Synchronous modeling and validation of schedulers dealing with shared resources[1]

Erwan Jahier, Nicolas Halbwachs, Pascal Raymond

Verimag Research Report n^o TR-2008-10

Jul 17 2008

Synchronous modeling and validation of schedulers dealing with shared resources[2]

Erwan Jahier, Nicolas Halbwachs, Pascal Raymond

Jul 17 2008

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 [8], 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

Reviewers: Florence Maraninchi

Notes:

How to cite this report:

@techreport { ,
title = { Synchronous modeling and validation of schedulers dealing with shared resources[3] },
authors = { Erwan Jahier, Nicolas Halbwachs, Pascal Raymond},
institution = { Verimag Research Report },
number = {TR-2008-10},
year = { },
note = { }
}

1 Introduction

The European project ASSERT is devoted to the safe model-driven design of embedded systems, with aerospace systems as 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 described in the AADL architecture description language [4, 16]. 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 Scade/Lustre and ADA.

AADL components are decorated with information like rates and WCET (Worst Case Execution Time) 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[4]. Our proposal in [8] 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 (using the concept of events). Therefore, in such cases, more realistic simulation and finer-grained formal verification can be performed.

The translation proposed in [8] takes into account various asynchronous aspects of AADL such as task execution time, periodic or sporadic activations, multitasking (using Rate Monotonic Scheduling [13]), 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 inter-locking can be automatically model-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 resulting executable model to model-check various kinds of properties (determinism, schedulability, absence of inter-locking), and to perform monitored simulations.

2 From AADL to synchronous programs

This section briefly recalls how the behavior of an (asynchronous) AADL model can be modeled by a non-deterministic synchronous program. This subject is presented in details in [8].

2.1 The AADL description language

We briefly recall here the main features of the Architecture Analysis & Design Language (AADL). The complete definition can be found in [4, 16].

An AADL model is made of a hierarchic assembly of software and hardware components. A component is defined by an interface (input and output *ports*), a set of sub-components, a set of *connections* linking

[4]Scade is the industrial version of Lustre[7] maintained and distributed by the Esterel-Technology company.

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 type 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 *threads* 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 port (called an *event* port).

Sub-program components are the leaves of this hierarchical description. Their implementations need to be provided in some host language. In our approach, if one wants to be able to formally analyse aperiodic threads which 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, insisting on the aspects that will be used later on to give a formal executable semantics to AADL descriptions.

A synchronous program is a reactive system: it executes a sequence of atomic reactions, periodically or sporadically, according to the way the program is activated. At each reaction, the program reads its inputs, computes its outputs, and updates its internal state. The main feature 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, synchronous paradigm provides an idealized representation of parallelism.

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. For instance, Figure 1.a pictures a Mealy machine with two inputs, x and y, one output z, and one state variable s. A step of this machine can be defined by the functions f_o and f_s, respectively giving the output and the next state from the current inputs and the current state:

$$z = f_o(x, y, s) \quad , \quad s' = f_s(x, y, s)$$

The behavior of that machine is the following: it starts in some initial state s_0. In a given state s, it deterministically reacts to an input valuation (x, y) by returning the output $z = f_o(x, y, s)$ and by updating its state into $s' = f_s(x, y, s)$ for the next reaction.

Such a machine is said to be *combinational* if it does nor use any register (memory).

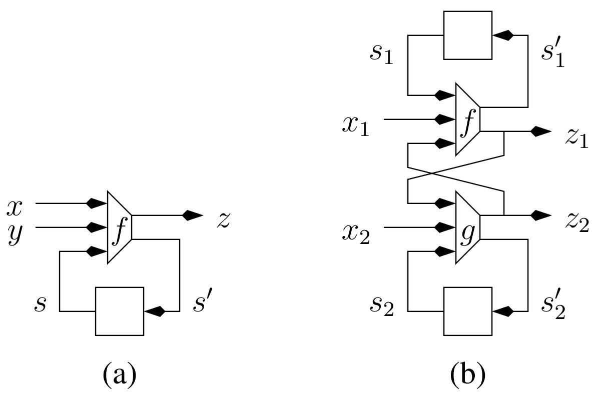


Figure 1: Synchronous machines and their composition

Synchronous composition Synchronous machines (also called *nodes*) are composed in parallel, by connecting one's outputs to the other's input (Figure 1.b). As long as the connections do not introduce loops in the combinational part, the behavior of the composition is straightforward, and defines yet another synchronous machine. In the example of Figure 1.b, we obtain a Mealy machine with two outputs and two state variables whose behavior is defined by:

$$z_1 = f_o(x_1, z_2, s_1)\ ,\ z_2 = g_o(x_2, z_1, s_2)$$
$$s'_1 = f_s(x_1, z_2, s_1)\ ,\ s'_2 = g_s(x_2, z_1, s_2)$$

Those equations have a straightforward solution as long as, either the results of $f_o(x_1, z_2, s_1)$ does not combinationaly depend on z_2, or the results of $g_o(x_2, z_1, s_2)$ does not combinationaly depend on z_1. In this paper, we only use such "correct" composition, free of combinational loops.

The delay machine Let us give two very simple examples of synchronous machines that will be used later. The first one is a single "delay" machine (the "`pre`" operator of Lustre, noted • and in the sequel): this machine receives an input i of some type τ, and returns i delayed by 1 step; it therefore has a state variable s of type τ, and can be defined by:

$$f_o(i, s) = s \quad , \quad f_s(i, s) = i$$

The sampler machine Our second example is a *sampler*, with two inputs: i of type τ and a Boolean b. This sampler returns the value of i when b is true, and its previous output when b is false. It is defined by:

$$f_o(i, s) = f_s(i, s) = \text{if } b \text{ then } i \text{ else } s$$[5]

Activation conditions All synchronous languages provide some extra mechanism to express sporadic activation (e.g., the "suspend" statement of Esterel, or the "clock" mechanism of Lustre and Signal).

In this paper, we use the Activation conditions of Scade: an *activation condition* is a meta-operator that takes a synchronous program P, a Boolean input b, and produces a new program called the *conditional activation of* P *by* b, and noted $P \blacktriangleleft b$. Figure 2 shows the graphical representation $P \blacktriangleleft b$. The behavior of $P \blacktriangleleft b$ is defined as follow: a new state variable z^- is introduced to hold the value of the output; and the transition function f' is defined using the transition function f of the program P as follows:

$$f'_o(x, b, s, z^-) = \begin{cases} z^- & \text{if } b = 0 \\ f_o(x, s) & \text{if } b = 1 \end{cases}$$
$$f'_s(x, b, s, z^-) = \begin{cases} (s, z^-) & \text{if } b = 0 \\ (f_s(x, s), f_o(x, s)) & \text{if } b = 1 \end{cases}$$

[5]In Lustre we would write "`current(`i `when` b`)`"

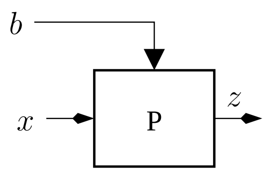


Figure 2: Activation condition

2.3 Modeling asynchrony in the synchronous framework

The ability of the synchronous framework to model asynchrony is well-known [14], and has been often used [1, 2, 6, 5, 12]. In [8], we used a similar technique for translating a subset of AADL into synchronous data flow equations. We briefly recall in this section the principles of this translation. It addresses the following problems:

- Synchronous programs are deterministic, while asynchronous composition introduces temporal and possibly functional non-determinism;
- In the synchronous framework, tasks take no time (or are executed within 1 logical instant, unless explicitly allocated among several instants);
- Asynchronous composition involves non-deterministic interleaving of atomic actions.

2.3.1 Modeling non-determinism

The key idea for modeling asynchrony with synchronous machines is to use additional inputs – often called "oracles" — whenever it is necessary to model the intrinsic non-determinism of asynchronous execution. This way of expressing non-determinism has some advantages over built-in non-deterministic constructs of many specification languages.

- On the one hand, non-determinism is clearly localized and controlled, and it is possible to replay the same execution twice, just by providing the same oracles.
- On the other hand, non-determinism can be restricted by imposing some constraints on oracles. We will make an intensive use of this feature, in particular to express known scheduling constraints.

2.3.2 Time consuming tasks

In order to model time-lasting tasks, we proposed to delay the update of its outputs. Since the exact computation time of a task is generally not known precisely, this delay is generally non-deterministic.

Figure 3 illustrates the modeling principle: the task itself is modeled by the program P, activated by an external clock C_p, the true occurrences of which model the beginning of the execution. When C_p is true, the outputs of P are immediately available. This is why we introduced an extra sampler component β, in order to hold the value. The output will be dispatched to the environment on the next occurrence of the clock c_β. This dispatching clock is provided by a *random delayer*: this program is restarted whenever it receives C_P, then waits until it decides that the computation time is reached, and delivers an output for c_β. Here again, non-determinism is achieved with the help of an oracle Ω_p. For instance, in AADL, it is often the case that the execution time (via the `compute_exec_time` property) is known to belong to some interval $[m, M]$; in this case, the delayer component is programmed in such a way that c_β randomly occurs between m and M global ticks after each occurrence of C_p.

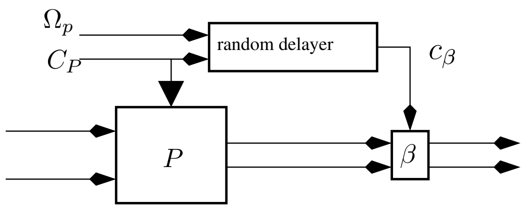


Figure 3: Modeling a time-lasting task

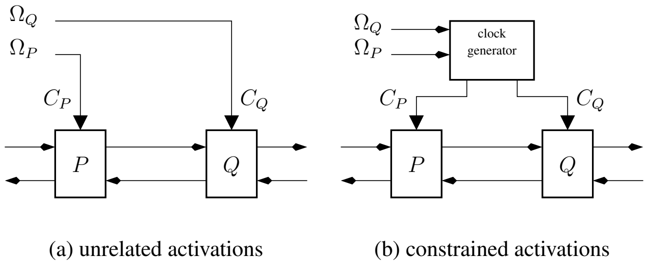


Figure 4: Modeling asynchronous parallel executions

2.3.3 Modeling asynchronous parallel executions

By combining activation conditions and oracle-driven non-determinism, we can express any non-synchronous composition of synchronous processes.

Consider the example of two processor components running *physically* in parallel, and suppose that those processors have already been modeled by two synchronous machines P and Q.

Each processor has its own “activation clock”. Without further information on those clocks, from an external observer point of view, one can state nothing more precise than that the processors are activated by some unrelated activation conditions. This is illustrated on Figure 4.a, where the processors are activated by unrelated and unconstrained oracles Ω_P and Ω_Q. The resulting non-deterministic synchronous model clearly outlines the fact that any interleaving of the executions can be observed, including (but not only) the case where both processors are actually working “at the same time”.

Note that this approach considers instants where none of the clocks are true, i.e., where nothing happens. For model-checkers, such instants cause no problem. But for simulation, such instants are useless; only instants where at least one clock is true are interesting. This can be achieved adding the constraints over the two oracle inputs. This illustrated in Figure 4.b, where a special program takes random oracles, and use them to compute clocks that satisfy the expected property. The equations of this generator could be:

$$C_p = \Omega_P \ ; \quad C_Q = \neg\Omega_P \vee \Omega_Q$$

Quasi-synchronous clocks This principle of random constrained generation can be easily extended to more sophisticated properties. For instance, it is often the case that, even if they are different, the actual clocks of the processors are known to be of (almost) the same rate. This information does not mean that the clocks are synchronous, but it is also clear that some interleavings become unrealistic (e.g., C_p occurs 3 times between 2 occurrences of C_q). This case is called *quasi-synchronous execution* [3], and can be finely modeled by programming a generator that forbids the clocks to differ too much; for instance, the generator may guarantee that each clock occurs at most twice between two occurrences of the other.

Multitasking The case of two concurrent processes sharing the same processor can be modeled in almost the same manner. The difference is that simultaneity is impossible: at each instant at most one process may run. If we have no particular information on the actual scheduling policy, we can safely model the interleaving as shown in Figure 4.b, provided that the clock generator guarantees that C_P et C_Q never occurs at the same time. A possible definition for such a generator is:

$$C_p = \Omega_P \ ; \quad C_Q = \neg\Omega_P \wedge \Omega_Q$$

Of course, more realistic scheduling policies can be specified. In our previous paper [8], we presented a solution for a simple policy with fixed priorities. In this article, we will consider more sophisticated policies that take into account shared resources with protected access, and all the problems they raise: priority inversion, inter-locking, etc.

3 Handling shared resources

In AADL, Data component accesses can be shared between several components. In contrast with 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 to them (read_only, write_only, or read_write), the necessary wires are added to the interface of the node (respectively one input, one output, or one input and one output).

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 often necessary to prevent the resource from being accessed by several components at the same time. For that purpose, several concurrency control protocols were defined [17], that modify the classical Rate Monotonic scheduling. In AADL, this is specified through the "Concurrency_Control_Protocol" property, which is 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. Once it obtains the 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 tl. But the problem with that protocol is that tl can block th without locking any resource. This is referred to as *the priority inversion problem* [17].

- BIP: The Basic Inheritance Protocol is a refinement of the previous one, defined in order to prevent priority inversions.

- PCP: The Priority Ceiling Protocol is a refinement of BIP defined in oder to prevent inter-blocking.

In the following, we describe those protocols more precisely, and explain how to implement them in a synchronous data-flow formalism.

3.1 No Lock

The simplest way of handling shared resources is to ignore them, and to give the CPU to the highest priority thread that asks for it. Even if this (absence of) protocol is straightforward and generally useless, we describe here the part of the scheduler that decides which thread the CPU will be attributed to, since this is the part that will be refined later for the other protocols.

Concretely, we need to generate a synchronous program that takes as inputs the 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 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., $Dispatched_k$).

3.2 Blocking

In order to take into account shared resources, we need additional inputs: the Boolean variable named $Asks\_cs^{t_i}_{r_\ell}$ indicates (via a suitable plug-in into the predefined AADL sub-programs Get_resource and Set_resource [16]) that the thread t_i wants to access the resource r_ℓ.

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

- the Boolean variable $has\_cs^{t_k}_{r_\ell}$ indicates that the thread t_k is in Critical Section on resource r_ℓ;
- the Boolean variable $t_i\_blocks^{t_k}_{r_\ell}$ 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 ($\bullet\ has\_cs^{t_k}_{r_\ell}$);[6]
- 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 $has\_cs^{t_k}_{r_\ell}$:[7]

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

Note that in the generated scheduler, the auxiliary variable $has\_cs^{t_k}_{r_\ell}$ is defined only if the thread t_k may access the resource r_ℓ (i.e., if there exists an access connection between the thread and the resource components in the AADL model). This remark holds for all the variables relating threads and resources.

Computing the blocks relation We say that a thread t_i blocks a thread t_k via a resource r_ℓ if t_k tries to access r_ℓ ($Asks\_cs^{t_k}_{r_\ell}$ is true) while r_ℓ is locked by t_i ($has\_cs^{t_i}_{r_\ell}$ is true).

$$\forall k, i \in [1,n], i \neq k, \forall \ell \in [1,m] :\ t_i\_blocks^{t_k}_{r_\ell} = Asks\_cs^{t_k}_{r_\ell}\ \wedge\ has\_cs^{t_i}_{r_\ell} \tag{3}$$

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

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

Note that those three definitions are cyclic: $t\_blocks^t_r$ depends on $has\_cs^t_r$, which depends on cpu, which depends on $t\_blocks^t_r$. In order to break that combinational loop, one solution is to slightly change the definition of $t\_blocks^t_r$ as follows:

$$\forall k, i \in [1,n], i \neq k, \forall \ell \in [1,m] :\ t_i\_blocks^{t_k}_{r_\ell} = Asks\_cs^{t_k}_{r_\ell} \wedge \mathbf{Asks\_cs^{t_i}_{r_\ell}}\ \wedge\ \bullet\mathbf{has\_cs^{t_i}_{r_\ell}} \tag{5}$$

Indeed, if the thread t_i was locking a resource r_ℓ at the previous instant, and if it still asks for the resource, it should keep the lock on r_ℓ. Therefore both formulations are equivalent (and the latter is loop-free thanks to the delay).

[6] All Boolean delays (●) are initialised to false.

[7] When we define such a relation (and we do it all along this Section), the quantification "$\forall k \in [1,n], \forall \ell \in [1,m]$" suggests that we generate $n \times m$ Lustre equations. But in fact, it is generally much less, since in the AADL description, all the threads may not have access to all the resources.

3.3 The Basic Inheritance Protocol

The Basic Inheritance Protocol was introduced [17] 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 puts a lock on it, the CPU is kept by t_3. Moreover, t_3 can be interrupted by t_2, which has a 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 idea of our (synchronous data-flow) implementation of the BIP is the following: we first consider the highest priority dispatched thread. If it is not blocked, it obtains the CPU. Otherwise, we 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 a thread that is not blocked, we give it the CPU. Hence, the first thing to do is to compute the transitive closure of the $t\_blocks_r^t$ 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 list of threads $\{t_{i=}t_{i0}, ..., t_{is} = t_k\}$ such that there exist 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 distincts. We note $Path(i,k)$ the set of cycle-free paths from t_i to t_k (this set is specified in the AADL source).

$$\forall i,k \in [1,n], i \neq k : \; t_i\_blocks^*_{t_k} = \overline{t_i\_is\_blocked} \;\wedge \bigvee_{p=\{i_0,...,i_s\}\in Path(i,k)} t_{i_0}\_blocks^{t_{i_1}}_{r_1} \;\wedge\; ... \;\wedge\; t_{i_{s-1}}\_blocks^{t_s}_{r_s} \quad (6)$$

where:

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

Note that we have added the $\overline{t_i\_is\_blocked}$ condition because we are only interested in obtaining a thread that is not blocked.

The protocol The BIP states that a thread in critical section on a resource *inherits* from the priority of any other more priority thread that asks for the same resource. The difficulty is to translate this “dynamic” condition (the priority of each thread depends on the history) into a Boolean condition. To do that we use an accumulator, (named ii for inhibiting index), that carries the value of the inhibitor of the thread that has the highest priority (if the highest priority dispatched thread is blocked). 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 $if\ c_1\ then\ x_1\ elseif\ c_2\ then\ x_2\ ...\ if\ c_n\ then\ x_n$
$ii_0 = 0$
$\forall k \in [1,n] : \; (cpu_k, ii_k) =$

$$\begin{aligned}
\overline{dispatched_k} &\rightarrow (False, ii_{k-1}) && (7)\\
(cpu_1 \vee ... \vee cpu_{k-1}) &\rightarrow (False, -1) && (8)\\
ii_{k-1} = k &\rightarrow (True, -1) && (9)\\
ii_{k-1} > 0 &\rightarrow (False, ii_{k-1}) && (10)\\
\{\; t_j\_blocks^*_{t_k} &\rightarrow (False, j)\;\}_{j\in[1,n],j\neq k} && (11)\\
True &\rightarrow (\overline{t_k\_is\_blocked}, 0) && (12)
\end{aligned}$$

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., lines 9 and 12 do not match):

- If t_k is blocked by a lower priority thread t_j (line 11), the inhibiting index takes the priority of the inhibitor, i.e., j. Then, the inhibiting index keeps this value (lines 7 and 10), until the index of the inhibitor is reached (line 9). In that case, the corresponding cpu variable is set to $True$, and the remaining values of cpu are set to $False$ (line 8).
- Otherwise (line 12), if t_k is not blocked at all, it gets the CPU, and all the remaining values of cpu are set to false (line 8). If it is blocked, the system deadlocks.

The inhibiting index is set to -1 when we are certain it will not be used anymore (cf lines 8 and 9).

3.4 Priority ceiling

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.

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. It is statically known from the AADL source; 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 tk_ask relation We first define an auxiliary relation that holds if a thread asks for one of the resources that it did not have at the previous instant (i.e., the thread wants to lock a recourse 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 Priority Ceiling of resources locked by threads other than k PC_k formal definition is just a direct translation of the informal definition given above.

$$\forall k \in [1,n] : \ PC_k = Min \ \{n+1\} \cup \left\{ PC(\ell) \ / \ Asks\_cs^{t_i}_{r_\ell} \quad \wedge \ \bullet \ has\_cs^{t_i}_{r_\ell} \right\}_{\substack{\ell \in [1,m], \\ i \in [1,n], i \neq k}} \tag{13}$$

Note that we use $Asks\_cs^{t_i}_{r_\ell} \wedge \bullet has\_cs^{t_i}_{r_\ell}$ instead of $has\_cs^{t_i}_{r_\ell}$ to avoid combinational loops, as in equation 5.

The protocol The PCP encoding is the same as the BIP one, except that we need to modify the definition of the $blocks$ relation defined in 5. 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 ($PC_k \leq k$), and where 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 ($PC(\ell) = PC_k$).

$$\forall k, i \in [1,n], i \neq k, \forall \ell \in [1,m] : \tag{14}$$
$$t_i\_blocks^{t_k}_{r_\ell} = Asks\_cs^{t_i}_{r_\ell} \ \wedge \ \bullet has\_cs^{t_i}_{r_\ell} \ \wedge \ (Asks\_cs^{t_k}_{r_\ell} \ \vee \ (asks\_cs^{t_k} \quad \wedge PC(\ell) = PC_k \leq k))$$

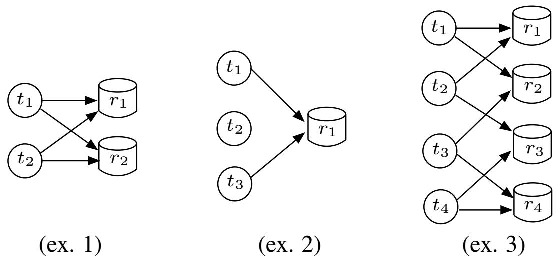


Figure 5: Examples

	Lock	BIP	PCP
ex. 1	ko, 5 steps	ko, 5 steps	ok
ex. 2	ok	ok	ok
ex. 3	ko, 9 steps	ok	ok

Figure 6: Experiments with the deadlock prop.

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[8]. The resulting Lustre program is a task scheduler, computing one Boolean variable (cpu_i) per thread, from Boolean inputs indicating which threads ask for the CPU, and which threads ask for which resource.

In the following, we illustrate the use of a model-checker to check various properties against 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.

4.1 Absence of deadlock

In order to prove the absence of deadlock, we just have to ask a model-checker [9] to prove that, whenever at least one thread asks for the CPU, at least one (in fact, exactly one) of the cpu_i is true.

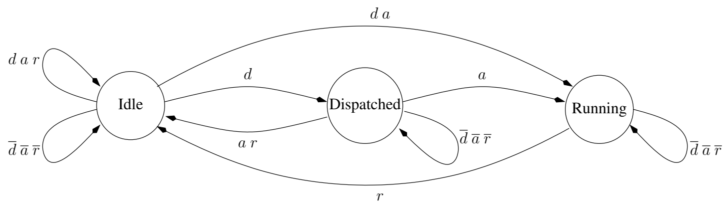
We performed this on several examples pictured in Figure 5. For instance, the first example (ex. 1) of Figure 5 consists in a system with two threads $t1$ and $t2$, that can access two resources $r1$ and $r2$. This example is precisely the one given in [17] to illustrate the fact that the BIP does not prevent deadlock, 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 [17] (and also in Section 3.4). Lesar also proved the absence of deadlock for the PCP. All the results are shown in Fig. 6. For the cases where the property is false, we indicate the number of steps of the counter-example provided by Lesar.

An interesting point in those experiments is that it is not always worth using the PCP (that is deadlock-free by construction) as 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 model-checker that the inputs of the scheduler are not completely random. For instance, a thread cannot change its requests for resources when it does not own the CPU.

[8]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/

	Lock	BIP	PCP
ex. 1	ok	ok	ok
ex. 2	ko, 4 steps	ok	ok*
ex. 3	ko, 4 steps	ok*[9]	ok

Figure 7: Experiments with the non-inversion prop.



$d\ a$
$d\ a\ r$
Idle
d
Dispatched
a
Running
$\overline{d}\,\overline{a}\,\overline{r}$
$a\ r$
$\overline{d}\,\overline{a}\,\overline{r}$
$\overline{d}\,\overline{a}\,\overline{r}$
r

Figure 8: The automaton recognizing well-scheduled systems. There is one such automaton per thread to schedule.

4.2 Priority inversion

The priority inversion corresponds to situations when a thread is blocked by a less priority thread. This occurs very naturally when two threads shares the same resource, locked by the less priority thread. Priority inversion is more problematic when it happens as in the example of Section 3.3 (which was the example given in [17] 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 less priority thread in critical section, the corresponding resource might be locked a long time.

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

As summarized in Figure 7, Lesar found counter-examples that falsify the non-inversion property for the last two examples of Figure 5. Note that the second example is the one given [17] (and also in Section 3.3), for motivating the introduction of the BIP. Here again, one can remark that in some configurations, it is not always worth the complexity of the BIP or the PCP to ensure the non-inversion.

4.3 Schedulability

The scheduler we generate in Section 3 take as input Boolean values ($Dispatched_1$, ..., $Dispatched_n$) indicating which threads ask for the CPU, and computes the thread the CPU is attributed to. But this scheduler is just a part of our AADL to Lustre translator; in particular, the values of the $Dispatched_i$ variables are also defined by this translator, using the thread WCET and period (or just the output of some software component for sporadic threads), as explained in [8], and recalled in Section 2.3.

In order to check the schedulability of the AADL program, 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 Figure 8. 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.

In other words, nothing prevents the resulting Lustre scheduler to generate a "dispatch" event between an "activate" and a "release" event. This is what typically 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 model-checking) that the system is schedulable.

The Cheddar tool [10, 18] 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 [19].

5 Conclusion

We have defined an automated translation of AADL models into a purely Boolean synchronous (Lustre) scheduler, that can be directly model-checked. The advantages are manyfold.

- Firstly, it 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.
- And finally, in presence of shared resources, the analytic schedulability criteria are very conservative, and may reject schedulable systems. Moreover, as soon as the system contains sporadic events (i.e., when the thread activation depends on the output of some other thread), the analytic method can be meaningless.

Of course, since the verification is automatic, we were not able to deal with generic mechanisms (e.g., to prove properties of resource management protocols whatever be the number of tasks and resources), but since the generation of models for verification is automatic, the verification can be played again for each instance of a generic mechanism.

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 [11]. 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 constraint the random input generator [15].

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 to turn 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.

References

[1] P. Baufreton. SACRES: A step ahead in the development of critical avionics applications. LNCS 1569, Springer-Verlag, 1999. 2.3

[2] P. Baufreton. Visual notations based on synchronous languages for dynamic validation of gals systems. In *CCCT'04 Computing, Communications and Control Technologies*, Austin (Texas), August 2004. 2.3

[3] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems, the quasi-synchronous approach. In *SAFECOMP'01*. LNCS 2187, 2001. 2.3.3

[4] P. H. Feiler, D. P. Gluch, J. J. Hudak, and B. A. Lewis. Embedded system architecture analysis using SAE AADL. Technical note cmu/sei-2004-tn-005, Carnegie Mellon University, 2004. 1, 2.1

[5] A. Gamatié and T. Gautier. The signal approach to the design of system architectures. In *10th IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2003)*, pages 80–88, Huntsville (Alabama), April 2003. 2.3

[6] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the signal language. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'2003)*, pages 144–151, Toronto, May 2003. 2.3

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 4

[8] N. Halbwachs, E. Jahier, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *Seventh International Conference on Embedded Software (EMSOFT 2007)*, Salzburg, Austria, October 2007. (document), 1, 2, 2.3, 2.3.3, 4.3

[9] N. Halbwachs, F. Lagnier, and C. Ratel. 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*, pages 785–793, September 1992. 4.1

[10] J. Hugues, B. Zalila, and L. Pautet F. Kordon. 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. 4.3

[11] E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. *International Journal on Software Tools for Technology Transfer (STTT)*, Special Section on Leveraging Applications of Formal Methods, 2006. 5

[12] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers, Special Issue on Application Specific Hardware Design*, April 2003. 2.3

[13] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973. 1

[14] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981. 2.3

[15] P. Raymond, E. Jahier, and Y. Roux. Describing and executing random reactive systems. In *SEFM*, pages 216–225. IEEE Computer Society, 2006. 5

[16] SAE. Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace, November 2004. 1, 2.1, 3.2

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. 3, 3.3, 4.1, 4.2

[18] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In J. W. McCormick and R. E. Sward, editors, *SIGAda*, pages 1–8. ACM, 2004. 4.3

[19] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. In *SIGAda*, 2005. 4.3