also be valid during reconfiguration. But after t_2 , services are described with properties of P_2 . Therefore, each property of P_1 implied by properties of P_2 must belong to P .

At last, a dynamic reconfiguration ensures total service continuity if all services of P_1 are preserved by dynamic reconfiguration i.e. after t_1 .

We note $P = \{p_1, p_2, \dots, p_n\}$

A dynamic reconfiguration is said to ensure total service continuity if and only if the following four assertions are true:

1. $\forall t < t_1, \forall p \in P_1, p = \text{true}$
2. $\forall t \in [t_1, t_2], \forall p \in P, p = \text{true}$

3. $\forall t > t_2, \forall p \in P_2, p = \text{true}$
4. $\forall p \in P_1, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow p) \wedge (p_{21} \wedge p_{22} \wedge \dots \wedge p_{2n} \Rightarrow p)$ (all services offered before t_1 are preserved).

4.2. Using observers to check properties

In order to model application properties that must be valid before, during and after dynamic reconfiguration, we propose to use the concept of observers (Jard, 1988). 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, such as its variable 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. 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. The way the observer technique can be applied to dynamic reconfiguration is further discussed in Section 4.4.

To demonstrate service continuity, we consider a *dynamic reconfiguration TURTLE model* obtained in three steps (figure 7):

- Step 1: Modeling the first software configuration (*Software modeling 1*) and the properties verified by the software in configuration 1 (*Observers of P1*).
- Step 2: Modeling the software after reconfiguration (*Software modeling 2*). *Observers of P2* model the properties to be satisfied by the software in configuration 2.
- Step 3: Writing the reconfiguration script—*Configuration Manager*—that makes software evolve from configuration 1 to configuration 2. Also, modeling the observers which check the properties to remain valid during the dynamic reconfiguration. Here, we address the observers which check for properties of P (*Observers of P*).

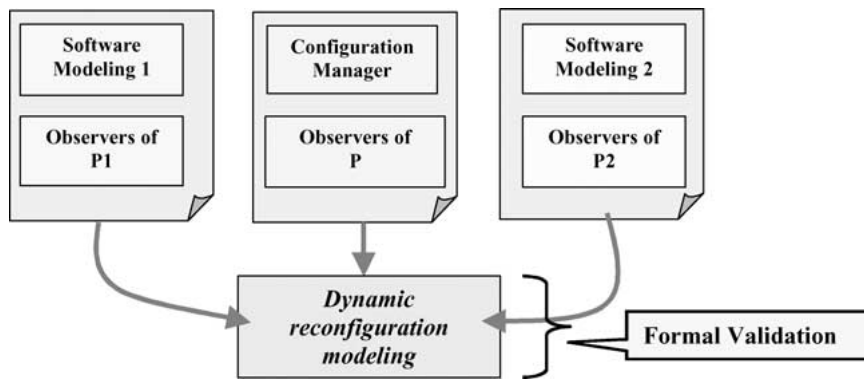


Figure 7. Dynamic reconfiguration process using TURTLE.

As explained in subsequent sections, all the models (Software Modeling 1, Configuration Manager, Software Modeling 2, all observers) are gathered to form the *dynamic reconfiguration modeling* to be formally validated against service continuity needs. This modeling includes the two software configurations together, and a description of how the dynamic reconfiguration is performed i.e. which dynamic reconfiguration operations are performed on the software, and how they are performed, etc. The execution of reconfiguration operations is described in the behavior diagram of a special *Tclass* named *ConfManager*. Full details about this modeling are given in the following sections.

4.3. Modeling software configurations

As explained in Section 2, the ADLs used in the field of dynamic reconfiguration (Purtilo, 1991; Stewart, 1997; Allen, 1998) were not designed with formal verification of service continuity in mind, even if that issue had been identified of great importance since Gupta (1996). As a consequence, a major concern is to define an ADL capable of offering formal validation of logical and real-time properties during dynamic reconfiguration.

Space-based embedded software is built upon tasks that communicate asynchronously. Therefore, we propose to associate a component per task, and we call such a component a “module”. Such association is a common practice in dynamically reconfiguration architecture modeling (Liskov, 1985; Stewart, 1997; Shrivastava, 1998). Modules offer to their environment communicating gates that we name *ports*. Each port is either an input port dedicated to data receiving, or an output port dedicated to data sending. Connectors, called *links*, connect an output port to an input one. TURTLE has not been primarily designed for software architecture in the sense used by Kramer (1985), Stewart (1997) and Shrivastava (1998). Therefore, we hereafter describe how such architecture may be modeled with TURTLE, and more particularly how TURTLE may be used to model *modules*, *ports*,² and *links*.

Thus, we consider a software architecture structured into modules. The TURTLE representation for a module is a *Tclass* (see figure 1). Modules communicate using ports. Therefore, for each port in a module, we create a gate in the relevant *Tclass*. Each gate must have a type. The TURTLE profile definition includes an abstract type *Gate* which is specialized in *InGate* and *OutGate* (see Section 3.2). The input and output ports of a module are modeled by attributes of type *InGate* and *OutGate* respectively. We introduce a *Module* abstract type from which concrete modules can be derived later on. This abstract class *Module* contains *active* and *stop*, two attributes of type *InGate*. In figure 8, a module *M1* inherits from *Module*. *M1*’s activity diagram starts with a synchronization on *active* and then offers a choice between what we refer to as its “normal activity” and the possibility to be stopped and reactivated. Whenever a *Tclass* performs synchronization on the *stop* Gate of *Module M1*, *M1* is supposed to reach its reconfiguration point as soon as possible. As a consequence, the system works under the assumption that modules such as *M1* periodically leave their normal activity to check whether synchronization can be performed or not on its *stop* gate (otherwise, it can never get into its reconfiguration point). The *active* gate allows reactivation of a module that is at its reconfiguration point. In our dynamic reconfiguration environment, it is the programmers’ responsibility to ensure that the *stop*

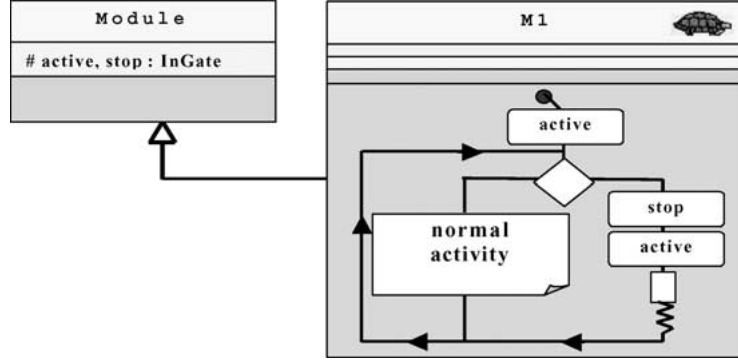


Figure 8. How a module may be modeled using TURTLE.

gate is checked as often as possible so that modules can reach their reconfiguration point as soon as possible. In the CONIC environment, Kramer (1990) has proved that the *quiescent* state (the “reconfiguration point” in CONIC) of a module is always reachable. Using the TURTLE formal validation process described in Section 3.3, system designers can predict whether modules can reach their reconfiguration point, and if so, if they can reach it as fast as required. Indeed, in case of dynamic reconfiguration failure identified at validation process, the reachability graph contains traces of the sequence of events leading to the failure. Thus, it becomes possible to determine if the failure is due to the fact a module couldn’t reach its reconfiguration point, or has been too slow to reach it.

A link connecting a module *OutGate* *og* to a module *InGate* *ig* is modeled as follows:

- The asynchronous semantic of links is modeled using an *Invocation* composition operators and a *Tclass* that models a message buffer on the receiving module’s side.
- The link behavior (delay, loss, etc.) is modeled in *Routing*, a *Tclass* common to all links. Thus, the software communication architecture can be modified by reconfiguring *Routing* only.

For example, consider the two links modeled in figure 9. The first one connects the *OutGate* *g1* of *M1* to the *InGate* *g3* of *M3*. There is an *Invocation* Operator between *M1* and *Routing*. An OCL formula $\{g1\}$ indicates that gates *g1* of both *M1* and *Routing* are involved in this invocation. The left part of *Routing* Behavior diagram describes the link behavior: it represents the link’s transmission delay and the buffer the message is forwarded to. For this link, the forwarding delay is at least 8 time units and at most 10 time units ($8 + 2$). These values can be obtained by simulations (see Section 5). An *Invocation* operator between *Routing* and *BufferPort1* models the forwarding of the message to the *Tclass* *BufferPort1* when *Routing* performs action *g1buf*. On the receiving side, *M3* is connected to *BufferPort1* with an *Invocation* operator. Two gates *g3* and *g3nb* make it possible for *M3* to read a message in *BufferPort1* and to get the number of available messages in *BufferPort1*, respectively.

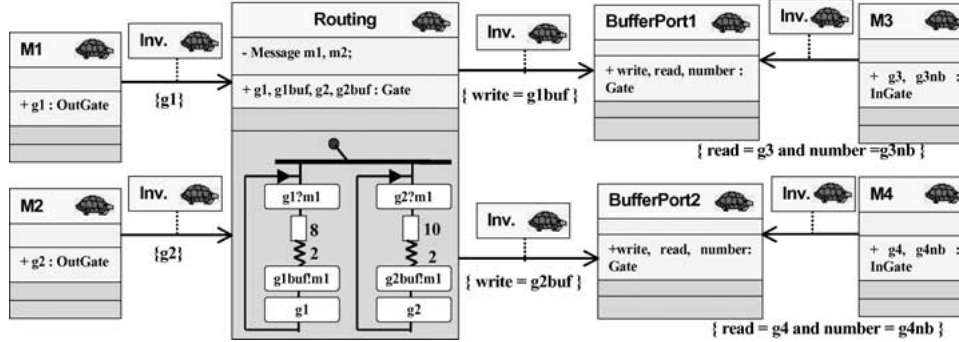


Figure 9. Modeling of software communication links with TURTLE.

The use of an intermediate class *BufferPort* makes it possible to model an asynchronous link between *Routing* and the receiving module. Using the *Routing Tclass* simplifies the dynamic reconfiguration of links. Indeed, a link behavior modification can be performed at *Routing* level. Further, a link interconnection can be modified by changing the output gate to which the message is forwarded. For example, if action “g1buf!m1” is switched by action “g2buf!m2” in *Routing*, then, the link from *M1* to *M3* becomes a link from *M1* to *M4* (see figure 9). Further information on link reconfiguration is provided in Section 4.5.

4.4. Modeling intrinsic and extrinsic application constraints

Intrinsic and extrinsic application constraints are described inside observers which are modeled as TURTLE *Tclasses*. Observer *Tclasses* are external to the application but belong to the application’s class diagram.

Observers analyze specific properties in a non-intrusive way; indeed, they do not modify the behavior of application modules, including modules observers get data from. For example, suppose that a *Tclass* O (Observer) has to get information data from a module M. To model data retrieval, we use a *Synchronization* composition operator between O and M. O may 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 is encountered.

Figure 10 describes an observer analyzing logical and real-time constraints. The two observed properties are:

- *Property 1 (logical constraint)*: the $2k + 1$ and $2(k + 1)$ integer values received by M on gate *g1* should be identical.
- *Property 2 (temporal constraint)*: no more than *t* time units should elapse before two synchronizations on gate *g2*.

VERIFYING SERVICE CONTINUITY IN A DYNAMIC RECONFIGURATION

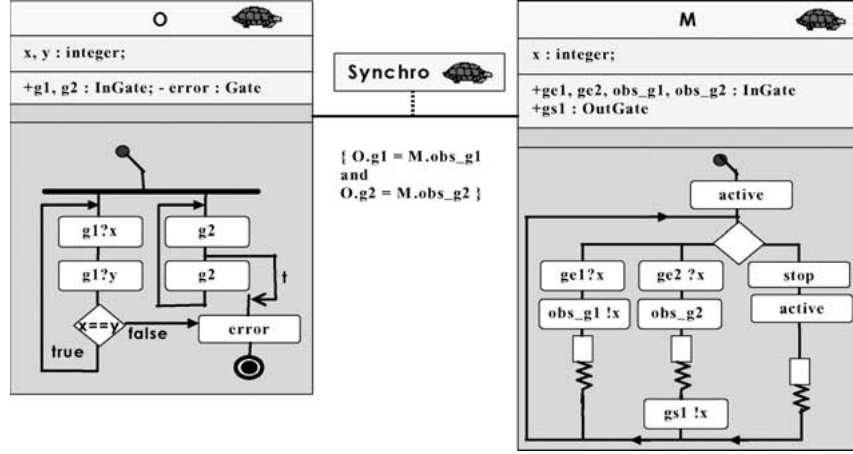


Figure 10. Observing logical and temporal properties in a module M.

Observer O analyzes the two properties by getting data from module M. Therefore, there is a *Synchro* operator between O and M. 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 corresponding observation gate *g-g* is either *g1* or *g2*—is not offered by the observer. Also, M will be stopped next time it synchronizes on *g*.

4.5. Modeling a dynamic reconfiguration script

This section explains how it is possible to build the *TURTLE dynamic reconfiguration* model. The building of this model relies on the modeling of software configurations introduced in Section 4.3 and on the modeling of observers introduced in Section 4.4.

A *TURTLE* class named *ConfManager* manages the initial software configuration, the execution of the application in this first configuration, and the execution of the reconfiguration script.

ConfManager first starts all the modules in configuration 1 by synchronization on their *active* gate. Then, it executes a (non-)deterministic delay to model the time during which the application runs in configuration 1. The lower portion of the activity diagram in *ConfManager* represents the execution of the reconfiguration script i.e. all dynamic reconfiguration operations possibly performed on our application. This script can modify:

- The general architecture of modules, in particular module addition or removal,
- The internal behavior of modules, and
- The interconnection architecture between modules, in particular links creation or destruction.

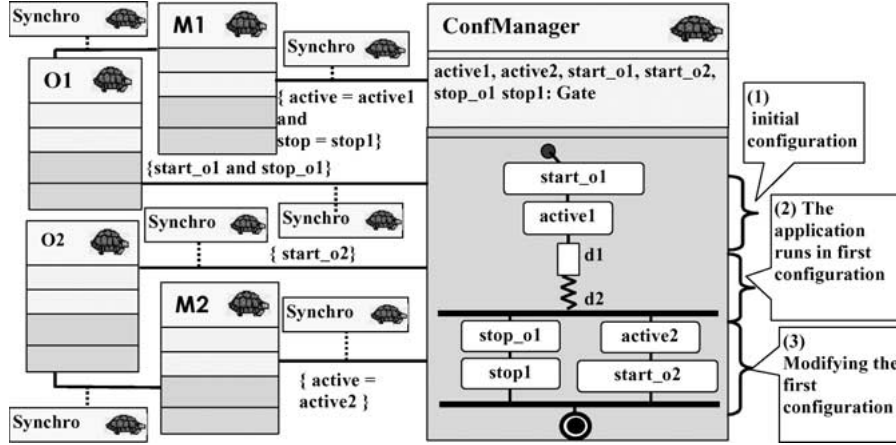


Figure 11. Modeling a module replacement.

As a first example, figure 11 depicts a dynamic reconfiguration that withdraws module *M1* and adds module *M2*.

First, *ConfManager* starts *Module M1* (*active1*). The application works in this state for at least *d1* time units and at most during *d1 + d2* time units. Then, the dynamic reconfiguration starts. The dynamic reconfiguration stops module *M1* (*stop1*) and starts module *M2* (*active2*).

During each configuration phase (initial configuration, execution of application in configuration 1, dynamic reconfiguration, and execution of the application in configuration 2), some application's intrinsic and extrinsic constraints must remain true and others are superseded. Indeed, when deleting from software a service to users, of course, properties relative to this service don't have to be true anymore. Therefore, we propose to activate an observer when the property it checks has to be verified by the system, and to inactivate it when the property it checks is not valid any more. For example, at figure 11, observers of *M1* should be activated before *M1* starts (*start_o1*) and should be stopped when *M1* is stopped (*stop_o1*). Also, the starting of *M2* should be followed (or preceded) by the activation of *M2* observers (*start_o2*).

As a second example, we address the internal behavior reconfiguration of a module (figure 12).

1. A module, *M1*, executes its 'first activity' once *ConfManager* has executed action *active1*.
2. After at least *d1* time units and at most after *d1 + d2* time units,
3. *ConfManager* executes *stop1* which stops *M1* after its current activity is completed (reconfiguration point). Then, *ConfManager* changes *M1* *conf* variable using synchronization on gate *chConf*. Finally, it reactivates *M1* (*active1*), which now executes 'Re-configured activity' instead of 'First activity'.

The third example deals with the dynamic reconfiguration of links between modules, an operation modeled by modifying the *Routing Tclass*. In figure 13, the link between modules

VERIFYING SERVICE CONTINUITY IN A DYNAMIC RECONFIGURATION

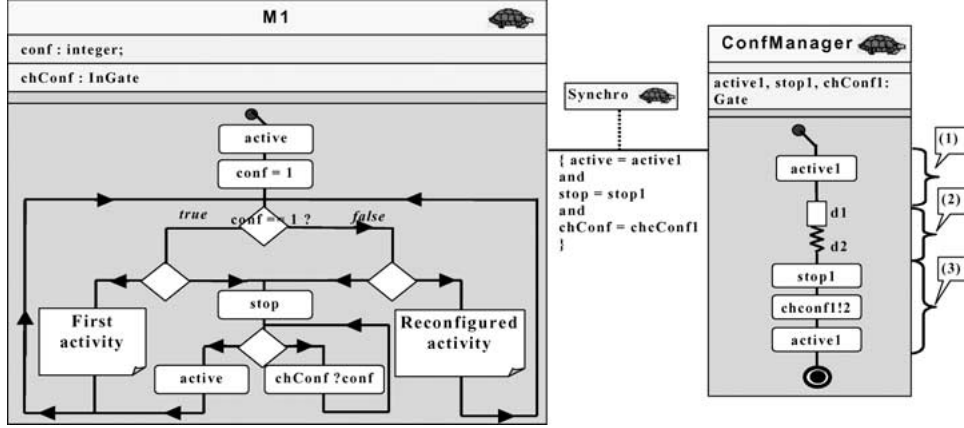


Figure 12. Modeling a module's Internal behavior reconfiguration.

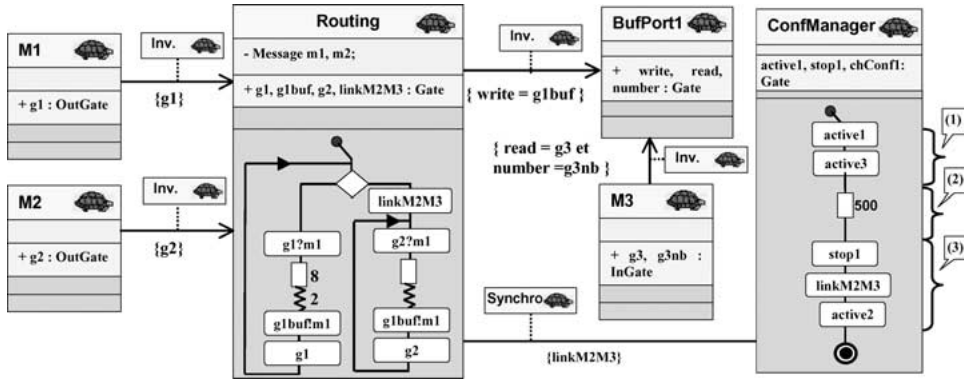


Figure 13. Modeling the dynamic reconfiguration of a link.

M1 and *M3* is modeled in the *Routing* *Tclass*: when *M1* sends a message on its *OutputGate* *g1*, *Routing* forwards it (*g1buf!m*) to the input buffer class of *M3* (*BufPort1*). Then, *M3* can receive the message by executing *g3*. The link reconfiguration consists in connecting a new module *M2* to the same input buffer of *M3*, and simultaneously to remove the link between *M1* and *M3*. First, *M1* and *M3* are started (1): *active1*, *active3* (Synchronizations between *ConfManager* and the Modules are not depicted). Then, after 500 time units (2), the reconfiguration occurs (3). First, Module *M1* is stopped (*stop1*). Then, the link between *M2* and *M3* is activated. To do so, the action *linkM2M3* waits for the link between *M1* and *M3* to be free, and then, it supersedes it with a link between *M2* and *M3*: messages sent by *M2* on *g2* are now forwarded to *g1buf*. Then, module *M2* is started. This modeling approach insures that no message is lost or mis-ordered during reconfiguration.

4.6. Formal validation of service continuity

As explained in Section 4.1, a set of properties characterizes services to be offered to users before, during and after reconfiguration. Therefore, if the configuration manager is modeled as in configuration 1, active observers check for properties P1. Then, during dynamic reconfiguration, active observers check for P. At last, after reconfiguration, active observers check for P2. As a consequence, formal validation of the *dynamic reconfiguration TURTLE model* makes it possible to prove service continuity. Indeed, when a service is no longer valid, an observer executes a specific action named, e.g., *error*. The latter is easily identified on the reachability graph, since the TURTLE to RT-LOTOS translation process builds up a correspondence table between actions in TURTLE diagrams and actions in the reachability graph (Lohr, 2002).

5. Case study

5.1. Context

The competition with high-speed terrestrial transmission technologies (Bigo, 2000) leads telecommunication satellite manufacturers to drastically optimize bandwidth by frequency reuse and by dynamic frequency allocation and temporal multiplex on uplinks and downlinks (Farserotu, 2000; Wittig, 2000). These issues have been investigated in the context of the French Research Minister project SAGAM (1998). This project focuses on the access to multimedia services using a multi-beam geostationary satellite. The core of the system consists in a fast and embedded ATM switch and a temporal multiplexer of ATM cells on downlinks. Switching and multiplexing are performed according to differentiated QoS.

The SAGAM project addresses the management of the embedded ATM switch and the management of uplink and downlink bandwidth. Bandwidth slots are allocated according to ATM active connections (Roullet, 1999). Assuming that all bandwidth management functionalities are embedded into the satellite and software implemented, we have modeled the following functionalities in TURTLE:

- The sending of a frame allocation report every 50 ms from the satellite to the users. The frame allocation report lets each user know which uplink slots he or she can send his or her data on.
- User sending *Dama-sig* signals to the satellite. A *Dama-sig* signal indicates that an ATM VBR connection wants to emit at its higher rate. Satellite software shares the remaining bandwidth among all *Dama-sig* requests.

More information about these functionalities are available in Roullet (1999), Combes (2001) and Aprille (2001a, 2002).

5.2. Software modeling

The *Dama-sig* signals computing and the frame allocation report sending are modeled within four modules:

VERIFYING SERVICE CONTINUITY IN A DYNAMIC RECONFIGURATION

- A *Damasig* module which receives *Dama-sig* signals, formats them, and then, forwards them to the *Dama* module.
- A *Dama* module which computes all pre-formatted *Dama-sig* signals. For each *Dama-sig*, it computes a new allocation for each ATM connection. The new allocation is sent to module *FarSender*.
- A *FarSender* module, which has two input ports. The first one is dedicated to receiving bandwidth allocation sent by *Dama* module. When receiving a message on the second input port, *FarSender* must send immediately a frame allocation report.
- A *Clock* module which sends two signals every 50 ms. One is sent to the second port of *FarSender*, and the other is sent to *Damasig* to indicate it that the next n *Dama-sig* signals can be computed (n being the number of active VBR connections).

Our TURTLE modeling methodology associates a *Tclass Buffer* with each input port of a module (see Section 4.3). The model includes six buffers: one per module, except *Damasig* and *FarSender* which have two input ports (so two buffers, see Section 4.3). An additional *Tclass* models message routing between each output port and its corresponding buffer.

Each of the modules listed above implements an algorithm which is not modeled *in extenso* in TURTLE but represented by its computation duration. Each duration value can be obtained either by simulations performed with specific simulators (ERC-32 simulators, see (ERC32, 1999)) or directly on target.

5.3. Modeling software constraints

The satellite embedded software should always respect the following two constraints on the service offered to users:

- Property 1. The sending of the frame allocation report is periodic with a period equal to 50 ms.
- Property 2. All received *Dama-sig* have to be fully computed before the next frame allocation report is sent.

One observer is modeled for each constraint (consequently, the system has two observers). Both observers synchronize with *FarSender* (see figure 14). The first one analyzes the sending date of the first frame allocation report. Then, it checks that the next frame allocation report is emitted exactly 50 ms after the previous one. The second observer checks, for each frame period, the number of allocations sent by *Dama* module to *FarSender*: this number must be equal to the number of *Dama-sig* signals to be computed by frame period. When one of the two properties becomes false, the observer checking for that property executes action *error*. Then, *FarSender* is stopped next time it synchronizes with this observer.

By applying the process depicted in figure 5 to this software modeling, we have proved that both properties are true in the first software configuration.

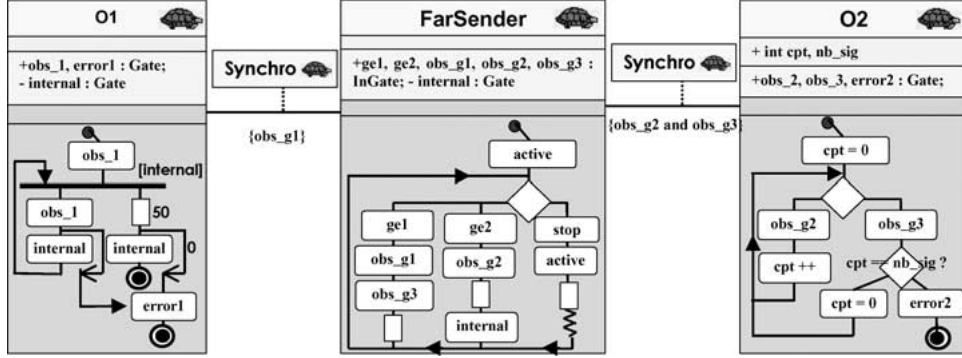


Figure 14. Modeling both properties with observers.

5.4. Checking a dynamic reconfiguration

Like satellite telecommunication payload algorithms in general, DAMA algorithms quickly evolve (Boutry, 2000). Therefore, the problem we focus on is the replacement of the *Dama* module while the software is running. The new *Dama* module, which supersedes the first one, is called *Dama2*. Its main algorithm takes more time to compute a *Dama-sig* that the first *Dama* module.

A simplified dynamic reconfiguration script contains the following: “stop *Dama*, instantiate *Dama2*, copy *Dama* state into a new instance of *Dama2*, start *Dama2*, destroy *Dama*”. The configuration manager models this script as follows: “stop *Dama*, wait for 30ms, start *Dama2*”. This duration has been obtained by simulations performed on an experimental platform that emulates the multimedia telecommunication system under study (Apvrille, 2001a).

Our objective is to perform dynamic reconfigurations with total service continuity. We indeed expect service to users to remain valid before, during, and after reconfiguration. User services are described by property1 and property2 that are checked out by observers O1 and O2, respectively.

Formal validation is performed on the TURTLE dynamic reconfiguration model obtained by applying the process described in Section 4.1. This model contains 15 classes: 5 modules, 6 buffers, a routing class, 2 observers and the dynamic reconfiguration manager. The reachability graph generated from the model is far too complex to be entirely drawn in this paper. Figure 15 shows an excerpt of this graph.

Reachability analysis has demonstrated that the software runs with a period of 50 ms: the dynamic reconfiguration succeeds if and only if it is started in the “middle” of a period. For instance, if t_0 is the starting date of a period, the reconfiguration has to be started between $t_1 = t_0 + 19$ ms and $t_2 = t_0 + 45$ ms. If the reconfiguration is started out of this time range, one can identify a property violation on the reachability graph (see figure 15), i.e. the service to users suffers discontinuity. On figure 15, *error2* transition indicates that property 2 can be violated. When started in the right time interval, the reconfiguration succeeds: both properties remain valid, which proves service continuity.

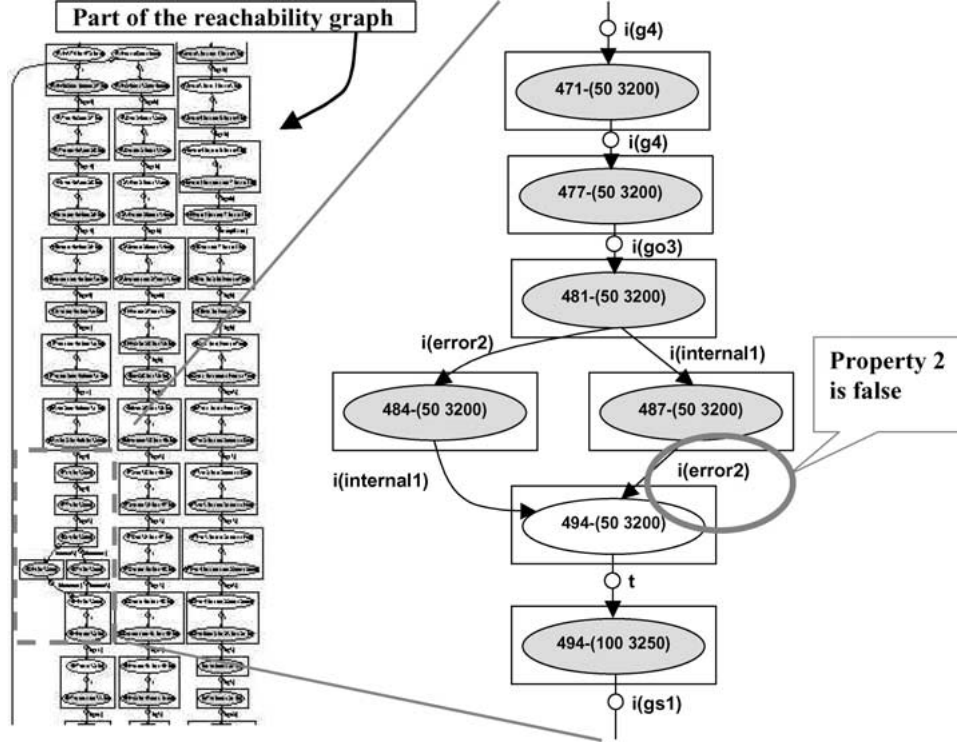


Figure 15. Reachability graph of the case study.

This result leads us to modify the reconfiguration script as follows: a new operation is inserted at the top of the script. This operation waits for the software to be in interval $[t1, t2]$ before starting the reconfiguration. The new reconfiguration script is the following: “wait for $t \in [t1, t2]$; stop Dama, instantiate Dama2, copy Dama state into a new instance of Dama2, start Dama2, destroy Dama”.

5.5. Lessons learned

The purpose of this section is to report our experience in applying TURTLE and the methodology proposed in the paper to the dynamic reconfiguration of an embedded real-time application. This work is part of a project funded by Alcatel Space.

In terms of architecture modeling capacity, we have defined, in Section 4.3, a framework for the modeling of software architectures. Using TURTLE as an ADL, software reconfigurations can be visually apprehended on a UML class diagram that features modules, interconnection schemes (communication channels), and reconfiguration manager. Unlike UML 1.5 and most ADLs, TURTLE also enables explicit modeling of real-time tasks and asynchronous communications between tasks.

Let us now address behavior modeling. The risk of combinatory explosion at reachability graph generation leads us to model task algorithms at a high level of abstraction. We consider that an algorithm should be modeled with a temporal operator representing all its possible computation durations. These durations can be obtained by simulations performed on an experimental platform (see Section 5.4). TURTLE's non-deterministic temporal operator provides an explicit way to model lower and upper limits of algorithms' duration. This is an advantage of TURTLE over UML 1.5 and other ADLs.

For the case study discussed in Section 5, a finite reachability graph was generated in less than five minutes on a SUN UltraSparc. Note that to reduce the size of the reachability graph, we had to limit as much as possible the use of variables and of non-deterministic delays. As a consequence, algorithms' duration were sometimes modeled by their maximal duration (deterministic delay) and not by a time interval starting at their minimal limit and finishing at their maximal limit. Also, the reconfiguration validation was successfully applied only when the number of messages transiting between modules was of reasonable size: tasks' buffer size was commonly limited to 250. Therefore, the size of modules' receiving buffers was reduced as much as possible (most of the time to 250 messages).

At last, we draw positive conclusions of using the observer technique, a simple way to formally validate the system's model against service continuity requirements. Observers are modeled using TURTLE classes, which are distinct from system tasks. Thus, the system software architecture remains unmodified. Also, the TURTLE synchronization operator makes it possible to model non-intrusive observers. Using a unique action identifier (error label) makes it simple to identify property violations in the reachability graph. The observer technique really helped us identifying non-trivial errors (see how we identified the violation of property 2 in previous section) which could have occurred when applying dynamic reconfiguration on the embedded system.

6. Conclusion

With an average lifetime of fifteen years, satellites must be regularly and dynamically reconfigured in order to adapt payloads to multimedia data stream evolution. Dynamic reconfiguration captures a service continuity problem. A dynamic reconfiguration should indeed not interrupt the software portion that is not modified, and preserve a set of properties that define the quality of service offered to end-users.

How to predict that a dynamic reconfiguration procedure guarantees service continuity is still an open issue. Gupta (1996) suggested that an avenue to explore is a priori validation, where a model of the software is simulated and verified against its expected properties before the software is actually implemented and tested. The work in the paper follows that approach with modeling in TURTLE, an enhanced real-time UML profile with a priori formal validation capabilities.

TURTLE extends UML class diagrams with composition operators that make it possible to explicitly model parallelism and synchronization between stereotyped classes named *Tclasses*. TURTLE also enhances UML activity diagrams with three temporal operators: a deterministic delay, a non-deterministic delay and a time limited offer. TURTLE has a formal semantics given in terms of translation to the Formal Description Technique

RT-LOTOS (Lohr, 2002). RT-LOTOS code derived from TURTLE models can be validated using the RTL toolkit (Courtiat, 2000) that implements efficient simulation algorithms and reachability analysis.

The paper has discussed the use of TURTLE and RTL to prove service continuity in a dynamic reconfiguration procedure. Our methodology relies on making up a dynamic reconfiguration model that contains software configurations before and after the reconfiguration, as well as the services to be continued during and after the upgrade. The methodology was successfully applied in the framework of SAGAM project. It can be applied as well to a wide range of software applications which capture dynamic reconfiguration problems.

A priori validation filters errors but does not exempt from testing software's implementation. We are presently working on deriving timed test sequences from RT-LOTOS specifications (Saqui, 2003), including those generated from TURTLE models. Also, our objective is to extend our reconfiguration methodology to the dynamic upgrade of distributed systems. Assuming that a distributed system runs on several sites, the TURTLE profile addressed in this paper does not enable explicit modeling of that distributed execution. Apvrille (2003) defines TURTLE-P, an enhanced TURTLE with deployment diagrams and their formal semantics. We plan to apply TURTLE-P to the validation of dynamic reconfiguration procedure in distributed systems. Last but not least, we think our approach could be used to formally prove that the integration of a user code into an active network does not lead to undesirable or unpredicted behaviors.

Notes

1. The TURTLE to RT-LOTOS translation renames gate identifiers.
2. A port in an ADL is different from a port in Rose RT.

References

- Allen, R. 1997. A formal approach to software architecture. Ph.D. Thesis, Carnegie Mellon University, School of computer Science, TR#CMU-CS-97-144.
- Allen, R., Douence, R., and Garlan, D. 1998. Specifying and analyzing dynamic software architectures. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, Lisbon, Portugal.
- André, C., Peraldi-Frati, M.-A., and Rigault, J.-P. 2002. Integrating the synchronou. In J.-M. Jézéquel, H. Hußmann, S. Cook, editors, *Paradigm into UML: Application to control-dominated systems. UML 2002—The Unified Modeling Language, 5th International Conference*, Dresden, Germany, LNCS 2460, Springer, ISBN 3-540-44254-5.
- Apvrille, L. 2002. Contribution to dynamic reconfiguration of embedded real-time software: Application to a satellite telecommunication environment Ph.D. dissertation (in French), TR #02298, LAAS-CNRS, Toulouse, France.
- Apvrille, L., Dairaine, L., Sénac, P., and Diaz, M. 2001a. Dynamic reconfiguration architecture of satellite network software services. In *Proceedings of the 19th AIAA International Communications Satellite Systems Conference*, Toulouse, France.
- Apvrille, L., de Saqui-Sannes, P., and Khendek, F. 2003. TURTLE-P: A UML profile for the validation of distributed Architectures. In *Proceedings of the Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'2003)*, Paris, France (in french).
- Apvrille, L., de Saqui-Sannes, P., Lohr, C., Sénac, P., and Courtiat J.-P. 2001b. A new UML profile for real-time System: Formal Design and Validation. In *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'2001)*, Toronto, Canada.

- Bigo, S. and Idler, W. 2000. Multi-terabits/s transmission over Alcatel TeraLight fiber. *The Alcatel Telecommunications Review*, 4th Quarter: 288–296. Available at: http://atr.alcatel.de/hefte/00i_4/gb/pdf_gb/11idlegb.pdf
- Boutry, L. 2000. *Network Evolution. France Telecom R&D Technical Memento*, vol. 15. Available at: <http://www.cent.fr/sas/mento15/chap5.html> (in French).
- Cailliau, D., Marin, O., and Folliot, B. 2001. A joint middleware/configuration language approach for space embedded software update. In *Proceedings of Data Systems In Aerospace (DASIA)*, Nice, France.
- Chen, T.M. 2000. Evolution to the programmable internet. *IEEE Communications Magazine*, 38(3):124–128.
- Clarck, R.G. and Moreira, A.M.D. 2000. Use of E-LOTOS in adding formality to UML. *Journal of Universal Computer Science*, 6(11):1071–1087.
- Combes, S., Fouquet, C., and Renat, V. 2001. Packet-based DAMA protocols for new generation satellite networks. In *Proceedings of the 19th AIAA International Communications Satellite Systems Conference*, Toulouse, France.
- Courtiau, J.-P., Santos, C.A.S., Lohr, C., and Outtaj, B. 2000. Experience with RT-LOTOS, a temporal extension of the LOTOS formal description technique. *Computer Communications*, 23(12):1104–1123.
- Delatour, J. and Paludetto, M. 1998. UML/PNO, a way to merge UML and Petri Net objects for the analysis of real-time systems. In *Proceedings of the Workshop on Object-Oriented Technology and Real Time Systems ECOOP'98*, Brussels, Belgium.
- Dupuy, S., and du Bouquet, L. 2001. A multi-formalism approach for the validation of UML models. *Formal Aspects of Computing*, 12:228–230.
- ERC32. 1999. Free simulation software for ERC-32, at ESTEC. Available at: <http://www.estec.esa.nl/wsmwww/erc32/freesoft.html>
- Farserotu, J., and Pradas, R. 2000. A survey of future broadband multimedia satellite systems, issues and trends, *IEEE Communications Magazine*, 128–133.
- Feiler, P., and Li, J. 1998. Consistency in dynamic reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDs '98)*, IEEE Press, ISBN 0-8186-8451-8, Annapolis, MD, USA.
- Frieder, O., Herman, G.E., Mansfield, W.H., and Segal, M.E. 1989. Dynamic program modification in telecommunications systems. In *Proceedings of the Seventh International Conference on Software Engineering for Telecommunication Switching Systems SETSS 89*, pp. 168–172.
- Gerard, S., Terrier, F., and Tanguy, Y. 2002. Using the model paradigm for real-time systems development: ACCORD/UML. In J.-M. Bruel, and Z. Bellahsene, editors, *Proceedings of the Advances in Object-Oriented Information Systems, OOIS 2002 Workshops*. Lecture Notes in Computer Science 2426, Springer: Montpellier, France, pp. 260–269.
- Gupta, D., Jalote, P., and Barua, G. 1996. A formal framework or on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131.
- Hoare, C.A.R. 1995. *Communicating Sequential Processes*. Prentice Hall.
- Hofmeister, C., and Purtilo, J. 1993. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference in Distributed Computing systems (ICDCS'93)*, IEEE Computer Society Press, pp. 101–110.
- ISO. 1998. LOTOS—A formal description technique based on the temporal ordering of observational behavior. International Standard 8807, International Organization for Standardization—Information processing Systems—Open Systems Interconnection, Geneva, Switzerland.
- Jard, C., Monin, J.-F., and Groz, R. 1988. Development of véda, a prototyping tool for distributed algorithms. *IEEE Transactions on Software Engineering*, 14(3):339–352.
- Kramer, J. and Magee, J. 1985. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436.
- Kramer, J. and Magee, J. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306.
- Liskov, B.H. 1985. The Argus language and system. Distributed systems: Methods and Tools for Specifications, Lecture Notes in Computer Science No. 190, Springer-Verlag, pp. 343–430
- Lohr, C. 2002. Contribution to real-time system specification relying on the formal description technique RT-LOTOS. Ph.D. dissertation (in French), Toulouse, France.

VERIFYING SERVICE CONTINUITY IN A DYNAMIC RECONFIGURATION

- Lohr, C., Apvrille, L., de Saqui-Sannes, P., and Courtiat, J.-P. 2003. New operators for the TURTLE real-time UML profile. In *Proceeding of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'2003)*, Paris, France.
- Medvidovic, N. and Taylor, R. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.
- Okamoto, A., Sunaga, H., and Koyanagi, K. Dynamic program modification in the non-stop software extensible system (NOSES). In *Proceedings of the IEEE International Conference on Communications (ICC '94)*, vol. 3, pp. 1779–1783.
- OMG. 2003. Unified modeling language specification, Version 1.5. Object Management Group, Available at: <http://www.omg.org/technology/documents/formal/uml.htm>
- Oreizy, P., Medvidovic, N., and Taylor, R.N. 1998. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'1998)*, Kyoto, Japan.
- Saqui-Sannes, P. de, Sadani, T., Lohr, C., and Courtiat, J.-P. 2003. First results in deriving timed test sequences from RT-LOTOS specifications. TR#03336, LAAS-CNRS, Toulouse, France.
- Purtilo, J.M. and Hofmeister, C. 1991. Dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 560–571.
- Purtilo, J.M. 1994. The polyolith software bus. *CM Transactions on Programming Languages and Systems*, 16(1):151–174.
- Rey, R. 1986. *Engineering and Operations in the Bell System*, second edition. AT&T Bell Laboratories, Murray Hill, NJ, USA.
- Rose, RT. 2003. Available at <http://www.rational.com/products/rosert/index.jsp>
- Roulet, L. 1999. SAGAM demonstrator of a G.E.O. satellite multimedia access system: Architecture & integrated resource manager. In *Proceedings of the European Conference on Satellite Communication*, Toulouse, France.
- SAGAM. 1998. SAtellite Géostationnaire pour Accès Multimédia. Projet RNRT (Réseau National de la Recherche en Télécommunication, France), France. <http://www.sagam-satellite.com>.
- Segal, M.E. and Frieder, O. 1993. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65.
- Shrivastava, S.K. and Wheeler, S.M. 1998. Architectural support for dynamic reconfiguration of distributed workflow applications. *IEE Proc-Software*, 145(5):155–162.
- Stevens, J.S. and Johnson, G.L.R. 2000. Updating the SOHO AOCS ACU On-board software. In *Proceedings of Data Systems In Aerospace (DASIA)*, Montreal, Canada, pp. 347–352.
- Stewart, D., Volpe, R., and Khosla, P. 1997. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759–776.
- Tau. 2003. Available at: <http://www.telelogic.com>
- Theelen, B.D., van der Putten, P.H.A., and Voeten, J.P.M. 2002. Using the SHE method for UML-based performance modeling. In *Proceeding of the Forum on Specification and Design Languages FDL'02*, Marseille, France.
- Traoré, I. 2000. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108.
- Wittig, M. 2000. Satellite onboard processing for multimedia applications. *IEEE Communications Magazine*, 134–140.
- Yan, D.K.Y. and Chen, X. 2001. Resource management in software-programmable router operating systems. *IEEE Journal on Selected Areas in Communications*, 19(3):488–500.