Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 280 (2011) 23–33

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

1 Introduction

The Architecture Analysis and Description Language (AADL) [5] 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]. Similar works have already been done, each time using translational semantics targeting a specific formal language (Fiacre [3], BIP (Behavior, Interaction, Priority)[7] [4], Signal/Polychrony[6]), ...). 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.

doi:10.1016/j.entcs.2011.11.015

2 AADL

AADL (Architecture and Analysis Description Language) [5] is a language standardized 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 architecture of the considered system as a set of communicating threads. Threads have real-time properties (dispatch protocol, period, relative deadline, WCET [1], etc.). They communicate through ports or shared data using several communication 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 periodically 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

3 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 invariants 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 considered. 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. 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.

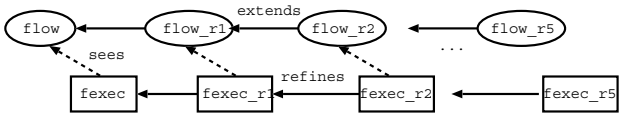


Fig. 1. Architecture of the development

3.1 Level 0: the computational model

3.1.1 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 : $connection \in Port \to Node$
axm2 : $inputs \in Node \to \mathbb{P}(Port)$
axm3 : $\forall n1, n2 \cdot (n1 \in Node \land n2 \in Node \land n1 \neq n2 \Rightarrow inputs(n1) \cap inputs(n2) = \varnothing)$
axm4 : $comp \in Node \to ((Port \nrightarrow Val) \to Val)$
axm5 : $\forall n, v1, v2 \cdot (n \in Node \land v1 \in dom(comp(n)) \land v2 \in dom(comp(n)) \land inputs(n) \lhd v1 = inputs(n) \lhd v2 \Rightarrow comp(n)(v1) = comp(n)(v2))$
axm6 : $\forall n, v \cdot (n \in Node \land v \in Port \nrightarrow Val \land inputs(n) \subseteq dom(v) \Rightarrow v \in dom(comp(n)))$
axm7 : $Node \neq \varnothing$
axm8 : $Port \neq \varnothing$
axm9 : $Val \neq \varnothing$
axm10 : $init \in Node \to Val$

END

3.1.2 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`). Operations 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 : $value \in Node \to Val$
inv2 : $entries \in Port \nrightarrow Val$
END

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 $transmit \mathrel{\widehat{=}}$
  any
  $dst, reset$
  where
  grd1 : $dst \in \mathbb{P}(Port)$
  grd2 : $reset \in \mathbb{P}(Port)$
  then
  act1 : $entries := reset \lhd (entries \lhd\!\!+ \{p \cdot p \in dst | p \mapsto value(connection(p))\})$
  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 $compute \mathrel{\widehat{=}}$
  any
  n
  where
  grd1 : $n \in Node$
  grd3 : $inputs(n) \subseteq dom(entries)$
  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.

3.2 Level 1: separation between computation and delay nodes

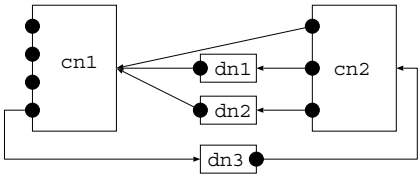
3.2.1 Static model

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] 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` : $CNode \in \mathbb{P}(Node)$
`axm2` : $DNode \in \mathbb{P}(Node)$
`axm3` : $partition(Node, CNode, DNode)$
`axm4` : $CPort \in \mathbb{P}(Port)$
`axm5` : $DPort \in \mathbb{P}(Port)$
`axm6` : $partition(Port, CPort, DPort)$
`axm7` : $input \in DNode \to DPort$
`axm8` : $\forall n \cdot (n \in DNode \Rightarrow inputs(n) = \{input(n)\})$
`axm9` : $\forall n \cdot (n \in CNode \Rightarrow inputs(n) \subseteq CPort)$
`axm10` : $\forall n, v \cdot (n \in DNode \wedge v \in Port \nrightarrow Val \wedge input(n) \in dom(v) \Rightarrow comp(n)(v) = v(input(n)))$
`axm11` : $connection[DPort] \subseteq CNode$
END

3.2.2 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.



MACHINE fexec_r1
REFINES fexec
SEES flow_r1
VARIABLES
`value1`, `args`, `fresh`

[2] $\texttt{partition}(E, E_1, \ldots, E_n) \equiv E = \bigcup_i E_i \wedge \bigwedge_{i \neq j} E_i \cap E_j = \varnothing$

INVARIANTS
inv1 : $value1 \in Node \rightarrow Val$
inv2 : $args \in CPort \nrightarrow Val$
inv3 : $args = (CPort \lhd entries)$
inv4 : $fresh \in \mathbb{P}(DNode)$
inv5 : $\forall n \cdot (n \in CNode \Rightarrow value1(n) = value(n))$
inv6 : $\forall n \cdot (n \in DNode \wedge n \notin fresh \Rightarrow value1(n) = value(n))$
inv7 : $\forall n \cdot (n \in DNode \wedge n \in fresh \Rightarrow input(n) \in dom(entries))$
inv8 : $\forall n \cdot (n \in DNode \wedge n \in fresh \Rightarrow value1(n) = entries(input(n)))$
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 communication 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.

3.3 Level 2: introduction of immediate and delayed links

3.3.1 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).

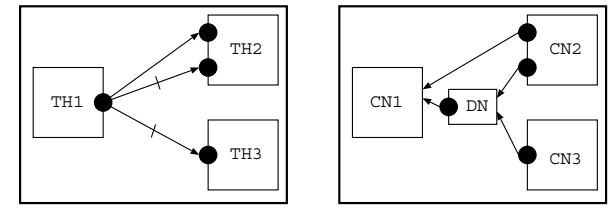


Fig. 2. Introduction of immediate and delayed connections

CONTEXT flow_r2
EXTENDS flow_r1
CONSTANTS
`dnode`, `immediate`, `delayed`
AXIOMS
axm1 : $immediate \in CPort \nrightarrow CNode$

axm2 : $delayed \in CPort \nrightarrow CNode$
axm3 : $dnode \in CNode \rightarrowtail DNode$
output buffer for delayed communication
axm4 : $partition(CPort, dom(immediate), dom(delayed))$
axm5 : $\forall p \cdot (p \in CPort \wedge p \in dom(immediate) \Rightarrow connection(p) = immediate(p))$
axm6 : $\forall p \cdot (p \in CPort \wedge p \in dom(delayed) \Rightarrow connection(p) = dnode(delayed(p)))$
axm7 : $\forall d \cdot (d \in DNode \Rightarrow dnode(connection(input(d))) = d)$
END

3.3.2 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 according 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 computational 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.

3.4 Level 3: Suppression of DNodes

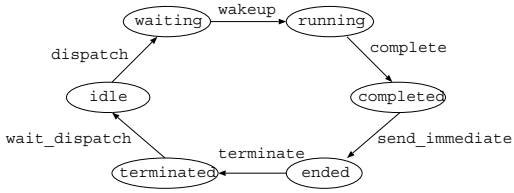
Variables attached to DNodes are replaced by variables attached to CNodes: the `output` variable of a CNode is Level 2 contents of the attached DNode `buffer` variable, *cfresh* is the set of CNodes of which output is up to date.

MACHINE fexec_r3
REFINES fexec_r2
SEES flow_r2
VARIABLES
`result`, `output`, `args`, `cfresh`
INVARIANTS
inv1 : $output \in CNode \rightarrow Val$
inv2 : $output = dnode; buffer$
inv3 : $cfresh \in \mathbb{P}(CNode)$
inv4 : $fresh = dnode[cfresh]$ I
END

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

3.5 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 appear: dispatch is delayed until nodes sending data through delayed links leave their *terminated* state, wakeup is delayed until all input ports become 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 $dispatch \widehat{=}$
refines $dispatch$
  any
    nds
  where
    grd1 : $nds \in \mathbb{P}(CNode)$
    grd2 : $terminated \notin state[delayed[union(inputs[nds])]]$
    grd3 : $state[nds] \subseteq \{idle\}$
  then
    act1 : $args := ((dom(immediate) \cap union(inputs[nds])) \lhd args) \lhd\!\!+ ((dom(delayed) \cap union(inputs[nds])) \lhd (delayed; output))$
    act2 : $state := state \lhd\!\!+ (nds \times \{waiting\})$
  end

3.6 Level 5: Add abstract scheduler

3.6.1 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 $\mathbb{N}$ 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 $\mathbb{N}$.

AXIOMS
  axm1 : $sequence = \{sq \cdot sq \in \mathbb{N} \rightarrowtail CNode \wedge finite(dom(sq)) | sq\}$
  axm2 : $merge \in sequence \times sequence \rightarrow sequence$
  axm3 : $mono = \{f \cdot f \in \mathbb{N} \rightarrowtail \mathbb{N} \wedge (\forall i, j \cdot i \in dom(f) \wedge j \in dom(f) \wedge i < j \Rightarrow f(i) < f(j)) | f\}$
  axm4 : $\forall s1, s2 \cdot (s1 \in sequence \wedge s2 \in sequence \Rightarrow ran(merge(s1 \mapsto s2)) = ran(s1) \cup ran(s2))$
  axm5 : $\forall s1, s2 \cdot (s1 \in sequence \wedge s2 \in sequence \Rightarrow (\exists p1, p2 \cdot p1 \in mono \wedge p2 \in mono \wedge ran(p1) = dom(s1) \wedge ran(p2) = dom(s2) \wedge partition(dom(merge(s1 \mapsto s2)), dom(p1), dom(p2)) \wedge merge(s1 \mapsto s2) = (p1; s1) \lhd\!\!+ (p2; s2)))$
  axm6 : $ordered = \{sq \cdot sq \in sequence \wedge (\forall i \cdot i \in dom(sq) \Rightarrow immediate[inputs(sq(i))] \subseteq sq[0 \mathinner{..} i-1]) | sq\}$

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.

3.6.2 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 : $tasks \in sequence$
inv1b : $\forall i \cdot i \in dom(tasks) \Rightarrow immediate[inputs(tasks(i)) \setminus dom(args)] \subseteq tasks[0 \,..\, i-1] \cup active$
inv2 : $state[ran(tasks)] \subseteq \{waiting\}$
inv3 : $active \in \mathbb{P}(CNode)$
inv4 : $state[active] \subseteq \{running, completed\}$
inv6 : $dom(delayed) \cap union(inputs[ran(tasks)]) \subseteq dom(args)$

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 connections is not in the `terminated` state are simultaneously launched: they become `waiting` and are added to the `tasks` sequence of nodes.
  Event *dispatch* $\widehat{=}$
  any
  sq
  where
  grd1 : $sq \in ordered$
  grd2 : $terminated \notin state[delayed[union(inputs[ran(sq)])]]$
  grd3 : $state[ran(sq)] \subseteq \{idle\}$
  with
  nds : `nds` $=$ `ran(sq)`
  then
  act1 : $args := ((dom(immediate) \cap union(inputs[ran(sq)])) \lhd args) \lhd ((dom(delayed) \cap union(inputs[ran(sq)])) \lhd (delayed; output))$
  act2 : $state := state \lhd (ran(sq) \times \{waiting\})$
  act3 : $tasks := merge(tasks \mapsto sq)$
  end

- wakeup: extracts the first element of the `tasks` sequence of nodes. It becomes `running` and is the only `active` node.
  Event *wakeup* $\widehat{=}$
  when
  grd1 : $dom(tasks) \neq \varnothing$
  grd2 : $active = \varnothing$
  with
  n : `n` $=$ `tasks(min(dom(tasks)))`
  then
  act1 : $state(tasks(min(dom(tasks)))) := running$
  act2 : $active := \{tasks(min(dom(tasks)))\}$
  act3 : $tasks := \{min(dom(tasks))\} \lhd tasks$
  end

- complete: the running node computes a result and becomes `completed`
  Event *complete* $\widehat{=}$
  any
  n
  where
  grd1 : $n \in active$
  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* $\widehat{=}$
any
n
where
grd1 : $n \in active$
grd2 : $state(n) = completed$
then
act1 : $args := (immediate^{-1}[\{n\}] \times \{result(n)\}) \lhd args$
act2 : $state(n) := ended$
act3 : $active := active \setminus \{n\}$
end

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

Event *terminate* $\widehat{=}$
any
n
where
grd1 : $n \in CNode$
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* $\widehat{=}$
any
$n1$
where
grd1 : $n1 \in CNode$
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.

3.7 Validation and Next steps

All the proof obligations associated to the proposed development have been discharged thanks to the Rodin platform [1]. It was quite difficult because of the complexity 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.

4 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 information. Incremental invariant preserving proofs guarantee the correctness of the abstract views defined by the first levels of the development and at last the compatibility 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 introducing timing information, and then by the encoding of either the synchronous model or a real-time scheduling algorithms under a schedulability hypothesis. Another 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.