How to Make AADL Specification More Precise

M. Benammar
Department of Computer Science, University of Batna, Algeria

F. Belala
Department of Computer Science
Mentouri University
Constantine, Algeria

ABSTRACT
AADL (Architectural Analysis and Design Language) is a textual and graphical language used to design and analyze software architecture of embedded real time systems. Many tools and models provide semantics and precise meaning for AADL architecture behavior. However, they are not supported by a well defined formal semantics. This paper suggets *Rewriting Logic* via its practical language Maude as an adequate formalism for modeling behavior concepts in an AADL architectural description. Besides, RT-Maude system offers a natural support to execute and prototype real-time object-oriented modules formalizing AADL architecture behavior composed of several communicating threads.

General Terms
Formal Methods, Embedded Systems, Software Architecture, Architecture Description Language (ADL).

Keywords
AADL, Behavioral Annex, Revised Rewriting Logic, Real Time Maude.

1. INTRODUCTION
In the last decade, software architecture emerged like a central concept in the software engineering. Its principal characteristic resides in the fact that it represents a significant abstract model of the structure and the behavior of software systems. Then, the problem of ensuring as early as possible the correctness of a software architecture occupies a great importance in the development life-cycle of software products. Formal methods should be used to describe software architectures and express their dynamic evolution so that one could reason on them. In particular, Architecture Description Languages (ADL) are formal notations for software architecture description of a system.

AADL (Architectural Analysis and Design Language) [1] is an Architecture Description Language which is especially effective for model-based analysis and specification of complex embedded real-time systems. It was standardized by the SAE (Society of Automotive Engineers) in November 2004.

AADL employs formal modeling concepts for the description of software/hardware architecture and non functional properties of real time systems in terms of distinct components and their interactions. It has the advantage that it assembles within the same notation the set of information concerning the application organization and its deployment. However, AADL is focused on the architectural aspects of the system components and their connections, but doesn't deal directly with their behavioral aspects. So, specification of system real-time behavior is one of major concern for AADL.

Some behavioral aspects can be described with the core of the AADL standard, such as mode change, actual behaviors of components rely on target source code. Besides, the AADL behavior annex proposed by IRIT in 2006 [2], is an extension of AADL which may offer another way to specify the behaviors of components without expressing them with the target language, therefore it can support more precise behavioral and timing analysis. However, it is not supported by a well defined formal semantics. Thus, the formal reasoning on AADL architecture or its analysis is far from being possible.

In this paper we propose a formal semantic framework based on *Revised Rewriting Logic* to describe the static structure and the behavior of an AADL architecture. We associate to each AADL component a mathematical model, represented by a revised rewrite theory $R = (\Sigma, E, \Phi, R)$, where (Σ, E) is a membership equational theory describing its static structure, including all the declared structures on the level of its AADL description. Let us note that the operators considered in Σ can freeze, in certain cases, their arguments thanks to the Φ function. The rewrite rules R describe the component behavior.

As in AADL a thread represents a sequential flow of execution and it is the only AADL component that can be scheduled, we illustrate our formalization approach on this fundamental unit of concurrent execution in AADL.

The AADL Meta model defines a thread component with two types of declarations: *Type* (defining its interfaces) and *Implementation* (defining its internal structure). Communication between threads can be realized through dataflow, call to server subprogram or access to shared variable. These various connection points are declared in the interface of the communicating components and are called features. They will be Ports, Server Subprograms or Data Access depending on the chosen communication paradigm. Its execution model specifies at runtime real-time patterns such as dispatch, communication and timing of components. Thus, the semantics of this component semantics that remains imprecise and insufficient will be formally defined using *Rewriting Logic* and its well-founded language Maude. Besides, we propose a generic implementation of this framework using of the object oriented modules of RT-Maude language (Real Time Maude) [3]. A multiset of objects and juxtaposed messages, interacting through some rewrite rules, will offer a natural and precise semantics for the execution model of an

architectural composition of threads governed by the communication mechanisms.

Maude is a declarative language, created by SRI (Stanford Research Institute) laboratory in the United States, implementing rewriting logic concepts. RT-Maude is an extension of Maude for the specification, prototyping and analysis of the real time systems.

In the remainder of this paper, we first introduce our used basic concepts of rewriting logic via its practical language Maude and the architecture description language AADL. Section 3 recalls some existing attempts for describing AADL architecture behavior. In section 4, we outline how it is possible to give a formal semantics for interacting entities (threads) defined by AADL. Object oriented real time modules of RT-Maude are built to implement execution model of thread component. Based on a case study (section 5) illustrating the proposed formalization approach, we show too how the verification process could be driven using the LTL model checker of Maude. Finally, we conclude the paper with constructive remarks and future perspectives.

2. FUNDAMENTAL CONCEPTS

In this section we give the fundamentals concepts of Maude and AADL language to facilitate the comprehension of our work. For more details, the reader can refer to [4] for Maude and [1] for AADL.

2.1 Maude Language

Rewriting logic, the theoretical basis of this work, is known as being logic of concurrent change that can deal naturally with state and with highly non-deterministic concurrent computations [5]. In this logic the basic axioms are rewrite rules of the form $t \rightarrow t'$ with t and t' expressions in a given language. There are two complementary readings of a rewrite rule, one computational, and another logical [6].

Computationally, the rewrite rule $t \rightarrow t'$ is interpreted as a local transition in a concurrent system; that is, t and t' describe patterns for fragments of the distributed state of a system, and the rule explains how a local concurrent transition can take place in a such system, changing the local state fragment from an instance of the pattern t to the corresponding instance of the pattern t'.

Logically, the rewrite rule $t \rightarrow t'$ is interpreted as an inference rule, so that we can infer formulas of the form t' from formulas of the form t.

The computational and logical viewpoints are not exclusive; they complement each other and are efficiently used to implement a wide range of logics and models of computation. So, rewriting logic has good properties as a general and flexible logical and semantic framework.

In rewriting logic, a concurrent system is represented by a rewrite theory (Σ, E, R) describing the complex structure of its states and the various possible transitions between them. Moreover, the recent extensions of this logic develop new semantic bases for a revised version [7] which supports several new characteristics. In this version of the rewriting logic, a revised rewrite theory is a four tuple $R = (\Sigma, E, \Phi, R)$ where (Σ, E) is a membership equational theory, Φ is a function assigning to each operator $f : k_1, \ldots, k_n \rightarrow k$ in Σ the subset $\Phi(f) \subseteq \{1, \ldots, n\}$ of its frozen arguments, and R is a set of labeled conditional rewrite rules. This rewrite rules can be of the form:

$$(\forall X) r : t \rightarrow t' \; if \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \rightarrow t_l'$$

Where r is the rule label, all terms $(p_i , q_i , w_j , s_j , t_l , t_l')$ are Σ-terms, and the conditions can be rewrite rules, memberships equations in (Σ, E), or any combination of both. Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ implies a formula $[t] \rightarrow [t']$ if and only if it is obtained by a finite application of the following deduction rules:

1. Reflexivity. For each term $[t]$ in $T_{\Sigma,E}(X)$, $\overline{[t] \rightarrow [t]}$
   Where $T_{\Sigma,E}(X)$ is a set of Σ-terms with variables.

2. Congruence. For each function $f \in \Sigma_n$, $n \in N$,

$$\frac{[t_1] \rightarrow [t_1'] \ldots [t_n] \rightarrow [t_n']}{[f(t_1, \ldots t_n)] \rightarrow [f(t_1', \ldots, t_n')]}$$

3. Replacement. For each rewrite rule

$$r : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)] \text{ in R,}$$
$$\frac{[w_1] \rightarrow [w_1'] \ldots [w_n] \rightarrow [w_n']}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w'}/\overline{x})]}$$

4. Transitivity.

$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Theoretical concepts of rewriting logic are implemented through Maude language, a high-performance declarative language, supporting both equational and rewriting logic specification of concurrent systems. It has been influenced by the OBJ3 equational logic language. Besides, Maude has been also used to develop, program and prototype a wide range of applications. Maude offers a comprehensive toolkit for the analysis of specifications, such as LTL model checker, Inductive Theorem Prover (ITP), Maude Termination Tool, Church Rosser Checker, Coherence Checker, etc [8].

In general, a Maude program represents a rewrite theory of rewriting logic, i.e., a signature and a set of rewrite rules. Computation in this language corresponds to the deduction in rewriting logic. Consequently, Maude supports three programming types, functional (functional theory), system (rewrite theory) and object oriented (object oriented rewrite theory). Thus, it was used successfully for the specification, prototyping, and checking of several object oriented applications [4]. A concurrent system in this case is modeled by a multiset of objects and juxtaposed messages, where the concurrent interactions between the objects are governed by rewrite rules. An object is represented by the term $< O : C \mid a_1 : v_1 , \ldots , a_n : v_n >$,where O is the object's name instance of the class C, a_i, $i \in 1..n$, the names of the object's attributes, and v_i, their corresponding values.

The class declaration follows this syntax: ***class C | $a_1 : s_1$, … , $a_n : s_n$.***,where ***C*** is a class's name and ***s_i*** is the sort of ***a_i*** attribute. It is also possible to declare the subclasses and profit from the heritance concept. The messages are declared by using the key word ***msg***. The general form of a rewrite rule in Maude's object oriented syntax is:

crl [r] : M_1 … M_n < O_1 : F_1 | at_1> … < O_m : F_m | at_m > => < O_{i1} : F'_{i1} | at'_{i1} > … <O_{ik} : F'_{ik} | at'_{ik} > M_1' … M_p' if Cond .

Where ***r*** is the label of the rule, ***M_S, S ∈ 1..n,*** et ***M'$_u$, u ∈ 1..p*** are messages, ***O_i, i ∈ 1..m***, and ***O_{il}, l ∈ 1..k***, are objects, and ***Cond*** is the rule's condition. If the rule is unconditional, we replace the key word ***crl*** by ***rl*** and we remove the clause ***if Cond***.

There are two main frameworks in Maude: (1) the Core Maude interpreter implemented in C++ and providing all of Maude basic functionalities and (2) the Full Maude, an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible module algebra than that available in Core Maude. RT-Maude language belongs to this interpreter. RT-Maude system [3] is a tool allowing the specification and analysis of the real times systems. It is particularly appropriate to the object oriented real time systems. In RT-Maude, an object oriented real time rewrite theory contains the following features:

- Specification of a data sort ***Time*** specifying time domain, which may be discrete or dense.
- The sort ***GlobalSystem*** and a free constructor ***{_}***, denoting that ***{t}*** is the whole system in the state ***t***,
- The ordinary rewrite rules model the instantaneous change
- And a particular rewrite *tick* rule, having the form ***crl [l] : {t} => {t`} in time D if cond,*** modeling the advance of a time ***D*** in the system whose state is ***t*** if a condition is checked.

RT-Maude language has been already used to simulate and analyze a set of real time and hybrid systems such as communication protocols [9], CASH scheduling algorithm [10], wireless sensor network algorithms [11], the *LfP* architecture description language [12], the reconfigurable and time constrained workflows [13], etc.

2.2 AADL Language

The standard version 1.0 of AADL (Architecture Analysis and Design Language) was published in November 2004 under SAE (Society of Automotive Engineers) authority. It is dedicated to the description of the real time embedded systems. Its advantage is that it includes within the same notation the whole information concerning the application organization and its deployment. AADL describes embedded system architecture using a set of interconnected components. It is based, like any other ADL, on the concepts of components connections and configurations.

AADL Components. The abstract declaration of AADL component is composed of *component type* and one or more *component implementations*. The *component type* declaration contains sub clauses representing: the features, the flows, and the property associations. A *component implementation* specifies an internal structure in terms of *subcomponents*, *connections* between the features of those subcomponents, *flows* across the subcomponents, *modes* to represent operational states of the system, and *properties*. Several categories of AADL component exist: *threads* are the schedulable units for the concurrent execution, *processes* represent spaces of virtual addresses, and *systems* support the hierarchical organization of the *threads* and *processes*. AADL also describes the execution platform in terms of *processors* which support the execution of threads, *memory* for the storage of data and code, and *bus* for the physical interconnection.

For instance, *AADL thread* is an active applicative component. Its implementation conditions are specified as properties: *deadline*, *dispatch protocol*, *period*, etc. The standard defines four types of dispatch protocol such as: *Periodic* (thread being executed with intervals of times), *Aperiodic* (thread starts by events invocation), or *Sporadic* (thread involves a limit for the rate of sporadic execution). Threads communicate through *data port*, *event port* or *event data port*, and/or the *data port group*.

AADL Connections. Connections specify the patterns of a control and data flow between different components at runtime. It is composed of several connections across the subcomponents. It has an ultimate source and an ultimate destination belonging to thread or device category of component.

AADL Configurations. AADL configurations of a system are represented by assemblies of software and hardware components. A configuration represents a graph of *components* and *connectors*. The *connectors* for AADL are specified by flows and modes. *Flows* indicate that the logical information which binds the out port (or port group) of departure component named (*flow source*) and the in port (or port group) of arrival component (*flow sink*). The *modes* represent the operational states of software, execution platform, and the compositional components in the modeled physical system. A change in mode can change the whole of active components and connections.

3. BEHAVIOR DESCRIPTION IN AADL ARCHITECTURE

AADL employs formal modeling concepts for the description of software/hardware architecture and non functional properties of embedded real time systems in terms of distinct components and their interactions. AADL language is precisely defined and stabilized as regards structural aspects. The behavioral aspect is described in a behavior annex which is not entirely validated. Also, AADL execution semantics is defined in the standard. But, this semantics will be improved to adapt it to possible variations as for instance, target platforms ones.

The behavior description is defined in an appendix that describes only action sequencing, sending and receiving messages as well as the temporal events. Operational semantics is defined in the standard with regard to process and thread management using just an operational mode and mode transitions. The mode transitions represent the commutation between system execution configurations. These can have the effect of activating and deactivating threads for execution, or still a changing the pattern of connections between threads, and finally the changes in component-internal characteristics. Consequently, a thread can be

active in a mode and inactive in another, only the active threads execute their instructions.

Thread behavior semantics is defined in the standard on the basis of automata representing thread states and the transition between these states. Figure 1 shows that a *thread* can be stopped, inactive, or in activity. An active thread can be waiting for dispatch, *AwaitingDispatch* state, or in execution, *Compute* state.

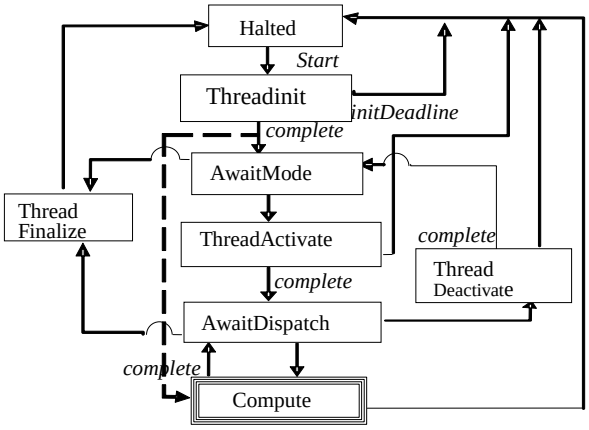

Figure 1. The state/transition model for AADL thread

Moreover, in the '*Compute*' hierarchical state of this automata, the thread can have others substates (figure 2) such as *Ready* (ready for execution), *Running* (in execution), or *Awaiting-Resource* (be blocked on the access of a resource).

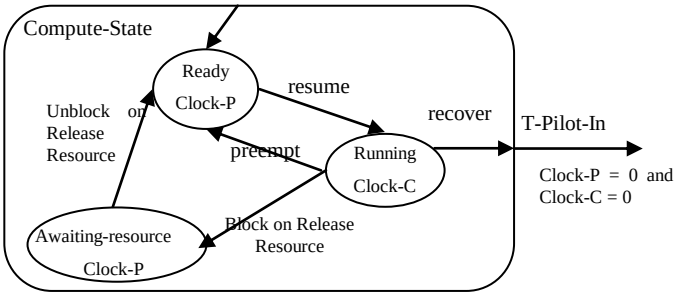

Figure 2. *'Compute'* hierarchical state of a thread

At this level of the *Compute* state, local state changes of the thread are visible through its configurations when, at runtime, it receives data and/or events, executes computation and sends signals (data and/or event) throughout ports. This execution model does not appear at the level of the thread AADL description, it has been recently described by some existing works, but its semantics remains difficult to define with an adequate level of abstraction [2].

An abrupt work is being done in literature tempting to formalize AADL architectural description while transforming it to other models that come with some convenient tools for example: TINA [14] or CADP [15]. Indeed, some well known formalisms such as timed automaton [16], Timed Petri Nets (TPN) [17, 18], real time process algebra (ACSR) [19] or Timed Abstract State Machine (TASM) [20] are recently used to provide AADL architectural description formal models. Nevertheless, all these contributions are restricted to only some AADL concepts formalization and the thread complex behavior is not formally defined. In this paper we would like to explore another alternative for describing a well-defined structure of a thread, its execution model and so its behavior.

4. A Semantic Model for AADL Architecture

The aim of this paper is to propose an unified semantic framework, based on object-oriented real-time rewrite theories, to specify AADL architecture of a real time embedded system. The proposed model is described in this case, as being a multiset of juxtaposed objects and messages, where the concurrent interactions between the objects are governed by rewrite rules. Thus, we propose a generic set of mapping rules allowing to abstract the most significant architectural elements of AADL in the syntax of rewriting logic. Then, we define a formal semantics of an AADL architecture composed of several communicating threads. On the one hand, we formalize the structure and the behavior of these execution concurrent units, and on the other hand, we formal check the flat execution model of this AADL architecture. We consider both the flow across declared connections and the execution properties. The implementation of this proposed model is achieved thanks to the RT-Maude system and tools that exist around it.

Table 1. Mapping AADL architectural elements to RT-Maude object-oriented concepts

AADL Architectural Element	RT-Maude Object -Oriented Concept
AADL Architecture	Real-time object-oriented rewrite theory
Thread component	*Thread* class
Component functional Interface	*PortState* sort for *IPort* and *OPort* attributes , *File* sort for *AccessData*, *InBufferPort* and *OutBufferPort* attributes
Thread state	*ThreadState* sort
Thread Configuration	Conditional Rewrite rules
Interaction	Transmit Message between objects, instances of thread class
Flow latency	Message transmission time
Thread Implementation	*ThreadImpl* class (subclass of class Thread)
Temporal execution properties	*Time* sort for *Period* and *Execution-time* Attributes

Our approach consists in associating to each AADL architectural element a formal concept of an extended real time rewrite theory which is implemented as RT-Maude module (see table 1).

According to these mapping rules, each AADL architecture may have a formal mathematic support represented by a real time extended rewrite theory $\mathcal{R} = (\Sigma, E, \Phi, R)$, where (Σ, E) is a membership equational theory describing the architecture static structure. The Σ signature specifies the set of sorts and subsorts, and the set of the useful operators to describe each clause of its total description: *features*, *properties*, *modes*, *type* and *implementation*. The set of the E equations contains in addition the attributes associated to some operators. The Φ function represents the set of operators considered in Σ which can freeze,

in some cases, their arguments. The rewrite rules ***R*** describe the components behavior according to follow-up of their configurations. Each component category belonging to the declared AADL architecture will be represented by a special class of the corresponding real time object oriented rewrite theory. We are particularly interested in this paper by the thread component of AADL as it is the fundamental unit of execution. So, it will be best considered to illustrate and motivate our proposed formal setting for AADL components.

As shown in table 1 and figure 3, thread type is modeled by the *Thread* class, whose attributes are respectively: 1) the functional interfaces *IPort* and *OPort*, 2) data subcomponent, represented as a buffer, related to each connection port *InBufferPort*, *OutBufferPort* and 3) *TState* to specify its state. The *PortState* sort represents its functional interfaces states.

Thread component internal structure or its implementation is specified by the *ThreadImpl* class (figure 3) having the attributes: 1) *Sstate* specifying substate of the "*Compute"* composite state, 2) *Time* (predefined sort) allowing to specify both temporal execution properties (*Period* and *Execution-Time*) and the *Clock-P* and *Clock-C* clocks for the warning of the period and execution time of the thread. It is obvious that this *ThreadImpl* class is declared as a subclass of the *Thread* class to profit from the inheritance concept.

```
class Thread | IPort : PortState, OPort :
PortState, TState : ThreadState, InBufferPort
: File,  AccessData : File, OutBufferPort :
File,  MaxPreempt : Nat, MaxData : Nat, NBS :
Set .
```

```
class ThreadImpl |  Substate : Sstate, Period
: Time,Execution-time : Time, Clock-P : Time,
Clock-C : Time .
```

```
subclass ThreadImpl < Thread .
subsort Sstate < ThreadState .

msg from_to_transfer_ : Oid  Oid Data -> Msg .
sort DlyMsg .
subsorts Msg < DlyMsg < Configuration .
op dly : Msg Time -> DlyMsg [ctor] .
var ms : Msg .
eq dly(ms, 0) = ms .
```

Figure 3. RT-Maude specification of AADL thread component

We represent the passage of data flow and/or event through threads connection, by messages transmission (figure 3) between a thread and their neighbors (threads). Each message declaration specifies the source and destination threads and the transmitted data/event type. The *source* and *destination* components are object instances of the *Thread* class. Message transmission time is taken into account by the *DlyMsg* sort modeling the flow latency time. The operator *dly(m, t)* indicates that it remains ***t*** time units for the arrival of the *msg* ***m*** (to its destination).

All static concepts, involved in the thread specification, are given in the corresponding real time object oriented module. We omit their presentation for paper ligibility.

The thread behavior formalization in this case must begin by a detailed and a complete modeling of concurrent and hierarchical thread states.

So, we declare, with constructor operators, the thread states (*wait* or *compute*), the thread substates (*Ready*, *Running*, *Awaiting-Resource*, *Complete* and *noSub*) and the port states (*waitIn*, *waitOut*, *receive* and *send*). Then, the behavior formalization aspect starts with the specification of the visible changes of the thread states, associated to its connection ports and materialized by the rewrite rules *Data-Receive* and *Data-send* (see figure 4). The *Data-Receive* rule prepares thread for a new execution period, after receiving a message. It changes the thread state from *wait* to *compute* and its substate from *noSub* to *Ready* initializing the clocks by the values, specified in the execution properties.

```
rl[Data-Receive]:(from T1 to T2 transfer DT) <
T2: ThreadImpl | IPort: waitIn, TState : wait,
Substate: noSub , Clock-P: 0, Clock-C: 0,
Period: R , Execution-time : R', InBufferPort:
L2 >
=>  < T2 : ThreadImpl | IPort: receive, TState:
compute , Substate: Ready, Period: R,Execution-
time: R', Clock-P: R, Clock-C: R', MaxData: 0,
MaxPreempt: 0, InBufferPort: DT; L2 >.
```

```
rl[Data-Send]: < T1: ThreadImpl | OPort:
waitOut, TState: compute, Substate: Complete,
OutBufferPort: L; DT, NBS:(T2 & R), Clock-P:
R1, Clock-C: R2, MaxPreempt: N, MaxData: N1 >

=>  < T1: ThreadImpl |  OPort: waitOut, TState:
wait , Substate: noSub, OutBufferPort: L , NBS:
T2 & R , Clock-P: 0, Clock-C: 0, MaxPreempt: 0,
MaxData: 0 > dly(from T1 to T2 transfer DT, R).
```

Figure 4. RT-Maude specification of AADL connection and interaction

The *Data-Send* rule putts the thread in its initial state after a period elapse and an execution time. It transforms the thread state from *compute* to *wait* and its substate from *Complete* to *noSub* and then, generates the message with the thread execution result for the transmission.

In this formalization, we take into account the composite state of the thread in its active hierarchical state *compute*. We define its substates and their corresponding transitions (see figure 2) and also the execution properties (*Compute-Execution-Time* and *Period*). The first rewrite rule, in figure 5, changes the thread substate from *Ready* to *Running* and prepares the received data treatment. The rewrite rule *finish* considers the particular case, where the thread doesn't have an out port and gives its initial state after elapse of period time.

```
Crl [init] : < Imp : ThreadImpl |InBufferPort:
L, AccessData: EmptyFile, Substate: Ready,
Clock-C: R'>
=>  < Imp: ThreadImpl | InBufferPort: Queu(L),
AccessData: Head(L), Substate: Running,  Clock-
C: R' > if (R' > 0) .
```

```
...
```

```
crl [complete-Rec] : < Imp: ThreadImpl | IPort
: receive , TState :  compute, Substate: subS ,
AccessData: Temp2 , OutBufferPort: L,
OPort: waitOut, Clock-P: R, Clock-C : R' >
=>  < Imp: ThreadImpl | IPort: waitIn , TState:
compute,  AccessData: EmptyFile ,
OutBufferPort: Temp2 ; L , Substate: Complete,
 OPort : waitOut , Clock-P : R, Clock-C : R' >
if (R == 0) .
```

```
rl[finish]: <Imp : ThreadImpl | IPort: receive,
TState: compute, Substate: Complete , OPort:
NoPort , Clock-P: R, Clock-C: R',
MaxPreempt: N, MaxData: N1 >
=>  < Imp: ThreadImpl | IPort: waitIn, TState:
wait, Substate: noSub, OPort: NoPort, Clock-P :
0, Clock-C : 0 , MaxPreempt : 0 , MaxData : 0
> .
```

```
crl [tick] : {C:Configuration}
=>   {delta(C:Configuration, R)} in time R if
(R <= mte(C:Configuration)) /\ (not
agerEnabled(C:Configuration)) [nonexec] .

eq delta(< Imp : ThreadImpl | Substate :
Running, Clock-P : R , Clock-C : R' >, R'') =
< Imp : ThreadImpl | Substate : Running, Clock-
P : R monus R'' , Clock-C : R' monus R'' > .

ceq mte(NeC NeC') = min(mte(NeC), mte(NeC')) if
(NeC =/= none) /\(NeC'  =/=  none) .
```

Figure 5. RT-Maude specification of AADL thread behavior

The conditional rewrite rules *resume*, *preempt*, *block-on-Release-Resource*, *Unblock-on-Release-Resource*, *recover* and *complete-rec* define the transitions between substates of the *compute* state. The *mte* function is used to calculate the maximum time elapse value possible from a given thread state, before a significant action is taken (here, it is about the minimum values of the two clocks). Moreover, the *delta* function models the effect of passage of a time *R* on the thread by decreasing one or both of its clocks according to the time elapsed. These two functions are used in the *tick rule* in order to calculate and apply the time elapse on the thread configuration (figure 5).

The *nonexec* attribute of the *tick rule* (figure 6) precises that this rule advances time when no other rule is executable. The *delta* operation modifies only the attributes of sort *time*. The equations in figure 6 calculate the *delta* operation effect on thread configuration and on the messages transmission. We introduce send and reception time for each message and we express too, the distribution of the *delta* operation on the whole thread configuration to make the time elapsing uniformly.

The *mte* operation evaluation considers a thread (if it is at *compute* state) with its clocks initialized by the execution properties values. Then, it considers the distribution of the *mte* operation on the configuration.

This formalization approach gives a real-time object-oriented rewrite module in RT-Maude, providing an executable mathematical model of AADL architectural system, composed essentially of several communicated threads. We can use these RT-Maude specifications to simulate and analyze the concurrent system behavior so formalized.

Under appropriate conditions, we can check that our mathematical model satisfies some important properties, or obtain a useful counter example showing that the property in question is violated. For instance, we can model check any linear temporal logic (LTL) property of AADL architectural system. We will deal with very useful properties such as *accessibility*, *safety* and *liveness*, while considering the formal description of AADL thread behavior in its "*compute"* particular state (see GPS example section).

5. A CASE STUDY: GPS EXAMPLE

We illustrate our proposed AADL formalization approach through a case study describing a GPS system example. This system should display the current position information for the user. It is composed of one sensor *GPS* and two threads: *TGPS* and *TScreen*. The *GPS* sensor captures information parameters from satellite and sends it to thread *TGPS*.

```
system  GPSyst
end GPSyst
```

```
system implementation GPSyst.impl
subcomponents
  GPS: device  GPS.impl;
  TGPS: thread  TGPS.impl;
  TScreen: thread  TScreen.impl;
connections
  data port GPS. OutBufPort -> TGPS.InBufPort;
  data port TGPS.OutBufPort ->
  TScreen.InBufPort;
end GPSyst.impl;
```

```
thread TGPS
  features
    InBufPort : in data port DataType;
    OutBufPort : out data port DataType;
end TGPS;
```

```
thread implementation TGPS.impl
   properties
     Dispatch-Protocol => Periodic ;
     Compute-Execution-Time => 10 ms ;
     Period => 20 ms ;
   end TGPS.impl;
```

```
thread TScreen
  features
    InBufPort : in data port  DataType;
    OutBufPort : out data port DataType;
end TSreen;
```

```
thread implementation TScreen.impl
   properties
     Dispatch-Protocol => Periodic;
     Compute-Execution-Time => 7 ms ;
     Period => 15 ms ;
 end TScreen.Impl;
```

Figure 6. AADL description of GPS example

TGPS reads these parameters and converts them into an internal representation, then it sends the result to thread *TScreen*. This one displays the recent position received periodically.

AADL description of thread *TGPS* in this architecture (figure 6) shows that is a periodic thread which operates according to temporal properties values: *Period* and *Compute-Execution-Time.*

This AADL model gives only static description of components and their connections including, for each thread description, the specification of implementation conditions. This is done by the properties declaration, such as: dispatch protocol, period and Compute-Execution-Time and their various values. We exploit this AADL declaration to describe the thread behavioral aspects using our proposed approach.

The obtained RT-Maude module is so generic and serves to simulate and analyze the any system behavior particularly the GPS one. It is clear that the code portion of this module in figure 7 defines an instantiation of the initial state of the considered architecture configuration declared in the equation part "*initstate"*. Similarly, we may formalize any system example.

```
ops GPS  TGPS TSCREEN : -> Oid [ctor] .
op initState : -> GlobalSystem .

eq initState = {(from GPS to TGPS transfer
data1 ) < TGPS: ThreadImpl |IPort : waitIn,
TState : wait, Substate : noSub , OPort :
waitOut, InBufferPort : EmptyFile, AccessData
: EmptyFile, OutBufferPort : EmptyFile,
Period : 20,  Execution-time : 10, Clock-P :
0, Clock-C : 0, MaxData : 0, MaxPreempt : 0 ,
NBS : (TSCREEN & 4) >

 < TSCREEN : ThreadImpl | IPort : waitIn,
TState : wait, Substate: noSub, OPort:
NoPort, InBufferPort : EmptyFile, AccessData
: EmptyFile, OutBufferPort : EmptyFile ,
Period : 15, Execution-time : 7, Clock-P: 0,
Clock-C: 0, MaxData : 0, MaxPreempt: 0 , NBS
: EmptySet >}.
```

Figure 7. The Maude model of the AADL GPS example

In addition, for each thread of this AADL architecture example, a check of the following property **:**

The final state (*Complete* substate) of the thread execution process can be reached in time, by the LTL model-checker of RT-Maude is launched by this command (figure 8):

```
 (mc initState1 |=t (<> CompleteStateTGPS)/\
(<>  CompleteStateTSCREEN)     in time <= 70 .)
```

This means that the thread execution finishes correctly (substate = *complete* reachable in time). The screen shot of figure 8 shows that this property for instance is then evaluated to true with this solution.

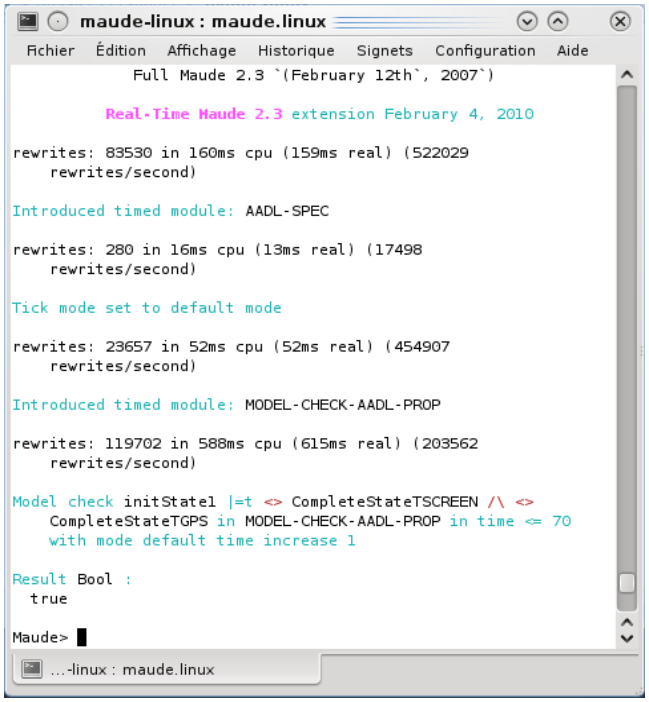

Figure 8. The model-check of AADL property example

For this purpose, we build the module *MODEL-CHECK-AADL-PROP* (figure 9) which imports the predefined module *TIMED-MODEL-CHECKER* and the module *AADL-SPEC* containing all the architecture specification of our *GPS* example including the *TGPS* thread specification. The specification of the previous property, of sort *Prop*, is made through the atomic proposition: *CompleteStateTGPS* and its corresponding equation.

```
(tomod MODEL-CHECK-AADL-PROP is
 including TIMED-MODEL-CHECKER .
 protecting AADL-SPEC .
 ops  CompleteStateTGPS : -> Prop [ctor].
 var REST : Configuration .
 var Imp : Oid .
 vars R R' R'' : Time .

eq {REST  < TGPS :  ThreadImpl |   Substate :
   Complete   >} |= CompleteStateTGPS = true .

endtom)
```

Figure 9. MODEL-CHECK-AADL-PROP module

6. CONCLUSION

AADL describes embedded system architecture using a set of interconnected components, abstracting away the functionality of components that is not precisely known at early stages of system development. This article deals with the formalization behaviour of its components, especially the thread component, known as the fundamental unit of concurrent execution in AADL. For such a

purpose, we have chosen *Rewriting Logic* via its practical language Maude as the underlined formalism for the proposed semantic framework.

This article presents a first step to achieve behaviour formalization of AADL architecture. The next steps must refine and complete the set of real-time object-oriented modules presented above. Other architectural elements must also be investigated (eg. AADL properties), we will extend the proposed formal analysis approach to other property kinds specifying space constraints and temporal ones.

7. REFERENCES

[1] SAE International 2008 Architecture Analysis and Design Language (AADL) Standard, Version 2. SAE Draft Standard AS5506 V2.

[2] Dissaux, P., Bodeveix, J.P., Filali, M., Gaufillet, P., and Vernadat, F. 2006 AADL Behavioral Annex. In Proceeding of the DASIA'06, Data Systems in Aerospace- Conference, Berlin, Germany, ESA SP-630.

[3] Ölveczky, P. C. 2007 Real-Time Maude 2.3 Manual. Department of Informatics, University of Oslo.

[4] Clavel, M., Duran, F., Eker, S., Marti-Oliet, N., Lincoln, P., Meseguer, J., and Talcott, C. 2008 Maude Manual (Version 2.4).

[5] Meseguer, J. 1992 Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96(1), pp. 73-155.

[6] Marti-Oliet, N., Meseguer, J. 1993 Rewriting logic as logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory.

[7] Bruni, R., Meseguer, J. 2006 Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360, pp. 386-414.

[8] Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J. F. 2002 Maude: Specification and programming in rewriting logic", Theoretical Computer Science, pp. 187–243.

[9] Ölveczky, P. C., Meseguer, J., and Talcott, C. 2006 Specification and Analysis of the AER/NCA Active Network Protocol suite in Real-Time Maude. In Formal Methods in System Design, vol. 29, pp. 253-293.

[10] Ölveczky, P. C., and Caccamo, M. 2006 Formal Simulation and Analysis of CASH Scheduling algorithm in Real-Time Maude. Electronic Notes in Theoretical Computer Science, L. Baresi and R. Heckel editors, Vol. 3922, pp. 357-372.

[11] Ölveczky, P. C., and Thorvaldsen, S. 2006 Formal Modeling and Analysis of Wireless Sensor Network Algorithms in Real-Time Maude. In IPDPS'2006: Parallel and Distributed processing Symposium.

[12] Jerad, C., Barkaoui, K., and Grissa-Touzi, A.. 2007 Hierarchical Verification in Maude of LfP Software Architectures. In ECSA'07, 1st European Conference on Software Architecture, Aranjuez (Madrid), LNCS , Springer, pp. 156-170

[13] Barkaoui, K., Boucheneb, H., and Hicheur, A. 2008 Modeling and Analyzing Time-Contrained Flexible Workflows with Time Recursive Petri Nets. In 5th International Workshop on Web Services and Formal Methods Co-Located with BPM 2008, LNCS volume 5387, Springer R. Bruni, pp. 19-35.

[14] Berthomieu, B., Ribet, P.-O., and Vernadat, F. 2004 The tool TINA-construction of abstract state spaces for Petri nets and time Petri nets, International Journal of Production Research, Vol. 42, pp. 2741-2756.

[15] Garavel, H., Mateescu, R., Lang, F., Serwe, W. 2007 CADP 2006 : A toolbox for the construction and analysis of distributed processes, In Werner Damm and Holger Hermanns editors CAV, LNCS, Springer vol. 4590, pp. 158-163.

[16] Chkouri, M., Robert, A., Bozga, M., and Sifakis, J. 2008 Translating AADL into BIP -Application to the Verification of Real-time Systems, In Proc. of MoDELSACES-MB Model Based Architecting and Construction of Embedded Systems, pp. 39-54.

[17] Renault, X., Kordon,, F., Hugues, J. 2009 Adapting models to modelcheckers, a case study : Analyzing AADL using Time or Colored Petri Nets. In Proceedings of the 20th International Symposium on Rapid System Prototyping, IEEE Computer Society, pp. 26-33.

[18] Renault, X., Kordon, F., Hugues, J. 2009 From AADL architectural models to Petri Nets: Checking model viability. 12th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'09), IEEE Computer Society, pp. 313-320.

[19] Sokolsky, O., Lee, I., and Clarke, D. 2009 Process-Algebraic Interpretation of AADL Models. 14th International Conference on Reliable Software Technologies, LNCS 5570, pp. 222-236

[20] Yang, Z., Hu, K., Ma, D., and Pi, L. 2009 Towards a Formal Semantics for The AADL Behavior Annex. In Design, Automation & Test in Europe DATE 2009, pp. 1166-1171.