HAL
open science

Virtual Execution of AADL Models via a Translation into Synchronous Programs

Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, David Lesens

To cite this version:

Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, David Lesens. Virtual Execution of AADL Models via a Translation into Synchronous Programs. EMSOFT 2007, Oct 2007, Salzburg, Austria. pp.134 - 143, 10.1145/1289927.1289951 . hal-00189563

HAL Id: hal-00189563

https://hal.science/hal-00189563

Submitted on 21 Nov 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Virtual Execution of AADL Models via a Translation into Synchronous Programs *

Erwan Jahier, Nicolas Halbwachs
Pascal Raymond, Xavier Nicollin
CNRS - Verimag, Grenoble, France
first_name.last_name@imag.fr

David Lesens
Astrium Space Transportation
Les Mureaux, France
david.lesens@astrium.eads.net

ABSTRACT

Architecture description languages are used to describe both the hardware and software architecture of an application, at system-level. The basic software components are intended to be developed independently, and then deployed on the described architecture. This separate development of the architecture and of the software raises the problem of early validation of the integrated system.

In this paper, we propose to solve this problem by translating the architecture into an executable model, which can be simulated and validated together with the software components. More specifically, we consider the case where the architecture is described in the AADL language, and the software components are developed in some synchronous language like Scade or Lustre. We show how the architecture can be automatically translated into a non-deterministic synchronous model, to which the actual software component can be integrated. The result is an executable integrated synchronous model, which can be validated with tools available for synchronous programs. The approach is illustrated on an industrial case study extracted from an actual spatial system.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, Languages, Verification

Keywords

Simulation, Formal Verification, Architecture Description Language, Synchronous Languages

*This work was partially supported by the European Commission under the Integrated Project Assert, IST 004033

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-825-1/07/0009 ...$5.00.

1. INTRODUCTION

Synchronous languages [15, 11, 3] are well-suited to program embedded software, as long as centralized, sequential code is targeted. The use of synchronous designs is for instance used by Astrium Space Transportation for the most critical space software. However, complex systems often need to be implemented on more complex architectures — e.g., multi-processors —, and executed according to more complex execution schemes — e.g., involving multitasking which generally limit the use of pure synchronous designs to the simplest or the most critical software. Methods for automatic translation of synchronous languages for those more complex targets have been proposed [6, 5, 24].

More classical approaches consist in programming separately software components using synchronous languages, and deploying them on the target architecture using classical design methods for asynchronous systems. In this case, the problem of validating the whole system is crucial: as a matter of fact, the execution of the software on the target architecture is generally asynchronous, maybe non-deterministic, and as a consequence, this phase of the design is the most error-prone.

Of course, the validation can be performed by testing the implementation. However, this will result in late error detection. Moreover, testing an asynchronous implementation is difficult: non-determinism can make the tests non-reproducible, the test coverage is difficult to define and to measure, and the observation of the system under test can modify its behavior. This is why it is useful to validate such a system on an early, realistic model.

The ASSERT project. ASSERT is a European project devoted to the safe model-driven design of embedded systems, with aerospace systems as main application domain. In the ASSERT process, several approaches — including synchronous programming with Scade — are possible to develop software components, and the target architecture is described in the AADL [8, 23] architecture description language. In the proposed methodology, models of the system under design should exist all along the design process, for validation purposes. As a consequence, one must be able to model software components deployed on an architecture described in AADL. When software components are described in a synchronous language, it is natural to model the whole system in the synchronous framework: this framework is well-known to be able to describe asynchrony as well [20], and synchronous modeling of asynchronous behaviors has

been proposed several times [1, 2, 10, 9, 18, 12, 14].

From AADL to synchronous programs. This is why, in the present paper, we propose a tool that takes as inputs (1) an architecture described in AADL, and (2) an implementation of software component in Scade or Lustre; the tool generates an executable model of the behavior of the software deployed on the architecture. Such a model is usable for early simulation, but also for formal verification, using tools available for Scade and Lustre.

Outline. The paper is organized as follows. The first two sections are partly borrowed from [14]. Section 2 recalls the bases of synchronous paradigm, and Section 3 explains how it can be used to describe asynchronous behaviors. The considered fragment of AADL is presented in Section 4, and Section 5 describes how it can be translated into a synchronous program. Section 6 presents a case study, adapted from a real aerospace system. Section 7 presents how the resulting synchronous program can be simulated and formally verified. Section 8 briefly presents the freely available prototype.

2. THE SYNCHRONOUS PARADIGM

In this section, we recall the only necessary features about synchronous programming which are of interest for our modeling activity. Externally, a synchronous program behaves as a sequence of atomic steps, which can be periodic or sporadic, according to the way the program is activated. To perform one step of the program, the environment has to provide its current inputs; the step itself consists in computing the current outputs, as a function of the current inputs and the current internal state of the program (which generally has remanent variables encoding this state), and in updating the state for the next step.

The specific feature of synchronous programs is the way internal components behave with each other: when several components are composed in synchronous parallelism, one step of the whole composition consists of a "simultaneous" step of all the components, which can communicate with each other during the execution of the step. This execution is guaranteed to be *deterministic*, a very important property of synchronous programs, since it makes much easier the understanding of programs, as well as their testing and verification.

To be more precise, and following the presentation of [12], a synchronous component is a straightforward generalization of synchronous circuits (Mealy machines) to work with arbitrary data-types: such a machine has a memory (a state), and a combinational part, computing the output and the next state as a function of the current input and the current state. For instance, Figure 1.a pictures a machine with two inputs, x and y, one output z, and one state variable s. One can define a step of the machine by the functions, say 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 the 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.

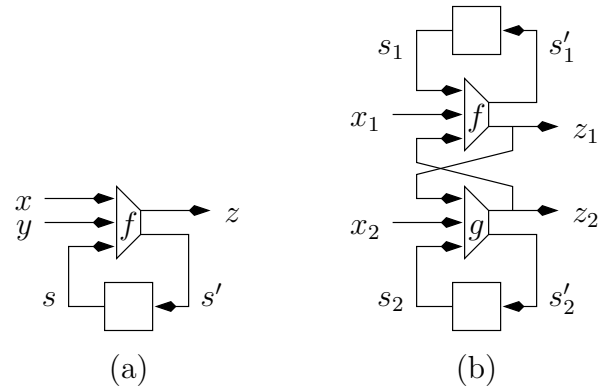

Figure 1: Synchronous machines and their composition

Those machines can be composed in parallel, with possible "plugging" of one's outputs into the other's inputs (Figure 1.b), as long as those wirings don't introduce any combinational loop. Such a composition is shown by Figure 1.b, where the variables are defined by

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

and where either the results of $f_o(x_1, z_2, s_1)$ should not depend on z_2, or the results of $g_o(x_2, z_1, s_2)$ should not depend on z_1, to avoid combinational loops.

To conclude this section, let us give two very simple examples of synchronous machines that we will use later. The first one is a single "delay" machine (the "`pre`" operator of Lustre): the machine δ receives an input i of some type τ, and returns its input delayed by 1 step; it has a state variable s of type τ and is defined by

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

Our second example is a *sampler* $\beta(b)$, with an input i of type τ and a Boolean input b, which returns the value of i when b is true, or its previous output when b is false (it would be written "`current(`i `when` b`)`" in Lustre). It is defined by

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

3. EXPRESSING ASYNCHRONY IN THE SYNCHRONOUS FRAMEWORK

Basically, the difference between synchrony and asynchrony is that, in the synchronous model, each significant "event" is precisely dated with respect to other events and with respect to the sequence of steps. As soon as synchrony is released, the date of some events becomes unknown, or not precisely known, meaning that the temporal behavior becomes non-deterministic.

According to our presentation of the synchronous paradigm, to express asynchrony, we need, on one hand, that components don't necessarily participate in all steps, and on the other hand, to express non-determinism (which is on purpose forbidden in synchronous languages!).

3.1 Sporadic activation

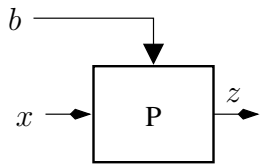

Figure 2: Activation condition

All synchronous languages propose some ways of preventing a component from reacting (the "suspend" statement of Esterel, or the "clock" mechanism of Lustre and Signal). However, we don't want to bother about "absent" values or signals, induced by those notions. In Scade, the notion of "activation condition" allows a node to be activated sporadically, its output and state keeping their previous values when the activation condition is false; therefore, an additional memory is implicitly requested to record the previous output. More precisely, if P is a synchronous machine, with input i, output o, and state s, defined by the functions f_o and f_s, the *conditional activation* of P by b, noted $P \blacktriangleleft b$, takes a new Boolean input b, its state is a pair (s, o^-), and its output and state functions f'_o and f'_s are as follows:

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

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

Graphically, we shall note $P \blacktriangleleft b$ with the activation condition as a black arrow input on top of the machine it controls, as in Figure 2.

3.2 Non-determinism

We will classically model non-determinism by means of additional inputs — often called "oracles" — to the model. Those oracles will be used to control non-deterministic choices. This way of expressing non-determinism has some advantages over built-in non-deterministic constructs of many specification languages:

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

3.3 General principles

By combining sporadic activation and oracle-driven non-determinism, we can express any non-synchronous composition of synchronous processes. The general construction is the following (see Figure 3): the processes are all sporadically activated according to activation conditions emitted by a global scheduler. The scheduler is non-deterministic: it receives one Boolean oracle for each condition it has to elaborate. But it can restrict this complete non-determinism by enforcing constraints among the conditions it actually emits towards the processes.

For instance, in the example of Figure 3, the scheduler could prevent the processes from being activated simultaneously, by setting

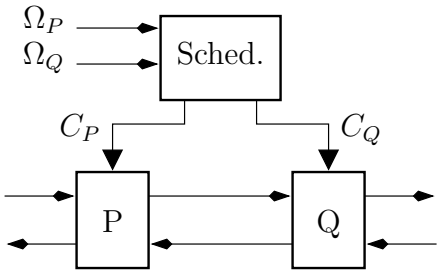

Figure 3: Modeling non-synchronous executions

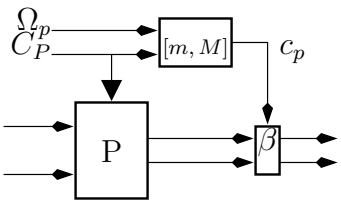

Figure 4: Modeling a non-instantaneous task

$$C_P = \Omega_P \ , \ C_Q = \Omega_Q \wedge \neg C_P$$

This would simulate two asynchronous processes, communicating through a shared memory: since, when P is not activated, its outputs keep their last values, then Q gets those last written values just as it would read them in a shared memory.

More complex communication mechanisms can be modeled as communication processes, which may need their own activation conditions: for instance, we will have to model tasks taking more than one synchronous step. In general, minimum m and maximum M execution times are known for such a task. In our synchronous setting, we will just express that the outputs of the task are available at least m steps, and at most M steps after its activation, assuming time is counted as number of steps. This is done using the sampler defined at the end of Section 2, triggered by a condition (c_p in Figure 4) non deterministically true once in the interval $[m, M]$. That can be done by a straightforward operator taking an additional oracle as input (Ω_p in Figure 4).

The modeling of a new composition mechanism then consists in expressing constraints on activation conditions, to be inserted in the scheduler. Since this task is quite difficult and error-prone, it is of course better to get the model by translation from a higher-level language. This is what we shall do from a significant fragment of AADL.

4. THE AADL LANGUAGE AND THE CONSIDERED SUBSET

4.1 Generalities

AADL components are made of a so-called *component type*, and each type is optionally associated with one (or several) *component implementation(s)*. A component type is made of a functional interface plus various typed attributes

that are called *properties*. An implementation inherits interface and properties from its corresponding component type. It also declares the sub-components it is made of, as well as the *connections* between them. Connections are made through the sub-components input and output *ports*. Ports can convey data, events (control), or both.

An AADL model is made of a hierarchic assembly of software and hardware component types and implementations. The top-level component is a *system*. A system is made of several *devices* and *processors*; each processor can run several *processes*; each process can run several *threads*; and each thread can run several *sub-programs*. Leaves of an AADL model are therefore either subprograms, or component types.

4.2 The supported AADL subset

Here, we describe the fragment of AADL taken into account by our translation. We also briefly explain how the supported features will be translated in the synchronous model. More details are given in the next section.

Since our goal is to model behaviors, most aspects concerning hardware and non-functional properties will be ignored.

4.2.1 System

A system is the top-level component of the AADL model, mixing hardware and software components. A system is translated into a synchronous machine which abstracts away the bindings between software and hardware components, and only reflects the functional behavior of the whole system.

4.2.2 Execution platform components

Devices. Device components are used to interface the AADL model with its environment. Therefore, devices are not translated as the other components: their inputs are considered as system outputs, and their outputs as system inputs. For simulation and verification purposes, behavioral models of devices can be provided by the user.

Processors. Processor components are an abstraction of hardware and software responsible for executing and scheduling processes. Basically, each processor will have its own physical clock, which is the base time of the components running on the processor. The `Clock_period` property, that declares the processor internal clock rate, is used in our translation to model the relations between the processors clocks (cf Section 5.1).

Memory. Memory components are used to specify the amount and the kind of memory that is available to other components. We assume that enough memory is available and thus ignore everything that is related to such components.

Buses. Bus components are used to exchange data between hardware components. Detailed models of specific buses can be provided (see, e.g., [10]). In our prototype tool, we just consider buses as usual connections.

4.2.3 Software components

Processes. Process components are an abstraction of software responsible for scheduling and for executing threads. Processes are scheduled the same way as periodic threads (see below), the main difference being that threads (executed by a process) can share a common memory whereas processes (executed by a processor) cannot.

Threads. Thread components are an abstraction of software responsible for scheduling and for executing subprograms. When several threads run under the same process, the sharing of the process is managed by a runtime scheduler. The `dispatch_protocol` property is used to specify the activation of a thread:

- `periodic` means that the thread must be activated according to the specified `period`;
- `aperiodic` means that the thread is activated via one of the other components output port, called an *event* port;
- a `sporadic` thread is a mixture between `aperiodic` and `periodic`: it can be activated either by events, or periodically;
- `background` threads are always active, but have the lowest priority.

The property `compute_exec_time` specifies a minimum and maximum execution time for the thread to execute its task. For thread implementations, execution times result from the `compute_exec_time` associated to their subprograms. Those bounds on execution time will be used to define the availability of outputs (cf. Figure 4).

Sub-programs. A thread can be made of a sequence of sub-program components, that represent elementary pieces of code that processes inputs to produce outputs. Only their interfaces are given in the AADL model; sub-program implementations ought to be provided in some host language. For our purpose, we require sub-programs to be given in a synchronous language (Scade or Lustre). Moreover, sub-programs must be provided with a `compute_exec_time` property in order to simulate accurately the time their computations take.

Data. Data components are not associated to synchronous machines, but to data types. The data type mapping is not handled by the tool, except for the base types `bool`, `int`, and `floats`; this mapping must be done by users or a third-party tool (such a tool is developed within the ASSERT project).

4.2.4 Other concepts

AADL defines a concept of *operational mode*, that is ignored in the current version of the tool. A concept of *Flow* is also introduced to allow users to declare the existence of logical flows of information between a sequence of components. Flows are used to perform various non-functional analysis. Therefore they are ignored in our translation too.

5. TRANSLATING AADL

System, process, thread, and subprogram components will be translated into synchronous machines which inputs/outputs are made of the component input/output ports. For component implementations, the synchronous machines wirings result from the component inner connections. The synchronous machines will also have as additional inputs:

- an activation condition;
- a termination condition (for delaying outputs);
- the activation and termination conditions of their submachines.

5.1 Processors and physical parallelism

As said before, each processor has its own clock. Therefore, a processor is translated into a machine with an activation condition `ck` corresponding to its clock. This clock will define the local notion of time of the subcomponents — processes, threads and subprograms — running on the processor. For instance, thread periods will be counted with respect to that clock.

One can consider those processors clocks to be completely unrelated, meaning that processors are completely asynchronous. This solution leads to a maximal non-determinism, but can be considered as unrealistic. In general, in the design of real-time distributed applications, some assumptions are made about the relative speeds of the processors. We illustrate below how such an assumption may be expressed in our modeling, and we consider a very common case, called *quasi-synchrony*.

Very often, people are using several processors of the same kind, and thus, running with internal clocks with the same period. Having the same period introduces strong relationship between the clocks, that must be taken into account in order to produce realistic simulations.

Simulating same-period clocks requires to take into account time drift: perfect distributed clocks are difficult to obtain (e.g., [17]), thus a period of T generally means $T \pm \varepsilon$, with $\varepsilon \ll T$. As a consequence some (unbounded) time drift may occur when observing two clocks with the "same" period. Modeling realistic time drift is rather complex, and a common solution is to over-approximate it with simple, local relations between the occurrences of the clocks. An example of such a relation, known as *quasi-synchrony*, was formalized by [5]. It formalizes that several clocks "are almost the same".

Quasi-synchronous composition. A set of clocks is said to be *quasi-synchronous* if, between two successive activations of any clock, each other clock is activated at most twice.

Such an interleaving is illustrated in Figure 5. The figure pictures the timing diagram of two quasi-synchronous clocks `ck1` and `ck2`. The "relative advance" of `ck1` over `ck2` (`a1`) and of `ck2` over `ck1` (`a2`) are represented for clarifying the principle: whenever `c1` (resp. `c2`) is true, `a2` (resp. `a1`) is reset; meanwhile, if `c2` (resp. `c1`) is false, `a1` (resp. `a2`) is incremented. Since, in the example, `a1` and `a2` are both bounded by 2, the clocks are (so far) quasi-synchronous.

The important property of quasi-synchronous composition of processes is that each process is guaranteed to miss at most one sample of the other's output in a row. Hence,

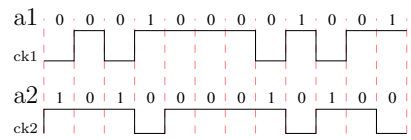

Figure 5: Quasi-synchronous clocks

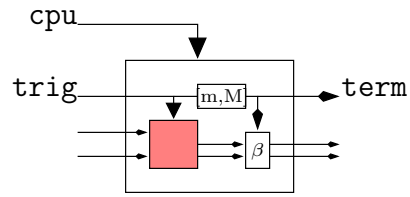

Figure 6: Model of a software component

programs that are designed to be robust to clock drift (if they read programs output in registers for example) have the same behavior with quasi-synchronous or synchronous clocks.

In [14], we gave a non-deterministic synchronous machine producing a pair of quasi-synchronous clocks. It can be easily generalized to several clocks. Our translation tool uses such generators for simulating processor clocks. Such a simulation allows, in particular, AADL systems robustness to be checked.

Quasy-synchronous extensions. For the time being, only quasi-synchronous clocks are handled. However, the principle can be easily extended to clocks with more complex relationships. For instance, if the period of a clock c is known to be twice the period of a clock c', then clock drift can be abstracted by stating that c is activated at most once between two activations of c', and c' at most three times between two activations of c.

To conclude this section, let us emphasize the fact that quasi-synchrony is somehow a coarse approximation of the relations between same-period clocks. For instance, quasi-synchrony allows one clock to appear twice more often than the other. Therefore, the observed periods may actually be some T for the fast one, and $2T$ for the slow one. This may be a problem when the processor clocks are used to count real-time and to time-stamp external events: the time-stamp of the same external event will be $t_0 + d$ on one processor, and $t_0 + 2d$ on the other. For systems where real-time delays matter, a more accurate modeling of clock relations is necessary.

5.2 Threads and concurrency

In addition to its inputs and outputs specified in its AADL interface, each software component translation — process, thread or subprogram — must be provided with a triggering condition `trig`, which determines when the component must be started, and an activity condition `cpu`, which expresses that the component owns the processor and can run. It returns a Boolean output `term` indicating when it terminates (Figure 6).

It is easy to define a generic synchronous machine generating a periodic condition, which is true every T steps (cf

Appendix B). Note that those machines are deterministic. They will be used to define the `trig` conditions of periodic processes and threads.

Several processes or threads that run on the same processor have to share the CPU. This sharing is managed by a runtime scheduler, which has to be modeled. In the current version of our tool, a simple rate-monotonic scheduler [19] is assumed. The scheduler computes an activity condition `cpu` for each process and thread of the processor (cf. Appendix B). This activity condition is used as a local clock to count the component execution time, since a component executes only when it owns the processor. To compute activity conditions, the scheduler must know whether a component needs the processor. This condition, `needs`, is computed by a two states synchronous machine set by the `trig` condition of the component, and reset by its `term` condition.

Figure 7 shows a non-deterministic transition system representing the behavior of a scheduled thread. Its internal state toggles from `needs=0` to `needs=1` when the input `trig` occurs. At the same instant, an internal counter x is reset. Once in this active state, the internal counter x is incremented whenever the input `cpu` occurs. The thread toggles back to `needs=0`, non-deterministically, provided that the counter x is within the computation bounds $[m, M]$. This transition also outputs the `term` signal.

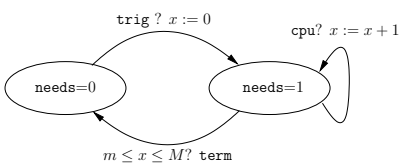

Figure 7: The possible thread states

5.3 Subprograms and sequentiality

Finally, when a thread is made of several subprograms, they must be activated in sequence. Subprograms inherit the activity condition of their parent thread. The first one is triggered by the triggering condition of the parent thread, while each other subprograms is triggered by the termination condition of the previous one in the sequence (Figure 8).

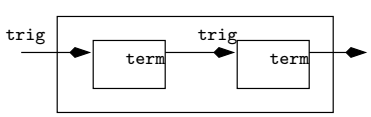

Figure 8: Sequencing subprograms

6. THE PFS CASE STUDY

The PFS ("Proximity Flight Safety") case study was proposed by Astrium Space Transportation, and concerns an equipment of the "Automated Transfer Vehicle" (ATV) in charge of supplying the International Space Station (ISS). The role of the PFS is to ensure the safety of the approach of the ATV to the ISS: when anything goes wrong, the PFS is in charge of performing a "collision avoidance maneuver" (CAM), i.e., to safely move the ATV apart from the ISS, and to orient it towards the sun, waiting for new instructions.

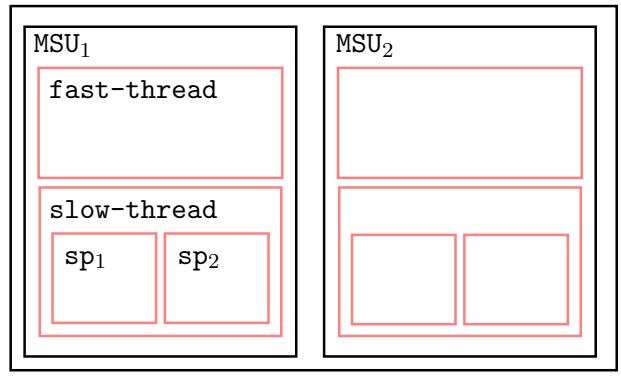

Figure 9: The PFS architecture

Here, we don't consider the actual system, but a version which has been complexified to illustrate more aspects of the AADL translation.

6.1 The PFS functions and architecture

The CAM is performed by two redundant units, called "Monitoring and Safing Units" (MSU), running on two computers. At each instant, one of them is the master, but can resign this role if it detects its own failure, in which case the other MSU becomes the master. However, the master may not change when a CAM is in progress. Once an MSU has given up its mastership, it never recovers.
Each MSU:

- detects anomalies, which can be failures of the main computer, abnormal state of the bus, erroneous position or speed of the ATV, "red button" pressed from inside the ISS;
- detects its own failures, which can involve a master change;
- is able to perform a CAM.

The whole system is supposed to tolerate at most three faults: the two ones which raise a CAM, and a fault of one MSU. As a consequence, one doesn't have to consider the case of two faulty MSUs.

In the version of the PFS used for this paper, each MSU is made of three tasks to be run periodically.

- One high priority task that lasts between 30 and 40 ms, and that should be activated every 100 ms. This task is modeled by an AADL thread named `fast-thread`.
- Two low priority periodic tasks, of period 500 ms: one that lasts between 1 and 100 ms, and one that last between 1 and 150 ms. Since those two tasks need to be executed in sequence, each one is modeled as an AADL subprogram. Those subprograms, sp_1 and sp_2, are themselves contained in a thread named `slow-thread` of period 500 ms.

Figure 9 pictures the PFS components architecture. Note that since there is only one process per processor, no distinction is made between processors and processes.

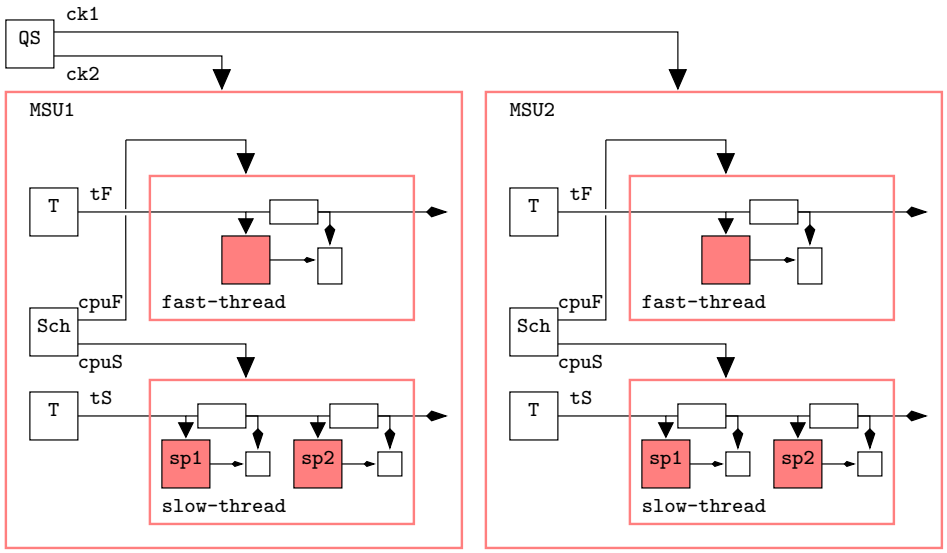

Figure 10: The synchronous model of the PFS

6.2 The synchronous model of the PFS

The translation of the AADL architecture produces a synchronous model pictured in Figure 10.

A generator of quasi-synchronous clocks, noted `QS`, produces the clocks `ck1` and `ck2` of the two processors. Inside each processor model, generators of periodic clocks are used to trigger the models of the fast thread (`tF`, period 100 ms) and of the first subprogram of the slow thread (`tS`,period 500 ms).

A model of the runtime scheduler computes the activation conditions `cpuF` and `cpuS` of the threads, corresponding to their allocation of the processor, and giving priority to the fast thread. Inside the thread models, each software component is provided with a delay operator, delaying its outputs according to the bounds on its execution time, those delays being counted in terms of ticks of the thread activation condition.

Finally, the second subprogram of the slow thread is triggered by the termination of the first subprogram. Notice that in Figure 10, the oracles needed by non deterministic operators (`QS`, `Sch`, and all the delay elements) are omitted for simplicity. We did not represent either the computation of the "`need`" condition of each thread, which should be fed back to the scheduler `Sch`.

6.3 A simulation of the executable model

To illustrate better the PFS model, let us detail the behavior of its control variables along some steps of simulation.

Figure 11 pictures 12 cycles of the MSU_1 scheduling (The behavior of MSU_2 being similar). Each cycle of this simulation is assumed to take 10 ms.

The clock `ck1` is the processor clock, generated to be quasi-synchronous with the clock of the other processor. Hence, that sequence of 12 synchronous cycles actually corresponds to a simulation of 80 ms of the processor life, since `ck1` is true 8 times.

The condition `needsF` (resp. `cpuF`) expresses that the fast thread needs (resp. has) the processor. `needsF` is computed

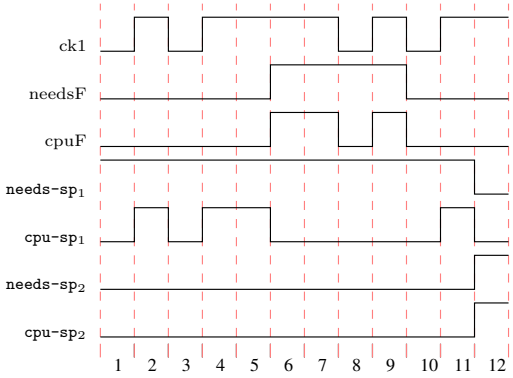

Figure 11: A timing diagram illustrating the scheduling of the MSU_1 sub-components

according to the automata shown in Figure 7. Its raising edge corresponds to a trigger provided by a deterministic counter (`tF` in Figure 10). This trigger occurs each 100 ms (fast thread period), i.e., once every 10 ticks of `ck1`. Since `fast-thread` has the highest priority, it gets the CPU as soon as it needs it, thus, the control variable `cpuF` is simply the conjunction of `needsF` and `ck1`. `cpuF` is used as an activation condition for the synchronous program executed by the `fast-thread`. From the synchronous program point of view, the outputs are computed instantaneously; but as explained in Section 5, the output values of this task are delayed in order to simulate the execution time of the task, which can be 3 or 4 cycles (3 in the example of Figure 11) where `cpuF` is true. The falling edge of `cpuF` corresponds to the `term` event.

Conditions `needs-sp`$_1$ and `cpu-sp`$_1$ play a similar role to activate the `sp`$_1$ task, and release its output. Note that at instant 6, `fast-thread` preempts the CPU to `sp`$_1$, and gives it back at instant 11. At instant 12, `needs-sp`$_1$ be-

come false, because cpu-sp$_1$ has been true 4 times in this sequence, which is correct since the AADL model specifies its execution can last between 1 and 10 cycles. At instant 12, since sp$_1$ has finished, sp$_2$ needs the CPU. It gets it immediately, since the CPU is not required by more priority tasks. Therefore, both needs-sp$_2$ and cpu-sp$_2$ become true.

7. USING THE EXECUTABLE MODEL

Given a description of the architecture in AADL and synchronous program for software components, we are able to obtain automatically a model of the whole system in Scade/Lustre. This model enjoys all the advantages of a synchronous model: it is executable, its properties can be expressed by means of synchronous observers, and it can be validated using existing tools. Concerning our case study, the obtained model was used for simulation and formal verification.

Simulation. The testing tool Lurette [16, 22] was used to extensively simulate the model. Lurette allows the environment to be specified, and uses this specification to generate realistic input sequences, which are provided to the model. Moreover, the simulation can be monitored by writing synchronous observers [13] of the (safety) properties of interest. The tool can then drive, in a completely automatic way, an arbitrary number of arbitrary long simulations, restricted to realistic input scenarios, and while checking that the specified properties are satisfied.

Formal verification. We also used the model-checker Lesar [21] to verify properties on the model of the PFS. For instance, we considered the property that, in presence of at most two faults, at each instant, one and only one MSU is the master. This property was first successfully proved on a purely synchronous model. On the generated model, it was disproved: as a matter of fact, because of the non-synchronous execution of the two processors, the property cannot be ensured: when the master resigns its role, the other MSU cannot take the mastership instantly. However, because of the quasi-synchrony assumption, it was possible to prove the actual, weaker property: when MSU1 resigns its role of master, then within at most two steps of the clock ck1, MSU2 takes the mastership.

8. THE PROTOTYPE

The ideas presented in the article have been implemented into a prototype named aadl2sync. This prototype is freely available on the tool page of the Synchron's group web site[1]. It has been developed under the scope of the ASSERT IST project.

aadl2sync can generate Lustre or Scade code. Given a set of textual AADL files[2] containing a top level system foo, it generates 5 files:

- foo_scheduler.lus that contains the scheduler driving all the additional inputs (quasi-synchronous clocks, activate condition, etc.) introduced during the translation.
- foo_components.lus that contains the translation in Lustre of all AADL components present in the model.
- fillme_foo_nodes.lus that contains the interface of nodes corresponding to leaves in the AADL model. The bodies of such nodes — in Lustre or Scade — must be provided by the user.
- fillme_foo_const.lus that contains various constants that needs to be defined, such as the components initial values, i.e., the values they ought to output at the first cycle (values that are used to provide default values in activation conditions).
- fillme_foo_types.lus that contains the Lustre type definitions of Data type components. This translation is not automated by **aadl2sync**, except for base types.

Those last 3 files, which are prefixed by fillme_, need to be filled in and renamed without the prefix. More details on the produced Lustre files, the supported AADL subset, etc., are provided in the aadl2sync Reference Manual [7].

Launched on the 4 AADL files containing the PFS case study, that are made of 1500 lines of code, aadl2sync produces 500 lines of Lustre for the scheduler, plus 1800 lines of Lustre code for the 4 other files. The main node has 10 inputs and 10 outputs, some of them being arrays. If we count the input/output number with arrays expanded, we have 48 inputs and 119 outputs. Therefore, one important and difficult task that have to be done by the user is to describe the set of possible inputs, in particular in order to initialize correctly the PFS.

9. CONCLUSION

In this paper, we presented an effective use of the synchronous modeling of asynchronous systems, to produce automatically a usable model of synchronous software deployed on an architecture. We chose to start from a description of the architecture in the AADL language, but the same approach could obviously be applied to other architecture description languages. A side effect of this translation is to provide a precise and executable semantics to the language. The obtained model can be used for early simulation and verification of systems. Since the proposed tool is based on existing popular formalisms (AADL and Scade), we think that it is likely to take place in a real model-driven process.

We consider that this work illustrates once again some advantages of using the synchronous framework:

- Intermediate situations between pure synchrony and pure asynchrony can be accurately modelled
- Several time scales can be used, which are inherited through the hierarchy of components: the time of subprograms is a sub-scale of the time of threads, which is a sub-scale of the time of processors.
- Synchronous code can be seamlessly embedded in the model of the architecture

The introduction of oracles is not the only way of modelling non-determinism, but it has the advantage of making

[1] http://www-verimag.imag.fr/∼synchron

[2] Actually, aadl2sync takes as input aaxl files, which are the xml counterparts of textual aadl files and that can be obtained with the eclipse plugin of OSATE (http://la.sei.cmu.edu/aadl-wiki)

the simulations reproducible, the behavior being deterministic as a function of inputs *and oracles*.

Coming back rapidly to related works, the same approach was used in many proposals for using synchronous languages — and especially the Signal language — for modeling non synchronous systems and developing system level design methodology: for instance, the ARINC avionic platform was modelled in Signal [10], and the Sildex tool was extended in the framework of the European projects Sacres and SafeAir [1, 2], resulting in the system-level tool box RT-Builder (see `http://www.tni-software.com/`). However, this common approach was never connected to a standard Architecture Description Language like AADL. On the other hand, the way we combine, in the same model, several so-called Models of Computation and Communication (MOCCs) — distributed, asynchronous or quasi synchronous composition, multitasking, synchrony — is similar to the approach proposed for long in environments like Ptolemy [4].

As short term further work, we plan to support additional AADL features. Indeed, we have implemented first the necessary features to be able to handle the PFS case study. However, some other notions will soon be useful and should be straightforward to translate into a synchronous/dataflow setting. For instance, the notion of event, which lets one describe event-triggered architecture, are quite easy to define in terms of activation condition. Moreover, all the component properties that are related to time delays (data transmission on buses, or data reading/writing on memory component, etc.) should be easy to handle in a similar manner as we model the execution time of sub-programs and threads. Our scheduler can also be enriched with locking mechanisms for reading and writing shared memory.

10. REFERENCES

[1] P. Baufreton. SACRES: A step ahead in the development of critical avionics applications. In F.W. Vaandrager and J.H. van Schuppen, editors, *Hybrid Systems: Computation and Control: Second International Workshop, HSCC'99*. LNCS 1569, Springer-Verlag, 1999.

[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.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[4] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A mixed-paradigm simulation/prototyping platform in c++. In *C++ At Work Conference*, Santa Clara, CA, 1991.

[5] 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.

[6] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *Proc. Safecomp'99*, volume 1698 of *Lecture Notes in Computer Science*. Springer Verlag, September 1999.

[7] P. Raymond E. Jahier, N. Halbwachs. *The AADL2sync User Guide*, 4 2007.

[8] 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.

[9] 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.

[10] 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.

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[12] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*. LNCS 2491, Springer Verlag, October 2002.

[13] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[14] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Sixth International Conference on Application of Concurrency to System Design, ACSD 2006*, Turku, Finland, June 2006.

[15] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[16] 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.

[17] H. Kopetz. The time-triggered architecture. In *ISORC'98*, Kyoto, Japan, April 1998.

[18] 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.

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

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

[21] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[22] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[23] SAE. Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace, November 2004.

[24] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, June 2004.

APPENDIX

A. A QUASI-SYNCHRONOUS CLOCK FILTER IN LUSTRE

We give below a Lustre V4 implementation of the quasi-synchronous clock generator/acceptor discussed in Section 5.1. We first define intermediary nodes. `compute_advance` computes the relative advance of ck1 w.r.t. ck2. `ba_fill` creates an array of size n filled with x. `ba_none` checks if all elements of the array are false.

```
node compute_advance(ck1, ck2 : bool)
returns (c : int);
var pc : int;
let
  pc = 0 -> pre c;
  c = if ck2 then 0 -- reset the advance of ck1
      else if ck1 then pc + 1 --
ck1 got ahead of 1 tick
      else pc;
tel

node ba_fill(const n: int; x : bool)
returns (t: bool^n);
let
  t = x^n;
tel

node ba_none(const n : int; I : bool^n)
returns ( ok : bool);
var Nor : bool^n;
let
  Nor = [not I[0]] | (Nor[0..n-2] and not I[1..n-1]);
  ok = Nor[n-1];
tel
```

`QS` is main node of the quasi-synchronous scheduler for processors with the same clock rate. It is paramaterized by:

- n is the number of clocks to generate;
- d is the maximal authorized clock drift, i.e., the maximum number of ticks authorized for the other clocks between 2 ticks of each clock;
- alea is an array of random values (n clock candidate values);
- select is the same alea, except for clocks that are d ticks late, that are forced to false.

```
node QS(const n:int; const d:int; alea:bool^n)
returns (select:bool^n);
var
  advance_max_is_reached, problems : bool^n^n;
  filter : bool^n;
  advance, padvance : int^n^n;
let
  advance_max_is_reached = padvance >= d^n^n;
  -- there is a problem if the max advance is reached
  -- and no tick occurs
  problems = advance_max_is_reached and not alea^n;
  -- we force the tick for clocks that would be
  -- more than d ticks late
  filter = ba_none(n^n, problems);
  select = alea and filter;
  advance = compute_advance(ba_fill(n^n, select),
                            select^n);
  padvance = 0^n^n -> pre advance;
tel
```

That program can be used to perform formal verification (e.g., model-checking) of the corresponding AADL model. In order to be able to perform simulations, the only thing that remains to be done is to generate arrays of random values.

B. PERIODIC THREADS SCHEDULING

Here are 2 Lustre programs corresponding to the description of Section 5.2. For both nodes, the input `qs_ck` comes from the node `QS` above.

A periodic clock generator

```
node clock_of_period(period :int; qs_ck:bool)
returns (activate_clock : bool);
let
  pcpt = period -> pre cpt;
  cpt = if activate_clock then period else
        if qs_ck then (pcpt - 1) else pcpt;
  activate_clock = true -> (pcpt = 1) and qs_ck;
tel
```

A rate-monotonic scheduler for 3 threads that eventually need the cpu.

```
node rms_3(
   qs_ck : bool; needs1, needs2, needs3 : bool)
returns (cpu1, cpu2, cpu3 : bool);
let
  cpu1 = qs_ck and needs1;
  cpu2 = qs_ck and needs2 and not cpu1;
  cpu3 = qs_ck and needs3 and not cpu1 and not cpu2;
tel
```