On the Use of Real-Time Maude for Architecture Description and Verification: A Case Study

Chadlia Jerad[1], Kamel Barkaoui[2] and Amel Grissa Touzi[3]
[1,3] LSTS-ENIT, Le Belvédère 1002 Tunis, BP 37, TUNISIA
[2] CEDRIC-CNAM, 292, Rue Saint-Martin, Paris 75003, FRANCE
[1]*chadlia.jerad@gmail.com,* [2]*barkaoui@cnam.fr,* [3]*amel.touzi@enit.rnu.tn*

Real-Time Maude is an executable rewriting logic language particularly well suited for the specification of object-oriented open and distributed real time systems. In this paper we explore the possibility of using Real-Time Maude as a formal notation for software architecture description and verification of real time systems. The system model is composed of two kinds of descriptions: static and dynamic. The static description consists in identifying the different elements composing the architecture, while the dynamic description is the definition of the rules governing the system behaviour in terms of the possible actions allowed. The correspondence between software architecture concepts and the Real-Time Maude concepts are developed for this purpose. The step towards verifying system architecture is realized by applying Real-Time Maude simulation and analysis techniques to the described model and the properties that must be satisfied. An example is used to illustrate our proposal and to compare it with other architecture description languages.

Keywords: ADL, Rewriting logic, Real-Time Maude

1. INTRODUCTION

During the past decade, architectural design has emerged as an important subfield of software engineering. In fact, a good architecture can help ensure that a system will satisfy key requirements. Consequently, a new discipline emerged, which concerns formal notations for representing and analyzing architectural designs: Architecture Description Languages (ADL) [4]. ADL allow to model hierarchical components systems. After modelling step, designers want to make sure that the built application is safe.

Rewriting Logic formalism has been used in the verification of systems models. This logic differs from the standard logics, as first or higher order logics, by the fact that it is a logic of change whose models are concurrent systems and whose deduction is concurrent computation in such systems. Real-Time Maude [16] is an executable specification language based on rewriting logic. This system is particularly well suited for formal specification and analysis of real-time systems.

In this paper we would like to explore another alternative for describing and verifying real time systems' architectures. We propose Real-Time Maude, since it is dedicated to the specification of real time object-oriented open and distributed systems.

This paper is organized as follows: in section 2, we recall basic concepts related to software architecture, rewriting logic and Real-Time Maude system. Section 3 introduces rewriting semantics of software architecture description, based on Real-Time Maude system, while section 4 presents how to perform formal verification and analysis. The following section

illustrates our approach through the case study of an aircraft aileron controller system. In section 6, we discuss related work. Finally, we conclude the paper in section 7 and give some perspectives about further work.

2. BASIC CONCEPTS

2.1 Software Architecture

Software architecture plays typically a key role as a bridge between requirements and code. By providing an abstract description (or model) of a system, the architecture exposes certain properties, while hiding others. Initially, the architectural design was largely an ad hoc affair [4]. One development has been the creation of formal notations for representing and analyzing architectural designs. ADL usually provide both a conceptual framework and a concrete syntax for characterizing software architectures [4, 9]. The use of these languages offers several advantages. Indeed, ADL allow: the reuse of architectural design specifications, rapid prototyping, supports use for evolution and re-engineering and better understanding of architectural designs through early errors detection, and quality of service analysis [4, 9].

A number of ADL have been proposed for specifying and verifying architectures, both within a particular domain and as general-purpose architecture modelling languages. As example of ADL, we note Rapide [5] and Wright [1]. There were many attempts to classify software architecture description languages [3, 6, 8, 9]. As Medvidovic and Taylor's classification and comparison framework [9] is the most recent and complete, we use their results as a basis for our work. Indeed, figure 1 identifies the desired kinds of representation, manipulation, and qualities of architectural models described in architecture description languages. We conclude from this figure that the main architectural building blocks are: components, components interfaces, connectors and configurations.

- ADL
  - Architecture modelling features
    - Components
      - Interface
      - Types
      - Semantics
      - Constraints
      - Evolution
      - Non-functional properties
    - Connectors
      - Interface
      - Types
      - Semantics
      - Constraints
      - Evolution
      - Non-functional properties
    - Architectural configurations
      - Understandability
      - Compositionality
      - Heterogeneity
      - Refinement and traceability
      - Scalability
      - Evolution
      - Dynamism
      - Constraints
      - Non-functional properties
  - Tool support
    - Active specification
    - Multiple views
    - Analysis
    - Refinement
    - Implementation generation
    - Dynamism

FIGURE 1: ADL classification and comparison framework. Essential modelling features are underlined.

2.2 Rewriting logic

The theory of rewriting logic has proved to be very useful to find a unifying framework for concurrency formalisms [10]. In [12], Meseguer showed in a number of examples that it is suitable as a common framework for concurrency. For rewriting logic, the entities in question are concurrent systems having states and evolving by means of transitions. Rewrite rules in the theory describe which elementary local transitions are possible in the distributed state by concurrent local transformations [7]. We give the definition of a labelled rewrite specification.

Definition 1. *A labelled rewrite specification is a 4-uplet $\Psi = (\Sigma, E, L, R)$, where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of labels and R is a set of pairs $R \subseteq L \times (T_{\Sigma,E}(X)^2)$ where the first component is a label and the second is a pair of equivalent classes of terms. The rewrite signature of Ψ is the equational theory (Σ, E). The elements in R are called rewrite rules and are often denoted by expressions of the form $r : [t] \rightarrow [t']$.*

Definition 2. *Given a rewrite specification Ψ, we say that Ψ entails the sequent $[t] \rightarrow [t']$ and write $\Psi \vdash [t] \rightarrow [t']$ if $[t] \rightarrow [t']$ can be obtained by finite application of deduction rules of reflexivity, congruence, replacement and transitivity.*

Readers may refer to [11] for more detailed information about rewriting logic.

In object oriented rewriting logic, object oriented message-passing style and functional styles can be freely mixed. Messages are then exchanged between objects, and cause communication events by application of rewrite rules. Concurrent rewriting modulo structural axioms of associativity, commutativity and identity capture abstractly the essential aspects of communication in a distributed object oriented configuration made up of concurrent objects and messages [7].

In recent years, several executable specification languages based on rewriting logic have been designed and implemented. Maude system is an efficient tool for systems' specification and analysis based on rewrite logic. In [14], it has been used for software architecture verification. Typically, Maude specifications are executable [2]. Maude system provides syntax for representing object oriented concepts. Maude's object oriented syntax models the concurrent state of the system (or configuration) as a multi-set of objects and messages. An object is given by the term: `< O : C | atv >`, where `O` is the object identifier instance of the class `C` and `atv` is its set of the attributes and there values. This set is given with respect to the following syntax : `a1 : v1 , ... , an : vn`, where `ai`, `i` ∈ `1..n`, are the object's attribute and `vi` their respective values. The concurrent interactions between the objects are controlled by rewriting rules. The general form of a conditional rewrite rule in Maude is as follows:

```
crl [RuleLabel] :
  M1 ... Mn < O1 : F1 | atv1 > ... < Om : Fm | atvm >
  => < Oi1 : Fi1 | atvi1 > ... < Oik : Fik | atvik>
     < Q1 : D1 | atv1" > ... < Qp : Dp | atvp" > M'1 ... M'q
if Cond .
```

where `RuleLabel` is the label of the rule, `Ms`, `s` ∈ `1..n`, and `M'r`, `r` ∈ `1..q` are messages, `i1`, ...,`ik` are numbers in `1..m` and `Cond` is the rule condition. If the rule is non conditional, then the keyword `crl` is replaced by rl and the "`if Cond`" clause is removed. For additional details about Maude's syntax, readers may refer to [2].

2.3 Real-Time Maude

Real-Time Maude is an extension of Full-Maude [15]. It is a language and a tool supporting the formal specification and analysis of real-time and hybrid systems. The specification formalism is particularly suitable to specify object oriented real-time systems. In [15], the authors introduced an approach for modelling real-time systems as real-time rewrite theories. A real-time rewrite theory contains:

- a specification of a data sort Time specifying the time domain, which may be discrete or dense,
- the sort GlobalSystem and a free constructor, denoting that t is the whole system in state t,
- ordinary rewrite rules that model instantaneous change,
- and tick rewrite rules, that model the elapsing of time in a system.

Tick rules have the following form:

```
crl [l] : {t} => {t'} in time D if cond .
```

where `t` and `t'` are of sort `System`, `D` is a term of sort `Time` denoting the duration of the tick rule and `cond` is the condition. Real-time rewrite theories are specified in Real-Time Maude as timed modules or object oriented timed modules at the user level by enclosing the module body between the keywords `tmod` and `endtm`, or between `tomod` and `endtom` for object oriented timed modules. In Maude system, we can use properties specifications, expressed in LTL and implemented model checking procedures, to check properties in a given module starting from an initial state [2]. As an extension of Maude, Real-Time Maude offers a wide range of analysis techniques, including timed rewriting for simulation purposes, untimed and time-bound search for states that are reachable from the initial state and match a given search pattern, and time-bound linear temporal logic model checking [16].

3. SOFTWARE ARCHITECTURE DESCRIPTION IN REAL-TIME MAUDE

In general, software architectures are modelled in terms of components, connectors and configurations. Components are system entities which contain information and offer services (through interfaces), while connectors ensure communication between them. A system is therefore composed of interacting components through connectors. The rules governing interaction events are specified by the means of configurations.

As previously defined in sub-section 2.1, architecture description may include other modelling elements, as semantics (high-level model of an element's behaviour), constraints (a property or assertion about a system or one of its parts, the violation of which will render the system unacceptable), non-functional properties (properties which are not entirely derivable from the specification of the element's semantics), compositionality (the mechanism that allows architectures to describe software systems at different levels of detail: Complex structure) and analysis (the ability to evaluate the properties of such systems in order to lessen the cost of any error).

The step towards building system architecture description is to identify the different elements for static and dynamic description, which can be done according to the following process:

1. Identify the components, the connectors and the configurations,
2. Identify the interfaces,
3. Identify compositionality,
4. Finally, identify the semantics.

Points 1, 2 and 3 deal with the static aspect of the system. Point 4 defines the rules governing the system's behaviour in terms of the possible actions allowed, that is its dynamic aspect. Once we have identified all these entities, we are ready to model the system.

For that, we propose Real-Time Maude, using the following modelling approach:

1. Each component is modelled by a Real-Time Maude class, whose members must include at least the attribute `computation` describing the instance status. The name of the class modelling a component is the same as the component name. The class may declare `clock` and `dly` attributes (which allow performing time advance), as it may declare any additional attributes (that characterize the component fulfilling its role).
2. Each connector is modelled by a set of rewrite rules describing communication events. These events are shown up by the mean of Real-Time Maude messages.
3. Types are modelled by sorts.
4. Configuration is a constant term of sort `System` fixed by an equation the set of initial components.

5. Each interface is modelled by a term of sort `Service`.
6. Compositionality is obtained by sub-class relationship.
7. Finally, behaviour semantic is modelled by rewrite rules. The left-hand side and guard of a rewrite rule represent the conditions that must satisfy a particular subsystem for a rule to be triggered on it, that is, what has to happen for an action to take place; its right-hand side represents the effect of such an action on such subsystem.

The following table summarizes the correspondence between software architecture concepts and the Real-Time Maude concepts used to model them.

TABLE 1: Correspondence between software architecture concepts and Real-Time Maude concepts.

Software architecture concepts	Real-Time Maude concepts
Component	Class
Component interface	A set of terms having the sort `Service` on top
Component computation	A set of rewrite rules
Connector	A set of rewrite rules
Types	Sorts
Communication events	Messages exchange
Configuration	A term having the sort `System` on top
Compositionality	Sub-class relationship

In order to carry on a modular description, the following rules are applied:

- All the declarations and rewrite rules that are relative to a given component are gathered in a distinct and single Real-Time Maude module. If the component should be equipped with a clock for time measuring purposes, then the built module is a real-time object oriented Real-Time Maude module, else it is an object oriented Real-Time Maude module.
- Connectors specifications are grouped in one single real-time object-oriented module.
- Configuration specification is set in a real-time object-oriented module that imports all components and connector modules.

Figure 2 shows a visual outline of these rules. Once the system specifications are written using this modelling approach, what we get is a rewrite logic specification of the system, which can be executed in Real-Time Maude, thus simulating its behaviour. This means that, in addition to being able to formally reason about such a system, we will have a running prototype.

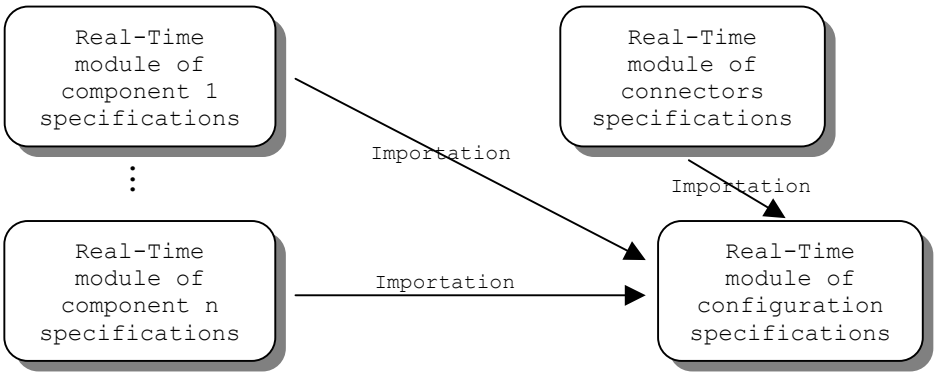

FIGURE 2: Visual outline of modular description.

4. SOFTWARE ARCHITECTURE VERIFICATION IN REAL-TIME MAUDE

Since we obtain a running prototype written in Real-Time Maude, we may perform analysis and verification tasks. Indeed, Real-Time Maude offers a wide range of analysis techniques, including timed rewriting for simulation purposes, un-timed and time-bounded search for states that are reachable from the initial state and match a given search pattern, and time-bounded linear temporal logic model checking [18]. It has been used to model and analyze scheduling algorithms [17] and sophisticated communication protocols [19].

The step towards verifying system architecture described with Real-Time Maude as introduced in the previous section is to identify these elements:

- The constraints and the non-functional properties,
- Analysis mechanisms.

The first point defines the properties that must be satisfied, while the second consists in applying Real-Time Maude simulation and analysis techniques.

An interesting issue worth pointing out is that designers may use Maude's strategies for controlling the execution or model-checking the system behaviour based on the different occurrence of the actions.

5. A CASE STUDY

In order to illustrate our proposal, we will specify in Real-Time Maude an aircraft controller system, inspired from [13]. This system is responsible for controlling the right aircraft aileron (figure 3). The system is synchronous and manages the communication between the different components. This example is a simplified aileron aircraft controller which aims at drive a motivator by generating periodically its control (2 time units period).

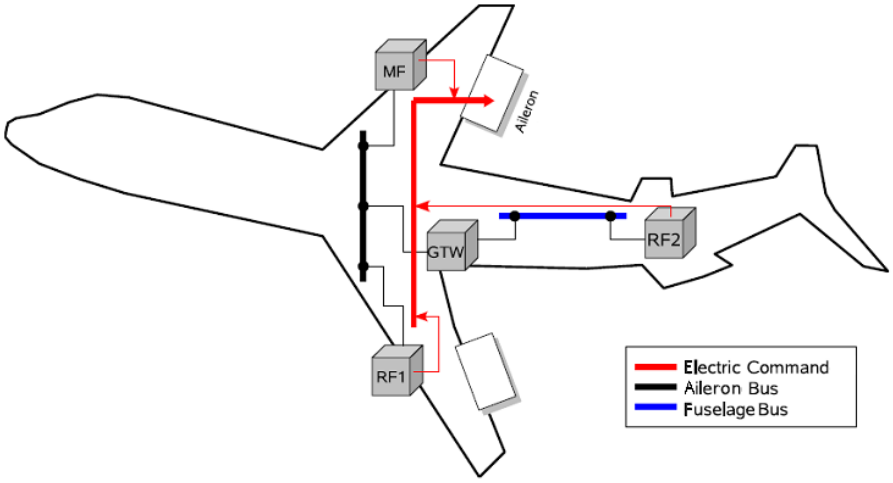

FIGURE 3: Aircraft controller system.

Since the aileron control is critical, this system is composed of three redundant functions implemented on different calculators which communicate via buses. These functions are as follows:

- The master function (MF): this function is initially in driving mode until its failure. While sending a command towards the aileron via an analogical bus, MF also sends a command towards the two other systems' functions (RF1 and RF2) via digital buses.
- The first rescue function (RF1): this function is initially in off mode. It passes in driving mode when MF fails. According to RF1, MF is considered as failing if this one did not receive any command coming from MF within 4 ms. While sending a command towards aileron, RF1 also sends a command towards the second rescue function (RF2).

- The second rescue function (RF2): this function is also initially in off mode. It passes in driving mode when both, the master function and the first rescue function, fail. According to RF2, the failure of MF and RF1 are interpreted by the non reception of any command coming from one of them within 10 ms.

This system contains two different types of buses:

- Analogical buses: these buses allow the effective transmission of the command towards the aileron. The transmission is instantaneous.
- Digital buses: these buses allow exchanging control signals between the control modules. The transmission is done within a variable elapse of time.

The communication buses are characterized by execution latencies. In table 2, we summarize the latencies of each bus.

TABLE 2: Bus communication latencies of the aileron controller system.

Bus	Minimal Latency	Maximal latency
Analogical buses connecting functions to the aileron	0 ms	0 ms
Digital bus connecting MF to RF1	0 ms	2 ms
Digital bus connecting MF to RF2	0 ms	4 ms
Digital bus connecting RF1 to RF2	0 ms	4 ms

5.1 Modelling Components

Let us begin with the system's components constituting its architecture. Following our proposed approach, we can identify here four main components, namely the master function, the first rescue function, the second rescue function and the aileron.

The master function component is modelled by the `MF` class, which has four attributes: `clock`, `dly`, computation and mode. As explained in previous section, the first two attributes are used for time measuring purposes, while the third is used for describing class computations. The additional attribute mode stores the information about the component's mode. This function may provide three services: `CmdService`, `RF1Service` and `RF2Service`, towards respectively the aileron, `RF1` and `RF2`.

```
class MF | clock : Time, dly : Time,
  computation : String, mode : String .
op CmdService RF1Service RF2Service : -> Service [ctor] .
```

The first rescue function component is modelled by the `RF1` class, which has the same attributes as `MF` class. In one hand, this function may provide two services: `CmdService` and `RF2Service`, towards respectively the aileron and RF2. In the other hand, it requires one service, which is `RF1Service`. The second rescue function component is modelled by the `RF2` class, which also has the same attributes as `MF` class. This function may provide only one service (that is `CmdService`) towards the aileron, and requires one service, which is `RF2Service`.
Finally, we have the aileron component. This last is modelled by the `Aileron` class, which has only three attributes: `clock`, `dly` and `computation`. This component requires only one service, which is `CmdService`. This service can be received from one of the three functions.

5.2 Rules modelling components behaviours

The behaviour of each component is specified through a set of rewrite rules. We start first by modelling the master function component behaviour. Indeed, this component computes the

position then sends the control towards the aileron and the other two functions at each cycle. The table 3 gives these rules.

In Real-Time Maude, the object attributes that are not relevant for a rewrite rule do not need to be mentioned. Indeed, attributes not appearing in the right-hand side of a rule will maintain their previous values unmodified [16].

In these rules, we notice the modification of computation attribute's value, which underlines the behaviour. We note as well the presence of three messages (provide messages) in the left-hand side of the second rule, which underlines communication events creation. In the third rule, the values 0 and step of dly attribute is the key concept for advancing time.

The behaviour of the first rescue function component is given in table 4. Indeed, it is initially in off mode. Each time this component receives a message (require(RF1Service)), its clock is reset to zero. If no message is received within 12 time units, the component mode becomes Cmd and it starts controlling the aileron in a way similar to the master function component.

The behaviour of the second rescue function component is similar to that of the first one. The main difference is that the time condition for moving to Cmd mode is 12 instead of 4.

The behaviour of aileron component is specified through two rewrite rules. The first describes the clock reset operation following the reception of a require message, while the second describes time advancing if such a message does not exist.

TABLE 3: Rules describing the master function component behaviour.

Master function behaviour	Real-Time Maude specification

Compute position	`rl [MF-Compute-position] :`
	`  < O : MF | mode : "cmd", computation : "Start" >`
	`=> < O : MF | mode : "cmd", computation : "CmpPos" > .`

Sending controls	`rl [MF-Sending] :`
	`  < O : MF | computation : "CmpPos" >`
	`=> < O : MF | computation : "Send" >`
	`  provide("CmdService") provide("RF1Service")`
	`  provide("RF2Service") .`

Period	`rl [MF-Period] :`
	`  < O : MF | clock : R, dly : 0, computation : "Send" >`
	`=> if R == 2 then`
	`  < O : MF | clock : 0, dly : 0, computation : "Start" >`
	`  else`
	`  < O : MF | clock : R, dly : step, computation : "Send" >`
	`  fi .`

MF failiure	`rl [MF-Failiure] :`
	`  < O : MF | >`
	`=> none .`

5.3 Modeling connectors

Connectors are responsible of service exchange between components. As we have shown is section 3, the corresponding Real-Time Maude concept is a set of rewrite rules. In our case, buses are treated as connectors. In table 5, we give the Real-Time Maude specification of these connectors.

The analogical bus behaviour is described by one rewrite rule, which transforms directly a `provide` message into a `require` message. However, the digital bus with maximal latency 2 is described by two rewrite rules. The first generates from the `provide` message a temporary message (`tempRequire`) of maximum time elapse 2. The second rule explains that the

transformation of the temporary message into a `require` message can happen at any time between the first and the second time arguments. The Real-Time Maude specification of the digital bus with maximal latency 4 is similar to that with maximal latency 2. The difference appears only in time delays of the `tempRequire` message.

TABLE 4: Rules describing the first rescue function component behaviour.

First rescue function behaviour	Real-Time Maude specification

Receive a message	`crl [RF1-Receive-require-message] :`
	`  < O : RF1 | clock : R, dly : R' >`
	`  require("RF1Service")`
	`=> < O : RF1 | clock : 0, dly : 0 >`
	`if R <= 4 .`

Advance time if no message is received	`crl [RF1-No-message-advance-time] :`
	`  {< O : RF1 | clock : R, dly : 0 > Conf}`
	`=> {< O : RF1 | clock : R, dly : step > Conf}`
	`if R <= 4 and not(existe(require("RF1Service"), Conf)) .`

Move to cmd mode	`crl [RF1-Move-to-cmd-mode] :`
	`  < O : RF1 | clock : R, dly : R', mode : "off",`
	`  computation : "Start" >`
	`=> < O : RF1 | clock : 0, dly : 0, mode : "cmd",`
	`  computation : "Cmd" >`
	`if R > 4 .`

Compute position	`rl [RF1-Compute-position] :`
	`  < O : RF1 | mode : "cmd", computation : "Cmd" >`
	`=> < O : RF1 | mode : "cmd", computation : "CmpPos" > .`

Sending controls	`rl [RF1-Sending] :`
	`  < O : RF1 | computation : "CmpPos" >`
	`=> < O : RF1 | computation : "Send" >`
	`  provide("CmdService") provide("RF1Service")`
	`  provide("RF2Service") .`

Period	`rl [RF1-Period] :`
	`  < O : RF1 | clock : R, dly : 0, computation : "Send" >`
	`=> if R == 2 then`
	`  < O : RF1 | clock : 0, dly : 0, computation : "Cmd" >`
	`  else`
	`  < O : RF1 | clock : R, dly : step, computation : "Send" >`
	`  fi .`

RF1 failiure	`rl [RF1-Failiure] :`
	`  < O : RF1 | mode : "Cmd" >`
	`=> none .`

5.4 Advancing time

Since all components clock attribute values describe elapsed time, we specify their increase with the synchronous rule below:

```
crl [tick] :
  {C:Configuration} => {delta(C:Configuration, R)} in time R
if mte(C:Configuration) == true [nonexec] .
```

In this rule, mte operation describes advancing time condition. Consequently, it is executed only when all dly attributes are different from 0 and there is no require or provide messages. Indeed, when such message appears, it should be immediately consumed. The operation delta models the effect of time elapse on the system. It increases all the clock attribute values and decreases all dly attribute values.

TABLE 5: Rules describing connectors.

Connectors behaviour	Real-Time Maude specification

Analogical bus	`rl [Analogical-bus] :`
	`  provide("CmdService")`
	`  => require("CmdService") .`

Digital bus with maximal latency 2	`rl [Digital-bus-2] :`
	`  provide("RF1Service")`
	`  => tempRequire("RF1Service", 2, 0, 0) .`
	`crl [tempRequire-to-require] :`
	`  tempRequire("RF1Service", R, R', R'')`
	`  => require("RF1Service")`
	`if R'' >= R' and R'' <= R .`

Digital bus with maximal latency 4	`rl [Digital-bus-4] :`
	`  provide("RF2Service")`
	`  => tempRequire("RF2Service", 4, 0, 0) .`
	`crl [tempRequire-to-require] :`
	`  tempRequire("RF2Service", R, R', R'')`
	`  => require("RF2Service")`
	`if R'' >= R' and R'' <= R .`

5.5 System verification

Considering the aileron aircraft controller system, we need to check these two properties:

- The aileron should not remain without control more than 16 ms (AileronCMD property).
- There must be only one function controlling the aileron at a time (NbrFCmd property).

Before expressing these properties, we need to specify the system's initial state, which is a set of instances. This is done by declaring, as follows, two types of constants: objects constants and `System` constant.

```
ops AileronIt MFInst RF1Inst RF2Inst : -> Oid [ctor] .

op initState : -> System [ctor] .

eq initState = { < AileronInst : Aileron | clock : 0, dly : 0, computation :
"Start" >
    < MFInst : MF | clock : 0, dly : 0, computation : "Start", mode : "Cmd"
>
    < RF1Inst : RF1 | clock : 0, dly : 0, computation : "Start", mode :
"Off" >
    < RF2Inst : RF2 | clock : 0, dly : 0, computation : "Start", mode :
"Off" >
  } .
```

Properties specification as well, is done by declaring constants. These constants have the term `Prop` on top. The constraint semantics is established by the mean of an equation. We illustrate properties declaration by `AileronCmd` property:

```
ops AileronCMD NbrFCmd : -> Prop [ctor] .

eq {< O : Aileron | clock : R', dly : R'', computation : S >
Rest:Configuration}
          |= AileronCMD = (R' > 16) .
```

The analysis performed using Real-Time Maude model checker gave the following results:

```
Maude > (mc {initState} |=u [] ~ AileronCMD .)
rewrites: 51475 in 116ms cpu (114ms real) (443723 rewrites/second)
```

```
Untimed model check {initState}  |=u []~ AileronCMD in AIRCRAFT-CHECK with
mode default time increase 1

Result Bool :
  true

Maude > (mc {initState} |=u [] ~ NbrFCmd .)
rewrites: 54617 in 96ms cpu (96ms real) (568891 rewrites/second)

Untimed model check {initState}  |=u []~ NbrFCmd in AIRCRAFT-CHECK with mode
default time increase 1

Result Bool :
  true
```

Therefore, we may conclude that both properties are satisfied. In addition, we can find out deadlock states. This is performed through Real-Time Maude's search command that can be unbound timed. The following execution result shows up deadlock freeness property.

```
Maude > (utsearch {initState} =>! GS:GlobalSystem .)
rewrites: 49226 in 88ms cpu (88ms real) (559348 rewrites/second)

Untimed search in AIRCRAFT-CHECK {initState} =>! GS:GlobalSystem with mode
default
          time increase 1 :

No solution
```

6 RELATED WORK AND DISCUSSION

Description and verification in architecture specification are key concepts employed in ADL. Among the proposed approaches, we will focus here on two axes: the axe that concerns the description and the axe that concerns using rewriting logic for verification in ADL.

In the first axe, the use of Real-Time Maude for specifying software architectures, among known ADL as Wright and Rapide, may present the following advantages:

- We showed in section 3 that we can express a set of essential architecture description elements using Real-Time Maude.
- Wright ADL does not support explicit time constrains expression, while Real-Time Maude supports it.
- Rapide ADL offers only three time operators that are *pause*, *after* and *delay*. However, these operators have relatively close semantics, while we can express more timing constrains using Real-Time Maude.

With regard to the verification in architecture description, there are other proposed formal notations like posets and CPS. However, Rewriting logic, which is the underlying formalism that Real-Time Maude is based on, is a common framework for concurrency formalisms. Consequently, this point emphasizes the main advantage of using simulation and analysis techniques of Real-Time Maude.

Despite the advantages we pointed out, some attributes of software architectures introduced in section 2.1 cannot be adequately captured in Real-Time Maude, like refinement and traceability [9]. Indeed, ADL may enable correct and consistent refinement of architectures into executable systems and traceability of changes across levels of architectural refinement. However, Real-Time Maude system does not yet support refinement techniques. In the same way, active specification and implementation generation cannot be captured using this system.

7 CONCLUSION

Real-Time Maude is an executable rewriting logic language especially well suited for the specification of object-oriented open and distributed systems. In this paper, we have explored the possibility of using Real-Time Maude for specifying the architecture of real time software

systems, and we are now in a position to formally reason about the specifications produced. The example of an aileron aircraft controller is used to illustrate our proposal.

Different languages have been proposed for architecture description. It has been shown that rewriting logic and its associated language Maude constitute an appropriate logical and semantic framework making much easier the integration and interoperability of different models and languages in a rigorous way. Thus, Real-Time Maude seems to be a promising option as a unifying framework for ADL.

An interesting topic of research, that is worth pointing out, is the use of the reflective capabilities of rewriting logic and Maude. These capabilities can be used for specifying and reasoning about different system properties, such as QoS, as well as specifying strategies for controlling the execution or model-checking the system behaviour based on the different occurrence of the actions.

REFERENCES

[1] Allen, R. J. (1997) *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, USA.

[2] Clavel, M. and al. (2004) *Maude manuel (version 2.1)*. SRI International, Menlo Park, CA 94025, USA (http://maude.cs.uiuc.edu/).

[3] Clements, P. C. (1996) A survey of architecture description languages. *Proceedings of the 8th International Workshop on Software Specification and Design*, pp. 16–25.

[4] Garlan, D. (2001) Software architecture. *In Encyclopedia of Software Engineering*, School of computer science, Carnegie Mellon University.

[5] Kenney, J. J. (1996) *Executable Formal Models of Distributed Transaction Systems Based on Event Processing*. PhD thesis, Department of Electrical Engineering, Stanford University, USA.

[6] Kogut, P. and Clements, P. C. (1995) Feature analysis of architecture description languages. *Proceedings of Software Technology Conference (STC'95)*.

[7] Marti-Oliet, N. and Meseguer, J. (2001) Rewriting logic: Roadmap and bibliography. *In Theoretical Computer Science*.

[8] Medvidovic, N. and Rosenblum, D. S. (1997) Domains of concern in software architectures and architecture description languages. *Proceedings of USENIX Conference, Domain-Specific Languages*, pp 199–212.

[9] Medvidovic, N. and Taylor, N. R. (2000) A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), pp 70-93.

[10] Meseguer, J. (1990) Rewriting as a unified model of concurrency. *Lecture Notes in Computer Science*, 458, pp 384–400.

[11] Meseguer J. (1992) Conditional rewriting logic as unified model of concurrency. *Theoretical computer science*, 96, pp 73–155.

[12] Meseguer J. (1996) Rewriting logic as a semantic framework for concurrency. *Proceedings of the 7th International Conference on Concurrency Theory, Lecture Notes in Computer Science*, 1119. Springer-Verlag.

[13] Aït Ameur, Y. and Delmas, R. and Wiels, V. (2004) Un cadre formel pour la spécification des systèmes avioniques. *In proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'04)*, France.

[14] Jerad, C. and Barkaoui, K. and Grissa Touzi, A. (2007) Hierarchical verification in Maude of L*f*P software architectures. *Proceedings of the 1st European Conference on Software Architecture (ECSA'07), Lecture Notes in Computer Science*, 4758, pp 156-170, Springer-Verlag.

[15] P. C. Ölveczky. *Specification and analysis of real-time and hybrid systems in rewriting logic*. PhD thesis, Department of Computer Science, University of Bergen, December 2000.

[16] Ölveczky, P. C. (2005) *Real-Time Maude 2.1 Manual*. University of Illinois (Department of Computer Science) and University of Oslo (Department of Informatics) (http://heim.ifi.uio.no/~peterol/RealTimeMaude/).

[17] Ölveczky, P. C. and Caccamo, M. (2006) Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. *Proceedings of Fundamental Aspects of*

Software Engineering (FASE'06), Electronic Notes in Theoretical Computer Science, 3922, pp 357–372.

[18] Ölveczky, P. C. and Meseguer, M. (2004) Specification and analysis of real-time systems using real-time Maude. *Proceedings of Fundamental Aspects of Software Engineering (FASE'2004), Lecture Notes in Computer Science*, 2984.

[19] Ölveczky, P. C. and Meseguer, J. and Talcott, C. (2006) Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29, pp 253–293.