Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)



[Electronic Notes in Theoretical Computer Science 280 (2011) 23–33](http://dx.doi.org/10.1016/j.entcs.2011.11.015)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Event B Development of a Synchronous AADL Scheduler

Jean-Paul Bodeveix and Mamoun Filali

*IRIT - Universit de Toulouse, France*

**Abstract**

This paper presents the modelisation of the semantics of a subset of the architecture description language AADL using Event-B. Elements of the semantics of the considered subset are gradually introduced in order to make possible the traceability of the formal text against the informal specification. Starting from a very general computational model, we incrementally add elements of AADL by constraining or instantiating it and finally introduce a family of schedulers. The Rodin platform is used to prove the correctness of this development.

*Keywords:* refinement, scheduling, architecture description languages

# Introduction

The Architecture Analysis and Description Language (AADL) [[5](#_bookmark8)] is the result of a long experience in the design of modeling languages targeting the development of safety critical real-time systems. Compared to most modeling languages, AADL has a rather precise semantics and has been designed to allow the early analysis of real- time systems. In order to show the correctness of analysis or code generation tools, it is necessary to define a formal semantics of AADL. In this paper, we presents the modelisation of the semantics of a subset of AADL using Event-B [[2](#_bookmark4)]. Similar works have already been done, each time using translational semantics targeting a specific formal language (Fiacre [[3](#_bookmark5)], BIP (Behavior, Interaction, Priority)[[7](#_bookmark10)] [[4](#_bookmark7)], Signal/Polychrony[[6](#_bookmark9)]), . . . ). Here, we propose a different approach which mainly aims at explaining the formal model by using a refinement-based development. Elements of the semantics of the considered subset are gradually introduced in order to make possible the traceability of the formal text against the informal specification. Then, an abstract scheduler is introduced as a refinement and proved to be correct. The paper is organized as follows. Section 2 introduces the AADL language and defines the subset we are interested in. Section 3 presents the Event-B development

of the semantics of this subset. Section 4 draws some conclusions.

1571-0661 © 2011 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

[doi:10.1016/j.entcs.2011.11.015](http://dx.doi.org/10.1016/j.entcs.2011.11.015)

# AADL

AADL (Architecture and Analysis Description Language) [[5](#_bookmark8)] is a language stan- dardized by the SAE for modeling real-time critical embedded systems. It allows the early analysis of real time systems. An AADL model defines the dynamic ar- chitecture of the considered system as a set of communicating threads. Threads have real-time properties (dispatch protocol, period, relative deadline, WCET [1](#_bookmark0) , etc.). They communicate through ports or shared data using several communica- tion protocols. The main goal of AADL is to allow to check that the hardware architecture which is made of processors, memories, buses and devices is well suited to the software architecture. Among the verified considered properties, we can cite schedulability (tasks complete before their deadline), response time, bus load, etc. In order to make such analysis possible, the execution model of AADL is precisely defined. Here, we consider a small *synchronous* subset which consists of threads pe- riodically dispatched, each having its own period. They only communicate through data ports. The AADL execution model defines the semantics of this subset. It can be summarized by the following points:

* At dispatch, a thread reads its input ports
* Dispatched thread execute (i.e. access to the processor) during at most their WCET, until completion, under the control of a scheduler.
* Completion must occur before deadline for the system to be schedulable.
* At completion, a thread writes its results computed from its input ports to its output ports. In the following, we suppose each thread has only one output port.
* If two threads linked by immediate connections are dispatched simultaneously, the sender’s completion occurs before the receiver’s start of execution, immediate links transfer data at that time and the corresponding input ports are read again by the thread, thus refreshing the value obtained at dispatch.
* The graph of nodes connected by immediate links should be acyclic.
* Delayed and unsynchronized immediate connectors read output ports at the sender deadline and update input ports at receiver dispatch. The update causally precedes the simultaneous reading of input ports.

These rules ensure deterministic executions whatever is the scheduling policy, as soon as deadlines are not missed. Models, which can scheduled, behave as if execution and communication where instantaneous. In such a way, this subset of AADL can be seen as a verifiable implementation of a synchronous model. In the following, we show how to derive by successive refinements an implementation satisfying the previous requirements. Refinement steps introduce elements of the specification that are preserved during the development.

1 Worst case execution time

# Event-B development

The goal of the proposed Event-B development is to establish that the scheduler obtained at the last refinement level satisfies the properties introduced as invari- ants in the successive refinements. To reformulate it differently, we prove that the scheduler is compatible with the semantics of the subset of AADL we have consid- ered. A complementary goal of this development is to introduce elements of AADL semantics incrementally, each element constraining or instantiating the previously defined computational model.

The more abstract level defines a very general computational model: a system is made of a set of nodes which compute a value from values computed before by connected nodes. Refinement levels restrict this non-deterministic model by introducing new AADL concepts such as immediate and delayed connections. So, we proceed using what are usually called horizontal refinements to gradually introduce elements of the specification. The structure of the development is given by Fig.

[1](#_bookmark1). Context machines (flow \*) describe the static structure of the model and are refined to introduce details such as immediate and delayed connections. Behavioral machines (fexec \*) introduce variables defining the contents of node ports and how they evolve when an event is managed. Refinements are introduced to take into account refined static information and implementation details.

flow\_r5

...



extends

flow

flow\_r1

flow\_r2

sees

refines

fexec\_r2

fexec\_r1

fexec

fexec\_r5

Fig. 1. Architecture of the development

* 1. *Level 0: the computational model*
     1. *Static model*

A system is a set of nodes, each having several input ports and one output port. Connections between input ports and the unique output of a node are represented by a function from input ports to nodes, which implies that an input port receives data from only one node, as for an AADL input data port. To each node is attached a function (comp) that computes a value given the value of its input ports. The function comp being partial and of higher order, two axioms are rather complex and need some explanations: axm5 says that the function comp only depends on the value of the input ports of the node on which it is applied; axm6 says that, given a node n, comp(n) can be applied to a valuation of ports if it associates a value to all the ports of n.

**CONTEXT** flow

**SETS**

Node – *set of computational nodes*

Port – *set of ports*

Val – *values contained in ports*

**CONSTANTS**

comp – *functions computed by nodes*

inputs – *associates input ports to nodes*

connection – *associates the node producing data to a port*

init – *initial value of output ports, attached to associated nodes*

**AXIOMS**

axm1 : *connection ∈ Port → Node*

axm2 : *inputs ∈ Node →* P(*Port* )

axm3 : *∀n1 , n2 ·*(*n1 ∈ Node ∧ n2 ∈ Node ∧ n1 /*= *n2 ⇒ inputs*(*n1* ) *∩ inputs*(*n2* )= ∅)

axm4 : *comp ∈ Node →* ((*Port →' Val* ) *→ Val* )

axm5 : *∀n, v1 , v2 ·*(*n ∈ Node ∧ v1 ∈ dom*(*comp*(*n*)) *∧ v2 ∈ dom*(*comp*(*n*)) *∧ inputs*(*n*) *Q v1* = *inputs*(*n*) *Q v2 ⇒ comp*(*n*)(*v1* )= *comp*(*n*)(*v2* ))

axm6 : *∀n, v ·*(*n ∈ Node ∧ v ∈ Port →'*

axm7 : *Node /*= ∅ axm8 : *Port /*= ∅ axm9 : *Val /*= ∅

axm10 : *init ∈ Node → Val*

**END**

*Val ∧ inputs*(*n*) *⊆ dom*(*v* ) *⇒ v ∈ dom*(*comp*(*n*)))

* + 1. *Dynamic model*

The dynamic state of the system is described by data contained in the input ports (variable entries) and data previously computed by nodes (variable value). Op- erations describe how data evolves. A node of which some ports have no value must wait before performing its computation. Initially, input ports have no value.

**MACHINE** fexec **SEES** flow **VARIABLES**

value*,* entries

**INVARIANTS**

inv1 : *value ∈ Node → Val*

inv2 : *entries ∈ Port →'*

**END**

*Val*

The system evolves through two events: *transmit* specifies the update of input ports while *compute* corresponds to a node performing a computation. The order in which these two events are sequenced is left unspecified. In fact, two consecutive transmit events act as a composed transmit event. Two consecutive compute events for the same node act as one. Furthermore, a node can only be computed if its inputs are known, which means that some transmit events should occur before.

* *transmit* takes as parameters two sets: the set of ports that loose their values (which means they become outdated) and the set of ports that receive their value from the node to which they are connected.

^

**Event** *transmit* =

**any**

*dst, reset*

**where**

grd1 : *dst ∈* P(*Port* )

grd2 : *reset ∈* P(*Port* )

**then**

act1 : *entries* := *reset Q−* (*entries Q− {p·p ∈ dst|p '→ value*(*connection*(*p*))*}*)

**end**

* compute: it takes as parameter a node of which all ports are valued. The value of the node is then computed and stored.

**Event** *compute* =

^

**any**

*n*

**where**

grd1 : *n ∈ Node*

grd3 : *inputs*(*n*) *⊆ dom*(*entries*)

**then**

act1 : *value*(*n*) := *comp*(*n*)(*entries*)

**end**

As a consequence, the value of a node is computed from a current or from a past value of the nodes to which its input ports are connected. The age of the inputs remains unspecified here. Furthermore, nodes may remain indefinitely inactive if some of their inputs are never transmitted.

* 1. *Level 1: separation between computation and delay nodes*
     1. *Static model*

We introduce here delay nodes (called DNodes). They will be associated to *delayed* connections of the AADL model. They have no computational behavior and act only as buffers. So, nodes are partitioned [2](#_bookmark2) into computational nodes (CNode) and delay nodes (DNode). Ports are also partioned into the CPort (ports of CNodes) and DPort (ports of DNodes). A delay node has one input (a DPort), which is connected to a CNode (function input). CPorts can be connected to CNodes or to DNodes. The computation function associated to a DNode is the identity function. So, its only purpose is to introduce a delay. These constraints are illustrated the figure where data flows in the reverse direction of connections.

**CONTEXT** flow r1 **EXTENDS** flow **CONSTANTS**

CNode*,* DNode*,* input*,* CPort*,* DPort

**AXIOMS**

axm1 : *CNode ∈* P(*Node*)

axm2 : *DNode ∈* P(*Node*)

axm3 : *partition*(*Node, CNode, DNode*)

axm4 : *CPort ∈* P(*Port* )

axm5 : *DPort ∈* P(*Port* )

axm6 : *partition*(*Port, CPort, DPort* )

axm7 : *input ∈ DNode → DPort*

axm8 : *∀n·*(*n ∈ DNode ⇒ inputs*(*n*)= *{input* (*n*)*}*)

axm9 : *∀n·*(*n ∈ CNode ⇒ inputs*(*n*) *⊆ CPort* )

axm10 : *∀n, v ·*(*n ∈ DNode ∧ v ∈ Port →'*

axm11 : *connection*[*DPort* ] *⊆ CNode*

**END**

*Val ∧ input* (*n*) *∈ dom*(*v* ) *⇒ comp*(*n*)(*v* )= *v* (*input* (*n*)))

* + 1. *Dynamic model*

Compared to the abstract model, we mainly introduce the variable *fresh* which contains the set of *DNodes* that have received a data in their input port, which has not been yet transmitted to the node itself. The value of the node and the value of the input port are identical for non fresh node.



dn1

dn3

cn2

cn1

dn2

**MACHINE** fexec r1 **REFINES** fexec **SEES** flow r1 **VARIABLES**

value1*,* args*,* fresh

*i*

2 partition(*E, E*1*,..., En*) *≡ E* = *Ei ∧*

*i/*=*Ej i ∩ Ej* = ∅

**INVARIANTS**

inv1 : *value1 ∈ Node → Val*

inv2 : *args ∈ CPort →' Val*

inv3 : *args* = (*CPort Q entries*)

inv4 : *fresh ∈* P(*DNode*)

inv5 : *∀n·*(*n ∈ CNode ⇒ value1* (*n*)= *value*(*n*))

inv6 : *∀n·*(*n ∈ DNode ∧ n ∈/ fresh ⇒ value1* (*n*)= *value*(*n*))

inv7 : *∀n·*(*n ∈ DNode ∧ n ∈ fresh ⇒ input* (*n*) *∈ dom*(*entries*))

inv8 : *∀n·*(*n ∈ DNode ∧ n ∈ fresh ⇒ value1* (*n*)= *entries*(*input* (*n*)))

**END**

CPorts and nodes take values. The fresh set contains DNodes that have received their inputs but have not been “computed” yet. Depending on wether the DNode belongs to this set or not, the value of the DNode is that of the corresponding level 0 node or port.

Events are specialized as follows to make easier the specification of AADL com- munication protocols:

* compute: it only applies to CNodes. However, a nop event must be added to refine DNode computations. It only updates the fresh set, which comes to saying that the node has read the input value.
* initCNodes: it specializes transmit so that transmissions originate from DNodes and their destinations are ports of a given set of nodes.
* sendToCNodes: it specializes transmit so that the source of transmissions is a given CNode and destinations are CPorts.
* sendToDNodes: it specializes transmit so that the source is a given CNode and the destinations are DNodes.
  1. *Level 2: introduction of immediate and delayed links*
     1. *Static model*

Connections are split into immediate connections and delayed connections. To each CNode is associated one DNode. It is used by outgoing delayed connections so that data flowing between to CNodes cross a DNode to arrive via a delayed link (Fig. [2](#_bookmark3)).



TH2

TH1

TH3



CN2

CN1

DN

CN3

AADL model: 1 imm, 2 delayed Event-B model

Fig. 2. Introduction of immediate and delayed connections

**CONTEXT** flow r2 **EXTENDS** flow r1 **CONSTANTS**

dnode*,* immediate*,* delayed

**AXIOMS**

axm1 : *immediate ∈ CPort →'*

*CNode*

axm2 : *delayed ∈ CPort →' CNode*

axm3 : *dnode ∈ CNode* >→ *DNode*

output buffer for delayed communication

axm4 : *partition*(*CPort, dom*(*immediate*)*, dom*(*delayed* ))

axm5 : *∀p·*(*p ∈ CPort ∧ p ∈ dom*(*immediate*) *⇒ connection*(*p*)= *immediate*(*p*)) axm6 : *∀p·*(*p ∈ CPort ∧ p ∈ dom*(*delayed* ) *⇒ connection*(*p*)= *dnode*(*delayed* (*p*))) axm7 : *∀d ·*(*d ∈ DNode ⇒ dnode*(*connection*(*input* (*d* ))) = *d* )

**END**

* + 1. *Dynamic model*

Variables are renamed to conform to the AADL view: the value computed by a CNode is called result while the value contained in a DNode is called buffer.

Events are redefined to use the new constants and variables and renamed ac- cording to AADL execution model.

* + - * The *dispatch* event is introduced as a renaming of *initCNodes*. A set of nodes is dispatched synchronously. It is delayed until ports connected through delayed links have been freshened. At dispatch, ports connected via immediate links are emptied while the value of ports connected via delayed links are read from the associated buffer.
      * The *sendImmediate* event refines *sendToCNodes*. It transmits the value computed by a CNode to input ports connected through an immediate link and that are not yet valued.
      * The *sendOutput* event refines the *sendToDNodes* event and transmits computa- tional results of CNodes to their associated DNode which become *fresh*.
      * The compute event assigns the computed result to the result variable of a CNode of which inputs are present.
      * The *nop* event removes one element from the *fresh* set: data may become not fresh at any time.
  1. *Level 3: Suppression of DNodes*

Variables attached to DNodes are replaced by variables attached to CNodes: the output variable of a CNode is Level 2 contents of the attached DNode buffer variable, *cfresh* is the set of CNodes of which output is up to date.

**MACHINE** fexec r3 **REFINES** fexec r2 **SEES** flow r2 **VARIABLES**

result*,* output*,* args*,* cfresh

**INVARIANTS**

inv1 : *output ∈ CNode → Val* inv2 : *output* = *dnode*; *buffer* inv3 : *cfresh ∈* P(*CNode*) inv4 : *fresh* = *dnode*[*cfresh*] I

**END**

Events are redefined to take into account the removal of DNodes and the new position of state variables.

* 1. *Level 4: Add states to specify thread lifecycle*

We introduce states in the static model. They correspond either to the thread states defined by the AADL execution model or to internal states used to manage data flows.

Events are renamed and redefined



so that guards mainly depend on the thread state. Scheduling constraints

dispatch

waiting

wakeup

running

complete

appear: dispatch is delayed until nodes sending data through delayed links leave their *terminated* state, wakeup is delayed until all input ports become

idle

wait\_dispatch

terminated

terminate

ended

completed send\_immediate

valued. Events are redefined to test the state variable of other threads and update the state of the threads they manage. For example, the following dispatch event defines the synchronous dispatch of the set of nodes nds: if they are all idle and source nodes of delayed connections are not in the intermediate state terminated, they read their incoming data and go to the *waiting* state.

**Event** *dispatch* =^

**refines** *dispatch*

**any**

*nds*

**where**

grd1 : *nds ∈* P(*CNode*)

grd2 : *terminated ∈/ state*[*delayed* [*union*(*inputs*[*nds*])]]

grd3 : *state*[*nds*] *⊆ {idle}*

**then**

act1 : *args* := ((*dom*(*immediate*) *∩ union*(*inputs*[*nds*])) *−Q*

*union*(*inputs*[*nds*])) *Q* (*delayed* ; *output* ))

act2 : *state* := *state Q−* (*nds × {waiting}*)

**end**

*args*) *Q−* ((*dom*(*delayed* ) *∩*

* 1. *Level 5: Add abstract scheduler*
     1. *Static model*

Dynamic tests on the availability of data are avoided by using a scheduler which defines the order in which nodes are computed. For this purpose, we manage ordered sequences of nodes. The static model introduces general definitions and operators over ordered sets:

* + - * sequence is the set of finite sequences of CNodes. It is defined as an injection from a finite subset of N to CNode.
      * merge is a function which merges two sequences while preserving the relative order of the elements of the two given sequences.
      * mono is the set of monotonous partial injections over N.

**AXIOMS**

axm1 : *sequence* = *{sq ·sq ∈* N >*'*

*CNode ∧ finite*(*dom*(*sq* ))*|sq}*

axm2 : *merge ∈ sequence × sequence → sequence*

axm3 : *mono* = *{f ·f ∈* N >*'* N *∧* (*∀i, j ·i ∈ dom*(*f* ) *∧ j ∈ dom*(*f* ) *∧ i < j ⇒ f* (*i* ) *< f* (*j* ))*|f }*

axm4 : *∀s1 , s2 ·*(*s1 ∈ sequence ∧ s2 ∈ sequence ⇒ ran*(*merge*(*s1 '→ s2* )) = *ran*(*s1* ) *∪ ran*(*s2* ))

axm5 : *∀s1 , s2 ·*(*s1 ∈ sequence ∧ s2 ∈ sequence ⇒* (*∃p1 , p2 ·p1 ∈ mono ∧ p2 ∈ mono ∧ ran*(*p1* ) = *dom*(*s1* ) *∧ ran*(*p2* ) = *dom*(*s2* ) *∧ partition*(*dom*(*merge*(*s1 '→ s2* ))*, dom*(*p1* )*, dom*(*p2* )) *∧ merge*(*s1 '→ s2* )= (*p1* ; *s1* ) *Q−* (*p2* ; *s2* )))

axm6 : *ordered* = *{sq ·sq ∈ sequence ∧* (*∀i ·i ∈ dom*(*sq* ) *⇒ immediate*[*inputs*(*sq* (*i* ))] *⊆ sq* [*0 .. i − 1* ])*|sq}*

**END**

These axiomatic declarations do not impose a specific scheduling strategy as soon as it is non preemptive. The main point is that the ordering is compatible with immediate links: preceding tasks must be executed before.

* + 1. *Dynamic model*

The scheduler manages two sets of of nodes: *tasks* is ordered and contains nodes in the waiting state, *active* contains nodes in running or completed state.

**MACHINE** fexec r5 **REFINES** fexec r4 **SEES** flow r4 **VARIABLES**

result*,* output*,* args*,* state*,* tasks*,* active

**INVARIANTS**

inv1a : *tasks ∈ sequence*

inv1b : *∀i ·i ∈ dom*(*tasks*) *⇒ immediate*[*inputs*(*tasks*(*i* )) *\ dom*(*args*)] *⊆ tasks*[*0 .. i − 1* ] *∪ active*

inv2 : *state*[*ran*(*tasks*)] *⊆ {waiting}*

inv3 : *active ∈* P(*CNode*)

inv4 : *state*[*active*] *⊆ {running, completed }*

inv6 : *dom*(*delayed* ) *∩ union*(*inputs*[*ran*(*tasks*)]) *⊆ dom*(*args*)

The operations are redefined to manage the sequences of nodes and thread states:

* dispatch: an ordered set of idle nodes of which the source of delayed input connec- tions is not in the terminated state are simultaneously launched: they become waiting and are added to the tasks sequence of nodes.

**Event** *dispatch* =

^

**any**

*sq*

**where**

grd1 : *sq ∈ ordered*

grd2 : *terminated ∈/ state*[*delayed* [*union*(*inputs*[*ran*(*sq* )])]]

grd3 : *state*[*ran*(*sq* )] *⊆ {idle}*

**with**

nds : nds = ran(sq)

**then**

act1 : *args* := ((*dom*(*immediate*) *∩ union*(*inputs*[*ran*(*sq* )])) *Q− args*) *Q−* ((*dom*(*delayed* ) *∩*

*union*(*inputs*[*ran*(*sq* )])) *Q* (*delayed* ; *output* )) act2 : *state* := *state Q−* (*ran*(*sq* ) *× {waiting}*) act3 : *tasks* := *merge*(*tasks '→ sq* )

**end**

* wakeup: extracts the first element of the tasks sequence of nodes. It becomes

running and is the only active node.

**Event** *wakeup* =

^

**when**

grd1 : *dom*(*tasks*) */*= ∅

grd2 : *active* = ∅

**with**

n : n = tasks(min(dom(tasks)))

**then**

act1 : *state*(*tasks*(*min*(*dom*(*tasks*)))) := *running*

act2 : *active* := *{tasks*(*min*(*dom*(*tasks*)))*}*

act3 : *tasks* := *{min*(*dom*(*tasks*))*} Q− tasks*

**end**

* complete: the running node computes a result and becomes completed

**Event** *complete* =

^

**any**

*n*

**where**

grd1 : *n ∈ active*

grd2 : *state*(*n*)= *running*

**then**

act1 : *result* (*n*) := *comp*(*n*)(*args*)

act2 : *state*(*n*) := *completed*

**end**

* send immediate: on completion, the computed result is transferred through the immediate links, the node becomes ended and is removed from the active set.

**Event** *sendImmediate* =

^

**any**

*n*

**where**

grd1 : *n ∈ active*

grd2 : *state*(*n*)= *completed*

**then**

act1 : *args* := (*immediate−1* [*{n}*] *× {result* (*n*)*}*) *Q− args*

act2 : *state*(*n*) := *ended*

act3 : *active* := *active \ {n}*

**end**

* terminate: the computed result is copied to the output buffer and the node becomes terminated.

^

**Event** *terminate* =

**any**

*n*

**where**

grd1 : *n ∈ CNode*

grd2 : *state*(*n*)= *ended*

**then**

act1 : *output* (*n*) := *result* (*n*)

act3 : *state*(*n*) := *terminated*

**end**

* wait dispatch: the node becomes idle and is ready for the next dispatch.

**Event** *waitDispatch* =

^

**any**

*n1*

**where**

grd1 : *n1 ∈ CNode*

grd3 : *state*(*n1* )= *terminated*

**then**

act1 : *state*(*n1* ) := *idle*

**end**

In fact, the proof obligations of this step validate the scheduler and state that it is possible to replace data dependent control by a scheduling strategy. Wether scheduling is static or dynamic can be decided later, once timing has been introduced to specify a finite control.

* 1. *Validation and Next steps*

All the proof obligations associated to the proposed development have been dis- charged thanks to the Rodin platform [[1](#_bookmark6)]. It was quite difficult because of the com- plexity of some set and sequence expressions, mainly in the last refinement level where sequence of tasks are introduced to specify the behavior of the scheduler. The validation effort is summarized in the following table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | level 0 | level 1 | level 2 | level 3 | level 4 | level 5 |
| obligations | 7 | 43 | 24 | 17 | 31 | 47 |
| automatic | 6 | 31 | 13 | 13 | 20 | 18 |
| assisted | 1 | 12 | 11 | 4 | 11 | 29 |

The scheduler introduced in the last refinement step is defined axiomatically. In fact these axioms are satisfied by several actual scheduling algorithms. Thus, the study could be pursued further by considering specific scheduling algorithms. Other refinement steps could consider optimized memory management avoiding useless memory transfers through the use of shared variables, then add time to model the

periodicity of tasks. The last refinement level makes a non deterministic choice of the set of nodes to be synchronously dispatched. This set should be determined by considering a global date and the periods of the AADL threads. This instantiation is not trivial because the considered nodes should be idle, which means their previous activation has been completed in time. So, a necessary condition to establish the refinement is that the system can be scheduled. This hypothesis can be satisfied by considering the synchronous hypothesis: computations take zero time.

# Conclusion

This paper describes a refinement-based approach to describe the semantics of a real-time execution model, illustrated by a subset of AADL. Contrary to existing translation-based semantics, it allows the incremental introduction of semantics in- formation. Incremental invariant preserving proofs guarantee the correctness of the abstract views defined by the first levels of the development and at last the compati- bility of the axiomatized family of schedulers with AADL execution model. Thanks to the Rodin platform, all the generated proof obligations have been discharged, either automatically or through user control. This work should be refined by in- troducing timing information, and then by the encoding of either the synchronous model or a real-time scheduling algorithms under a schedulability hypothesis. An- other perspective concerns the study of data structures which make proofs easier to discharge either automatically or manually.

# References

1. *Event-b and the rodin platform.* <http://www.event-b.org/>.
2. Abrial, J.-R., “Modeling in Event-B - System and Software Engineering,” Cambridge University Press, 2010, I-XXVI, 1-586 pp.
3. Berthomieu, B., J.-P. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali and F. Vernadat, *Formal verification* *of AADL specifications in the topcased environment*, in: *Ada-Europe*, 2009, pp. 207–221.
4. Chkouri, M. Y., A. Robert, M. Bozga and J. Sifakis, *Translating AADL into BIP - application to the verification of real-time systems*, in: *MoDELS Workshops*, 2008, pp. 5–19.
5. Feiler, P., B. Lewis and S. Vestal, *The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems*, in: *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, 2006, pp. 1206–1211.
6. Ma, Y., J.-P. Talpin, S. K. Shukla and T. Gautier, *Distributed simulation of AADL specifications in a polychronous model of computation*, Embedded Software and Systems **0** (2009), pp. 607–614.
7. Pi, L., J.-P. Bodeveix and M. Filali, *Modeling AADL data communication with BIP*, in: *Ada-Europe*, 2009, pp. 192–206.