Synchronous modeling and validation of priority inheritance schedulers*

Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond

VERIMAG (CNRS, UJF, INPG)
Grenoble, France,
{Erwan.Jahier,Nicolas.Halbwachs,Pascal.Raymond}@imag.fr,
URL: http://www-verimag.imag.fr

Abstract. Architecture Description Languages (ADLs) allow embedded systems to be described as assemblies of hardware and software components. It is attractive to use such a global modelling as a basis for early system analysis. However, in such descriptions, the applicative software is often abstracted away, and is supposed to be developed in some host programming language. This forbids to take the applicative software into account in such early validation. To overcome this limitation, a solution consists in translating the ADL description into an executable model, which can be simulated and validated together with the software. In a previous paper [1], we proposed such a translation of AADL (Architecture Analysis & Design Language) specifications into an executable synchronous model.
The present paper is a continuation of this work, and deals with expressing the behavior of complex scheduling policies managing shared resources. We provide a synchronous specification for two shared resource scheduling protocols: the well-known basic priority inheritance protocol (BIP), and the priority ceiling protocol (PCP). This results in an automated translation of AADL models into a purely Boolean synchronous (Lustre) scheduler, that can be directly model-checked, possibly with the actual software.

keywords: Embedded systems, Simulation, Scheduling, Formal Verification, Architecture Description Languages, Synchronous Languages.

1 Introduction

The European project ASSERT is devoted to safe model-driven design of embedded systems, with aerospace systems as a main application domain. Such systems are deployed on specific architectures that need to be described and simulated in order to allow early validation of the integrated system.

The approach taken in the ASSERT project is to describe the execution architecture separately from the software components. The target architecture is

* This work was partially supported by the European Commission under the Integrated Project ASSERT, IST 004033, which ended in 2008

described in the AADL architecture description language [2, 3]. AADL provides a collection of classical systems components, which can be instantiated and assembled to describe the actual execution platform. In a typical AADL description, a system is made of several *computers*, communicating through *buses*; a computer is made of *memory* and *processors*, and a processor runs a *scheduler* and several *tasks*; at last, tasks are running *applicative software*. Those software components can be developed using several programming languages, including ADA, C, or even Scade and Lustre via a C wrapping.

AADL components are decorated with information like rates and Worst Case Execution Time (WCET) for periodic tasks, scheduling policy, etc. Those informations are intended to be used in the validation of the platform, mainly by checking properties like the absence of deadlocks, or the respect of deadlines. The functional part is expressed by the software components, and thus generally completely ignored, although it may influence some non-functional aspects. For instance, a software component may produce some event that wakes up a task; the scheduling environment and the execution times are then modified.

Our main objective is to perform simulation and validation that take into account both the system architecture and the functional aspects. We consider the case where software components are implemented in the synchronous programming language Lustre/Scade[1]. Our proposal in [1] is to build automatically a simulator of the architecture, expressed in a synchronous language like the software components. This approach presents several advantages: first, synchronous languages are well-known to be able to express non-synchronous behaviors, while the converse is more difficult; now, getting all aspects in the same model allows both functional and system aspects to be considered jointly. For instance, in AADL, sporadic tasks can be activated by the output of some other components. Therefore, in such cases, more realistic simulation and finer-grained formal verification can be performed.

The translation proposed in [1] takes into account various asynchronous aspects of AADL such as task execution time, periodic or sporadic activations, multitasking (using Rate Monotonic Scheduling [5]), and clock drifts. The result is an executable integrated synchronous model, combining architecture behavior with actual software components, which can be validated with tools available for synchronous programs.

In this paper, we propose to extend this work by taking into account shared resources using different protocols (no lock, blocking, basic inheritance, priority ceiling). We also show how various properties related to determinism, schedulability, or the absence of deadlock can be automatically checked on given architecture models.

The article is organized as follows. We first recall in Section 2 the principles of simulation of AADL in the synchronous paradigm. Then we describe in Section 3 how to deal with shared resources and various shared access protocols in a synchronous program. Finally, we show in Section 4 how one can use the

[1] Scade is the industrial version of Lustre[4], and is maintained and distributed by the Esterel-Technology company.

resulting executable model to check various kinds of properties (determinism, schedulability, absence of deadlock), and to perform monitored simulations.

2 From AADL to synchronous programs

This section recalls the main features of the Architecture Analysis & Design Language (AADL), as well as the synchronous paradigm. Then it briefly recalls how the behavior of an (asynchronous) AADL model can be modeled by a non-deterministic synchronous program. This subject is presented in detail in [1].

2.1 The AADL description language

An AADL model is made of an arborescent assembly of software and hardware components [2, 3]. A component is defined by an interface (input and output *ports*), a set of sub-components, a set of *connections* linking up the subcomponents ports, and a set of typed attributes (called *properties*). The main kinds of AADL components are the following.

Systems are top-level components; they describe the mapping between software and hardware components. *Device* components model hardware responsible for interfacing the system with its environment. They are typically used to represent sensors or actuators. From a functional point of view, they correspond to the inputs and the outputs of the system. *Processor* components are abstractions of hardware and software responsible for scheduling and executing threads.

Memory components (hardware) are used to specify the amount and the kind of memory that is available to other components.

Data components (software) are used to represent data types in the source text. Other components might have a shared access to data components. The access policy is controlled by the `Concurrency_Control_Protocol` property (lock, priority ceiling protocol, cf. Section 3). *Bus* components (hardware) are used to exchange data between components on different processors.

Process components are abstractions of software responsible for defining a memory space that can be accessed by the *thread* sub-components it contains. *Thread* components are abstractions of software responsible for executing applicative programs. When several threads run under the same processor, the sharing of the processor is managed by a runtime scheduler. The `dispatch_protocol` property is used to specify that scheduling policy. For instance, the value `periodic` means that the thread must be activated according to the specified `period`; the value `aperiodic` means that the thread is activated via one of the other components' output ports (called *event* ports). *Sub-program* components are the leaves of this arborescent description. Their implementations need to be provided in some host language. In our approach, if one wants to be able to formally analyze aperiodic threads whose activation depends on the functional output of some program component, one needs to provide for it a synchronous program (or at least a wrapper), e.g., written in Scade or Lustre. The property `compute_exec_time` specifies a range for the worst case execution time (WCET)

of the program. In the sequel, we use the term *task* to denote a thread running a program.

2.2 The synchronous paradigm

We present now the essentials of the synchronous paradigm, focusing on the aspects that will be used later on.

A synchronous program (also called a node) is a dynamic system evolving on a discrete time scale. It has an internal memory made of state variables, inputs and outputs, and its behavior is a (virtually infinite) sequence of atomic reactions. Each reaction consists in reading current inputs, computing outputs, and updating the internal memory (state). In other terms, synchronous programs are a straightforward generalization of synchronous circuits (i.e., sequential circuits or Mealy machines), where data can be of arbitrary types rather than just Boolean values.

A synchronous program is characterized by a vector of inputs $\boldsymbol{i}$, a vector of outputs $\boldsymbol{o}$, a vector of state variables $\boldsymbol{s}$. Its semantics is defined by its initial state s_0 (the initial value of $\boldsymbol{s}$), and the functions f_o and f_s, respectively returning the output and the next state from the current inputs and the current state. For each instant t:

$$o_t = f_o(i_t, s_t) \quad ; \quad s_{t+1} = f_s(i_t, s_t)$$

A program without state is called a *combinational node*; usual functions like arithmetic or logical operators are then naturally lifted to synchronous programs.

For any data-type τ with a well-defined default value d, a "delay" (or register) program can be defined as follows:

$$s_0 = d \quad ; \quad f_o(i, s) = s \quad ; \quad f_s(i, s) = i$$

In the sequel we mainly use Boolean (resp. integer) registers, with false as default value (resp. 0), and represented by the symbol $\bullet$.

The main characteristic of synchronous programs is the way they are composed: when connecting several sub-programs, a reaction of the whole program consists of a *simultaneous* reaction of all the components. In other words, the synchronous paradigm provides an idealized representation of parallelism.

An important consequence is that a big synchronous program can be described as a parallel composition of smaller sub-programs. In this approach, a program is described as a data-flow network of synchronous programs connected by wires. Fig. 1 shows the data-flow network of a synchronous counter, made of a delay node and three combinational nodes (two "if-then-else" and an adder). For the sake of conciseness, we use sets of equations rather than drawings for representing such networks. For instance, the set of equations equivalent to the counter is the following:

$$c = \text{if } r \text{ then } 0 \text{ else } s \quad ; \quad s = \bullet c + \text{if } x \text{ then } 1 \text{ else } 0$$

Note that such a set of equations has a straightforward solution as long as it does not contain combinational loops. In other words, any feed-back loop in

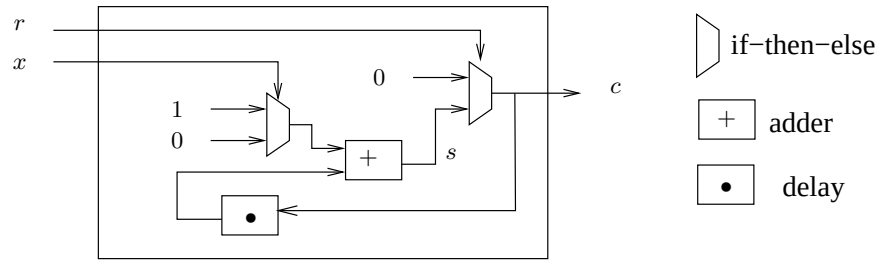

Fig. 1. The data-flow network of a synchronous counter.

the network should pass through a delay operator. In the following, we will take care to define only such well-founded data-flow networks.

At last, all synchronous formalisms are providing a notion of under-sampling, also called activation condition, or clock-enable in the domain of synchronous circuits. The activation condition is an higher-order operator that takes a synchronous node P, a Boolean input b, and produces a new node. The behavior of that node is defined as follows: whenever b is true, it behaves exactly like P, and whenever b is false, both the internal state and the outputs are "frozen" (i.e. they keep their previous value).

2.3 Modeling asynchrony in the synchronous framework

The ability of the synchronous framework to model asynchrony is well-known [6], and has been often used [7–11]. In [1], we used a similar technique for translating a subset of AADL into synchronous data flow equations.

This goal is mainly achieved by using "oracles" (i.e., additional inputs) for modeling non-determinism, and activation conditions for modeling the asynchronous aspects: time-consuming tasks, multi-tasking, clock jitter.

However, in this previous work, multi-tasking was only considered in the case of simple fixed priorities rate monotonic scheduling. In this article, we consider more sophisticated policies that take into account shared resources with protected access, and all the problems they raise: priority inversion, and deadlock.

3 Handling shared resources

In AADL, Data component accesses can be shared between several components. In contrast to other kinds of components (thread, process, sub-program) which are translated into nodes, data components are translated into local variables of the surrounding component node. Depending on the kind of access that is associated with them (read_only, write_only, or read_write), the necessary wires are added to the interface of the node: a data component that has a write (resp., read) access to a resource has an additional output (resp., an additional input), and the data update is performed at its dispatch time using an activation condition.

In order to guarantee the data integrity, it is necessary to prevent the resource from being accessed by several components at the same time. For that

purpose, several concurrency control protocols were defined [12], that modify the classical Rate Monotonic scheduling. In AADL, this is specified through the "Concurrency_Control_Protocol" property, attached to a data component. In this section, we explain how to implement four kinds of concurrency control protocol:

- NoneSpecified: components access the shared resource with no constraint at all (no lock mechanism).
- Lock: Before accessing a shared resource, a component should ask for it, and gets it only if no other component has locked it before; otherwise, it is suspended until the resource is unlocked. When a component obtains a resource, we say that the component enters a *critical section*.
  Hence, a low priority thread tl can block a high priority one th if th wants to access a resource that is locked by tl. The problem with this protocol is that tl can block th, even when tl is not in critical section. This is referred to as *the priority inversion problem* [12].
- BIP: The Basic Inheritance Protocol, also known as Priority Inheritance Protocol, refines the previous one to prevent priority inversions.
- PCP: The Priority Ceiling Protocol is a refinement of BIP defined in order to prevent deadlock.

In the following, we describe those protocols more precisely, and explain how to implement them in terms of synchronous data-flow equations. Defining a scheduling protocol consists of defining a node, called hereafter a scheduler, that decides at each instant which thread the CPU is attributed to.

3.1 The *No Lock* protocol

The simplest way of handling shared resources is to ignore them, and to always give the CPU to the highest priority thread. This (absence of) protocol is straightforward and generally useless for systems involving shared resources. But this simplest scheduler is refined in later sections for the other protocols. It is basically the scheduler used in [1].

Concretely, we need to generate a synchronous program that takes as inputs Boolean variables indicating which threads ask for the CPU ($Dispatched_1$, ..., $Dispatched_n$), and that returns Boolean variables indicating which thread is elected (cpu_1, ..., cpu_n). Of course, at most one among the cpu_i should be true at each instant. The program that computes the $Dispatched_i$ variables is derived from the period and the WCET of threads, which is specified in the AADL code.

The convention here is that t_i has priority over t_j if $i < j$. A possible way of implementing that node is as follows:

$$\forall k \in [1, n] : \ cpu_k = Dispatched_k \wedge \bigwedge_{0<i<k} \overline{cpu_i} \tag{1}$$

Henceforth, the convention is that the program input variables begin with an uppercase letter (e.g., $Dispatched_k$); and $\overline{cpu_i}$ stands for the negation of cpu_i.

3.2 The *Blocking* protocol

In order to take into account shared resources, we need additional inputs: the Boolean variable named $Asks\_cs_{r_\ell}^{t_i}$ indicates that the thread t_i wants to access the resource r_ℓ (their values come from the output of the predefined AADL subprograms Get_resource and Set_resource [3]).

In order to ease the definition of cpu_k, we introduce the following auxiliary variables:

- the Boolean variable $has\_cs_{r_\ell}^{t_k}$ indicates that the thread t_k is in Critical Section on resource r_ℓ;
- the Boolean variable $t_i\_blocks_{r_\ell}^{t_k}$ indicates that the thread t_k asks for a resource r_ℓ, which is locked by another thread t_i.

Computing which thread is in critical section. A thread t_k is in critical section for a resource r_ℓ if it asks for the resource, and if either

- it was in critical section before ($\bullet\ has\_cs_{r_\ell}^{t_k}$);[2]
- or it enters in critical section at the current instant. It enters a critical section when and only when it obtains the CPU.

Hence, the following definition of $has\_cs_{r_\ell}^{t_k}$:

$$\forall k \in [1,n], \forall \ell \in [1,m] :\ has\_cs_{r_\ell}^{t_k} = Asks\_cs_{r_\ell}^{t_k}\ \wedge\ (cpu_k\ \vee\ \bullet\ has\_cs_{r_\ell}^{t_k}) \qquad (2)$$

Note that when we define such a relation, the quantification "$\forall k \in [1,n], \forall \ell \in [1,m]$" suggests that we generate $n \times m$ equations for defining the scheduler. But in fact, it is generally much less, since in the AADL model, all threads may not have access connections to all resources. This remark actually holds for all the variables relating threads and resources in the following.

Computing the blocks relation. We say that a thread t_i blocks a thread t_k via a resource r_ℓ if both threads ask for the resource, and if the thread t_i was owning r_ℓ at the previous instant.

$$\forall k, i \in [1,n], i \neq k, \forall \ell \in [1,m] :$$
$$t_i\_blocks_{r_\ell}^{t_k} = Asks\_cs_{r_\ell}^{t_k}\ \wedge\ Asks\_cs_{r_\ell}^{t_i}\ \wedge\ \bullet has\_cs_{r_\ell}^{t_i} \qquad (3)$$

Computing the elected thread. Once we have defined those two auxiliary relations, cpu_k can easily be defined similarly as in Section 3.1: the highest priority thread obtains the CPU, except if it is blocked by some other thread:

$$\forall k \in [1,n] :\ cpu_k = Dispatched_k \wedge \bigwedge_{0<i<k} \overline{cpu_i} \wedge \bigwedge_{i \neq k, \ell \in [1,m]} \overline{t_i\_blocks_{r_\ell}^{t_k}} \qquad (4)$$

Note that those three sets of equations defines a valid synchronous program, since they do not contain any combinational cycle (cf Section 2.2).

[2] All Boolean delays ($\bullet$) are implicitly initialized to false.

3.3 The *Basic Inheritance* Protocol

The Basic Inheritance Protocol was introduced [12] to avoid the *priority inversion problem*. Indeed, with the previous protocol, when a high-priority thread t_1 wants to access a resource shared by a lower priority thread t_3, which have put a lock on it, then t_3 keeps the CPU. Moreover, t_3 can be interrupted by t_2 of lower priority than t_1, even though t_2 does not try to access any shared resource.

The idea of the Basic Inheritance Protocol (BIP) is to modify the priority of t_3 in such a way that it inherits the priority of t_1, when t_3 has the lock on a resource r_ℓ requested by t_1. Indeed, this prevents t_2 to interrupt t_3, and hence prevents the priority inversion.

The intuition of our BIP (synchronous data-flow) encoding is the following: first consider the dispatched thread with the highest priority. If it is not blocked, it must obtain the CPU. Otherwise, consider its blocking thread, and check if it is itself blocked, and so on until we find a thread that is not blocked. When we find the thread that is not blocked[3], we give it the CPU. Hence, the first thing to do is to compute the transitive closure of the $t\_blocks^t_r$ relation.

Computing the $t_i\_blocks^*_{t_k}$ relation. Let an *inhibition path from a thread t_i to a thread t_k* be a set of $s+1$ threads $\{t_i = t_{i0}, ..., t_{is} = t_k\}$ such that there exist s resources $r_1, ..., r_s$, that may be respectively accessed by t_{i0} and t_{i1}, t_{i1} and t_{i2}, ..., t_{is-1} and t_{ks}. Such a path is said to be *cycle-free* if all the threads in the path are distinct. Let $Path(i,k)$ be the set of cycle-free paths from t_i to t_k (this set can be computed from the AADL source code). A thread t_i $blocks^*$ another thread t_k if t_i is not itself blocked, and if there exists an inhibition path in $Path(i,k)$ that is true.

$$\forall i,k \in [1,n], i \neq k : \tag{5}$$

$$t_i\_blocks^*_{t_k} = \overline{t_i\_is\_blocked} \wedge \bigvee_{p=\{i_0,\ldots,i_s\}\in Path(i,k)} t_{i_0}\_blocks^{t_{i_1}}_{r_1} \wedge \ldots \wedge t_{i_{s-1}}\_blocks^{t_s}_{r_s}$$

where:

$$\forall k \in [1,n] : \; t_k\_is\_blocked = \bigvee_{\ell\in[1,m], j\in[1,n], j\neq k} t_j\_blocks^{t_k}_{r_\ell}$$

The protocol. The BIP states that a thread in critical section on a resource *inherits* the priority of any other higher priority thread that asks for the same resource. The difficulty is to translate this "dynamic" condition[4] into a Boolean condition. To do that we use an accumulator, (named *ii*, which stands for inhibiting index), that carries the value of the inhibitor of the thread that has the highest priority, if the dispatched thread with the highest priority is blocked (*ii* is set to 0 or -1 otherwise). For readability, we use a switch-like notation, where $c_1 \rightarrow x_1, c_2 \rightarrow x_2, ..., c_n \rightarrow x_n$ stands for $if\ c_1\ then\ x_1\ else\ if\ c_2\ then\ x_2\ \ ...\ if\ c_n\ then\ x_n$.

[3] if such a thread does not exist, the model-checker will tell us (cf Section 4.1).

[4] The priority of each thread depends on the history. But by chance, it only depends on a very short history, that is, the previous instant.

$$ii_0 = 0$$

$$\forall k \in [1,n]: \ (cpu_k, ii_k) = \ \overline{dispatched_k} \rightarrow (False, ii_{k-1}) \quad (6)$$

$$(cpu_1 \ \vee \ ... \ \vee \ cpu_{k-1}) \rightarrow (False, -1) \quad (7)$$

$$ii_{k-1} = k \rightarrow (True, -1) \quad (8)$$

$$ii_{k-1} > 0 \rightarrow (False, ii_{k-1}) \quad (9)$$

$$\{ \ t_j\_blocks^*_{t_k} \rightarrow (False, j) \ \}_{j \in [1,n], j \neq k} \quad (10)$$

$$True \rightarrow (\overline{t_k\_is\_blocked}, 0) \quad (11)$$

For each $k > 0$, cpu_k and ii_k depend on cpu_{k-1} and ii_{k-1}, which means that cpu_1 and ii_1 are computed first, and then cpu_2 and ii_2, and so on, until cpu_n and ii_n. At the beginning, the inhibiting index is equal to 0 ($ii_0 = 0$). Then, the pairs (cpu_1,ii_1), ..., (cpu_n,ii_n) are computed in turn. As long as cpu_{k-1} is set to $False$ (i.e., when conditions of lines 8 and 11 do not hold):

- If t_k is blocked by a lower priority thread t_j (line 10), the inhibiting index takes the priority of the inhibitor j. Then, the inhibiting index keeps this value (lines 6 and 9), until the index of the inhibitor is reached (line 8). In that case, the corresponding cpu variable is set to $True$, the remaining values of cpu are set to $False$ (line 7), and ii_k is unused for bigger k (-1).
- Otherwise (line 11), if t_k is not blocked at all, it gets the CPU, and all the remaining values of cpu are set to false (line 7). If it is blocked, the system deadlocks.

3.4 The *Priority Ceiling* Protocol

The problem with the BIP is that it does not prevent deadlocks. Indeed, consider the following scenario, where 2 threads t_1 and t_2 share 2 resources r_1 and r_2:

1. t_2 asks for the CPU ($Dispatched_1$) and gets it.
2. t_2 locks r_1.
3. t_1 asks for the CPU. It has a higher priority than t_2, hence t_1 gets the CPU.
4. t_1 locks r_2.
5. t_1 tries to lock r_1. But t_2 has locked it. Therefore t_2 gets the CPU.
6. t_2 tries to lock r_2. But t_1 has locked it. Nobody can get the CPU. The system is blocked (or deadlocks).

One solution is to (statically) forbid such intertwined use of locks. Another solution is to use the so-called Priority Ceiling Protocol (PCP). The PCP is a refinement of the BIP.

The *priority ceiling of a resource* r_ℓ is the maximal priority of all the threads that may use r_ℓ; we note it $PC(\ell)$. The *priority ceiling of a thread* t_k is the maximum of the priority ceilings of the resources locked by other threads; we note it PC_k. Contrary to $PC(\ell)$, PC_k is a dynamic value. The PCP consists in adding the following constraint to the BIP: t_k can lock a resource r only if its priority is higher than its priority ceiling ($k < PC_k$).

The Priority Ceiling of resources locked by threads other than k. PC_k formal definition is just a direct translation of the definition given above.

$$\forall k \in [1,n]: \ PC_k = \tag{12}$$
$$Min\ \{n+1\} \ \cup \ \left\{ PC(\ell) \ / \ Asks\_cs^{t_i}_{r_\ell} \ \wedge \ \bullet has\_cs^{t_i}_{r_\ell} \right\} \begin{matrix} \ell \in [1,m] \\ i \in [1,n], i \neq k \end{matrix}$$

The tk_ask relation. We first define yet another auxiliary relation that states whether a thread wants to enter a critical section at the current instant (i.e., a thread asks for a resource that it hasn't locked yet).

$$\forall k \in [1,n]: \ asks\_cs^{t_k} = \bigvee_{\ell \in [1,m]} (Asks\_cs^{t_k}_{r_\ell} \ \wedge \ \overline{\bullet has\_cs^{t_k}_{r_\ell}})$$

The protocol. The PCP encoding is the same as the BIP one, except that we modify the definition of the *blocks* relation (previously defined in equation 3). Indeed, there is now a second reason for a thread t_k to be blocked by another thread t_i: if t_k wants to enter a critical section ($asks\_cs^{t_k}$) when its priority ceiling PC_k is not higher than its own priority ($PC_k \leq k$). Note that the priority ceiling of t_k (i.e., the value of PC_k) is a consequence of the lock that t_i has on the resource ℓ ($PC(\ell) = PC_k$).

$$\forall k, i \in [1,n], i \neq k, \forall \ell \in [1,m] : t_i\_blocks^{t_k}_{r_\ell} = \tag{13}$$
$$Asks\_cs^{t_i}_{r_\ell} \ \wedge \ \bullet has\_cs^{t_i}_{r_\ell} \ \wedge \ (Asks\_cs^{t_k}_{r_\ell} \vee (asks\_cs^{t_k} \ \wedge \ PC(\ell) = PC_k \leq k))$$

Here again, those set of equations defines a valid synchronous program as they do not contain any combinational loop.

4 Validation

We have encoded all the equations given in the previous Section into an OCAML (meta-)program that, given a set of tasks, a set of resources, and a set of task/resource pairs, generates a LUSTRE program[5]. The resulting LUSTRE program is a task scheduler, computing one Boolean variable (cpu_i) per thread (t_i), from Boolean inputs indicating which threads ask for the CPU ($Dispatched_{t_i}$), and which threads ask for which resource ($Asks\_cs^{t_i}_{r_l}$).

In the following, we illustrate the use of a state-explorer (i.e., a model-checker) to prove various properties of this generated program. This was very useful to debug the equations given in this paper, and also to debug the OCAML encoding of those equations. We'll also argue why we believe it might also be useful for AADL end-users.

[5] We put a copy of this OCAML program as well as a copy of the resulting LUSTRE programs at the url `http://www-verimag.imag.fr/~jahier/aadl-schedul/`

4.1 Absence of deadlock

In order to prove the absence of deadlock, we used LESAR [13], a LUSTRE model-checker. This tool implements state-of-the-art state-reachability algorithms, based on Binary Decision Diagrams. We used both a enumerative algorithm which complexity is related to the number of states, and a symbolic algorithm, which complexity is related to the diameter of the state space.

We proved with LESAR that, whenever at least one thread asks for the CPU, at least one of the cpu_i is true. Actually, we even prove that exactly one cpu_i is true in that case, which simply proves that our scheduler is correct in the sense that it does not give the processor to more than one thread:

$$(\bigvee_{i\in[1,n]} \ Dispatched_i) \Rightarrow \bigvee_{i\in[1,n]} \ cpu_i$$

We performed this on the examples of Fig. 2. For instance, the first example (ex. 1) of Fig. 2 consists of a system with two threads $t1$ and $t2$, that can access two resources $r1$ and $r2$. This example is precisely the one given in [12] to illustrate the fact that the BIP does not prevent deadlock, and which motivates the definition of PCP.

LESAR was indeed able to generate a counter-example that exhibits a deadlock; the scenario it provides is almost the same as the one given [12] (and also in Section 3.4). LESAR proved the absence of deadlock for the PCP on the three examples. The results of those experiments are outlined in Fig. 3. When the property is false, we indicate the length of the counter-example. When the property is true, we indicate the diameter of the graph, and its number of states. All runs lasted less that a second.

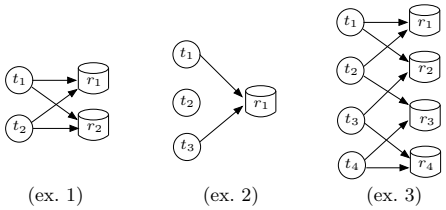

Fig. 2. Examples of tasks accessing shared resources.

An interesting point in those experiments is that it is not always worth using the PCP (that is deadlock-free by construction) since the BIP and the lock protocol can provably be deadlock-free in some configurations (e.g., in ex. 2). Note that in order to avoid false alarms, we need to tell the state-explorer that the inputs of the scheduler are not completely random. For instance, it was necessary to assert that a thread cannot change its requests for resources when it does not own the CPU.

	Lock	BIP	PCP
ex. 1	ko: 5	ko: 5	ok: 5/40
ex. 2	ok: 6/96	ok: 7/96	ok: 7/96
ex. 3	ko: 9	ko: 9	ok: 12/2316

Fig. 3. Deadlock property exp.

Lock	BIP	PCP
ok: 6/46	ok: 6/46	ok: 5/40
ko: 4	ok: 7/96	ok: 7/96
ko: 4	ok: 10/3708	ok: 12/3216

Fig. 4. Priority-inversion property exp.

4.2 Priority inversion

The priority inversion corresponds to situations when a thread is blocked by a lower priority thread. This occurs very naturally when two threads share the same resource, locked by the lower priority thread. Priority inversion is more problematic when it happens as in the example of Section 3.3 (which was the example given in [12] to motivate the introduction of the BIP). Indeed, threads are generally supposed to remain in critical section for a short time. Now, if a thread that does not lock any resource preempts a lower priority thread in critical section, the corresponding resource might be locked for a long time.

Therefore, we check the following property: if a thread t_k gets the CPU, when a higher priority thread asks to enter in critical section, then t_k should have at least a lock on one of the resource. In other words, we want to be sure that a thread that does not lock any resource cannot block any higher priority thread:

$$\forall i \in [2,n], \forall j \in [1,i-1] : \ (cpu_i \ \wedge \ asks\_cs^{t_j}) \Rightarrow \bigvee_{\ell \in [1,m]} has\_cs^{t_i}_{r_\ell}$$

We actually ask the model-checker to prove a slightly higher refined property, which is that for any system that does not deadlock, there is no priority inversion. Indeed, as soon as two tasks deadlock, any other thread can get the CPU even if it is not supposed to, according to the priorities defined by the protocol. This is the kind of subtlety that can be discovered using a model-checker.

As summarized in Fig. 4, Lesar found counter-examples that falsify the property for the last two examples of Fig. 2 using the lock protocol. And it proved the property with the BIP and the PCP. The second example is the one given [12] (and in Section 3.3) for motivating the introduction of the BIP.

4.3 Schedulability

The thread scheduler we generate in Section 3 is just a part of the Aadl2lustre translator [1]. The program that computes the values of the $Dispatched_i$ variables is derived from the AADL code (from the threads period and WCET).

In order to check the schedulability of an Aadl system, we look at the sequences of values taken by the $Dispatched_i$ and cpu_i variables. The set of valid sequences is defined by the automaton of Fig. 5. In this automaton, d stands for "dispatch", and is defined as the $Dispatched$ rising edge; a stands for "activate", and is defined as the cpu rising edge; and r stands for "release", and is defined as the $Dispatched$ falling edge. All omitted transitions in this automaton target the "scheduling-error" state. A system is well-scheduled if this error state is never reached. Indeed, nothing prevents the generated scheduler to issue a "dispatch" event between an "activate" and a "release" event. This is what occurs when the system is not schedulable, i.e., when some deadline is missed. Once encoded into a Lustre formula, this automaton can be used to prove (by state-exploration) that the system is schedulable.

Note that this schedulability property somehow does not only concern the part of the Aadl2lustre translator described in this article. But we mention

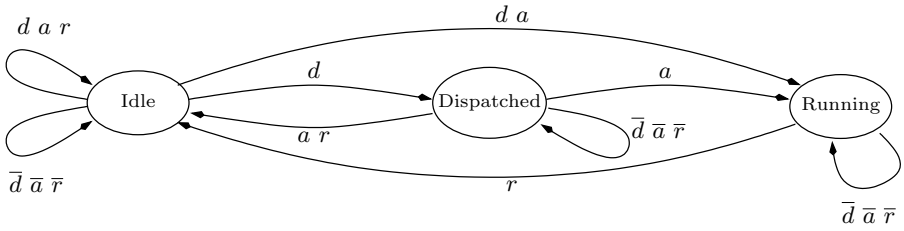

Fig. 5. The automaton recognizing well-scheduled systems. There is one such automaton per thread to schedule.

it here because we believe this analysis is particularly interesting in presence of shared resources.

5 Related work

The Cheddar tool [14, 15] can perform Schedulability analysis over AADL specification, but it ignores the functional aspects of AADL components, and it is more oriented towards quantitative analysis: resource usage, number of preemptions, number of context switches, etc. Cheddar allows users to define dedicated (user defined) schedulers and perform simulations [16].

Using a synchronous framework to model software architectures is not a new idea [9, 10, 8, 11]. Gamatié et al. [9, 10] defined a framework that provides a library of components, written in SIGNAL [17] and C++, suitable for modeling systems following the ARINC (Aeronautical Radio Incorporated) 653 standard. They demonstrate how to use the SIGNAL language as an ADL – whereas we translate AADL architecture models into LUSTRE. They mention that model-checking could be possible since the system is described in SIGNAL, but the task scheduler is implemented in C++, which would make its model-checking difficult – they do not pretend to be able to check the scheduler tough. Anyway, they do not mention any particular protocol with respect to shared resource handling.

Formal verification of priority inheritance protocols has also been conducted using the PVS theorem prover [18]. The kind of outcome that one obtains using a theorem prover is of course different of what can be achieved with a model-checker. With PVS, Dutertre proves very general property about the PCP correctness. On the contrary, we model-check the protocol together with the system architecture description, plus the functional components. We are therefore able to prove much more fine-grained properties, not only about the whole system behavior, but also about the scheduling protocol itself. Moreover, some protocol properties can be false in the general case; for example, we proved that the second system of Fig. 2 does not deadlock with the BIP.

Penix et al. used a model-checker to verify a rate monotonic scheduler of a real time operating system [19]. But as their scheduler model is very detailed, here again the rest of the architecture is kept abstract. Elaborated protocols for dealing with shared resources are not addressed either.

6 Conclusion

Defining an automated translation from AADL models to a purely Boolean synchronous scheduler, that can be directly model-checked, has many advantages.

- Firstly, the model-checker was very useful to debug our scheduler generator.
- Secondly, we claim it can also be useful for the AADL end-users; for example, the PCP is a refinement of the BIP that has been introduced to avoid deadlocks. However, for some particular topologies of threads and resources, it may happen that deadlocks cannot occur even with the BIP scheduler, and that a model-checker is able to prove it on our model.
- Finally, in presence of shared resources, the analytic schedulability criteria may be too conservative, and reject schedulable systems. Moreover, as soon as the system contains sporadic events (when the thread activation depends on the output of some other thread), the analytic method can be meaningless. Consider for example two components activated by a third one, which both outputs cannot be true at the same instant.

Of course with our technique, one cannot deal with generic properties (i.e., for any number of tasks and resources), but since the generation of models for verification is automatic, the verification can be replayed for each instance.

When the verification problem is too large, an exhaustive verification can be untractable. However, our encoding can still be useful to perform intensive automatic simulations using testing tools like Lurette [20]. The absence of deadlocks, the schedulability, and the non-inversion properties are used as test oracles (i.e., runtime monitors). The assertions on the scheduler inputs (e.g., no rising edges for the asking of a resource by threads that do not have the CPU) are used to constrain the random input generator [21].

Another case where such tests and simulations are the only tractable methods is when the AADL model contains sporadic threads activated by software components that are not implemented in Lustre (or in any other language with formal semantics). A way around this problem would be to have a Lustre abstraction of all the possible behavior of such components; but such an abstraction is not always easy to define.

Acknowledgments

We thank Karine Altisen for fruitful (nearby office) discussions about scheduling.

References

1. Halbwachs, N., Jahier, E., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Seventh International Conference on Embedded Software (EMSOFT 2007), Salzburg, Austria (October 2007)
2. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded system architecture analysis using SAE AADL. Technical note cmu/sei-2004-tn-005, Carnegie Mellon University (2004)

3. SAE: Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace (November 2004)
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE **79**(9) (September 1991) 1305–1320
5. Liu, C.L., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. JACM **20**(1) (1973) 46–61
6. Milner, R.: On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ. (1981)
7. Baufreton, P.: SACRES: A step ahead in the development of critical avionics applications. In Vaandrager, F., van Schuppen, J., eds.: Hybrid Systems: Computation and Control: Second International Workshop, HSCC'99, LNCS 1569, Springer-Verlag (1999)
8. Baufreton, P.: Visual notations based on synchronous languages for dynamic validation of gals systems. In: CCCT'04 Computing, Communications and Control Technologies, Austin (Texas) (August 2004)
9. Gamatié, A., Gautier, T.: Synchronous modeling of avionics applications using the signal language. In: 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'2003), Toronto (May 2003) 144–151
10. Gamatié, A., Gautier, T.: The signal approach to the design of system architectures. In: 10th IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2003), Huntsville (Alabama) (April 2003) 80–88
11. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for system design. Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design (April 2003)
12. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Computers **39**(9) (1990) 1175–1185
13. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems (September 1992) 785–793
14. Hugues, J., Zalila, B., Kordon, L.P.F.: Rapid prototyping of distributed real-time embedded systems using the AADL and Ocarina. In: 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07). (2007)
15. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar: a flexible real time scheduling framework. In McCormick, J.W., Sward, R.E., eds.: SIGAda, ACM (2004) 1–8
16. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with AADL. In: SIGAda. (2005)
17. Guernic, P.L., Benveniste, A., Bournai, P., Gautier, T.: SIGNAL, a data flow oriented language for signal processing. IEEE-ASSP **34**(2) (1986) 362–374
18. Dutertre, B.: Formal analysis of the priority ceiling protocol. In: In IEEE Real-Time Systems Symposium (RTSS'00. (2000) 151–160
19. Penix, J., Visser, W., Engstrom, E., Larson, A., Weininger, N.: Verification of time partitioning in the deos scheduler kernel. In: ICSE. (2000) 488–497
20. Jahier, E., Raymond, P., Baufreton, P.: Case studies with Lurette V2. International Journal on Software Tools for Technology Transfer (STTT) **Special Section on Leveraging Applications of Formal Methods** (2006)
21. Raymond, P., Jahier, E., Roux, Y.: Describing and executing random reactive systems. In: SEFM, IEEE Computer Society (2006) 216–225