Development of a Synchronous Subset of AADL *

M. Filali[1] and J. Lawall[2]

[1] IRIT-CNRS ; Université de Toulouse ; 118 route de Narbonne, F-31062 Toulouse, France
[2] DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen, Denmark

Abstract. We study the definition and the mapping of an AADL subset: the so called synchronous subset. We show that the data port protocol used for delayed and immediate connections between periodic threads can be interpreted in a synchronous way. In this paper, we formalize this interpretation and study the development of its mapping such that the original synchronous semantics is preserved. For that purpose, we use refinements through the Event B method.

1 Introduction

Model-based design has emerged as one of the most important design paradigms in recent years. High level models allow the developer to concentrate on the functionality to be offered rather than implementation details. The Architecture Analysis and Design Language (AADL) [11] is by now considered as a mature alternative for modeling embedded and real time systems. AADL has been standardized by the SAE [19], and features of AADL have influenced the MARTE OMG standard [10]. As a successor of the MetaH language [16] developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications, AADL capitalizes on more than 10 years of experience. AADL also builds on the experience acquired during the development of Architecture Description Languages (ADLs) such as ACME and Wright [3].

In this paper, we study the AADL data port protocol, which defines the semantics of delayed and immediate connections between periodic threads. This is a fundamental protocol that lies at the heart of any embedded AADL-based platform. We show that the data port protocol can be interpreted in a synchronous way [6]. Nevertheless, this interpretation does not provide a satisfactory basis for implementation in embedded systems, as the stack depth entailed by recursive calls is only bounded by the least common multiple of the periods of all of the threads, which can be very large. We thus present the development of a mapping of the synchronous semantics of the AADL data port protocol into an iterative implementation, such that the original synchronous semantics is preserved. For this purpose, we use refinements through the Event B method [2].

After a brief overview of AADL and Event B in Sections 2 and 3, we motivate the proposed development in Section 4. Section 5 presents the successive refinements of the development. Section 6 outlines the validation of the development. Before concluding, we review some related work in Section 7.

* This work was partly supported by the French AESE project Topcased and by the region Midi-Pyrénées.

2 AADL

AADL includes all the standard concepts of any ADL, e.g., components, connectors to describe the interface of components, and connections to link components. AADL distinguishes between three kinds of components: software components (process, thread, thread group, subprogram, and data), hardware components (processor, bus, memory, device), and system components.

2.1 AADL threads

In AADL, threads are the only components that have an execution semantics. AADL supports the classic types of thread dispatch protocols: a thread can be declared to be periodic, aperiodic, sporadic or background. All of the standard properties (worst case execution time (WCET), deadline, etc.) used to describe a real-time system exist in AADL. In the following, we consider periodic threads only. A periodic thread is dispatched within every period.[3] When a thread *completes* its execution, it goes to the "awaiting_dispatch" state until its next period. The thread's actual execution time is bounded by its WCET and must end by its deadline.

2.2 AADL data port protocol

AADL defines three types of ports: data, event and event data ports. Data ports allow communication via a single word (a register). Event and event data ports represent buffered communications. In this paper, we consider only data ports.

A data port connection can be declared as *delayed* or *immediate*. If the connection is delayed, data is available at the deadline of the sending thread. If the connection is immediate the receiving thread must wait for the sending thread to complete to start its execution. Figure 1 illustrates the instants where execution and these communications take place. One key aspect of the AADL data port protocol is that communications between a thread and its environment occur at well defined instants:

- In general, a message is copied at the dispatch (data event in Fig. 1); in the case of an immediate connection, a message is copied at the start of execution (immediate data in Fig. 1).
- A message is actually sent at the completion, in the case of an immediate connection, or at the deadline, in the case of a delayed connection.

3 A Brief Overview of Event B

Event B stems from the B method [1]. One of the goals of Event B is to reason about so called reactive systems [15]. Like B, Event B is based on set theory together with first-order logic. It proposes refinements as the main software development concept. However, instead of B operations, Event B proposes events, which are simpler.

[3] We consider here AADL V1 which does not take into account the phase of a periodic thead.

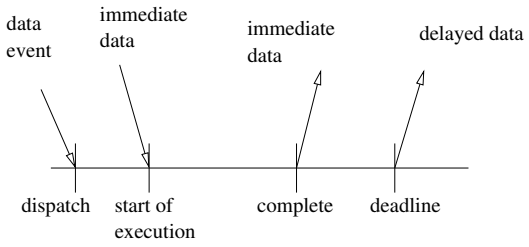

Fig. 1. *Communication through data ports in AADL.*

3.1 Basic principles

The basic structuring concepts of Event B are the *context* and the *machine*. A context contains *sets*, *constants* and their properties (*axioms* and *theorems*). A machine contains a system specification. A system specification *sees* contexts and defines a static and a dynamic part. The static part defines a state space through *variables*. These variables are "typed" and more generally specified by *invariants*. The dynamic part defines a behavior through an initialization event and a set of events. Each event can be considered as a non deterministic guarded command [9]. The guard is specified by a conjunction of predicates and the command is specified as a set of substitutions.

3.2 Notation

For the most part, Event B uses standard set notation. Some notation that is specific to Event B is as follows:

- **pair construction:** Pairs are constructed using the maplet operator $\mapsto$. A pair is thus denoted $(a \mapsto b)$ instead of (a,b).
- **restriction to the domain:** $F \lhd R = \{x \mapsto y \mid (x \mapsto y) \in R \wedge x \in F\}$
- **overwrite:** $Q \lhd\!\!+ R = ((dom(Q) \setminus dom(R)) \lhd Q) \cup R$

4 Motivation of the Development

In this section, we motivate our development by presenting the specification view of the data port protocol and some features of the operational view. The specification view is intended to be used when reasoning over the protocol. The operational view is intended to be used for an actual implementation. For instance, the operational view takes into account the times where:

- the computations take place: at the period;
- the outputs are made avaliable: at the completion or at the deadline.

The goal of the development is to establish that the operational view refines the specification view.

First, we illustrate the two views through a toy AADL architecture, shown in Figure 2. In this architecture, we have three periodic threads: `t1`, `t2` and `t3`. The period and deadline of `t1` are both 10. That of `t2` are 10 and 5, respectively. That of `t3` are 15 and 5, respectively. Thread `t1` has two output ports `o1,o2` and two input ports `i4` and `i5`. Thread `t2` has one output port `o5` and two input ports `i1` and `i3`. Thread `t3` has two output ports `o3` and `o4` and one input port `i2`. In each case, output port `o`*i* is connected to input port `i`*i*, via either a delayed (`d`) or an immediate (`i`) connection. We furthermore adopt the convention that inside a thread, input and output ports are linked through implicit immediate connections (an output can be linked to any subset of inputs).

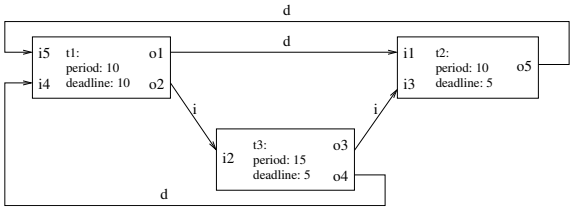

Fig. 2. *A toy AADL architecture.*

4.1 The specification view

In the specification view, computations are assumed to occur at precise time instants. In this study, we assume that computations occur at the beginning of their period, and do not take time. Still, their results are available only at the deadline. This ensures conformity with an effective implementation in which computations do take time but respect the deadlines. Computations can depend on each other in:

- either a *delayed* way: at time `t`, the computation for port `p` depends on the computation that occured for port `p'` at its most recent deadline.
- Or in an *immediate* way: at time `t`, the computation for port `p` depends on the computation that occured for port `p'` at the same time `t`, if any. The result of this computation is then buffered in `p` for use in subsequent computations until a new result is available from `p'`.

These two causality relations are given through the following relations:

$\texttt{Pred\_D} \in \mathbb{P}(\texttt{Port} \times \texttt{Port})$ // `delayed port predecessor relation`
$\texttt{Pred\_I} \in \mathbb{P}(\texttt{Port} \times \texttt{Port})$ // `immediate port predecessor relation`

The `Pred_D` and `Pred_I` port predecessor relations for the example shown in Figure 2 are the following:

Delayed port predecessor relation	Immediate port predecessor relation
$\{(i1 \mapsto o1), (i4 \mapsto o4), (i5 \mapsto o5)\}$	$\{(i2 \mapsto o2), (i3 \mapsto o3)\}$

Remark. The "hidden" immediate relations between each input port and each output port of a thread are not shown.

The constants `C_D` (resp. `C_I`) are used to define the computation tasks for each delayed (resp. immediate port). A computation task is parameterized by

- the identity of the port at the end of a connection,
- the values of the ports at the beginning of a connection, at preceding times for delayed ports or at the current time for immediate ports.

As delayed connections only appear between threads, `C_D` only transfers values from a single output port to a single input port. `C_I` does the same for connections between threads, and carries out the thread's computation for the "hidden" connections within threads. Then, the recursive `Compute` function is defined for delayed and immediate ports as follows:[4]

	$\text{Compute}(t)(p)$
$\text{delayed}(p)$ $\wedge\ t\%\text{period}(p) = 0$	$\text{C\_D}(p)(\{q \mapsto \text{Compute}(\text{Deadline}(t \mapsto q))(q) \mid q \mapsto p \in \text{Pred\_D}\})$
$\text{immediate}(p)$ $\wedge\ t\%\text{period}(p) = 0$ $\wedge\ \forall q \in \text{Pred\_I}^{-1}(p).$ $t\%\text{period}(q) = 0$	$\text{C\_I}(p)(\{q \mapsto \text{Compute(t)}(q) \mid q \mapsto p \in \text{Pred\_I}\})$
else	$\text{Compute}(t-1)(p)$

where t is not equal to 0.

Remark: In the preceding table, we have adopted the usual mathematical notation $\{\text{exp} \mid \text{boolean\_exp}\}$ for set comprehensions. In event B, the quantified variables would be made explicit and the order of the terms changed as follows: $\{\text{vars.boolean\_exp} \mid \text{exp}\}$.

The expression $\texttt{Compute}(t)(p)$ deserves some explanation. For a delayed port it is computed over the predecessor ports q (according to the `Pred_D` relation) at their respective deadlines: $\texttt{Compute}(\texttt{Deadline}(t \mapsto q))(q)$. For an immediate port, it is computed over the predecessor ports q (according to the `Pred_I` relation) at the current time t: i.e., $\texttt{Compute}(t)(q)$. In general, the value of a port p is computed at its period: $\text{period}(p)$; for an immediate port, it is actually computed if its period aligns with those of its predecessors; otherwise it remains unchanged.

Discussion. This initial specification is functional. It follows that, assuming the termination of the `Compute` function, the specification can be considered as executable. However, we remark that such code cannot be considered as executable in the context of an embedded system. In particular, the memory resources needed for executing such a code are not a priori bounded, i.e., the code does not have the constant space property. Indeed, the depth of the required stack to handle recursive calls depends on the least

[4] % denotes the infix modulo function.

common multiple of the periods of the various threads. Then, from a technical point of view, the goal of the refinements that will be introduced in Sections 5.3 and 5.4 can be seen as the implementation of this recursivity through bounded memory independently of parameter values.

4.2 The operational view

The operational view introduces a scheduler that manages the data port architecture. The information needed by this scheduler is given by the `RealTime` context. `Idle` and `Deadline` are the basic structures used by such a scheduler.

$\texttt{Idle} \in \mathbb{N} \rightarrow \mathbb{P}(\texttt{Port})$: for a time t, the list of idle ports.
$\texttt{Deadline} \in \texttt{Port} \times \mathbb{N} \rightarrow \mathbb{N}$: for a delayed port p and a time t, the (time) value of the most recent deadline

We illustrate these static structures by instantiating them according to the architecture shown in Figure 2.

Data structures of the architecture example

– `Idle` returns the list of idle ports at time t.

Idle	$1-9$	Port
Idle	10	$\{i2, o3, o4\}$
Idle	$11-14$	Port
Idle	15	$\{i1, i3, i4, i5, o1, o2, o5\}$
Idle	$16-19$	Port
Idle	20	$\{i2, o3, o4\}$
Idle	$21-29$	Port

– `Deadline` is a function which gives for a delayed port p and a time t, the (time) value of the most recent deadline.

Deadline$(o1, 0-9)$	0	Deadline$(o4, 0-4)$	0	Deadline$(o5, 0-4)$	0
Deadline$(o1, 10-19)$	9	Deadline$(o4, 5-19)$	4	Deadline$(o5, 5-14)$	4
Deadline$(o1, 20-29)$	19	Deadline$(o4, 20-29)$	19	Deadline$(o5, 15-24)$	14
				Deadline$(o5, 25-29)$	24

Algorithm of the scheduler. B does not offer mechanisms for real-time programming, such as dedicated primitives for awaiting clock interrupts. In the proposed Event B machines, guards model the real-time clock triggers. Once an event is triggered, all the enabled events at that time are executed until none of them is still enabled. Then, the processor idles until the next clock tick. In this paper, we consider that the time between two clock ticks is sufficient to handle all the enabled events. In that way, we do not lose any clock tick and all the events take place in zero time according to the synchronous abstraction. The main loop consists of time-triggered iterations. For our toy example, iterations are triggered at $0, 10, 15, 20, 30$, etc., which correspond to the

periods of the threads of our example. Each iteration first handles the ports that should run at that time and then prepares the next iteration. Handling ports is done through the `ComputeDelayed` and `ComputeImmediate` events. Preparing the next iteration is done through the `Tick` event. In the last refinement, these events are exclusive and deterministic.

Synthesis. The operational view of the considered AADL subset is a mix between a time-triggered machine and a data-flow machine. Ports are updated according to their period. Immediate connections enforce a send-receive synchronization.

5 Abstracting and Refining the AADL Data Port Protocol

In this section, we formalize the fact that if we restrict AADL to connections with the data port protocol, we have a synchronous computation model. For that purpose, we first exhibit a model of the protocol from which we derive another model, based on histories, close to the description given in the previous section. A third model is derived for considering implementation related issues with respect to the boundedness of the used memory and the time for evaluating new port values. To summarize, we consider the following refinement-based development, where $\sqsubseteq$ is B notation for "refined by":

MACHINE	Spec	$\sqsubseteq$	I_Spec	$\sqsubseteq$	P_Spec	$\sqsubseteq$	M_Spec	$\sqsubseteq$	Scheduler
CONTEXT	Ports		I_Ports		M_Ports		S_Compute		

- Spec is the initial specification representing the abstraction of the AADL protocol.
- I_Spec is the refinement where *Idle* ports are introduced.
- P_Spec is the refinement where a *Partition* of ports is introduced. Immediate ports are computed.
- M_Spec is the refinement where port buffering through a memory is introduced. Delayed ports are computed.
- Scheduler is the final refinement where port updates are scheduled according to a total order.

5.1 The specification

This is the initial specification for the AADL data port protocol as the *time* parameterized function `Compute`. The variable *ports* records the value of this function at each point in time through the `Initialisation` and `Tick` events. This recording is done atomically so that no value returned by `Compute` is lost: between two events, `Compute` stutters.

The static description. We have three variables:

- t is the current time,
- *ports* maps ports to their current value,

– b is a previous time, such that *ports* has not changed since b until t (excluded). `inv5` states that we have not missed any value in the interval $b..t-1$: the range of the *Compute* function over this interval is a singleton.

MACHINE Spec
SEES Ports

VARIABLES t ports b

INVARIANTS
`inv1` : $t \in \mathbb{N} \land 0 < t$
`inv2` : $ports \in Port \rightarrow Val$
`inv4` : $b \in \mathbb{N} \land b < t$
`inv3` : $ports = Compute(b)$
`inv5` : $Compute[b\,..\,t-1] = \{Compute(b)\}$

The dynamic description. The basic idea is that, in order to preserve our basic invariant `inv5` , the time t is advanced to a new value t' such that ports remain constant from t to $t'-1$.

Initialisation
begin
`act1` : $t :\mid t' \in \mathbb{N} \land 0 < t' \land Compute[0\,..\,(t'-1)] = \{Compute(0)\}$
`act2` : $ports := Compute(0)$
`act3` : $b := 0$
end

Event *Tick* $\widehat{=}$
begin
`act1` : $t :\mid t' \in \mathbb{N} \land t < t' \land Compute[t\,..\,(t'-1)] = \{Compute(t)\}$
`act2` : $ports := Compute(t)$
`act3` : $b := t$
end

5.2 Introducing idle ports and atomicity breaking through silent steps

In this refinement, we introduce `Idle` ports: a port is `Idle` at time t if it has the same value as at time $t-1$ (see the definition of the `Compute` function in Section 4.1). Intuitively, a port is idle when the thread to which it belongs is not active, i.e., after the deadline. Moreover, non idle ports are now not updated atomically: we introduce a silent `Step` event for updating them incrementally through the variable `compute`.

Basic sets. We introduce the constant time parameterized function `Idle`:

CONTEXT I_Ports
EXTENDS Ports
CONSTANTS `Idle`
AXIOMS
`axm1` : $Idle \in \mathbb{N} \rightarrow \mathbb{P}(Port)$
`axm3` : $\forall t \cdot (t \in \mathbb{N} \Rightarrow (\forall p \cdot p \in Port \Rightarrow (p \in Idle(t+1) \Rightarrow Compute(t+1)(p) = Compute(t)(p))))$
END

The static description. We introduce the variable `compute` to incrementally record port updates: recorded ports define the domain of the compute function (array). The invariant `inv2` states the correctness of this recording; any recorded slice is equal to the range of the *Compute* function over the same slice. The invariant `inv4` states that idle ports are implicitly recorded.

VARIABLES t ports compute b

INVARIANTS
`inv1` : $compute \in Port \pfun Val$
`inv2` : $\forall d \cdot (d \subseteq dom(compute) \Rightarrow d \domres compute = d \domres Compute(t))$
`inv4` : $Idle(t) \subseteq dom(compute)$

The dynamic description.

Initialisation
begin
`act3` : $t, compute :\mid t' \in \mathbb{N} \land 0 < t'$
$\land\ (t' \neq 1 \Rightarrow Idle[1 \upto (t'-1)] = \{Port\})$
$\land\ compute' = Idle(t') \domres Compute(0)$
`act1` : $ports := Compute(0)$
`act4` : $b := 0$
end

A silent step can occur if there exists some port not yet recorded:

Event *Step* $\widehat{=}$
any p
where
`grd1` : $p \in Port$
`grd2` : $p \notin dom(compute)$
then
`act1` : $compute(p) := Compute(t)(p)$
end

A tick can occur if all the ports have been recorded:

Event *Tick* $\widehat{=}$ **refines** ***Tick***
when `grd1` : $dom(compute) = Port$
then
`act1` : $t, compute :\mid t' \in \mathbb{N} \land t < t'$
$\land\ (t' \neq t+1 \Rightarrow Idle[(t+1) \upto (t'-1)] = \{Port\})$
$\land\ compute' = Idle(t') \domres compute$
`act2` : $ports := compute$
`act3` : $b := t$
end

5.3 Partitioning the ports

In this refinement, we partition ports into delayed and immediate ports. We also introduce the computation pattern for immediate ports. At time t, the computation function for an immediate port takes into account the values of other ports at the same time t. It follows that the value of such predecessor ports should have been computed before and more generally that the predecessor relation should be acyclic.

Basic sets.

CONTEXT P_Ports
EXTENDS I_Ports
CONSTANTS Delayed Immediate
AXIOMS
axm1 : $Delayed \subseteq Port$
axm2 : $Immediate \subseteq Port$
axm3 : $partition(Port, Delayed, Immediate)$
END

The static description.

VARIABLES t ports compute b

The dynamic description.

Event *ComputeImmediate* $\widehat{=}$ **refines** ***Step***
any p
where
grd1 : $p \in Immediate$
grd2 : $p \notin dom(compute)$
grd3 : $Pred\_I^{-1}[\{p\}] \subseteq dom(compute)$
then
act1 : $compute(p) := C\_I(p)(Pred\_I^{-1}[\{p\}] \lhd compute)$
end

5.4 Introducing port buffering

In this refinement, we make precise the computation pattern for delayed ports. At time t, the computation function for a delayed port takes into account the values of other ports at their last deadline. In order to give access to such *past* values, we use a buffering mechanism. The boundedness of such a buffering is ensured thanks to the properties of the `Deadline` function (see properties (1) of Section 6.2).

Event *ComputeDelayed* $\widehat{=}$ **refines** ***ComputeDelayed***
any p **where**
grd1 : $p \in Delayed$
grd2 : $p \notin dom(compute)$
then
act1 : $compute(p) := C\_D(p)(Pred\_D^{-1}[\{p\}] \lhd mem)$
end

5.5 Port update scheduling

This is our last refinement step. As already discussed in Section 4, the events of this refinement are deterministic and the choice between them is exclusive. Although, unlike classic B, an implementation refinement is not supported by Event B, we believe that this refinement is significant with respect to a true implementation of an AADL data port scheduler. In fact, the only data structure that remains as non implementable with respect to classic B, is the compute partial function. The implementation of partial functions can be considered now as part of the folklore and could be done by automatic refinements as proposed by [18].

6 Development Validation

In this section, we relate some facts about the proposed development. The first one concerns the development proofs and the second one concerns a technical aspect about the resources needed to handle recursive calls.

6.1 Proof obligations

Most of the development has been done with the Rodin platform. There remain, however, some proofs that cannot be done with Rodin mainly related to the last refinement. This refinement relies on lists (B sequences) which are not yet supported by Rodin. It was thus easier for us to translate (manually) the development to Isabelle [17] and carry out all of the proofs within its proof environment. In fact, thanks to the locale mechanism of Isabelle it is easy to simulate Event B context extensions and machine refinements. However, since Isabelle is a general purpose theorem prover and not a method dedicated prover like Rodin, proof obligations related to invariant preservation, refinement and event feasibility had to be generated by hand. Fortunately, Isabelle decision procedures are very powerful and most of the proofs were straightforward.

6.2 Recursive function patterns

In this section, we present the basic ideas underlying the proposed implementation of recursive calls with bounded memory. In our representation of the AADL data port protocol, we have essentially two patterns:

- **well-founded recursion**: this pattern was used for the computation of immediate ports (see Section 4.1). Let us recall that the values of these ports depend on other immediate port values.

  $$\text{Compute}(t)(p) = \text{C\_I}(p)(\{q \mapsto \text{Compute(t)}(q) \mid q \mapsto p \in \text{Pred\_I}\})$$

  Such a computation is possible because we assume that that `Pred_I` is an acyclic relation. Thus, the computation proceeds according to a total order compatible with that acyclic relation. It follows that when an element is processed, all the lower elements have been processed already. Thus, the computation uses a finite number of finite resources: the number of port buffers. We note that each element is processed once. Such a property is not provided by a basic implementation of recursivity. Memoization could have been used; but, since all the elements are, a priori, known and have to be processed, the proposed order-based evaluation strategy is more efficient since it avoids testing if an element has already been processed.
- **past recursion**: this pattern was used for the computation of delayed ports:

  $$\text{Compute}(t)(p) = \text{C\_D}(p)(\{q \mapsto \text{Compute}(\text{Deadline}(t \mapsto q))(q) \mid q \mapsto p \in \text{Pred\_D}\})$$

  where `Deadline` has the following properties:

  $$\text{Deadline}(0) = 0 \wedge \text{Deadline}(t+1) \neq \text{Deadline}(t) \Rightarrow \text{Deadline}(t+1) = t \qquad (1)$$

In fact, thanks to these properties of the `Deadline` function, such a pattern can be implemented through the following primitive recursive pattern:

$$f(0) = c \wedge f(n+1) = g(f(n), n+1)$$

Actually, for such a pattern, the value of $f(n)$ can be computed with one *register* and one *counter*: initializing the register with c and the counter with 0, we compute the successive values of $f(i)$ until the counter values reaches n. Correctness is ensured by the invariant $counter \leq n \wedge register = f(counter)$ and termination by the variant $n - counter$.
The underlying idea of the preceding proposed implementation (Section 5.4) can be summarized as follows: in order to compute $f(\text{Deadline}(t))$ without recursion, we define an auxiliary primitive recursive function a such that:

$$a(0) = \text{Deadline}(0) \wedge a(n+1) = \textbf{if } \text{Deadline}(n+1) = \text{Deadline}(n) \textbf{ then } a(n) \textbf{ else } f(n)$$

We show by induction on n that $\forall n.\ a(n) = f(\text{Deadline}(n))$. Then, since a is primitive recursive, $f(\text{Deadline}(n))$ can be computed in an iterative way with finite memory resources. It follows that the computation of $f_i(n)$ which requires the knowledge of $f_{k \in I}(\text{Deadline}(n))$ also requires finite memory resources since the set I is a priori known.

Remarks.

- We have given here one *underlying* idea of the proposed implementation. It can be reused as a pattern for implementing a recursive function with an unknown recursion depth with bounded memory resources. In a similar way, the other idea concerns the implementation of well-founded recursion.
- The proposed implementation (Section 5.4), does not recompute the result of iterations from one call to another.

7 Related Work

It is becoming acknowledged that one way to make things abstract is to consider them at a level where we have a coarse grain of atomicity. Implementation details are then introduced progressively while maintaining the properties of the coarse grain events. For instance, along these ideas, bus protocols have been developed starting from a synchronous view [12]. In these protocols, the concern is to ensure the correct behavior of the devices with respect the bus lines while establishing basic properties like *mutual exclusion* between the connected devices. We have been concerned by another safety property: the preservation of a *precedence relation* given by a functional specification.

With respect to the specific domain we have been concerned with: computation scheduling, we can cite the work of Stoddard et al. [20] about interrupt scheduling. We note that they are especially concerned by interrupt handling and not by communication aspects. Their work is also concerned by making proofs for an unknown number of tasks. Scheduling aspects have also been dealt with in [13]. Here, the main concern was

to provide a constructive specification of the problem such that certified code could be extracted by the Coq system [5].

The work of [14] has also a semantic concern with respect to AADL. Its aim is to provide a synchronous execution platform for AADL. A Lustre [8] translation semantics of the basic mechanisms is proposed. Thanks to this approach, a model of the whole system is obtained. This model is executable and its properties can be expressed by means of synchronous observers; also, it can be validated or simulated thanks to the specification of the environment and the automatic generation of input sequences. As we have said, this is a translation semantics approach, whereas our work is concerned by the validation of an operational semantics with respect to a denotational semantics.

8 Conclusions

In this paper, we have been concerned by the formalization of an existing protocol offered by the AADL architecture description language. Although the protocol description was precise, we believe that the proposed abstraction through a functional specification is interesting since it is compact and allows to reason about the protocol without going to the intricacies of the implementation. Moreover, although, an implementation can be obtained directly from such a functional description, our proposed implementation relies on, a priori known, finite resources, as is mandatory for an embedded environment. Technically, we have shown that, in some cases, the memory resources needed to handle recursive calls can be, a priori, bounded even if the recursion depth depends on the parameters. It would be worth studying how to facilitate the reuse of such techniques through specification patterns as proposed by [4] and [7].

Concerning future work, we envision to introduce quantitative timing aspects. In this paper, we have made the assumption that computations take *zero* time, or more concretely, that all the required computations take place between two ticks and respect the real time specification deadlines. More generally, we are interested in providing patterns for implementing abstract functional synchronous languages [6] or subsets of AADL [11] on top of concrete asynchronous architectures.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial, D. Cansell, and D. Méry. Refinement and reachability in event$_b$. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer, 2005.
3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
4. E. Ball and M. Butler. Event-B patterns for specifying fault-tolerance in multi-agent interaction. In M. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 104–129. Springer-Verlag, 2009.

5. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. http://coq.inria.fr.
6. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
7. S. Blazy, F. Gervais, and R. Laleau. Reuse of specification patterns with the B method. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland*, volume 2651 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, June 2003.
8. P. Caspi, N. Halbwachs, and P. Pilaud. Lustre: a declarative language for programming synchronous systems. In *Proceedings of the 14th annual symposium on principles of programming languages*, pages 178–188, january 1987.
9. E. Dijkstra. *A Discipline of Programming*. Englewood Cliffs New Jersey: Prentice Hall, 1976.
10. M. Faugère, T. Bourbeau, R. de Simone, and S. Gérard. MARTE: Also an UML profile for modeling AADL applications. In *ICECCS*, pages 359–364. IEEE Computer Society, 2007.
11. P. H. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop 2003*, pages 1–10, May 2003.
12. R. B. Franca, L. Buss Becker, J.-P. Bodeveix, J.-M. Farines, and M. Filali. Towards safe design of synchronous bus protocols in Event-B. In *Brazilian Symposium on Formal Methods, Gramado Brazil, 19/08/2009-21/08/2009*, volume 5902 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2009.
13. N. Izerrouken, M. Pantel, and X. Thirioux. Machine checked sequencer for critical embedded code generator. In *International Conference on Formal Engineering Methods (ICFEM), Rio de Janeiro, Brazil, 09/12/2009-12/12/2009*. Springer-Verlag, December 2009.
14. E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software EMSOFT 2007*, pages 134 – 143, Salzburg, Austria, 2007. ASSERT.
15. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification*. Springer, 1991.
16. MetaH. *http://www.htc.honeywell.com/metah/*. 1997.
17. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
18. A. Requet. Bart: A tool for automatic refinement. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 345–345, Berlin, Heidelberg, 2008. Springer-Verlag.
19. SAE. Aerospace information report. avionics architecture description language. Technical Report AS5506, SAE, march 2002.
20. B. Stoddart, D. Cansell, and F. Zeyda. Modelling and proof analysis of interrupt driven scheduling. In J. Julliand and O. Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 2007.