



Unité Mixte de Recherche CNRS - INPG - UJF

Centre Equation  
2, avenue de VIGNATE  
F-38610 GIERES  
tel : +33 456 52 03 40  
fax : +33 456 52 03 50  
<http://www-verimag.imag.fr>

## Verimag Manual

### **The aadl2sync User Guide**

*Erwan Jahier, Louis Mandel,  
Nicolas Halbwachs, Pascal Raymond*

Initial version: January 12, 2007

Last update: January 24, 2008

Software Version: 0.31

Last svn checkin: "2008-01-24 10:26:46 "

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>The AADL to Lustre translation scheme</b>	<b>2</b>
<b>3</b>	<b>The AADL component to Lustre node mapping</b>	<b>3</b>
3.1	Systems	4
3.2	Hardware components	4
3.2.1	Devices	4
3.2.2	Processors	4
3.2.3	Memory	4
3.2.4	Buses	4
3.3	Software components	5
3.3.1	Processes	5
3.3.2	Threads	5
3.3.3	Sub-programs	5
3.3.4	Data	6
3.4	Other concepts	6
<b>4</b>	<b>Modeling the AADL virtual machine</b>	<b>6</b>
4.1	Quasi-synchronous clocks	6
4.2	Multi-tasking/Time-sharing	7
4.3	Activation conditions	7
4.4	Release conditions	8
4.5	Formal verification versus simulation	9
<b>5</b>	<b>The aadl2sync compiler</b>	<b>9</b>
5.1	Installation	9
5.2	Usage	9
5.2.1	The aadl2sync compiler	9
5.2.2	Fixing the OSATE aaxl generated files	11
<b>6</b>	<b>Examples XXX NOT FINISHED YET</b>	<b>11</b>
6.1	Two threads, one processor	12
6.1.1	The scheduler ( <code>schedule_multitask.lus</code> )	12
6.1.2	The scheduled top-level node ( <code>multitask.lus</code> )	15
6.1.3	The Lucky file ( <code>multitask.luc</code> )	15
6.1.4	The translated components ( <code>multitask_nodes.lus</code> )	17
6.1.5	The model leaves to be Filled in ( <code>fillme_multitask_nodes.lus</code> )	17
6.1.6	A filled model ( <code>fillme_multitask_nodes.lus</code> )	18
6.1.7	Constants ( <code>fillme_multitask_const.lus</code> )	18
6.1.8	Data types ( <code>fillme_multitask_types.lus</code> )	18
6.2	Two processors	18
<b>7</b>	<b>A test session using Lurette</b>	<b>21</b>
<b>8</b>	<b>A formal proof session using Lesar</b>	<b>21</b>
<b>9</b>	<b>The currently supported AADL subset</b>	<b>21</b>
9.1	Ignored concepts	21
9.2	Not supported concepts	21
9.3	Supported concepts	22
9.4	Some design decisions	22
9.5	Known limitations	22

## **A A quasi-synchronous clock generator/acceptor in Lustre V4**

**23**

## 1 Introduction

Architecture Description Languages (ADL) aim at defining systems by describing jointly software and hardware components and their interactions. ADL can describe functional interfaces to components and non-functional aspects (reliability, availability, timing, responsiveness, etc.). The objective is to be able to perform various analysis (schedulability, sizing analysis, safety analysis) very early in the development process.

AADL (Architecture Analysis and Design Language) is one such ADL, that specifically targets real time embedded systems. It lets one describe the structure of such systems as an arborescent assembly of software components mapped onto an execution platform. The leaves of that arborescent description are made of component interfaces that are left un-implemented.

Given an AADL model and implementations for the component leaves, the `aadl2sync` allows one to perform automatically simulation and formal verification. This is done by translating the AADL model into the Lustre programming language, for which simulation and formal verification tools exist. This tool chain allows one to focus on functional properties far before machine code generation and deployment phases.

The main difficulty in this translation is to model intrinsically non-deterministic and a-synchronous AADL descriptions into a synchronous language. To do that, we use techniques based on sporadic activation condition (stuttering)[6], input addition (oracles), and quasi-synchronous clocks [1].

In this document, we explain of the translation is performed, and how the tool can be used. Please refer to [4] for a complementary description of the process.

In the following, Lustre means either the academic Lustre, or Scade (cf Section for a description of the various code generation options). We illustrate our presentation using the academic Lustre syntax. We suppose the reader in familiar with the AADL concepts (cf. [2, 7]) as well as with Lustre [3].

**Outline.** Section 2 describes the general principles of the translation, and Section 3 how the translation operates component by component. Section 4 describes the scheduler driving the various oracles variables introduced during the translation. Section 5 presents the tool installation and usage and Section 6 illustrates its use on an example. Finally, Section 9 sums up which subset of AADL is supported, as well as the limitations of the current version of the tool.

## 2 The AADL to Lustre translation scheme

The objective of the translation in Lustre is to be able to use the Lustre simulation and formal verification tools to validate AADL designs very early in the development process.

In other words, we want to define in Lustre an executable/formal semantics for a subset of AADL. Since Lustre is a discrete-time, synchronous and deterministic language, a-synchronous and non-deterministic aspects of AADL call for a particular attention.

**From continuous to discrete time.** It is not really specified in the AADL model if the time is continuous or discrete, but what is certain is that, for Lustre, the time is discrete.

In order to discretize the time, we map the basic clock to the smallest time units that appears in the AADL model. For example, if the smallest time units that is used in the model are the milli-seconds (ms), then the basic clock will be rated at 1 milli-second.

**Modeling implicit non-determinism.** The first source of non-determinism in AADL comes from the fact that, when defining a set of programs/threads/processes running on different processors, the time (clocks) when every components is activated, or deliver its output is left unspecified. In order to model uncertainty in Lustre, a classical solution is to introduce oracles: an oracle is a Boolean variable that is added to the program inputs. For example, here, we can add a Boolean input that is true when the thread ought to be activated. As it is an input, formal verification tools (e.g., model checkers) would verify that the properties one wants to check holds for all the possible values of that oracle. Note however that oracles are generally not completely free variables; they respect some constraints such as, for example, that 2 threads can not run

together on the same process. In the following, we call oracles that control the activation of components *activation conditions*.

**Modeling non-instantaneous executions.** In some other cases, the non-determinism is stated explicitly. For example a thread or a sub-program may have the property that its `compute_execution_time` ranges from 10 to 100 Ms. In such a case, we add a boolean input that controls whether to retain the previous value of the component, or to deliver a new value. This oracle has the property to be true between 10 and 100 cycles after the thread is activated. In the following, we call oracles that control the cycle at which the components output values are available *release conditions*.

**Modeling a-synchronous components.** Programs that try to exchange values run on different processors might be sensible to clock drift. One way to weaken the synchronous hypothesis to model a-synchronous components is to use *quasi-synchronous* clocks [1]. Clocks are said to be *quasi-synchronous* (resp *n quasi-synchronous*) if, between two successive activations of one clock, the other clocks are activated at most twice (resp *n* times). Such an interleaving is illustrated in Figure 1. 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, `a1` (resp. `a2`) 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.

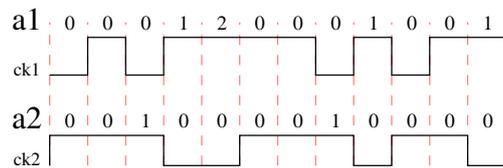


Figure 1: Two processes running on quasi-synchronous clocks

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. Hence, 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. Using quasi-synchronous clocks allow one to check that robustness for AADL models.

**An AADL component to Lustre node mapping.** Basically, the translation consists in mapping AADL components to Lustre nodes, where the input/output ports of components correspond to the input/output of nodes, and where node calls directly result from component wiring (connections).

Since we focus on functional properties, we will (safely) ignore in our translation most of the hardware components. We also ignore most properties, as they generally concern non-functional aspects. Nodes have additional inputs for carrying the oracles of their sub-nodes. Those oracles are controlled by a scheduler that is described in Section 4. More details about this component to node mapping is given in the following Section.

### 3 The AADL component to Lustre node mapping

AADL components are made of a *Type*, and each *Type* is optionally associated to one (or several) *Implementation(s)*. A component *Type* describes the functional interface of the component. An *Implementation* inherits of the corresponding component *Type* attributes. It also declares the sub-components it is made of, as well as the *connections* between them. Those connections are made through the sub-components input and output *ports*.

An AADL model is made of an arborescent assembly of software and hardware component types or implementations. 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 subprograms. Leaves of the AADL model are therefore either subprograms, or component types. We need actual (Lustre) code for those leaves to be able to perform formal verifications and simulations.

### 3.1 Systems

A system is the top-level component of the AADL model. It is translated into a (top-level) Lustre node. Each input (resp output) port of the system is mapped into an input (resp output) of the node. For system implementations, each sub-component is mapped into a Lustre node obtained by the translation. The node calls result straightforwardly from its inner connections.

**Properties.** No property is taken into account for systems.

**Sub-components.** The sub-components of a system can be processors, processes, devices, and data. It can also be memory and bus, but those are abstracted away by the tool. It could also be systems, but this is not supported yet.

### 3.2 Hardware components

#### 3.2.1 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.

**Properties.** No property is taken into account for devices.

**Sub-components.** None.

#### 3.2.2 Processors

Processor components are an abstraction of hardware and software responsible for executing and scheduling processes. Basically, each processor will have its own 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. More details on this scheduler are provided in Section 4.

**Properties.** The `Clock_period` is used when the system contains several processors to generate the quasi-synchronous clocks (cf Section 4.1).

**Sub-components.** The only possible sub-component for processors are memories, but those are abstracted.

#### 3.2.3 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.

#### 3.2.4 Buses

Bus components are used to exchange data between hardware components. Detailed models of specific buses can be provided. In our prototype tool, we just consider buses as usual connections.

### 3.3 Software components

As for systems, each software component (except data) is mapped into a Lustre node which inputs/outputs is made of the process input/output ports. For process implementations, the node calls result from its inner connections.

#### 3.3.1 Processes

Process components are an abstraction of software responsible for scheduling and for executing threads. Processes are scheduled the same way 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.

**Properties.** When several processes run on the same processor, the `dispatch_protocol` property, attached to each process, is used by the scheduler to activate it. It can be set to `Periodic` only, and the `period` property must be set.

**Sub-components.** The sub-components of a process can be a thread or a data. Thread groups are not supported.

#### 3.3.2 Threads

Thread components are an abstraction of software responsible for scheduling and for executing sub-programs. When several threads run under the same process, the sharing of the process is managed by a runtime scheduler.

**Properties.** 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`, is necessary for thread Types so that the scheduler knows when to make the outputs available. For thread implementations, this execution time results from the `compute_exec_time` associated to their sub-programs.

**Sub-components.** Thread implementations are made of sub-program calls (and data).

#### 3.3.3 Sub-programs

Sub-program components 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.

**Properties.** A sub-program must define a `compute_exec_time` property, so that the scheduler knows when to release its outputs.

**Sub-components.** None.

### 3.3.4 Data

Data components are not associated to Lustre nodes, but to Lustre types. They are not handled by **aadl2sync** (except for the base types `bool`, `int`, and `floats`) and must be set by users or a third party tool (one such tool is developed within the ASSERT project).

Each data subcomponent is translated into a local variable of the containing component. If the containing component provides an access to that data, then its interface is modified:

- `read_only`: one output is added
- `write_only`: one input is added
- `read_write`: one output and one input are added
- `by_method`: the interface remains unchanged

## 3.4 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.

## 4 Modeling the AADL virtual machine

In order to model non-determinism, oracles are added to node inputs. Each node carries all the oracles necessary to control its sub-nodes. The top-level node therefore potentially have quite a lot of such oracles: two per component leave instances (one for the activation condition, and one for the release condition), plus as many quasi-synchronous clocks as there are processors. All those oracles are controlled by a centralized scheduler that is automatically generated. This scheduler models the behavior of the AADL virtual machine. We describe below what this generated scheduler looks like.

### 4.1 Quasi-synchronous clocks

We describe in this Section a possible implementation of a Quasi-synchronous (QS) clocks generator/acceptor. This program is parameterized by:

- `n`: the number of QS clocks to generate, and
- `d`: the maximal clock drift, i.e., the maximum number of tics that are authorized for the other clocks between 2 tics of each clock.

The idea is, starting from a `n`-array of Boolean values (e.g., chosen randomly) containing candidate values for the QS clocks, to check that no clock drift excess occurs, and to force the culprit clocks to be false when necessary to avoid this drift excess.

More precisely, we compute the relative advance of each clock w.r.t. the  $(n-1)$  other clocks. If:

- the relative advance of a clock `clk1` w.r.t. another clock `clk2` is equal to `d`,
- `clk1` is true,
- `clk2` is false,

then we force `clk1` to be false. Such a Lustre program (in the V4 syntax) is provided in extenso in appendix [A](#).

## 4.2 Multi-tasking/Time-sharing

Several threads that run on the same process need to share the CPU. Therefore we define one scheduler per process that runs more than one thread.

In the current version of **aadl2sync**, this scheduler is quite basic: it is a Rate-monotonic scheduler (preemptive, fix priority). Higher priority is given to threads that have the smallest period. For example, in order to schedule 3 threads, **aadl2sync** would generate the following task scheduler:

### A rate-monotonic scheduler for scheduling 3 threads

```
node cpu_from_needs_3(tick : bool; needs1, needs2, needs3 : bool)
returns (cpu1, cpu2, cpu3 : bool);
let
  cpu1 = tick and needs1;
  cpu2 = tick and needs2 and not cpu1;
  cpu3 = tick and needs3 and not cpu1 and not cpu2;
tel
```

where `tick` is the quasi-synchronous clock of the processor, and `needs1` (resp `needs2`, `needs3`) is a boolean that indicates whether the first (resp second, third) thread claims to be active. `cpu1` is true if the control is assigned to the first (resp second, third) thread. This scheduler can straightforwardly be generalized to  $n$  processes. The variables `cpu1` and `needs1` are associated to the higher priority thread, i.e, to the thread with the smallest period.

Note that we neglect the time necessary to perform context switch between threads, but it could be taken it into account tough quite easily.

## 4.3 Activation conditions

An activation condition is added to Periodic and sporadic components (processes and threads). An activation condition is a Boolean variable that is set to true when the node needs to be activated. For a component  $C$  of period  $p$ , the scheduler define a Boolean variable `C_activate_clk` that is true every  $p$  cycles of the outer processor.

### A periodic clock generator

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

For example, suppose that we have an AADL model leaf which is a periodic thread type, and which corresponds to the Lustre node `incr` that increments an integer.

### Example 1 A Lustre node that increments a integer

```
node incr(x: int) returns (y: int)
let
  y = x+1;
tel;
```

The components that call this thread would not call `incr` directly, but would call `conduct_incr`.

#### Example 2 The conduct of node incr

```
node conduct_incr (clk:bool; y_def:int; x:int) returns (y:int)
let
  y = if clk then current (incr (x when clk)) else y_def -> pre y;
tel;
```

The node `conduct_incr` has 2 additional inputs. A Boolean `clk` that is used to activate the the node `incr`; when `clk` is true, the node `incr` is called, and otherwise, the node `incr` outputs the value, stored in a memory, computed the last time the node was activated. The second additional input `y_def` is used to be able to provide an initial value to the memory.

In Scade, there exists a **cond\_act** operator that do exactly the same thing. Therefore, when generating Scade code, we generate `conduct (incr, clk, y_def, x)` instead of `conduct_incr (clk, y_def, x)`, and we do not need to define `conduct_incr` at all.

## 4.4 Release conditions

In the synchronous framework, nodes compute outputs from inputs instantaneously. In order to simulate that subprograms and threads do take time, we simply add barriers that retain the output values a certain amount of cycles. That number of cycles is determined by the `Compute_Execution_time` property that is associated to the corresponding component. For example, the integer output of node `incr` is filtered by a node `barrier_incr` defined as follows.

#### Example 3 The barrier of node incr

```
node barrier_incr (clk:bool; y_def:int; y:int) returns (y_delayed:int)
let
  y_delayed = y_def -> pre (if clk then y else y_delayed);
tel;
```

The clock `clk` of that barrier is computed by the node `consume`. That node takes as input

- an integer `in_min_max`, which may contain any value defined by the `Compute_execution_time` property;
- the activation condition `trig` and
- the `cpu` variable that is true when the component have the `cpu`.

When the activation condition `trig` becomes true, the `in_min_max` input is used to set the `alea` local variable, and a counter `cpt` is set to 0. Then, each step where `cpu` is true, `cpt` is incremented. When `cpt` becomes equal to `alea`, the release condition is emitted (i.e., becomes true for 1 cycle).

#### Computing the release condition

```
node consume (in_min_max : int; trig, cpu: bool)
returns (needs, release: bool);
var
  next_needs, edge_trig : bool;
  cpt, pcpt : int;
  alea : int;
let
  alea = if edge_trig then in_min_max else (0 -> pre alea);

  needs = false -> pre next_needs;
  (next_needs, edge_trig) = if needs then (not release, false)
                           else (trig, trig);
  cpt = if needs then (if cpu then pcpt + 1 else pcpt) else 0;
  pcpt = 0 -> pre cpt;
  release = cpu and (cpt = alea);
tel
```

When a thread implementation runs several sub-programs, those are executed in sequence. The release condition of the a sub-program is therefore plugged onto the activate condition of the following sub-program. The release condition of the last sub-program is plugged onto the release condition of the outer thread.

## 4.5 Formal verification versus simulation

In order to perform formation verification (e.g., model-checking), one should take into account that the oracles that have been added during the translation are not completely random. For example, The `in_min_max` variables ought to be between a minimum and a maximum value. This is why an additional node is defined in node `schedule_ex1.lus`:

### Oracles properties

```
node schedule_ex1_oracles_properties(  
  in_min_max11, in_min_max12, in_min_max13 : int;  
  in_min_max21, in_min_max22, in_min_max23 : int;  
  alea1, alea2 : bool)  
returns (ok:bool);  
let  
  ok =  
    ( 1 <= in_min_max11 and in_min_max11 <= 5) and  
    ( 1 <= in_min_max12 and in_min_max12 <= 5) and  
    ( 10 <= in_min_max13 and in_min_max13 <= 30) and  
    ( 1 <= in_min_max21 and in_min_max21 <= 5) and  
    ( 1 <= in_min_max22 and in_min_max22 <= 5) and  
    ( 10 <= in_min_max23 and in_min_max23 <= 30) and  
  true;  
tel
```

For simulation purposes, a lucky program named `ex1.luc` is generated, which also take those constraints into account to generate random input. This generated file can serve as a basis to describe the behavior of the AADL model environment to generate realistic input sequences.

## 5 The **aadl2sync** compiler

### 5.1 Installation

**aadl2sync** is a stand-alone executable. Therefore, one just needs to put it somewhere accessible via the `PATH` environment variable.



The environment variable `AADL_SCHEMA_DIR` ought to be set to a directory containing the `aaxl` schema definitions. Those can be found on the OSATE website, or in the `aadl-schema` directory of the **aadl2sync** distribution.

The Makefile in the example directory assumes that the Lurette tools are accessible from the `PATH`. This tool suite can be downloaded there: <http://www-verimag.imag.fr/synchron/index.php?page=lurette/download>

### 5.2 Usage

#### 5.2.1 The **aadl2sync** compiler

Actually, **aadl2sync** does not take as input `aadl` files, but their `aaxl` counter-parts. Those can be obtained via the Eclipse plugin of OSATE. If you edit your AADL programs with an external editor (`emacs`, `vi`, etc.),

one just needs to open the AADL file with the Eclipse/OSATE editor, and the aaxl files are automatically generated<sup>1</sup>.

If one launches the aadl2sync compiler using the -h option, one gets:

This is aadl2sync version 0.30

```
Usage: aadl2sync <options> <file>.aaxl+
<options> are:
  --output-file <str>
  -o <str>
      output file base name

  --lustre
      Generate academic lustre code for the Verimag compiler (the default).
  --scade
      Generate lustre code that is compatible with Scade.

  --verbosity-level <int>
  -v <int>
      Set the verbosity level.

  --gen-fake-body
      Generate a fake body to bodyless nodes

  --toplevel-system
  -tc
      Set the top-level system (one of the system is used if left unset).

  --gen-random-func
      Generates random functions instead of adding oracle
      variables as node inputs.

  --one-file
      Generates only one file (instead of 6).

  --v4-arrays
      Use Lustre V4 arrays in the generated code.

  --check-schedulability
      Add an additional boolean output (schedul_ok) that is true as long as no dead-li

  --show-scheduling-vars
      Add the internal scheduling variable at node outputs
      (for debugging or/and to understand what's going on).

  --version
  -version
      Show the version.

  --help
  -help
  -h
      Display this help message
```

---

<sup>1</sup>It ought to be possible to call this aadl2aaxl translator outside, from Eclipse; if anyone knows how to do it, please let me know

For instance, the command line call `aadl2sync ex1.aaxl` would generate 6 files:

- `ex1.lus` that contains the top-level nodes `ex1.simu` and `ex1.verif` that can be respectively used for simulation and formal verification. It also includes the other files described below.
- `ex1_nodes.lus` that contains the translation in Lustre of all AADL components present in the model.
- `ex1_scheduler.lus` that contains the scheduler driving all the oracles introduced during the translation.
- `fillme_ex1_nodes.lus` that contains the interface of nodes corresponding to leaves in the AADL model. The bodies of such nodes need to be provided.
- `fillme_ex1_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_ex1_types.lus` that contains the Lustre type definitions of Data type components. This translation is not automated by **aadl2sync**, but external tools exist that translate `asn1` type definition to Lustre for example.

Those last 3 files, which are prefixed by `fillme_`, need to be filled in and renamed without the prefix. If **aadl2sync** is launched with the `--gen-fake-body` option, fake values are provided for each of the entity to be defined.

The content of those 6 files is illustrated and described further in Section 6.

### 5.2.2 Fixing the OSATE aaxl generated files



As any xml document, aaxl files ought to define in their header (second tag) the URI where to find the xml schema they are supposed to conform to. However, the aaxl files generated by OSATE defines to a wrong URI (`http://AADL/2`). In order to turn around this problem, **aadl2sync** copies those schema files in the `/tmp/aadl-schema/` directory (using the `AADL_SCHEMA_DIR` environment variable). Therefore, one just needs to fix the aaxl file (e.g., with `sed`) so that it points to this directory. The purpose of the `fix_aaxl` rule of the `Makefile` provided in the `examples` directory of the distribution, is precisely to automate this boring process.

## 6 Examples XXX NOT FINISHED YET



THIS SECTION IS NOT FINISHED YET, SORRY.

We demonstrate in this section the use of the tool on two (a bit silly) examples that ought illustrate most of the AADL features we take into account. The first one is an AADL program that uses two threads of different periods that run on a single processor. The second one is made of 2 processors; each processor is running one process; each process is running 2 periodic threads: a (slow and low-priority) thread type, and (fast and high-priority) thread implementation that runs 2 sub-programs.

very simple exemple to illustrate - what file are generated - how thread are simulates

<sup>2</sup>it might be possible to say to the OSATE plugin of eclipse how to redefine that; please tell me (jahier@imag.fr) if you know how.

## 6.1 Two threads, one processor

Here is an AADL program that runs two periodic threads running on a single processor. The first thread, of period 16 ms, returns its value between 6 and 8 ms (of cpu time) after it has started. The second thread, of period 4 ms, returns its value after between 1 and 2 ms. Each thread exchanges an integer value. The top-level system has no input, and returns the output value of the first thread.

### The multitask.aadl file

```

system multitask end multitask;

system implementation multitask.IMPL
subcomponents
  slow_cmd : device An_Actuator;
  fast_cmd : device An_Actuator;
  a_processor : processor A_Processor.IMPL;
  a_process : process A_Process.IMPL;
connections
  data port a_process.O1 -> slow_cmd.I;
  data port a_process.O2 -> fast_cmd.I;
properties
  Actual_Processor_Binding =>
    reference a_processor applies to a_process;
end multitask.IMPL;

device An_Actuator
features
  I: in data port int;
end An_Actuator;

processor A_Processor end A_Processor;
processor implementation A_Processor.IMPL
end A_Processor.IMPL;

process A_Process
features
  O1 : out data port int;
  O2 : out data port int;
end A_Process;

process implementation A_Process.IMPL

subcomponents
  s : thread SlowThread;
  f : thread FastThread;
connections
  data port f.O -> s.I;
  data port s.O -> f.I;
  data port s.O -> O1;
  data port f.O -> O2;
end A_Process.IMPL;

thread SlowThread
features
  I: in data port int;
  O: out data port int;
properties
  Dispatch_Protocol => Periodic;
  Period => 15 ms;
  Compute_Execution_Time => 1 ms .. 12 ms;
end SlowThread;

thread FastThread
features
  I: in data port int;
  O: out data port int;
properties
  Dispatch_Protocol => Periodic;
  Period => 10 ms;
  Compute_Execution_Time => 1 ms .. 6 ms;
end FastThread;

data int end int;

```

This file can be found in the examples/multitask directory of the distribution. Launch "make demo" to generate an executable simulation of it.

Figure 2 illustrates XXX...

Note that `_thr1_activate` is true every 4 tics, whereas `_thr2_activate` is not always true every 10 tics. It is because a thread is activated after *period* tics as soon as it has the cpu. And since `thr2` is not the most priority thread, sometimes it is activated a little bit later than its period, i.e. once `thr1` release the CPU.

Here are the files generated by `aadl2sync` to obtain that simulation are described in the sequel.

### 6.1.1 The scheduler (schedule\_multitask.lus)

The file named `schedule_multitask.lus` defines the node `schedule_multitask` that schedules the top-level system. The inputs named `in_min_max.<i><j>` are used to simulate the compute execution time of the two threads (`<i>` refers to the processor number, and `<j>` refers to the thread number of processor `i`). Their values are set by the (generated) program Lucky file `multitask.luc` 6.1.3) described below.

This node outputs the various scheduling variables that indicate when a thread is activated (`_thr<j>_activate_clk`), when its output should be made available to other components (`_thr<j>_release_clk`). `need<i><j>` is true when the `j`th thread of the `i`th processor ask for the CPU. `cpu<i><j>` is true when the `j`th thread of the `i`th processor has the CPU.

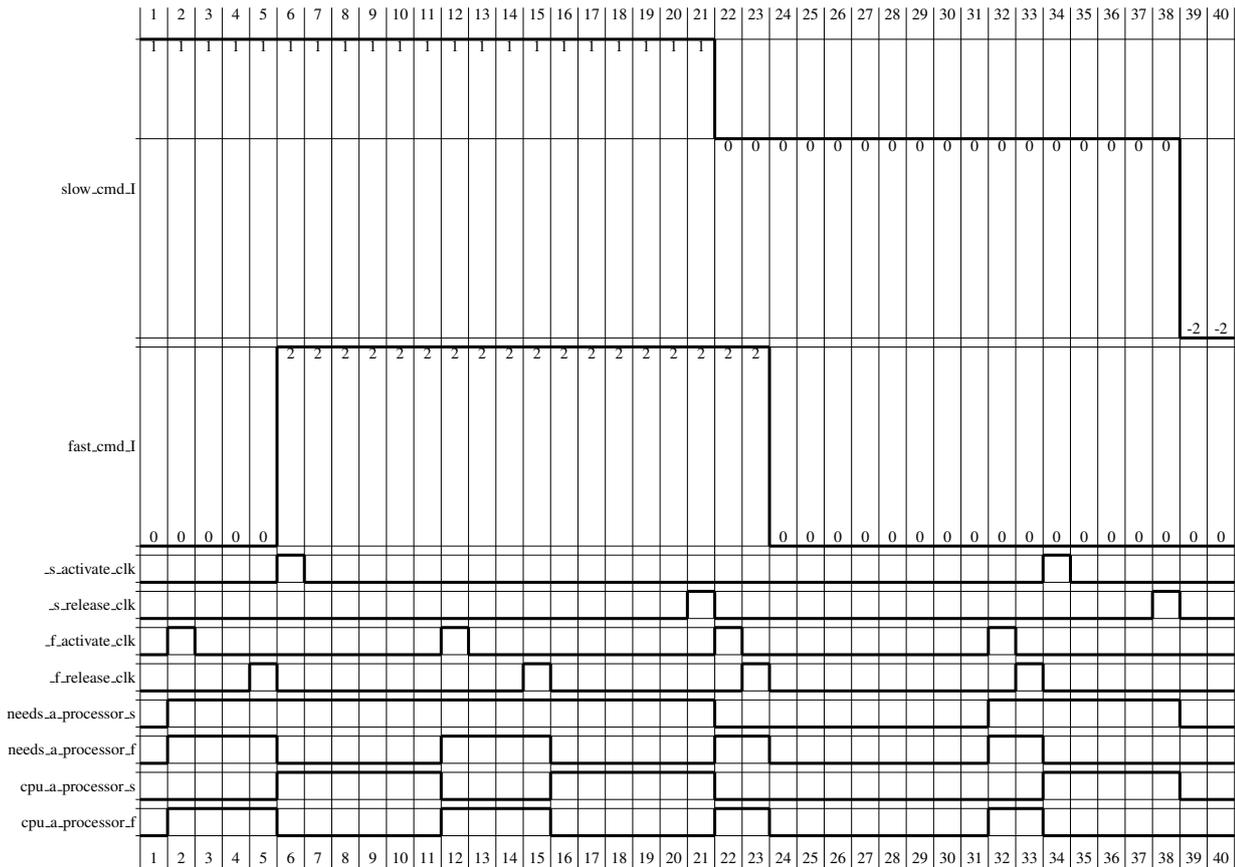


Figure 2: A timing diagram obtained from a simulation of multitask.aadl

The scheduler (schedule\_multitask.lus)

```

-- This file was generated by aadl2sync version 0.28.
-- ../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--multitask multitask.aaxl
-- on peouvou the 29/6/2007 at 14:50:11

include "fillme_multitask_types.lus"
include "fillme_multitask_const.lus"

-- interruption generator
node timer(const period : int; qs_tick:bool) returns (trig: bool);
var cpt, pcpt:int;
let
  pcpt = period -> pre cpt;
  cpt = if trig then period
        else if (qs_tick) then pcpt -1 else pcpt;
  trig = true -> (pcpt = 1) and qs_tick;
tel

node rising_edge(x:bool) returns (y:bool);
let
  y = false -> x and not (pre x);
tel

node first_rising_edge(x,reinit:bool) returns (y:bool);
-- y is true when x becomes true for the first time
-- since the beginning or the last reinit
var
  -- this var is used to encode a 2-state automaton
  y_has_been_emitted : bool;
let
  y = false ->
    if ((not pre y_has_been_emitted) and rising_edge(x)) then true
    else if (pre y_has_been_emitted and reinit) then false
    else false ;
  y_has_been_emitted = false ->
    if ((not pre y_has_been_emitted) and rising_edge(x)) then true
    else if (pre y_has_been_emitted and reinit) then false
    else pre y_has_been_emitted;
tel

-- cpu time counter
node consume(in_min_max : int; activate, cpu: bool)
returns (next_needs, term: bool);
var
  needs, edge_activate : bool;
  cpt, pcpt : int;
  alea : int;
let
  -- in_min_max is ignored, except when activate becomes true
  alea = if edge_activate then in_min_max else (0 -> pre alea);
  needs = false -> pre next_needs;
  next_needs, edge_activate =
    if needs then (not term, false) else (activate, activate);
  cpt = if needs then (if cpu then pcpt + 1 else pcpt) else 0;
  pcpt = 0 -> pre cpt;
  term = cpu and (cpt = alea);
tel

let
  cpul = qs_tick and needs1;
  cpu2 = qs_tick and needs2 and not cpul;
tel

node schedule_a_processor_threads(
  in_min_max1, in_min_max2 : int;
  qs_tick : bool)
returns (
  _s_activate_clk, _s_release_clk : bool;
  _f_activate_clk, _f_release_clk : bool;
  needs2 : bool;
  needs1 : bool;
  cpu2 : bool;
  cpul : bool);
var
  _s_activate_clk_candidate : bool;
  _f_activate_clk_candidate : bool;
  next_needs1, next_needs2 : bool;
let
  _s_activate_clk_candidate = timer(15, qs_tick);
  _f_activate_clk_candidate = timer(10, qs_tick);
  next_needs2, _s_release_clk = consume(
    in_min_max1, _s_activate_clk_candidate, cpu2);
  next_needs1, _f_release_clk = consume(
    in_min_max2, _f_activate_clk_candidate, cpul);
  needs1 = false -> pre next_needs1;
  needs2 = false -> pre next_needs2;
  cpul, cpu2 = cpu_from_needs_2(qs_tick, needs1, needs2);
-- We read inputs when the thread starts having the cpu
_s_activate_clk = first_rising_edge(cpu2, not needs2);
_f_activate_clk = first_rising_edge(cpul, not needs1);
tel
node schedule_multitask(
  _s_release_clk_min_max:int;
  _f_release_clk_min_max:int ) returns (
  _s_activate_clk : bool;
  _s_release_clk : bool;
  _f_activate_clk : bool;
  _f_release_clk : bool;
  needs11, needs12 : bool,
  cpul1, cpul2 : bool);
let
  _s_activate_clk,
  _s_release_clk,
  _f_activate_clk,
  _f_release_clk,
  needs11, needs12,
  cpul1, cpul2
= schedule_a_processor_threads(
  _s_release_clk_min_max,
  _f_release_clk_min_max,
  true
);

```

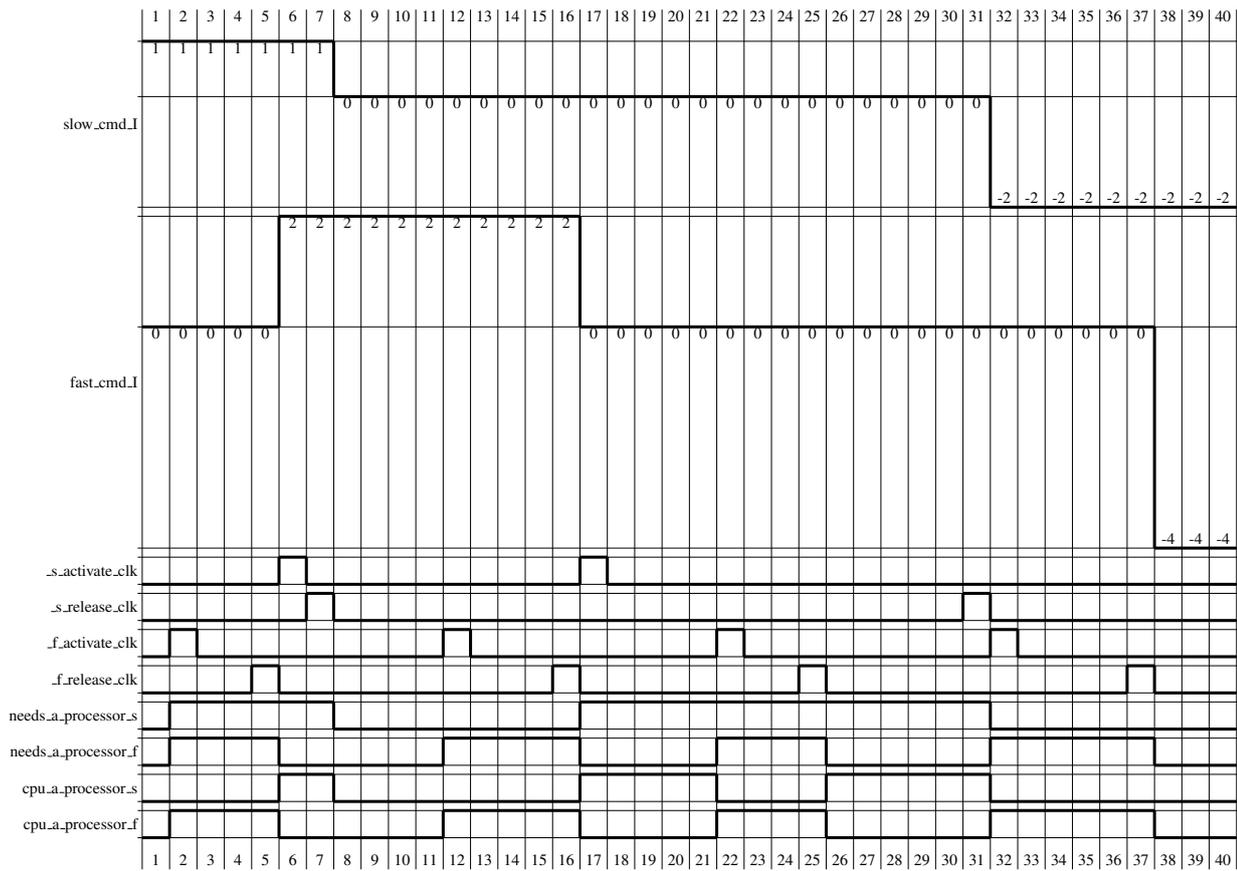


Figure 3: Another timing diagram obtained from a simulation of multitask.aadl the second activation of the fast thread was able to read the result of the slow thread at step 7, whereas in previous diagram, this result is only available at step 21

## 6.1.2 The scheduled top-level node (multitask.lus)

### The scheduled top-level node

```

-- This file was generated by aadl2sync version 0.28.
-- ../../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--multitask multitask.aaxl
-- on peouvou the 29/6/2007 at 14:50:11

include "schedule_multitask.lus"
include "multitask_nodes.lus"

-----
-- scheduled version of node multitask__multitask_IMPL

node multitask(
  _s_release_clk_min_max:int;
  _f_release_clk_min_max:int)
returns (
  slow_cmd_I : multitask__int;
  fast_cmd_I : multitask__int;
-- we output the scheduler clocks to see what is going on (cf --show-scheduling-vars option).
  _s_activate_clk : bool;
  _s_release_clk : bool;
  _f_activate_clk : bool;
  _f_release_clk : bool;
  needs_a_processor_s, needs_a_processor_f : bool;
  cpu_a_processor_s, cpu_a_processor_f : bool);

let
  _s_activate_clk,
  _s_release_clk,
  _f_activate_clk,
  _f_release_clk,
  needs_a_processor_s,
  needs_a_processor_f, cpu_a_processor_s,
  cpu_a_processor_f
  = schedule_multitask(
    _s_release_clk_min_max,
    _f_release_clk_min_max
  );

slow_cmd_I,
fast_cmd_I
  = multitask__multitask_IMPL
(
  _s_activate_clk,
  _s_release_clk,
  _f_activate_clk,
  _f_release_clk
);
tel;

```

This node outputs various internal scheduling variables that are there to illustrate the behavior of the scheduler. Without the `--show-scheduling-vars` option, this node would have no output. The inputs for that node are generated by the stochastic Lucky program of Section 6.1.3.

## 6.1.3 The Lucky file (multitask.luc)

This Lucky program only generates values for the `in_min_max<i><j>` variables. But if the top-level component had input, then this program would have them in its output interface. It would be up the user to define constraints over those variables. The internal variables are there because the compiler has been invoked with the `--show-scheduling-vars` options, and because lucky imposes that the set of inputs (resp outputs) matches exactly the set of outputs (resp inputs) of the node its generates inputs for (cf the Lucky manual [5]).

### The Lucky file

```
-- to be used for simulation
inputs {
  slow_cmd_I : int;
  fast_cmd_I : int;
  _s_activate_clk : bool;
  _s_release_clk : bool;
  _f_activate_clk : bool;
  _f_release_clk : bool;
  needs_a_processor_s : bool;
  needs_a_processor_f : bool;
  cpu_a_processor_s : bool;
  cpu_a_processor_f : bool
}

outputs {
  _s_release_clk_min_max:int;
  _f_release_clk_min_max:int}

states { loop : stable }
start_state { loop }

transitions {
  loop -> loop ~cond
  1 <= _s_release_clk_min_max and _s_release_clk_min_max <= 12 and
  1 <= _f_release_clk_min_max and _f_release_clk_min_max <= 6 and
  true
}
```

## 6.1.4 The translated components (multitask\_nodes.lus)

### The translated components

```

-- This file was generated by aadl2sync version 0.28.
-- ../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--multitask multitask_axl
-- on peouvou the 29/6/2007 at 14:50:11

include "fillme_multitask_nodes.lus"

-----
---- multitask_A_Process_IMPL
node multitask_A_Process_IMPL(_s_activate_clk : bool;
  _s_release_clk : bool;
  _f_activate_clk : bool;
  _f_release_clk : bool) returns (O1 : multitask__int;
  O2 : multitask__int);
var
  f_O_conduct : multitask__int;
  f_O : multitask__int;
  s_O_conduct : multitask__int;
  s_O : multitask__int;
let
  f_O =
    barrier_multitask_FastThread(_f_release_clk,
      multitask_FastThread_O_dft , f_O_conduct
    );
  f_O_conduct =
    conduct_multitask_FastThread(_f_activate_clk, s_O,
      multitask_FastThread_O_dft
    );
  O2 = f_O;
  s_O =
    barrier_multitask_SlowThread(_s_release_clk,
      multitask_SlowThread_O_dft , s_O_conduct
    );
  s_O_conduct =
    conduct_multitask_SlowThread(_s_activate_clk, f_O,
      multitask_SlowThread_O_dft
    );
  O1 = s_O;
tel;

-----
---- multitask_multitask_IMPL
node multitask_multitask_IMPL(_s_activate_clk : bool;
  _s_release_clk : bool;
  _f_activate_clk : bool;
  _f_release_clk : bool) returns (slow_cmd_I : multitask__int;
  fast_cmd_I : multitask__int);
var
  a_process_O1 : multitask__int;
  a_process_O2 : multitask__int;
let
  fast_cmd_I = a_process_O2;
  slow_cmd_I = a_process_O1;
  a_process_O1, a_process_O2 = multitask_A_Process_IMPL(_s_activate_clk,
    _s_release_clk,
    _f_activate_clk,
    _f_release_clk
  );
tel;

-----
----- Barrier of multitask_SlowThread
node barrier_multitask_SlowThread(
  clk:bool; defl:multitask__int;
  x1:multitask__int)
returns ( y1:multitask__int );
let
  y1 = defl -> pre (if clk then x1 else y1);
tel;
-- End of node barrier_multitask_SlowThread
-----
----- Conduct multitask_SlowThread
node conduct_multitask_SlowThread(clk:bool; x1:multitask__int;
  defl:multitask__int )
returns ( y1:multitask__int );
var
  x1_when_clk : multitask__int when clk;
let
  x1_when_clk = x1 when clk;

  y1 =
    if clk then current(multitask_SlowThread(x1_when_clk))
    else ((defl)
    -> pre (y1));
tel;
-- End of node conduct_multitask_SlowThread
-----
----- Barrier of multitask_FastThread
node barrier_multitask_FastThread(
  clk:bool; defl:multitask__int;
  x1:multitask__int)
returns ( y1:multitask__int );
let
  y1 = defl -> pre (if clk then x1 else y1);
tel;
-- End of node barrier_multitask_FastThread
-----
----- Conduct multitask_FastThread
node conduct_multitask_FastThread(clk:bool; x1:multitask__int;
  defl:multitask__int )
returns ( y1:multitask__int );
var
  x1_when_clk : multitask__int when clk;
let
  x1_when_clk = x1 when clk;

  y1 =
    if clk then current(multitask_FastThread(x1_when_clk))
    else ((defl)
    -> pre (y1));
tel;
-- End of node conduct_multitask_FastThread

```

This file contains the lustre translation of AADL components, as described in Section 4. The node activations and the variables releases are controlled by the scheduler variables.

## 6.1.5 The model leaves to be Filled in (fillme\_multitask\_nodes.lus)

### The model leaves to be Filled in

```

node multitask_SlowThread(I : multitask__int) returns (O : multitask__int);
let
  O = I - 2 ;
tel;

node multitask_FastThread(I : multitask__int) returns (O : multitask__int);
let
  O = I * 2;
tel;

```

When invoked with the "--gen-fake-body" option, the compiler put default arbitrary values to unknown entities such as undefined output variables (but also to data types, and constant values, as illus-

trated below). One have to search for the string "XXX" all through the generated files, in order to check whether those arbitrary values are correct.

### 6.1.6 A filled model (fillme\_multitask\_nodes.lus)

Here is a possible way to fill the file above

```
node multitask__SlowThread(I : multitask__int) returns (O : multitask__int);
let
  O = I - 2 ;
tel;

node multitask__FastThread(I : multitask__int) returns (O : multitask__int);
let
  O = I * 2;
tel;
```

### 6.1.7 Constants (fillme\_multitask\_const.lus)

Constants

```
-- This file was generated by aadl2sync version 0.28.
-- ../../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--multitask multitask.aaxl2
-- on peouvou the 27/6/2007 at 16:54:17

const multitask__SlowThread_O_dft = 1 ;
const multitask__FastThread_O_dft = 0 ;
```

All constants are defined in this file. Default values are provided when the compiler is invoked with the "--gen-fake-body" option.

### 6.1.8 Data types (fillme\_multitask\_types.lus)

Data types

```
-- This file was generated by aadl2sync version 0.28.
-- ../../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--multitask multitask.aaxl
-- on peouvou the 29/6/2007 at 14:50:11

{type} multitask__int = int; -- XXX Fix me?
```

Data types are defined in this file. Here, the compiler guessed it was an integer because "int" is a sub-string of "multitask\_\_int". This guess is only done if the compiler is invoked with the "--gen-fake-body" option.

## 6.2 Two processors

Here is an AADL program that illustrates the translation of a model with several processors.

We only show the scheduler, as the other generated files are very similar to the ones of the first example. One can have a look at all the generated files in the `examples/twoproc` directory of the distribution after launching the "make demo" command.

The `twoproc.aadl` file

```

system twoproc
end twoproc;
-----
system implementation twoproc.IMPL
subcomponents
  t1 : device A_Sensor;
  t2 : device A_Sensor;
  cmd1_1 : device An_Actuator;
  cmd1_2 : device An_Actuator;
  cmd1_3 : device An_Actuator;
  cmd2_1 : device An_Actuator;
  cmd2_2 : device An_Actuator;
  cmd2_3 : device An_Actuator;

  processor1 : processor A_Processor.IMPL;
  processor2 : processor A_Processor.IMPL;
  process1 : process A_Process.IMPL;
  process2 : process A_Process.IMPL;

  -- so that the 2 processes can communicate
  System_bus : bus A_Bus;
  memo : memory Some_Memory;

connections
  bus access System_bus -> processor1.proc_bus;
  bus access System_bus -> processor2.proc_bus;
  bus access System_bus -> memo.mem;

  data port process1.O1 -> cmd1_1.I;
  data port process1.O2 -> cmd1_2.I;
  data port process1.O3 -> cmd1_3.I;
  data port process2.O1 -> cmd2_1.I;
  data port process2.O2 -> cmd2_2.I;
  data port process2.O3 -> cmd2_3.I;
  data port t1.O -> process1.I;
  data port t2.O -> process2.I;

properties
  Actual_Processor_Binding =>
    reference processor1 applies to process1;
  Actual_Processor_Binding =>
    reference processor2 applies to process2;
end twoproc.IMPL;
-----
Data int end int;
-----
device A_Sensor
features
  O : out data port int;
end A_Sensor;
-----
device An_Actuator
features
  I : in data port int;
end An_Actuator;
-----
bus A_Bus end A_Bus;
-----
memory Some_Memory
features
  mem : requires bus access A_Bus;
end Some_Memory;
-----
processor A_Processor
features
  proc_bus : requires bus access A_Bus;
end A_Processor;
processor implementation A_Processor.IMPL
end A_Processor.IMPL;
-----
process A_Process
features
  I : in data port int;
  O1 : out data port int;
  O2 : out data port int;
  O3 : out data port int;
end A_Process;

process implementation A_Process.IMPL
subcomponents
  thr1 : thread Thread1;
  thr2 : thread Thread2.IMPL;
connections
  data port I -> thr1.I;
  data port I -> thr2.I1;
  data port thr1.O -> thr2.I2;
  data port thr1.O -> O1;
  data port thr2.O1 -> O2;
  data port thr2.O2 -> O3;
end A_Process.IMPL;
-----
thread Thread1
features
  I : in data port int;
  O : out data port int;
properties
  Dispatch_Protocol => Periodic;
  Period => 100 ms;
  Compute_Execution_Time => 10 ms .. 30 ms;
end Thread1;
-----
thread Thread2
features
  I1 : in data port int;
  I2 : in data port int;
  O1 : out data port int;
  O2 : out data port int;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end Thread2;

thread implementation Thread2.IMPL
calls {
  add : subprogram ADD;
  mult : subprogram MULT;
};
connections
  parameter I1 -> add.I1;
  parameter I2 -> add.I2;
  parameter I1 -> mult.I1;
  parameter I2 -> mult.I2;
  parameter add.O -> O1;
  parameter mult.O -> O2;
end Thread2.IMPL;
-----
subprogram ADD
features
  I1 : in parameter int;
  I2 : in parameter int;
  O : out parameter int;
properties
  Compute_Execution_Time => 1 ms .. 5 ms;
end ADD;
-----
subprogram MULT
features
  i1 : in parameter int;
  i2 : in parameter int;
  O : out parameter int;
properties
  Compute_Execution_Time => 1 ms .. 5 ms;
end MULT;

```

## The scheduler

```

-- This file was generated by aadl2sync version {0.27}.
-- ../bin/aadl2sync --show-scheduling-vars --gen-fake-body -o
--twoproc twoproc.aaxl
-- on peouyou the 12/6/2007 at 11:19:31

include "fillme_twoproc_types.lus"
include "fillme_twoproc_const.lus"

-- We skip definitions provided in the previous example
-- timer definition skipped ...
-- rising_edge definition skipped ...
-- first_rising_edge definition skipped ...
-- consume definition skipped ...

-- Compute the relative advance of clk1 w.r.t. clk2
node compute_advance(clk1, clk2 : bool) returns (c : int);
var pc : int;
let
  pc = 0 -> pre c;
  c = if clk2 then 0 -- reset the advance of clk1
      else if clk1 then pc + 1 -- clk1 got ahead of one tic
      else pc;
tel

-- A fix priority scheduler for 3 threads.
node cpu_from_needs_3(qs_tick, needs1, needs2, needs3 : bool)
returns (cpu1, cpu2, cpu3 : bool);
let
  cpu1 = qs_tick and needs1;
  cpu2 = qs_tick and needs2 and not cpu1;
  cpu3 = qs_tick and needs3 and not cpu1 and not cpu2;
tel

node schedule_processor1_threads(
  in_min_max1, in_min_max2, in_min_max3 : int;
  qs_tick : bool)
returns (
  _add_activate_clk, _add_release_clk : bool;
  _mult_activate_clk, _mult_release_clk : bool;
  _thrl_activate_clk, _thrl_release_clk : bool;
  needs1, needs2, needs3 : bool;
  cpu1, cpu2, cpu3 : bool );
var
  _add_activate_clk_candidate : bool;
  _mult_activate_clk_candidate : bool;
  _thrl_activate_clk_candidate : bool;
  next_needs1, next_needs2, next_needs3 : bool;
let
  _add_activate_clk_candidate = timer(20, qs_tick);
  _mult_activate_clk_candidate = _add_release_clk;
  _thrl_activate_clk_candidate = timer(100, qs_tick);
  next_needs1, _add_release_clk = consume(
    in_min_max1, _add_activate_clk_candidate, cpu1);
  needs1 = false -> pre next_needs1;
  next_needs2, _mult_release_clk = consume(
    in_min_max2, _mult_activate_clk_candidate, cpu2);
  needs2 = false -> pre next_needs2;
  next_needs3, _thrl_release_clk = consume(
    in_min_max3, _thrl_activate_clk_candidate, cpu3);
  needs3 = false -> pre next_needs3;
  cpu1, cpu2, cpu3 = cpu_from_needs_3(qs_tick, needs1, needs2, needs3);

-- We read inputs when the thread starts having the cpu
_add_activate_clk = first_rising_edge(cpu1, not needs1);
_mult_activate_clk = first_rising_edge(cpu2, not needs2);
_thrl_activate_clk = first_rising_edge(cpu3, not needs3);
tel

node schedule_processor2_threads(
  in_min_max1, in_min_max2, in_min_max3 : int;
  qs_tick : bool)
returns (
  _add_activate_clk_2, _add_release_clk_2 : bool;
  _mult_activate_clk_2, _mult_release_clk_2 : bool;
  _thrl_activate_clk_2, _thrl_release_clk_2 : bool;
  needs1, needs2, needs3 : bool;
  cpu1, cpu2, cpu3 : bool );
var
  _add_activate_clk_2_candidate : bool;
  _mult_activate_clk_2_candidate : bool;
  _thrl_activate_clk_2_candidate : bool;
  next_needs1, next_needs2, next_needs3 : bool;
let
  _add_activate_clk_2_candidate = timer(20, qs_tick);
  _mult_activate_clk_2_candidate = _add_release_clk_2;
  _thrl_activate_clk_2_candidate = timer(100, qs_tick);
  next_needs1, _add_release_clk_2 = consume(
    in_min_max1, _add_activate_clk_2_candidate, cpu1);
  needs1 = false -> pre next_needs1;
  next_needs2, _mult_release_clk_2 = consume(
    in_min_max2, _mult_activate_clk_2_candidate, cpu2);
  needs2 = false -> pre next_needs2;
  next_needs3, _thrl_release_clk_2 = consume(
    in_min_max3, _thrl_activate_clk_2_candidate, cpu3);
  needs3 = false -> pre next_needs3;
  cpu1, cpu2, cpu3 = cpu_from_needs_3(qs_tick, needs1, needs2, needs3);

-- We read inputs when the thread starts having the cpu
_add_activate_clk_2 = first_rising_edge(cpu1, not needs1);
_mult_activate_clk_2 = first_rising_edge(cpu2, not needs2);
_thrl_activate_clk_2 = first_rising_edge(cpu3, not needs3);
tel

-- A quasi synchronous scheduler for 2 processors.
node quasi_synchronous_scheduler(alea1, alea2 : bool)
returns (qs_1, qs_2 : bool);
var
  advance_max_is_reached_1_2 : bool;
  advance_max_is_reached_2_1 : bool;
  problems_1_2 : bool;
  problems_2_1 : bool;
  filter_1 : bool;
  filter_2 : bool;
  advance_1_2 : int;
  advance_2_1 : int;
  padvance_1_2 : int;
  padvance_2_1 : int;
let
  -- We compute the relative advance of various processors
  advance_1_2 = compute_advance(qs_1, qs_2);
  advance_2_1 = compute_advance(qs_2, qs_1);

  padvance_1_2 = 0 -> pre advance_1_2;
  padvance_2_1 = 0 -> pre advance_2_1;
  -- Is the maximum advance reached?
  advance_max_is_reached_1_2 = (padvance_1_2 >= 2);
  advance_max_is_reached_2_1 = (padvance_2_1 >= 2);

  -- there is a problem if the maximum advance is reached
  -- and no tic occurs
  problems_1_2 = advance_max_is_reached_1_2 and alea1;
  problems_2_1 = advance_max_is_reached_2_1 and alea2;

  -- we force the tic for clocks that are d tics late
  filter_1 = (not problems_1_2);
  filter_2 = (not problems_2_1) and true;
  qs_1 = alea1 and filter_1;
  qs_2 = alea2 and filter_2;
tel

node schedule_twoproc(
  in_min_max11, in_min_max12, in_min_max13 : int;
  in_min_max21, in_min_max22, in_min_max23 : int;
  alea1, alea2 : bool) returns (
  _add_activate_clk : bool;
  _add_activate_clk_2 : bool;
  _add_release_clk : bool;
  _add_release_clk_2 : bool;
  _mult_activate_clk : bool;
  _mult_activate_clk_2 : bool;
  _mult_release_clk : bool;
  _mult_release_clk_2 : bool;
  _thrl_activate_clk : bool;
  _thrl_activate_clk_2 : bool;
  _thrl_release_clk : bool;
  _thrl_release_clk_2 : bool;
  qs_tick1, qs_tick2 : bool;
  needs11, needs12, needs13 : bool;
  cpu11, cpu12, cpu13 : bool;
  needs21, needs22, needs23 : bool;
  cpu21, cpu22, cpu23 : bool);
let
  _add_activate_clk,
  _add_release_clk,
  _mult_activate_clk,
  _mult_release_clk,
  _thrl_activate_clk,
  _thrl_release_clk,
  needs11, needs12, needs13,
  cpu11, cpu12, cpu13
  = schedule_processor1_threads(
    in_min_max11, in_min_max12, in_min_max13,
    qs_tick1
  );
  _add_activate_clk_2,
  _add_release_clk_2,
  _mult_activate_clk_2,
  _mult_release_clk_2,
  _thrl_activate_clk_2,
  _thrl_release_clk_2,
  needs21, needs22, needs23,
  cpu21, cpu22, cpu23
  = schedule_processor2_threads(
    in_min_max21, in_min_max22, in_min_max23, qs_tick2);
  qs_tick1, qs_tick2 = quasi_synchronous_scheduler(alea1, alea2);
tel

-- to be used for formal verification
node schedule_twoproc_oracles_properties(
  in_min_max11, in_min_max12, in_min_max13 : int;
  in_min_max21, in_min_max22, in_min_max23 : int;
  alea1, alea2 : bool)
returns (ok:bool);
let
  ok =
    ( 1 <= in_min_max11 and in_min_max11 <= 5) and
    ( 1 <= in_min_max12 and in_min_max12 <= 5) and
    ( 10 <= in_min_max13 and in_min_max13 <= 30) and
    ( 1 <= in_min_max21 and in_min_max21 <= 5) and
    ( 1 <= in_min_max22 and in_min_max22 <= 5) and
    ( 10 <= in_min_max23 and in_min_max23 <= 30) and
    true;
tel

```

## 7 A test session using Lurette

## 8 A formal proof session using Lesar

## 9 The currently supported AADL subset

Not all AADL concepts are supported. Here is an (hopefully) exhaustive list of concepts that are supported and not supported by the current version of aadl2sync.

As the terms of components and features have a precise meaning in AADL, we use in the following the (hazy) term of concept to distinguish what is and what is not supported by the tool.

### 9.1 Ignored concepts

Some concepts are ignored in the translation (which means that they can appear in the AADL model without breaking the tool):

- processors (the fact that several processes run on the same processor is taken into account though);
- memory;
- buses †;
- flows;
- most properties (except the ones cited above) †;
- the base literal for integers (base 10 is assumed) †;
- the time unit (i.e., one should not use different time units) †.

Among those ignored concepts, note that taking into account processors, memory, flows, and most properties would definitely be useless for our purposes (validating the functional part of the model via simulation and formal verification). Concerning the other items (flagged by a †) in the contrary, some could be taken into account, at least partially, in future versions of the tool.

### 9.2 Not supported concepts

The following concepts could prevent the translation to occur if used in the AADL model:

- port groups mixing input and output ports (`in_out`) †;
- systems as sub-component of systems †;
- thread groups †;
- modes †;
- `sporadic`, `aperiodic` and `background` threads; execution fault handling in threads †;
- event ports: since event (data) ports are used to trigger non periodoc thread or or to cause a mode switch, it would be meaningless to take them into account yet †;
- subprograms as features (remote procedure calls) †;
- subcomponent accesses (outside their containment hierarchy) †.

All those concepts could be supported by the tool one day.

### 9.3 Supported concepts

The translation of those concepts was described in Section ???. We simply recall them here for the sake of completeness.

- systems;
- devices;
- processes;
- periodic and sporadic threads;
- sub-programs, sub-programs calls, sub-program parameters;
- data ;
- ports and port groups;
- some properties:
  - `compute_exec_time`, an interval associated to threads and subprograms;
  - `Period`, associated to processes and threads
  - `dispatch_protocol`, which must be equal to `Periodic` or `Sporadic`.

### 9.4 Some design decisions

- If a component implementation exists, no node is generated for the corresponding component Type.
- Devices are not translated into a Lustre node as other components; instead, their inputs are considered as system outputs, and their outputs are considered as system inputs.
- In AADL, it is possible to have several wires that comes into the same port. The AADL standard says that the messages are queued in such a case. For Data ports, I don't know what it means. Probably it is only allowed for event ports in AADL. Anyway, we reject such programs (when several wires target the same data port). For event port, we perform a logical and between the several wires (event port = Boolean variable).
- A bus is translated into a unit delay. That modelisation could be refined.

### 9.5 Known limitations

- Only Ms are taken into account as time unit.
- The inverse of a port group should be defined after the port group.
- The character '>' in comment kills the xml parser (e.g., if you comment a line like “Period => 15 ms;” ...). This is due to a bug in the OSATE eclipse plugin which ought to escape the '>' character (i.e., it should generate '&gt;' instead of '>').
- the `.xsd` files directory is wrong in the `axl` (cf `fix_axl` rule in `examples/Makefile`)

## A A quasi-synchronous clock generator/acceptor in Lustre V4

We give below a Lustre V4 implementation of the quasi-synchronous clock generator/acceptor discussed in Section 4.1. 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.

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

- `n` is the number of clocks to generate;
- `d` is the maximal authorized clock drift, i.e., the maximum number of tics authorized for the other clocks between 2 tics 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` tics late, that are forced to false.

#### A quasi-synchronous clock generator/acceptor in Lustre V4

```
-- Compute the relative advance of clk1 w.r.t. clk2
node compute_advance(clk1, clk2 : bool) returns (c : int);
var pc : int;
let
  pc = 0 -> pre c;
  c = if clk2 then 0 -- reset the advance of clk1
      else if clk1 then pc + 1 -- clk1 got ahead of one tic
      else pc;
tel

-- Create an array of size n filled with x
node ba_fill(const n: int; x : bool) returns (t: bool^n);
let
  t = x^n;
tel

-- Check if all elements of the array are false
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

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 tic occurs
  problems = advance_max_is_reached and not alea^n;
  -- we force the tic for clocks that would be more than d tics 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.

Note that this program is used if **aadl2sync** is launched with the `--v4-arrays` options. Otherwise, it is an array-expanded version of it (less readable) that is generated.

## References

- [1] 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. 1, 2

- [2] 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](#)
- [3] 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. [1](#)
- [4] 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. [1](#)
- [5] E. Jahier and P. Raymond. The Lucky Language Reference Manual. Technical Report TR-2004-6, Verimag, 2004. [www-verimag.imag.fr/~synchron/tools.html](http://www-verimag.imag.fr/~synchron/tools.html). [6.1.3](#)
- [6] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983. [1](#)
- [7] SAE. Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace, November 2004. [1](#)