Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude

Peter Csaba Ölveczky, Artur Boronat, José Meseguer

Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude

Peter Csaba Ölveczky[1], Artur Boronat[2], and José Meseguer[3]

[1] University of Oslo
[2] University of Leicester
[3] University of Illinois at Urbana-Champaign

Abstract. AADL is a standard for modeling embedded systems that is widely used in avionics and other safety-critical applications. However, AADL lacks a formal semantics, and this severely limits both unambiguous communication among model developers, and the development of simulators and formal analysis tools. In this work we present a formal object-based real-time concurrent semantics for a behavioral subset of AADL in rewriting logic, which includes the essential aspects of its behavior annex. Our semantics is directly executable in Real-Time Maude and provides an AADL simulator and LTL model checking tool called *AADL2Maude*. *AADL2Maude* is integrated with OSATE, so that OSATE's code generation facility is used to automatically transform AADL models into their corresponding Real-Time Maude specifications. Such transformed models can then be executed and model checked by Real-Time Maude. We present our semantics, and two case studies in which safety-critical properties are analyzed in *AADL2Maude*.

1 Introduction

AADL [15] is both a modeling language for real-time embedded systems and an international standard widely used in industry. It has features to model the real-time aspects of embedded systems and to represent both the software and hardware architectures of the components making up such systems. It does however lack a formal semantics. This lack is particularly important for real-time embedded systems, because many of them —in areas such as avionics, motor vehicles, and medical systems— are *safety-critical* systems, whose failures may cause great damage to persons and/or valuable assets. Furthermore, AADL models are not executable, which limits not just the possibility of formal analysis of their safety and liveness properties, but even the possibility of simulating them.

It seems clear that overcoming these limitations of AADL is highly desirable, but requires in an essential way the use of formal methods, because in the absence of a precise mathematical semantics any pretense of achieving formal verification is meaningless. Furthermore, these formal methods should be supported by tools that are integrated into the AADL tool chain. A further, highly desirable goal is to have a formal semantics of AADL that can be used to automatically generate *formal executable specifications* of AADL models, since the first and most

basic way of analyzing AADL models should be the capacity to *simulate* such models; and since such formal executable specifications can then also be used for automatic verification of safety and liveness properties by *model checking*.

This paper reports on our experience in defining an object-based real-time concurrent formal semantics for a substantial subset of AADL in the Real-Time Maude formal specification language [14]; and in directly using this semantics to simulate and formally analyze AADL models. We have found Real-Time Maude particularly well suited for this task for the following reasons:

- *Support for nested objects*. AADL models are structured in nested hierarchies of components. Much would be lost in translation if such structure were not preserved. Real-Time Maude's support for object classes with a "Russian dolls" nested structure (see [10]), provides an essentially isomorphic formal counterpart for an AADL model.
- *Support for real-time concurrency*. All real-time aspects of AADL, as well as the concurrent interactions between AADL components, can be directly and naturally formally modeled by means of *real-time rewrite theories*.
- *Wide range of formal analysis capabilities*. By automating the AADL formal semantics with the *AADL2Maude* tool, one can automatically generate formal executable specifications of AADL models in Real-Time Maude for *simulation*, *reachability analysis*, and *LTL model checking purposes*.
- *Completeness of the formal analysis*. In spite of the generality of the AADL models, their object-based semantics ensures that *time-bounded LTL properties* are *decidable* under very mild checkable conditions [13].

Our Contribution. To the best of our knowledge (see the related work discussion in Section 4), our work is the first that provides a *formal executable semantics* for AADL models with different modes, and whose thread behavior is specified in AADL's behavior annex; it is also the first that supports simulation, reachability, and LTL model checking of such models *directly in the semantic formalism itself* through the *AADL2Maude* tool. *AADL2Maude* is an OSATE plug-in that uses OSATE's code generation facility to automatically generate Real-Time Maude specifications from AADL models. Furthermore, our semantics directly supports *hierarchical objects* that communicate asynchronously with each other in real time and capture the hierarchical nature of AADL components. This makes the representational distance between the original AADL model and its rewriting logic semantics quite small, making it easier to understand the results of formal analysis. User-friendliness is also enhanced by a syntax for state predicates based on AADL notation to ease the specification of LTL properties. Finally, we summarize two case studies, one on safe interoperation of medical devices and one on the safety of avionics systems, demonstrating the usefulness of this semantics and tool in concrete examples.

The paper is organized as follows. Section 2 gives a brief introduction to AADL and Real-Time Maude. Section 3 presents the Real-Time Maude semantics of a behavioral subset of AADL, and shows how AADL models can be formally analyzed in *AADL2Maude*. Section 4 discusses related work on the use of formal methods for AADL, and Section 5 presents some concluding remarks.

2 Preliminaries on AADL and Real-Time Maude

AADL. The *Architecture Analysis & Design Language* (AADL) [15] is an industrial standard used in avionics, aerospace, automotive, medical devices, and robotics communities to describe a performance-critical embedded real-time system as an assembly of software components mapped onto an execution platform.

An AADL model describes a system as a hierarchy of hardware and software components. A component is defined by its *name*, its *interface* consisting of input and output ports, its *subcomponents* and their interaction, and other type-specific *properties*. System components are the top-level components, and can consist of other system components as well as of hardware and software components. Hardware components include: *processor* components that schedule and execute threads; *memory* components; *device* components representing devices like sensors and actuators that interface with the environment; and *bus* components that interconnect processors, memory, and devices. Software components include: *thread* components modeling the application software to be executed; *process* components defining protected memory that can be accessed by its thread subcomponents; and *data* components representing data types. In AADL, thread behavior is typically described using AADL's *behavior annex* [6], which models programs as transition systems with local state variables.

An AADL model specifies how the different components interact and are integrated to form a complete system. The AADL standard also describes the runtime mechanisms for handling message and event passing, synchronized access to shared resources, thread scheduling when several threads run on the same processor, and dynamic reconfiguration that are specified by *mode transitions*.

AADL has a MOF meta-model, and the OSATE modeling environment provides a set of plug-ins for front-end processing of AADL models on top of Eclipse.

Real-Time Maude. A Real-Time Maude [14] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [5] theory with Σ a signature[4] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written with syntax `rl [`l`] :` t `=>` t', where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t'. The rules are applied *modulo* the equations E.[5]
- TR is a set of *tick (rewrite) rules*, written with syntax

[4] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[5] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

`rl [`l`] : {`t`} => {`t'`} in time` τ `.`

that model time elapse. `{_}` encloses the global state, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [5] for more details on the syntax of Real-Time Maude.

In *object-oriented* Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a state is represented as a term `<` O `:` C `|` $att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a *concurrent* object-oriented system, the state is a term of sort `Configuration`. It has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : 0, a2 : y, a3 : w, a4 : z >  =>
          < O : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

defines a parametrized family of transitions which can be applied whenever the attribute `a1` of an object `O` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. "Irrelevant" attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system *up to a certain duration*. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state. The command which searches for n such states has syntax `(utsearch [`n`]` t `=>*` *pattern* `such that` *cond* `.)`.

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions*, possibly parametrized, can be predicates characterizing properties of the state and/or properties of the global time of the system. A temporal logic *formula* is constructed by state propositions and

temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). A time-bounded model checking command for initial state t and temporal logic formula *formula* has syntax `(mc` t `|=t` *formula* `in time <=` τ `.)`.

3 Real-Time Maude Semantics for a Subset of AADL

This section gives an overview of the Real-Time Maude semantics for a behavioral subset of AADL. Section 3.1 presents the chosen subset of AADL, Section 3.2 presents its Real-Time Maude semantics, Section 3.3 explains how AADL models can be simulated and formally analyzed in Real-Time Maude and illustrates such formal analysis with a medical devices example, and Section 3.4 summarizes an avionics safety example. A technical report giving more details, the entire executable Real-Time Maude semantics, and some AADL models and the corresponding automatically generated Real-Time Maude models are all available at `http://www.ifi.uio.no/RealTimeMaude/AADL`.

3.1 Overview of a Behavioral Subset of AADL

In AADL, a system is modeled as a collection of software and hardware components. Since we focus on the software parts of AADL, the following description only deals with the software components and features.

A component is given by its *type* and its *implementation*. A component type specifies the component's *interface* in terms of *features* and *properties*. In the software portion, features are just input and output *ports*. A component implementation specifies the internal structure of the component in terms of a set of *subcomponents*, a set of *connections* linking the ports of the subcomponents, and *modes* that represent operational states of components. *System* components are the top level components. A *process* component contains a set of *thread* components that define the dynamic behavior of the process.

Connections link ports to enable the exchange of data and events among components. A port is either a *data* port, an *event* port, or an *event data* port. *Buffers* associated to event ports and event data ports support queuing of, respectively, "events" and message data, while buffers of *data* ports only keep the latest data.

Modes represent the operational states of components. A component can have mode-specific property values, subcomponents, and connections. Mode transitions are triggered by events.

The *dispatch protocol* property of a thread determines when the thread is executed. A *periodic* thread is activated at time intervals of the specified period T; an *aperiodic* thread is activated when an event arrives at a port of the thread; a *sporadic* thread is activated when an event arrives *and* the interval between two dispatches is at least T; and a *background* thread is always active.

The dynamic behavior of a thread is defined using AADL's *behavior annex* [6]. Given finite sets of *states* and *state variables*, the behavior of a thread

is defined by a set of state transitions of the form s `-[`*guard*`]->` s' `{`*actions*`}`, where s and s' are states, and where *guard* is a Boolean condition on the values of the state variables and/or the presence of events or data in the thread's input ports. The *actions* that are performed when a transition is applied may update the state variables, generate new outputs, and/or suspend the thread for a given amount of time. Actions are built from basic actions using a small set of control structures allowing sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is nondeterministically selected and applied; if the resulting state s' is not a *complete* state, another transition is applied, and so on, until a complete state is reached (or the thread is suspended).

An AADL Example. As an example of a specification within our subset of AADL, consider a network of medical devices, consisting of a *controller*, a *ventilator machine* that assists a patient's breathing during surgery, and an *X-ray* device. Whenever a button is pushed to take an X-ray, and the ventilator machine has *not* paused in the past 10 minutes, the ventilator machine should pause for two seconds, starting one second after the button is pushed, and the X-ray should be taken after two seconds. To execute the system, we add a *test activator* that pushes the button every second.

The following AADL model was developed by Min-Young Nam at UIUC.

The entire system `Wholesys` is a closed system that does not have any features (i.e., ports) to the outside world. Hence, its *type* (interface) is empty:

```
system Wholesys
end Wholesys;
```

The *implementation* of the entire system describes the architecture of the system, with four subcomponents and the connections linking these subcomponents:

```
system implementation Wholesys.imp
 subcomponents TestActivator: system TA.impl;   Xray: system XM.impl;
               Controller: system CTRL.impl;    Ventilator: system VM.impl;
 connections
  C01: event data port Controller.xmContrOutput -> Xray.ctrlInput;
  C02: event data port Controller.vmContrOutput -> Ventilator.ctrlInput;
  C03: event data port Ventilator.feedback -> Controller.feedback;
  C04: event data port TestActivator.pressEvent -> Controller.commandInput;
end Wholesys.imp;
```

The test activator, which generates an event every second, is an instance of a *system* of type `TA`, having as interface the output port `pressEvent`. Its implementation consists of a process `taPr`, which again consists of a single thread `taTh` that is an instance of the following `taThread.impl`:

```
thread taThread
  features      pressEvent: out event data port Behavior::integer;
  properties    Dispatch_Protocol => periodic;    Period => 1 sec;
end taThread;
```

```
thread implementation taThread.impl
  annex behavior_specification {**
    states           s0: initial complete state;
    transitions      s0 -[ ]-> s0 {pressEvent!(1);}; **};
end taThread.impl;
```

The thread `taTh` is dispatched every second. When the thread is dispatched, the transition is applied once (since the resulting state `s0` is a complete state), and the action performed is to output the value `1` through the port `pressEvent`.

3.2 Real-Time Maude Semantics of AADL

This section outlines the object-based real-time rewriting logic semantics for the behavioral subset of AADL presented in Section 3.1. We first show how an AADL model is represented in Real-Time Maude, and then formalize the real-time concurrent semantics of AADL models.

Representing AADL Models in Real-Time Maude. The semantics of a component-based language can naturally be defined in an object-oriented style, where each component instance is modeled as an object. The hierarchical structure of AADL components is reflected in the nested structure of objects, in which an attribute of an object contains its subcomponents as a multiset of objects.

Any AADL component instance is represented as an object instance of a subclass of the following class `Component`, which contains the attributes common to all kinds of components (systems, processes, threads, etc.):

```
class Component | features : Configuration,    subcomponents : Configuration,
                  properties : Properties,     connections : ConnectionSet,
                  modes : Modes,               inModes : ModeNameSet .
```

The attribute `features` denotes the features of a component (i.e., its ports), represented as a multiset of `Port` objects (see below); `subcomponents` denotes the subcomponents of the object; `properties` denotes its *properties*, such as the dispatch protocol for threads; `connections` denotes the set of port connections of the object (see below); `modes` contains the object's mode transition system; and `inModes` gives the set of modes (of the immediate supercomponent) in which the component is available (if the component is not a mode-specific subcomponent of the containing component, then this attribute has the value `allModes`).

In our AADL subset, the classes `System` and `Process`, denoting system and process components, do not have other attributes than those they inherit from their `Component` superclass. The `Thread` class is declared as follows:

```
class Thread | behavior : ThreadBehavior,    status : ThreadStatus,
               deactivated : Bool .
subclass Thread < Component .
```

The `behavior` attribute denotes the transition system associated with the thread. The `status` indicates the current status of the thread (`active`, `completed`, `suspended`, etc.). The attribute `deactivated` indicates whether the thread is deactivated because it is not in the current "active" modes of the system.

Ports and connections. A port is modeled as an object instance of a subclass of the class `Port`, whose subclasses define outgoing and incoming ports, as well as data, event, and event data ports. See [12] for details. An immediate level-up connection, linking an outgoing port P in a subcomponent C to the outgoing port P' in the "current" component, is modeled as a term $C\,.\,P$ `-->` P'. Immediate same-level and level-down connections are terms of the forms, respectively, P_1 `-->` P_2 and P `-->` $C\,.\,P'$.

Representing Thread Behavior. The transition system associated with a thread is modeled as a term of the form:

`states`	`current:` s `complete:` $s_1 \ \ldots \ s_k$ `other:` $s_{k+1} \ \ldots \ s_n$
`state variables`	var_1 `|->` $val_1 \quad \ldots \quad var_m$ `|->` $value_m$
`transitions`	s `-[`$guard$`]->` s' `{`$actions$`}` `;` $\ldots$ `;` s'' `-[`$guard'$`]->` s' `{`$actions'$`}`

We have also defined some additional "syntactic sugar" functions (e.g., allowing the definition of `initial` states, omitting the declaration of the store when no state variables are declared) that reduce to the above form. The sets of transitions, locations, and variable mappings have the structure of a multiset, using a multiset union operator that is declared to be associative and commutative.

Translating an AADL Model into an Object-Based Real-Time Maude Module. One main goal of our semantics is to make the "representational distance" between an AADL model and the corresponding Real-Time Maude module as small as possible. In particular, this simplifies an automatic translation from an AADL model to a similar-looking Real-Time Maude module.

Consider a type declaration of a component (`System`, `Process`, or `Thread`):

`system` *typeName* [`features:` *ports*] [`properties:` *properties*] `end` *typeName*`;`

This declaration binds *typeName* to a set of ports and a set of properties. We can therefore consider `system` as a function that, given a name, returns the interface of that name; hence the above AADL declaration translates to the equation

`eq system(`*typeName*`) = features` $ports_{RTM}$ `properties` $properties_{RTM}$ `.`

where $ports_{RTM}$ denotes the Real-Time Maude representation of *ports*.

A component *implementation*, such as

```
system implementation typeName.implName
   ...
end typeName.implName;
```

defines an component *template*, which is instantiated to a concrete instance of the component in AADL declarations of the form (the 'in modes' part is optional)

instanceName`: system` *typeName*`.`*implName* [`in modes (`*mode names*`)`]

Therefore, the above implementation declaration translates to an equation

```
var INSTANCE-NAME : Oid .   var MNS : ModeNameSet .
eq INSTANCE-NAME system typeName . implName in modes MNS =
   < INSTANCE-NAME : System | features : features(system(typeName)),
                              properties : properties(system(typeName)),
                              inModes : MNS,  subcomponents : ...,   ... >
```

In addition, the "generic" equation

```
var SI : SystemId .   var SN : SystemName .   var IN : ImplName .
eq SI system SN . IN = SI system SN . IN in modes allModes .
```

allows us to declare component instances that are not mode-dependent. The above AADL component instance declaration therefore translates to the term

instanceName `system` *typeName* `.` *implName*`.`

Example 1. Consider the AADL model of the medical system in Section 3.1. The definition of the implementation `Wholesys.imp` in Section 3.1 is translated to

```
eq INSTANCE-NAME system Wholesys . imp in modes MNS =
   < INSTANCE-NAME : System |
      modes : noModes,   inModes : MNS,
      features : features(system(Wholesys)),
      properties : properties(system(Wholesys)),
      subcomponents :
        (TestActivator system TA . impl)    (Xray system XM . impl)
        (Controller system Controller . impl)  (Ventilator system VM . impl),
      connections :
        (Controller . xmContrOutput --> Xray . ctrlInput) ;
        (Controller . vmContrOutput --> Ventilator . ctrlInput) ;
        (Ventilator . feedback --> Controller . feedback) ;
        (TestActivator . pressEvent --> Controller . commandInput) > .
```

The test activator thread `taThread` and its implementation `taThread.impl` are translated as follows:

```
eq thread(taThread) = features (pressEvent out event data thread port)
                      properties DispatchProtocol(Periodic); Period(1 Sec).

eq  INSTANCE-NAME thread taThread . impl in modes MNS =
  < INSTANCE-NAME : Thread |
       modes : noModes,   inModes : MNS,
       features : features(thread(taThread)),
       subcomponents : none,  connections : none,
       properties : properties(thread(taThread)),
       behavior :  states   initial: s0  complete: s0
                   transitions  s0 -[]-> s0 {(pressEvent ! (1))} > .
```

where `pressEvent out event data thread port` is defined to be the object `< pressEvent : OutEventDataThreadPort | buffer : nil >`.

Real-Time Concurrent Semantics. This section formalizes the operational semantics of AADL in Real-Time Maude. The real-time concurrent semantics is defined by equations and rewrite rules specifying "message" transportation, mode switches, thread dispatch, thread execution, and timed behavior. We give only a small sample of our semantic definitions, and refer to [12] for more details.

Thread Dispatch and Execution. The *execution status* of a thread is either *active*, *completed*, *sleeping*, or *inactive*. When a *completed* thread is dispatched, the thread enters the *active* status to perform the computation. Upon successful completion of the computation, the thread returns to the *completed* status. Once an active thread executes a *delay* action, it enters the *sleeping* status, suspends for a period of time, and becomes active after that time period. Finally, a thread is *inactive* if it is not part of the "active" mode of the system.

The following rule models the dispatch of a *periodic* and *completed* thread when the "dispatch timer," i.e., the second parameter to `periodic-dispatch` is `0`. The thread is dispatched, that is, its `status` is set to `active`, the "timer" is reset to the length `T` of its period, and the input ports are "dispatched" as well:

```
rl [periodic-dispatch] :
   < O : Thread | properties : periodic-dispatch(T, 0)  PROPS,
                  status : completed,  features : PORTS >
  =>
   < O : Thread | properties : periodic-dispatch(T, T)  PROPS,
                  status : active,  features : dispatchInputPorts(PORTS) > .
```

The following rewrite rule specifies the execution of an *active* thread. If the thread is in state `L1`, and there is a transition from `L1` whose guard evaluates to `true`, then the transition is executed. The resulting `status` is `sleeping(...)` if the statement list `SL` contains `delay` statements; otherwise, the thread is `completed` or `inactive` if the resulting state `L2` is a complete state, and remains `active` if `L2` is not a complete state:

```
crl [apply-transition] :
    < O : Thread | status : active,  deactivated : B,  features : PORTS,
                   behavior :
                     states current: L1 complete: LS1 others: LS2
                     state variables VAL
                     transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
   =>
    < O : Thread | status : (if SLEEP then sleeping(SLEEP-TIME) else
                             (if (not L2 in LS1) then active else
                              (if B then inactive else completed fi) fi) fi),
                   features : NEW-PORTS,
                   behavior :
                     states current: L2 complete: LS1 others: LS2
                     state variables NEW-VALUATION
                     transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
    if evalGuard(GUARD, PORTS, VAL)
```

```
/\ transResult(NEW-PORTS, NEW-VALUATION, SLEEP-TIME) :=
      executeTransition( L1 -[GUARD]-> L2 {SL}, PORTS, VAL)
/\ SLEEP := SLEEP-TIME > 0 .
```

The function `executeTransition` executes a given transition in a state with a given set `PORTS` of ports and assignment `VAL` of the state variables. The function returns a triple `transResult`(p, σ, t), where p is the state of the ports after the execution, σ denotes the resulting values of the state variables, and t is the sum of the `delay`s in the transition actions. The transitions are modeled as a multiset of single transitions; therefore, *any* enabled transition can be applied in the rule.

Time Behavior. We model time elapse in the system by a single tick rule

```
crl {SYSTEM}  =>  {delta(SYSTEM, T)} in time T if T <= mte(SYSTEM) .
```

The function `delta` defines the effect of time elapse in a system, and the function `mte` defines the *m*aximal *t*ime *e*lapse possible until an action must be taken. These functions distribute over the elements in a (sub)configuration, propagate to the subcomponents of `system` and `process` components, and must be defined for single thread objects to define the time behavior of a system.

The following must be taken into account when defining these functions: (i) periodic threads must dispatch at the correct times; (ii) threads in `sleep` status must wake up when their sleep time expires; (iii) time must not elapse when there are "untreated" messages in the system, since an aperiodic thread is dispatched when it receives an event; and (iv) time cannot advance when a thread is in `active` state, as the thread should execute a transition when it is `active`.

The function `delta` modeling the effect of time elapse decreases the "timer" t in a `periodic-dispatch(T,`t`)` property of a thread, and the timer t' in the `sleeping(`t'`)` status of a thread, according to the elapsed time:

```
eq delta(< THR : Thread | subcomponents : C, status : TS, properties : PROPS >, T)
  = < THR : Thread | subcomponents : delta(C, T), status : delta(TS, T),
                     properties : delta(PROPS, T) > .

op delta : ThreadStatus Time -> ThreadStatus .
eq delta(sleeping(T), T') = sleeping(T - T').  eq delta(TS, T') = TS [owise] .
op delta : Properties Time -> Properties .
eq delta(periodic-dispatch(T,T') PROPS, T'') =
      periodic-dispatch(T, T' - T'') PROPS .
eq delta(PROPS, T) = PROPS [owise] .
```

The function `mte` (maximum time elapse) ensures that `mte` is 0 when an "untreated" message list, that is, one of the form `transfer(`ml`)`, is present in some port buffer; in addition, it ensures that time cannot advance beyond the wake-up time of a sleeping thread, or beyond the dispatch time of a periodic thread. In addition, time cannot advance when a thread is `active`:

```
eq mte(< THR : Thread | features : PORTS, subcomponents : C,
                        status : TS, properties : PROPS >)
   = min(mte(PORTS), mte(C), mte(TS), mte(PROPS)) .

eq mte(< P : Port | buffer : ML :: transfer(ML') :: ML''  >) = 0 .
eq mte(< P : Port | buffer : ML >) = INF [owise] .
op mte : ThreadStatus -> TimeInf .
eq mte(active) = 0 .  eq mte(completed) = INF .  eq mte(sleeping(T)) = T .
eq mte(inactive) = INF .
op mte : Properties -> TimeInf .
eq mte(periodic-dispatch(T, T') PROPS) = T' .  eq mte(PROPS) = INF [owise].
```

3.3 Formal Analysis of AADL Models: A Medical Devices Example

The Real-Time Maude verification model synthesized from an AADL design model can be formally analyzed in different ways. This section presents some functions allowing the user to define system properties in terms of an AADL model without having to understand its Real-Time Maude representation. We illustrate the formal analysis features with the plug-and-play interoperation of medical devices example.

Defining Initial States and Simulation. An AADL system definition declares a component template. An initial state is an instance of such a template. In the medical example, if `MAIN` is a system component name, the initial state is `{MAIN system Wholesys . impl}`. In addition, a function `initialize` is used to correctly initialize the `status` and `deactivated` attributes in the threads, since a thread may be inactive if a mode-specific component much higher in the containment hierarchy is not part of the "current" mode.

A first form of formal analysis consists of simulating *one* of the many possible system behaviors up to a given duration using *timed rewriting*:

```
Maude> (tfrew initialize({MAIN system Wholesys . impl}) in time < 20 .)
```

Reachability Analysis. Real-Time Maude's `tsearch` and `utsearch` commands can be used to analyze whether or not a *state pattern* can be reached from the initial state. To avoid requiring the user of *AADL2Maude* to know the Real-Time Maude representation of AADL models to define his/her state patterns, our tool defines some useful functions. The term

`value of` v `in component` *fullComponentName* `in` *globalComponent*

returns the value of the state variable v in the thread identified by the full component name *fullComponentName* in the system in state *globalComponent*. The full component name is defined as a `->`-separated path of component names, from the outermost to the innermost. Likewise, the term

`location of component` *fullComponentName* `in` *globalComponent*

gives the current location/state in the transition system in the given thread.

In our medical example, `MAIN -> Xray -> xmPr -> xmTh` denotes the full component name of the `xmTh` thread. The system must ensure that the ventilator machine is pausing when an X-ray is being taken, so that the X-ray is not blurred. The following search command analyzes this property by checking whether an *undesired* state, where the X-ray thread `xmTh` is in state `xray` while the ventilator thread `vmTh` is *not* in state `paused`, can be reached from the initial state (the unexpected result shows a concrete unsafe state that can be reached from the initial state):

```
Maude> (utsearch [1]
         initialize({MAIN system Wholesys . impl}) =>* {C:Configuration}
          such that
           ((location of component (MAIN -> Xray -> xmPr -> xmTh)
               in C:Configuration) == xray
            and (location of component (MAIN -> Ventilator -> vmPr -> vmTh)
                   in C:Configuration) =/= paused) .)

Solution 1    C:Configuration --> ...
```

LTL Model Checking. For LTL model checking purposes, our tool has pre-defined useful parametric atomic propositions, such as *full thread name* `@` *location*, which holds when the thread is in state *location*.

We can use time-bounded LTL model checking to verify that an X-ray must be taken within three seconds of the start of the system (this command returned a counter-example revealing a subtle and previously unknown design error):

```
Maude> (mc initialize({MAIN system Wholesys . impl}) |=t
           <> ((MAIN -> Xray -> xmPr -> xmTh) @ xray) in time <= 3 .)

Result ModelCheckResult :  counterexample( ... )
```

3.4 An Active Standby Avionics Example

The *AADL2Maude* tool has been used by Edgar Pek to verify an AADL model developed by Abdullah Al-Nayeem of an *active standby* specification by Steve Miller from Rockwell-Collins for deciding which of two computer systems is active in an aircraft [11]. The *active standby* system is a simplified example of a fault-tolerant avionics system. In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. Aircraft applications are implemented using the resources in the cabinets. There are always two or more cabinets that are physically separated on the aircraft so that physical damage (e.g., an explosion) doesn't take out the computer system. The active standby system considers the case of two cabinets and focuses on the logic of deciding which side is *active* in a setting where each side can fail, and where the user/pilot can toggle the active status of these sides.

All the desired system properties have been verified by unbounded LTL model checking of the synthesized Real-Time Maude verification model. We refer to [12] for more details on this case study.

4 Related Work

The applications of formal methods to analyze AADL models can be divided into: (i) those that handle AADL models *without* the behavior annex; (ii) those that add to an AADL model an *external behavior specification*; and (iii) those that handle AADL models whose behavior is specified in AADL's behavior annex.

Work in class (i) includes [16, 7]; they focus on analyzing schedulability and/or behavior of an architectural subset of AADL where thread behavior is only characterized by dispatch protocol and execution time. Work in class (ii) includes [9, 1, 2, 8]; they all assume that thread behavior is specified *outside* AADL, but differ on how this is done. [9] uses the Lustre synchronous language; [1] uses communicating timed automata; [2] uses rewrite rules; and [8] uses Ada.

Work in class (iii) includes [3, 4, 17] and our own work. The main difference between [3, 4] and our work is that we give a *formal executable semantics* to an AADL model with a behavior annex specification of its thread behavior, associating to it a real-time rewrite theory. Instead, both [3] and [4] are based on *translations into imperative languages*, which are themselves in need of a formal semantics. Specifically, [3] maps AADL models into the Fiacre language, which contains assignments, conditionals, while loops and sequential composition constructs; and [4] maps AADL models to the BIP language, in which state transitions are defined using code written in C. The paper [17], like us, proposes a formal semantics, in their case in the Timed Abstract State Machine (TASM) formalism; however, they deal with a smaller subset (periodic threads, no modes) and do not support model checking analysis, for which they suggest using the UPPAAL timed automata-based tool.

In summary, to the best of our knowledge our work is the first that provides a formal executable semantics for AADL models with modes, and whose thread behavior is specified in AADL's behavior annex; and also the first that supports simulation and LTL model checking of such models in the semantic formalism.

5 Conclusions

AADL's current lacks of a formal semantics and of executability are two severe limitations, particularly for certifiable safety-critical embedded systems. In this work we solve these two problems for a substantial subset of AADL by providing a formal object-oriented real-time rewriting semantics of it in Real-Time Maude, and by deriving from this semantics a tool, *AADL2Maude*, that connects the OSATE AADL tool with Real-Time Maude and supports simulation, reachability, and LTL model checking analyses of AADL models in this subset. Furthermore, we have illustrated the use of *AADL2Maude* with two case studies, one of safe medical device interoperation, and another on safety of an avionics system.

Our experience is quite encouraging, but much work remains. Increasingly larger AADL subsets should be given a formal rewriting logic semantics to achieve the goal of giving a formal semantics to the entire AADL standard and having simulation and formal analysis tools for AADL based on such a semantics. Also, further experimentation to extend and perfect our approach should

be carried out. We also plan to make it even easier for users to specify formal properties of AADL models in an AADL "formal property annex," so that such properties can be expressed solely in terms of the given AADL model.

Acknowledgments. We thank Abdullah Al-Nayeem, Min Young Nam, Lui Sha, and Mu Sun, for many fruitful discussions on AADL, Xiokang Qiu for his work on a previous prototype, and Edgar Pek for his work on the active standby example. Partial support from Rockwell Collins and Lockheed Martin, from the Research Council of Norway, and from NSF under Grant CNS 08-34709 is gratefully acknowledged.

References

1. Abdoul, T., Champeau, J., Dhaussy, P., Pillain, P.Y., Roger, J.C.: AADL execution semantics transformation for formal verification. In: ICECCS'08. IEEE (2008)
2. Benammar, M., Belala, F., Latreche, F.: AADL behavioral annex based on generalized rewriting logic. In: Proc. RCIS 2008. IEEE (2008)
3. Berthomieu, B., Bodeveix, J.P., Chaudet, C., Dal-Zilio, S., Filali, M., Vernadat, F.: Formal verification of AADL specifications in the Topcased environment. In: Ada-Europe'09. LNCS, vol. 5570. Springer (2009)
4. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP - application to the verification of real-time systems. In: MoDELS Workshops. LNCS, vol. 5421. Springer (2009)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Talcott, C.: All About Maude, LNCS, vol. 4350. Springer (2007)
6. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: ICECCS. IEEE (2007)
7. Gui, S., Luo, L., Li, Y., Wang, L.: Formal schedulability analysis and simulation for AADL. In: ICESS'08. IEEE (2008)
8. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Trans. Embedded Comput. Syst. 7(4) (2008)
9. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proc. EMSOFT'07. ACM (2007)
10. Meseguer, J., Talcott, C.: Semantic models for distributed object reflection. In: Proceedings of ECOOP'02, Málaga, Spain, June 2002. Springer LNCS 2374 (2002)
11. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics (2009), submitted for publication
12. Ölveczky, P.C., Boronat, A., Meseguer, J., Pek, E.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude (2010), report available at `http://www.ifi.uio.no/RealTimeMaude/AADL/`
13. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Electronic Notes in Theoretical Computer Science 176(4), 5–27 (2007)
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
15. SAE AADL Team: AADL homepage (2009), `http://www.aadl.info/`
16. Sokolsky, O., Lee, I., Clarke, D.: Process-algebraic interpretation of AADL models. In: Ada-Europe. LNCS, vol. 5570. Springer (2009)
17. Yang, Z., Hu, K., Ma, D., Pi, L.: Towards a formal semantics for the AADL behavior annex. In: Proc. DATE'09. IEEE (2009)