Virtual prototyping AADL architectures in a polychronous model of computation

Yue Ma, Jean-Pierre Talpin, Thierry Gautier

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Virtual prototyping AADL architectures in a polychronous model of computation

Yue Ma — Jean-Pierre Talpin — Thierry Gautier

N° 6479

June 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6479--FR+ENG

Virtual prototyping AADL architectures in a polychronous model of computation *

Yue Ma, Jean-Pierre Talpin, Thierry Gautier

Thème COM — Systèmes communicants
Projets Espresso

Rapport de recherche n° 6479 — June 2007 — 22 pages

Abstract: While synchrony and asynchrony are two distinct concepts of concurrency theory, effective and formally defined embedded system design methodologies usually mix the best from both synchronous and asynchronous worlds by considering locally synchronous processes composed in a globally asynchronous way to form so called GALS architectures. In the avionics domain, for instance, the Architecture Analysis and Design Language (AADL) may be used to describe both the hardware and software architecture of an application at system-level. Yet, a synchronous design formalism might be preferred to model and validate each of the critical components of the architecture in isolation. In this paper, we illustrate the use of the polychronous (multi-clocked synchronous) paradigm to model partially asynchronous applications. The specification formalism SIGNAL is used to describe real-world avionic applications using concepts of Integrated Modular Avionics (IMA). We show how an AADL architecture can be automatically translated into a synchronous model in SIGNAL using these modeling concepts. We present a case study on the design of generic system architecture. The approach is being implemented in the framework of the ANR project TopCased.

Key-words: formal methods, embedded systems, program analysis, synchronous paradigm

* This work is partially supported by the ANR project TopCased

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Prototypage virtuel d'architectures AADL dans un modèle de calcul synchrone [†]

Résumé : Nous utilisons le modèle de calcul polychrone pour le prototypage virtuel de spécifications AADL (Architecture Analysis and Design Language) en utilisant principalement le concept d'IMA (Integrated Modular Avionics) implanté dans l'atelier Polychrony au moyen d'une libraire mettant en oeuvre les service du système d'exploitation temps-réel ARINC-653. Nous montrons comment traduire automatiquement une spécification AADL par un programme synchrone utilisant ces services. Nous illustrons notre approche par quelques exemples. Ce travail est mis en oeuvre dans le cadre du projet ANR Topcased.

Mots-clés : méthodes formelles, systèmes embarqués, analyse de programmes, paradigme synchrone

[†] Travaux supportés par le projet ANR TopCased

1 Introduction

It is well admitted, embedded systems affect most aspects of our everyday lives. New development frameworks that allow designers to perform efficient exploration of design alternatives and analyze system properties early in the design cycle are commonly needed. Several proposals for Model-driven development of embedded systems have been defined, see [1, 2]. However, these modeling principles for architectural modeling of large embedded systems do not have a universal recognition. An important recent development in this respect is the emergence of AADL.

Architecture Analysis and Design Language (AADL [3]) is a standard for providing formal modeling concepts for the description and analysis of application system architecture in terms of distinct components and their interactions. It provides support to hierarchically describe how software components are mapped onto computational hardware elements of the execution platform. The modeling aspect of system design activity is becoming increasingly essential, since it allows prototyping and experiments without necessarily having a physical implementation of the system at hands. Meanwhile, component-based approaches provide a way to significantly reduce overall development costs through modularity and re-usability.

For the early validation and testing, some methods for the modeling and automated translation of those complex architectures into synchronous languages [4] have been proposed. For instance, the paradigm of "Globally asynchronous locally synchronous system" (GALS [5]) has been proposed to describe general asynchronous systems, while keeping as much as possible the advantages of synchronous components. Other approaches are proposed to program separately software components using synchronous languages, and deploy them to the target architecture using classical design methods for asynchronous systems. In such cases, the separate development of the architecture and software makes it difficult to validate the integrated system.

In order to support the virtual prototyping, simulation and formal validation of early, component-based, embedded architectures, we define a model of the AADL into the polychronous model of computation of the SIGNAL programming language [6]. TopCased [7] is a large open-source project devoted to the design of critical embedded systems. In the TopCased process, several meta-models are proposed, including those for describing architectures in AADL and those for modeling synchronous components. In this framework, we propose a methodology to describe asynchrony using a synchronous multi-clocked formalism.

The main difficulty in this translation is to model intrinsically non-deterministic and asynchronous AADL descriptions into a polychronous model. We challenge this difficulty by using existing techniques and library of the SIGNAL environment, consisting of a model of the APEX-ARINC-653 real-time operating system services. It proves a suitable and adequate library to model embedded architectures in the specific case of Integrated Modular Avionics (IMA [8]) considered in the TopCased project.

General principles In this article, we describe the general principles of the translation for each AADL component. The main one is to use APEX-ARINC services. ARINC 653 (Avionics Application Standard Software Interface [11]) is a standard that specifies an API for software of avionics, following the architecutre of Integrated Modular Avionics. It defines an APplication EXecutive (APEX) for space and time partitioning. An ARINC PARTITION is a logical allocation unit resulting from a functional decomposition of the system. PARTITIONs are composed of PROCESSes (to distinguish from the AADL process and SIGNAL *process*) that represent the executive units. AADL components are mapped onto ARINC PARTITIONs. Each component corresponds to some instances of APEX services (Figure 1). The scheduler and communication of the AADL components are also translated to SIGNAL *processes* using some improved APEX services.

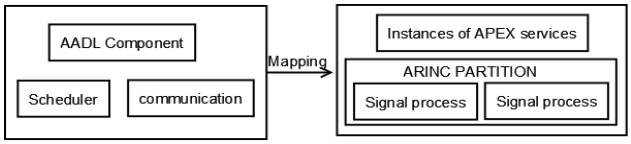


Figure 1: Architecture of modeling the AADL system using APEX services

Plan The paper is organized as follows. The following three sections recall a basic introduction of AADL, IMA architecture and SIGNAL. An AADL example is described in section 2. Section 5 explains the general principles of the translation for each AADL component. It is illustrated with the translation in SIGNAL of components used in our example, using the SIGNAL library of APEX services. In section 6, we state some related works. Finally, we summarize the extent of our contribution to virtual prototyping AADL, as well as the current limitations of our approach, and draw conclusions in section 7.

2 A Summary of AADL

The Architecture Analysis and Design Language (AADL) is a SAE standard aimed at the high level design and evaluation of the architecture of embedded systems. The language is used to describe the structure of such systems as an assembly of software components that are mapped to an execution platform. The purpose of a model in AADL is to describe the execution characteristics of the system. Because such characteristics depend on the hardware executing the software, an AADL model includes the description of both software and hardware.

AADL focuses on the description of systems using the component-based paradigm. A sample client-server AADL system is presented in Figure 2. The client sends a signal to the server, which in turn sends a message containing data. The server is made up of two sensors linked to a processor, which performs a certain number of calculations and sends the result to the client when it demands. In this graphical representation, all the basic components are depicted.

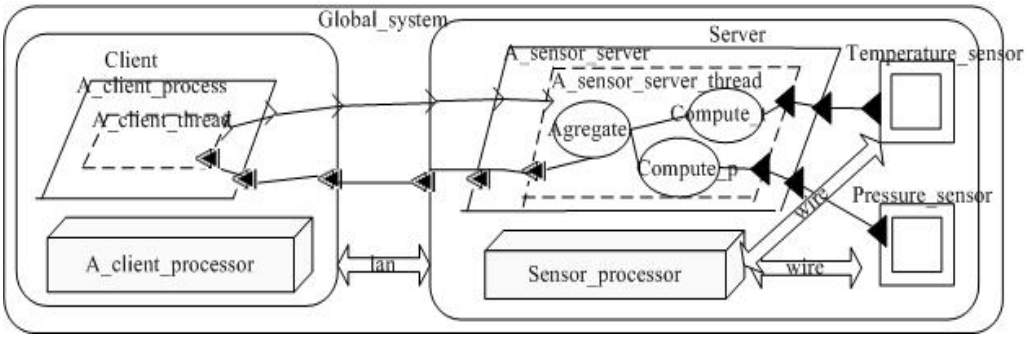


Figure 2: An AADL client-server model

Component categories AADL components are separated into three categories: composite components, application software components and execution platform components.

Composite components model components consisting of both hardware and software. A **system** component models a component containing execution platform, application software and other composite components. The **Global_system** in Figure 2 contains two subsystems: **Client** and **Server**.

Application software components include process, thread, thread group, subprogram, and data components. A **process** component models a protected virtual address space, and it contains at least one thread. A **thread** component is an abstraction of a schedulable unit of concurrent execution. A **subprogram** component models a procedure call as in imperative programming languages. The **A_sensor_server** process of **Server** subsystem in Figure 2 contains a thread **A_sensor_server_thread**, and this thread calls three subprograms for computation: **Compute_t, Compute_p** and **Agregate**.

Execution platform components model the hardware part of the system, and it includes the processor, memory, device, and bus components. A **processor** component is an abstraction of hardware and possibly embedded software that schedules and executes threads. It may contain memory, and can access memory or device components through a bus component. A **device** component is an abstraction for a component with complex behavior that interfaces with and represents a part of the external environment. A **bus** component is an abstraction for an execution platform component which provides communication of data and event messages between processor, memory and device components. In Figure 2, **Sensor_processor** is a processor, and it represents a hardware central processing unit. **Temperature_sensor** and **Pressure_sensor** are two devices, they communicate with **Sensor_processor** processor via **wire** bus.

Component type and implementation Each component is described in AADL with two parts. The first one, the *type*, represents the functional interface of the component and externally observable attributes, what is visible by other components. The second one, the *implementation*, describes the contents of the component, as well as the connections between them [10]. Each *type* may be associated with zero, one or more *implementation(s)*.

Properties A **property** provides information about component types, implementations, subcomponents, features, connections, flows, and subprogram calls.

Connections Components can be connected and bound to each other in a number of manners. A **connection** is a linkage between component features that represents the communication of data and control between components.

3 IMA Architecture

The APEX interface, defined in the ARINC standard [11], provides an avionics application software with the set of basic services to access the operating-system and other system-specific resources. Its definition relies on the Integrated Modular Avionics (IMA) architecture. A main feature in an IMA architecture is that several avionics applications can be hosted on a single, shared computer system (see Figure 3). This is addressed through a functional partitioning of the applications with respect to available time and memory resources [12]. The allocation unit that results from this decomposition is the PARTITION.

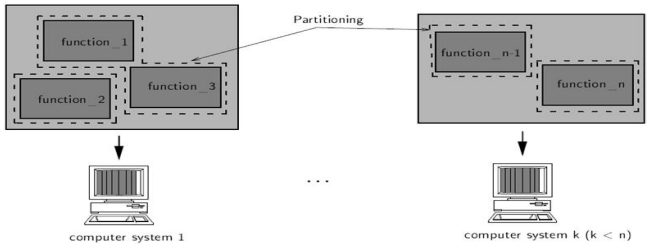


Figure 3: IMA: different functions can share a fault-tolerant computer

A processor is allocated to each PARTITION for a fixed time window within a major time frame maintained by the *module-level OS*. A PARTITION is composed of PROCESSes which represent the executive units. When a PARTITION is activated, its owned PROCESSes run concurrently to perform the functions associated with the PARTITION. Each ARINC PROCESS is uniquely characterized by information useful to the *partition-level OS*, which is responsible for the correct execution of PROCESSes within a PARTITION. Suitable mechanisms and devices are provided for communication and synchronization between PROCESSes (e.g. *buffer, event, semaphore*) and PARTITIONs (e.g. *ports* and *channels*).

The APEX interface allows IMA applications to access the underlying OS functionalities. The interface includes both services to achieve communications and synchronizations, and services for the management of PROCESSes and PARTITIONs.

4 The SIGNAL Language

4.1 Language features

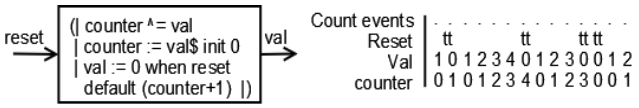
SIGNAL is a dataflow relational language that relies on the polychronous model [13]. It handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as `x` and implicitly indexed by discrete time. At any instant, a *signal* may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. The set of instants where a *signal* `x` is present represents its *clock*, noted $\hat{x}$. A SIGNAL *process* (to distinguish from the AADL process and ARINC PROCESS) is a system of equations over *signals* that specifies relations between values and clocks of the *signals*. A *program* is a *process*. SIGNAL relies on six primitive constructs that define *elementary processes*:

- *Relations.* `y:= f(x1,...,xn)` $\stackrel{def}{\equiv}$ $\forall t$: $(y_t = \perp \Leftrightarrow \forall i\ xi_t = \perp \wedge \exists i\ xi_t = \perp \Rightarrow \forall i\ xi_t = \perp \wedge \exists xi_t \neq \perp \Rightarrow y_t = f(x1_t, ..., xn_t))$
- *Delay.* `y:= x $ 1 init c` $\stackrel{def}{\equiv}$ $\forall t > 0, x_t \neq \perp \Leftrightarrow y_t \neq \perp \wedge x_t \neq \perp \Rightarrow y_t = x_{t-1}, y_0 = c.$
- *Undersampling.* `y:= x when b` where `b` is Boolean $\stackrel{def}{\equiv}$ $y_t = x_t$ if $b_t = true$, else $y_t = \perp$. The expression `y:= when b` is equivalent to `y:= b when b`.
- *Deterministic merging.* `z:= x default y` $\stackrel{def}{\equiv}$ $z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
- *Composition.* `P1|P2` $\stackrel{def}{\equiv}$ conjunction of equations of `P1` and `P2`.
- *Hiding.* `P where x` $\stackrel{def}{\equiv}$ `x` is local to the *process* `P`.

Derived operators are defined from the kernel of primitive operators, for example:
Clock extraction: `h := ^x` specifies the clock `h` of `x`, and can be defined as: `h := (x = x)`.
Synchronization: `x1 ^= x2` specifies that `x1` and `x2` have the same clock, and is defined as: `(| h := (^x1 = ^x2) |) where h`.
Clock union: `h:= x1 ^+ x2` specifies the clock union of `x1` and `x2`, which is also defined as: `h:= ^x1 default ^x2`.
Memory: `y:= x cell b init y0` allows to memorize in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. It is defined as:
`(| y := x default (y$1 init y0) | y ^= x ^+ (when b) |).`

SIGNAL offers a *process* frame that enables the definition of *sub-processes* (declared in the `where` scope). *Sub-processes* that are only specified by an interface without internal behavior are considered as external (e.g. C++ / JAVA functions), and may be separately compiled *processes* or physical components. Any *process* can be abstracted by an interface which specifies properties on its input-output *signals*. These properties essentially concern clock relations and dependencies between *signals*. All these features favor modularity and reusability.

Example (counter) We consider the definition of a counter: **Count**. It accepts an input **reset** *signal* and delivers the integer output *signal* **val**. The local variable **counter** is initialized to 0 and stores the previous value of the *signal* **val** (equation **counter := val$ init 0**). When an input **reset** occurs, the signal **val** is reset to 0 (expression (**0 when reset**)). Otherwise, the *signal* **val** takes an increment of the variable **counter** (expression (**counter+1**)). The activity of **Count** is governed by the clock of its output **val** which has higher frequency than its input **reset**.



SIGNAL is associated with a design environment, called POLYCHRONY [13], which offers a graphical user interface, a compiler and a model-checker that support the trustworthy design of systems.

4.2 Modeling of ARINC concepts in SIGNAL

The POLYCHRONY design environment includes a library in SIGNAL containing real-time executive services defined by ARINC [11]. It relies on a few basic blocks [14], which allow to model PARTITIONS: APEX-ARINC 653 services, an RTOS model and executive entities.

APEX services The APEX services modeled in SIGNAL include communication and synchronization services used by PROCESSes (e.g. *SEND_BUFFER*, *WAIT_EVENT*, *READ_BLACKBOARD*), PROCESS management services (e.g. *START*, *RESUME*), PARTITION management services (e.g. *SET_PARTITION_MODE*), and time management services (e.g. *PERIODIC_WAIT*).

PARTITION-level OS The role of the PARTITION-level OS is to ensure the correct concurrent execution of PROCESSes within the PARTITION (each PROCESS must have exclusive control on the processor). A sample model of the PARTITION-level OS is depicted in Figure 4.

Figure 4: Interface of the PARTITION-level OS model

In Figure 4, the input `Active_partition_ID` represents the identifier of the running PARTITION selected by the module-level OS, and it denotes an execution order when it identifies the current PARTITION. Whenever the PARTITION executes, the `PARTITION_LEVEL_OS` selects an active PROCESS within the PARTITION. The PROCESS is identified by the value carried by the output signal `Active_process_ID`, which is sent to each PROCESS. The other input/output signals can be referenced in [15].

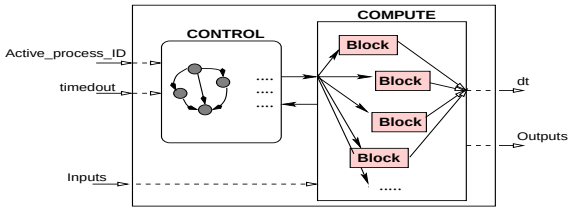


Figure 5: ARINC PROCESS model.

ARINC PROCESSES The definition of an ARINC PROCESS model basically takes into account its computation and control parts. This is depicted in Figure 5. Two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*. Any PROCESS is seen as a reactive component, which reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timedout` notifies PROCESSes of time-out expiration. In addition, there are other inputs (resp. outputs) needed for (resp. produced by) the PROCESS computations. The CONTROL and COMPUTE sub-components cooperate to achieve the correct execution of the PROCESS model.

The CONTROL sub-component specifies the control part of the PROCESS. Basically, it is a transition system that indicates which statements should be executed when the PROCESS model reacts. Whenever the input `Active_process_ID` identifies the ARINC PROCESS, this PROCESS "executes". Depending on the current state of the transition system representing the execution flow of the PROCESS, a *block* of actions in the COMPUTE sub-component is selected to be executed *instantaneously*.

The COMPUTE sub-component describes the actions computed by the PROCESS. It is composed of *blocks* of actions. They represent elementary pieces of code to be executed without interruption. The statements associated with a *block* are assumed to *complete within a bounded amount of time*. In the model, a *block* is executed instantaneously.

PARTITIONS After the initialization phase, the PARTITION gets activated (i.e. when receiving *Active_partition_ID*). The PARTITION-level OS selects an active PROCESS within the PARTITION. Then, the CONTROL subpart of each PROCESS checks whether or not the concerned PROCESS can execute. In the case a PROCESS is designated by the OS, this action is performed: the PROCESS executes a *block* from its COMPUTE subpart, and the duration corresponding to the executed *block* is returned to the PARTITION-level OS in order to update time counters. The execution of the model of the PARTITION follows this basic pattern until the *module-level OS* selects a new PARTITION to execute.

The subclauses of an ARINC SIGNAL system declaration can be summarized as follows:

```
SYSTEM ::= process defining_system_identifier =
    DEFINITION_OF_INTERFACE
    {PARTITION}+
    {MODULE_LEVEL_OS}
    [PARTITION_INTERACTION(sampling_port | queuing_port)] end;
DEFINITION_OF_INTERFACE::= [PARAMETER][INPUTS][OUPUTS]
PARAMETER::={type parameter_identifer;}*
INPUTS::=?{type input_identifier;}*
```

```
OUTPUTS::=!{type output_identifier;}*
PARTITION::= process defining_partition_identifier =
    DEFINITION_OF_INTERFACE
    {PROCESS}+
    {PARTITION_LEVEL_OS}

    [GLOBAL_OBJECTS(buffer|blackboard|semaphore)] end;
PROCESS::= process defining_process_identifier =
    DEFINITION_OF_INTERFACE
    {CONTROL}

    {COMPUTE} end;
CONTROL::= process defining_control_identifier =
    DEFINITION_OF_INTERFACE

    {CONTROL_BODY} end;
COMPUTE::= process defining_compute_identifier =
    DEFINITION_OF_INTERFACE

    {BLOCK}+ end;
```

5 Mapping AADL component to SIGNAL *process*

Here we present general rules to translate AADL systems into the SIGNAL programming language. We put our translation to work by studying the similarity relationship between AADL and APEX-ARINC services.

An AADL system model describes the architecture and runtime environment of an application system in terms of its constituent software and execution platform components and their interactions. In the following, we present the translation rules from four main categories: system, software components, hardware components and component interactions. For each category, we select some classical components. And for each component, we present a general mapping rule, show how its corresponding SIGNAL *process* is, then describe some details of the translation, and an example translation for the system presented in Figure 2 is given.

5.1 System

The system is the top-level component of the AADL model, It can be mapped into a top-level SIGNAL *process*.

General rules:

1. Each system can correspond to an ARINC PARTITION.
2. Each input (output) port of the system is mapped into an input(output) of the PARTITION.
3. For system implementations, each sub-component is mapped into a SIGNAL *process*, for example, an AADL process can be mapped as an ARINC PROCESS, a thread can be a *block*, and all the PROCESS and *block* can be modeled in SIGNAL as described in section 4.2.

4. The SIGNAL *process* calls result straightforwardly from their inner connections.

5. The connections between systems are implemented as the communications between the PARTITIONs, that are the *ports* and *channels* in APEX.

For instance, the client system (Figure 6 left) has one input port, one output port and two subcomponents. The corresponding SIGNAL model is a PARTITION (Figure 6 right):

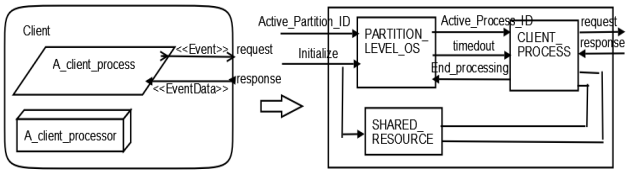


Figure 6: An AADL system to SIGNAL ARINC model mapping example

• The addtional two other inputs: **initialize** and **active_partition_ID** are generated by the PARTITION scheduler.

• The SHARED_RESOURCE includes **buff** and **sema**, used for the ARINC PROCESS communication.

Here an AADL system can only contain one processor, the case in which one system contains several processors is not considered yet. So that one AADL system can be mapped into one PARTITION.

5.2 Software components

Each software component (except data and thread group) is mapped into a SIGNAL *process* whose inputs/outputs are made of the component input/output ports. For component implementations, the SIGNAL *process* calls result from their inner connections.

5.2.1 Process

The AADL process component represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside the process.

Here we consider that the AADL processes executed on the same processor constitute a PARTITION (in the assumption that a system only has one processor), in other words, the processes in one system are mapped into one PARTITION.

General rules:

1. Each AADL process represents an ARINC PROCESS.

2. The input (output) ports of the AADL process become the inputs (outputs) of the PROCESS.

3. The AADL process is responsible for scheduling and for executing threads, while the CONTROL *process* schedules the *blocks* which are translated from the threads and sub-programs.

4. An AADL process must contain a thread, so the corresponding ARINC PROCESS has to contain a *block* in the COMPUTE sub-program.

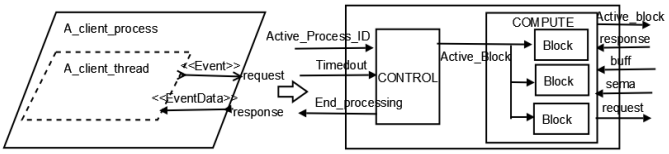


Figure 7: Process mapping example

Figure 7 is a simple example of AADL process mapping (the same example as described in section 2):

• The ARINC PROCESS attribute (process property in AADL) must be recorded in the PARTITION_LEVEL_OS which is the scheduler for the PROCESSes in the PARTITION.

• The CONTROL input **timedout** notifies PROCESSes of time-out expiration, and the other input **active_process_ID** notifies current active PROCESS which is scheduled by PARTITION_LEVEL_OS. The output **end_processing** is emitted by the PROCESS after completion, and **active_block** is transfered to the COMPUTE part to activate the corresponding *block*.

• There are other inputs needed for the ARINC PROCESS computations in actual programming.

• The input (output) ports of the AADL process component which correspond to the parent system inputs/outputs are translated as the ARINC PROCESS inputs (outputs); the other ports which are used for communication between AADL processes are not translated directly as PROCESS inputs (outputs), they can be translated as *buffer* or *blackboard* for the PROCESS communication, that will be represented in detail in component interaction section.

5.2.2 Thread

Thread component is an abstraction of software responsible for scheduling and for executing sub-programs.

When several threads run under the same AADL process, the sharing of the process is managed by a runtime scheduler. Threads are responsible for the subprogram execution, so the thread component can be translated as the execution of the ARINC PROCESS, that is the COMPUTE part of the PROCESS.

General rules:

1. The threads that belong to the same AADL process constitute the COMPUTE *process*.
2. Each thread can be a *block* or several *blocks* according to the subprograms it contains.
3. The inputs/outputs of COMPUTE correspond to the inputs/outputs of the parent PROCESS.
4. One more important input is needed: **active_block**, for activating the selected *block*.
5. Some communication services may be needed, in such case, more inputs will be added, like *port* and *buffer* names for identifying the communication scheme.

A generic interface of the SIGNAL *process* that specifies the COMPUTE sub-component mapping is given in Figure 8. Two *blocks* are made from the two subprograms. The *blocks* are scheduled by the CONTROL part.

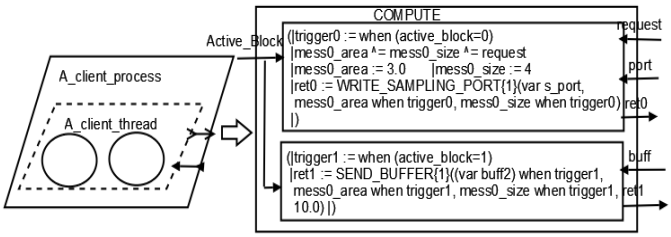


Figure 8: Generic interface of the COMPUTE sub-component

The **dispatch_protocol** property is used to specify the activation of a thread, it can be periodic/ aperiodic/ sporadic/ background. This must be recorded in the PARTITION_LEVEL_OS as an attribute of the parent process.

5.2.3 Subprogram

The subprogram is a callable component with or without parameters that operates on data or provides functions to components that call it. Subprogram components represent elementary pieces of code that processes inputs to produce outputs. Calls to subprograms are declared in call sequences in threads and subprogram implementations. Only their interfaces are given in the AADL model; subprogram implementations ought to be provided in some host language (such as C or Java).

The subprogram component can be mapped into a *block*, the code should be executed without interruption. The detailed implementation of the function can be programmed in C/JAVA language.

General rules:

1. Each subprogram becomes a *block* schema in SIGNAL. The *block* is part of the COMPUTE *process*.

2. Each *block* is identitied by a **BLOCK_ID**. Only when the current **active_block** equals to its **BLOCK_ID**, this *block* is executed.

3. Some *subprocess* may be needed for detailed computation of the execution of the subprogram.

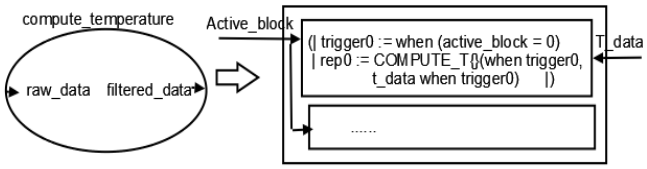


Figure 9: A simple subprogram mapping example

The same small example is used for subprogram translation (see Figure 9). Here **COMPUTE_T** processes the incoming data **T_data** when this *block* is triggered to produce some output data, the detailed output data producing is programmed in another SIGNAL *process*, which can be provided by a SIGNAL program or some C program.

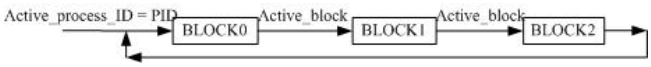


Figure 10: *Block* scheduler sequence

When a thread is made of several subprograms, they must be activated in sequence. In the SIGNAL library of ARINC services, the *blocks* are activated in sequence too. This is implemented by the CONTROL part. The *block* is activated only when the **active_block** equals to its **BLOCK_ID**. The **active_block** is computed in CONTROL, which is activated by the **active_process_ID** condition of the parent *process*, the ID value is increased each time the previous one is terminated, so that each *block* is executed in turn (see Figure 10).

5.3 Hardware components

Hardware components represent computational and interfacing resources within a system. Each hardware component can be mapped into a SIGNAL *process*, the translation is more intricate than software components. Here we consider some basic components for the translation. The device component is translated as an external interface, the processor as a scheduler, and the bus as a communication component.

5.3.1 Device

Device components are used to interface the AADL model with its environment. Devices are not translated as the other components, they are modeled outside the PARTITION, the implementation can be provided in some host language.

General rules:

1. The device can be a SIGNAL *process* outside the PARTITION.
2. The *process* inputs/outputs are mapped from the component input/output ports. The inputs are considered as PARTITION outputs, and the outputs as PARTITION inputs.

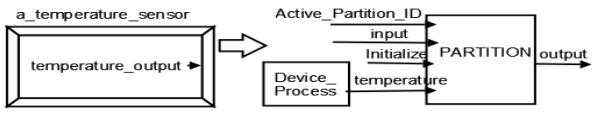


Figure 11: Device mapping example

Figure 11 is a device example which has one output data port **temperature_output**, in SIGNAL it becomes a *process* with an output **temperature**, and this output is transfered to the corresponding PARTITION as one input.

5.3.2 Processor

Processor component is an abstraction of hardware and software responsible for executing and scheduling processes. Basically, each processor has its own clock, which is the base time of the components running on the processor. Several processes or threads that run on the same processor have to share the resources such as CPU. The sharing is managed by a runtime scheduler.

The processor can be translated as the scheduler of the AADL processes which are bounded to the processor, corresponding to the PARTITION_LEVEL_OS in SIGNAL. The subclauses of PARTITION_LEVEL_OS declaration can be summarized as follows:

```
PARTITION_LEVEL_OS ::= process defining_PLOS_identifier =
            DEFINITION_OF_INTERFACE
            {PROCESS_CREATION}+
            {PROCESS_START}+
            {PROCESS_SCHEDULINGREQUEST}
            {PROCESS_GETACTIVE}
            {UPDATE_COUNTERS}

            {SUSPEND_SELF} end;
PROCESS_CREATION ::=
   process defining_PROCESS_CREATION_identifier =
        DEFINITION_OF_INTERFACE
        {PROCESS_RECORDING}+

        {ATTRIBUTE_RECORDING}+ end;
```

General rules:

1. The processor is translated as a PARTITION_LEVEL_OS of the PARTITION.
2. All the ARINC PROCESSes must be recorded and created in it.

3. When the PARTITION is activated, the PROCESS scheduling starts, a priority preemprive scheduling policy is provided.

4. The PROCESS will suspend when it finishes.

A SIGNAL translation for the example server part processor is given and commented below:

```
process PARTITION2_LEVEL_OS = { integer Partition_ID; }
( ? PartitionID_type active_partition_ID; event initialize, end_processing;
! ProcessID_type active_process_ID;
   [MAX_NUMBER_OF_PROCESSES]boolean timedout;)
(| pid1 := PROCESS_S_CREATION(initialize) %create the PROCESSes%
| return_code1 := START{}(pid1)
%any created PROCESS needs to be started to be active%
| partition_is_running := when (active_partition_ID = Partition_ID)
| success := PROCESS_SCHEDULINGREQUEST{} (when partition_is_running) %On receiving the input active signal,
a priority preemptive scheduling is tried to be performed%
| (active_process_ID,status) := PROCESS_GETACTIVE{}(when success) %invoked after each rescheduling request to
get the current active PROCESS%
| timedout := UPDATE_COUNTERS{}()
%manage the time counters associated with PROCESSes%
| timedout ^= when partition_is_running
| return_code2 := SUSPEND_SELF{}(7.0 when end_processing)
|) where
boolean success; event partition_is_running; ProcessStatus_type status;
ProcessID_type pid1; ReturnCode_type return_code1,return_code2;

process PROCESS_S_CREATION =
( ? event initialize; ! ProcessID_type pid1;)
(| recorded1 := PROCESS_RECORDING{}("process_server" when initialize)
| att1.Name := "process_server" | att1.Entry_Point := 0.1
| att1.Stack_Size := 1 | att1.Base_Priority := 3
| att1.Period := -1.0 | att1.Time_Capacity := 1.0
| att1.Deadline := #SOFT | att1 ^= when recorded1
| (pid1,ret1) := CREATE_PROCESS{}(att1)
%record the PROCESS attributes%
|) where
boolean recorded1; ProcessAttributes_type att1; ReturnCode_type ret1;
end; end; %end of PARTITION1_LEVEL_OS%
```

5.3.3 Bus

A bus component represents hardware and associated communication protocols that enable interactions among other execution platform components (ie., memory, processor and device). For example, a connection between two threads, each executing on a separate processor, is through a bus between those processors. This communication is specified in AADL using **access** and **binding** declarations to a bus.

Because memory is ignored in this article, we only discuss the bus interaction between processor and device components.

Bus between two processors In this case, it means that the bus connects two different sub-systems. The bus is used for exchange of communication data. As mentioned, each subsystem is mapped as an ARINC PARTITION, the communication between PARTITIONs in ARINC services is via *ports* and *channels* (Figure 12). There are two transfer modes in which *channels* may be configured: *sampling* mode and *queuing* mode. In the former, no

message queuing is allowed. A message remains in the source port until it is transmitted by the *channel* or it is overwritten by a new occurrence of the message. During transmissions, *channels* ensure that messages leave source ports and reach destination ports in the same order. A received message remains in the destination port, until it is also overwritten. In the *queuing* mode, *ports* are allowed to store messages from a source PARTITION in queues, until they are received by the destination PARTITION.

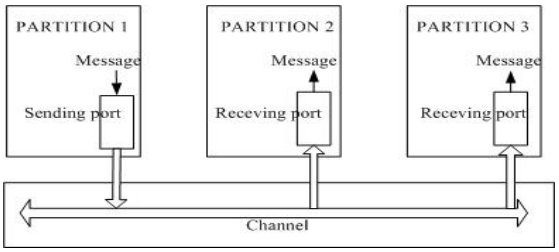


Figure 12: ARINC *port* mechanism

A simple way to implement bus access in SIGNAL is to use the *port* mechanism.
General rules:

1. The APEX SAMPLING_PORT mechanism can be used for AADL bus.
2. Some property checking must be added.
3. The source and destination PARTITIONs need to declare the use of SAMPLING_PORT, and identify the direction: source or destination.

Following is an example for the CREATE_SAMPLING_PORT interface. Here three new inputs (it maybe that more than three properties need to be checked) are added: **transmission_time**, **message_Size**, **access_protocol**, which correspond separately to Transmission_Time, Allowed_Message_Size, Allowed_Access_Protocol property in AADL. For the other APEX SAMPLING_PORT interfaces, similar property checking must be added.

```
bus lan
end lan;
bus implementation lan.ethernet
properties
        Transmission_Time => 1ms .. 5ms;
        Allowed_Message_Size =>1b .. 100kb;
        Allowed_Access_Procotol => Device_Access;
end lan.ethernet;

process CREATE_SAMPLING_PORT =
( ? Comm_ComponentName_type samplingPort_Name;
    SamplingPortSize_type samplingPort_Size;
    PortDirection_type port_direction;
    SystemTime_type refresh_period;
    SystemTime_type transmission_time;
    SamplingMessageSize_type message_Size;
    SamplingPortAccessProtocol access_protocol;
  ! Comm_ComponentID_type samplingPort_ID;
    ReturnCode_type return_code;)
```

```
(| (| (| exceeded := SAMPLINGPORT_CHECKCAPACITY() .....
    | size_OK := (samplingPort_Size <= MAX_SAMPLING_PORT_SIZE) when present ..... |) |) |);
```

Bus between a processor and a device In this case, the processor and device are in the same sub-system, it is the communication between a PARTITION and a device *process* in SIGNAL. A set of new BUS SIGNAL *processes* is provided:

- CREATE_BUS: create a new bus, record the predeclared properties.
- WRITE_BUS: input some messages to the bus, make property checking.
- READ_BUS: read the current message from the bus.

The detailed programming can be implemented in C code. In the programming, two things must be done: check the property whether the message is available for transfer, and if available then record the message in the bus, otherwise ignore it.

5.4 Component Interactions

An AADL port represents a communication interface for the directional exchange of data, events, or both between components. Ports are classified as:

- **data port**: interfaces for typed state data transmission among components without queuing. Connections between data ports are either immediate or delayed.
- **event port**: interfaces for the communication of events raised by subprograms, threads, processors, or devices that may be queued.
- **event data port**: interfaces for message transmission with queuing. These interfaces enable the queuing of the data associated with an event.

A port connection instance represents the actual flow of data and control between components of a system instance model. In case of a fully specified system, this flow is a transfer between two thread instances, a thread instance and a processor instance, or a thread instance and a device instance, at least one thread must be included. Each input port has a fresh variable to define the state of the port, if a port has not received anything between two thread dispatches, this variable is set to false. A buffer (to distinguish from the ARINC *buffer* mechanism) is also associated with each input port, when an output port sends a data or an event it modifies these buffers. On the dispatch of a thread, these buffers are copied into the local memory of the thread.

For the AADL port connection translation, we define a thread and its parent process parent sub-system as an **enclosing set**. The port connection can be divided into two types:

- Type A: the sequence of data connection is within an enclosing set, for example from a thread to its parent process, or from process to thread (within the same enclosing set) (see Figure 13 left).
- Type B: the sequence of data connection is between two enclosing sets, for example, the sequence of data connection from a thread to its parent process, to the second process, and to the thread contained in the second process (see Figure 13 right).

For type A, we just consider it as usual connection, like parameter transfering.

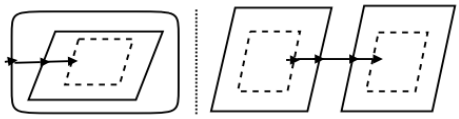


Figure 13: Port connection

For type B, it can be translated as *blackboard* or *buffer*, according to the communication scheme. If it is queuing, then it can be mapped into *buffer*; if queuing is not allowed, then *blackboard* can be used. A more detailed description of type B mapping in the following:

1. For data port, queuing is not allowed, and the connection can be either immediate or delayed. APEX *blackboard* is used to display and read messages: no message queues are allowed, and any message written on a *blackboard* remains there until the message is either cleared or overwritten by a new instance of the message [15]. That is to say, the output message is either synchronous with the input or delayed of several ticks. So data port connection communication can be mapped as read/write *blackboard*.

2. For event data port, queuing is allowed. APEX *buffer* allows to send and receive messages following a FIFO policy. So *buffer* can be used for event data port connection.

3. For event port, it may be queued. For simple, we image it is queued, and consider it the same as event data port.

6 Related work

A number of related approaches have been proposed. Dissaux [16] presents an approach for AADL model transformations. This approach concentrates on the analysis of components from legacy code aimed specifically towards use with the HOOD Stood tool [16]. Bertolino and Mirandola [17] propose an approach for the specification and analysis of performance related properties of AADL components using the RT-UML profile. Although the approach also uses a UML profile, it is not targeted towards model driven development.

Also, a number of tools are available that address the issues discussed in this paper. CHEDDAR [18] is a free real-time scheduling tool. It is designed for checking task temporal constraints of a real time application/system which is described with AADL or a CHEDDAR specific language. CHEDDAR provides a number of features to ease the development of specific schedulers and task models, and it relies on OCARINA [19] to provide schedulability analysis of AADL models. OCARINA allows model manipulation, generation of formal models, to perform scheduling analysis and generate distributed applications. OCARINA allows code generation from AADL descriptions to Ada. GME [20] is engaged in work on a DARPA-sponsored metamodeling framework, AADL capture and role-based system security analysis, model transformation and integration.

Some related approaches are proposed to modeling nonsynchronous systems using synchronous languages and developing system level design methodology. For instance, AADL2SYNC [21] tool is an AADL to synchronous programs translator, which is extended in the framework of the European project ASSERT, resulting in the system-level tool box translating AADL to LUSTRE. Although the approach also translates AADL to a synchronous language, it considers a purely synchronous model of computation (that of LUSTRE) in which clocks need to be totally ordered (by contrast to the relational, multi-clocked MoC considered here). This limitation requires the emulation of asynchrony by using a specific protocol of quasi-synchronous communication. This protocol correctly emulates asynchrony by simulating variable drifts using random-number generators. Still its expressive capability is limited compared to simply abstracting asynchrony using partially-ordered clock relations, which the MoC of SIGNAL allows, and yields a compositional translation of AADL constructs.

7 Conclusion

We are interested in a representation of the AADL meta model, which permits us to specify and prove correct transformations of AADL models. The aim of our approach, which was illustrated on a simple example, is to ease the task of evaluating dependability measures. Our approach has two main characteristics: 1) it is incremental, as it needs to support and trace model evolution, 2) it is based on model transformation, from AADL dependability models to SIGNAL that can be processed by existing technologies and services.

In this paper, we presented a way to use the APEX-ARINC services modeling of asynchronous systems, to produce automatically a usable model of synchronous architecture. Our technique efficiently reuses most of existing ARINC libraries and services in order to implement our proposal, which justices presenting it in sufficient details in the present article.

The advantage of our mapping modeling approach is that it provides a quite systematic way of modeling asynchronous behaviors, and it allows a significant reduction of the mapping cost, since the synchronous description of the synchronous parts generally involves the existing concepts and components.

Not all components and properties are supported at this moment. The following AADL concepts can be supported: system, sub-system, device, process, processor, thread, subprogram, bus and port. Some concepts are ignored in this translation: memory, flow, property and data. Some concepts are not considered yet: port group, thread group and so on.

After having defined the approach, the main purpose of the work carried out until now is to assess its feasibility. The next step of the work concerns the formalisation of transformation rules in order to automate model transformation and support more additional features.

References

[1] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey, **Embedded system design using UML and platforms**, In Forum on Specification and Design Languages, September 2002

[2] M. Edwards and P. Green, **UML for hardware and software object modeling**, In UML for real: design of embedded real-time systems, Kluwer Academi Publishers, 2003

[3] P.H. Feiler, D.P. Gluch, J.J. Hudak. **The Architecture Analysis & Design Language (AADL): An Introduction**, Technical Note CMU/SEI-2006-TN-011, February 2006

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

[5] M. Krstić, E. Grass, F.K. Gürkaynak, P. Vivet, **Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook**, IEEE Design and Test of Computers, vol. 24, no. 5, pp. 430-441, September-October, 2007

[6] 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, World Scientific, April 2003

[7] TopCased project, http://www.topcased.org

[8] Airlines Electronic Engineering Committee. **Design Guidance for Integrated Modular Avionics**. ARINC Report 651-1, November 1997

[9] Bernard HOUSSAIS, IRISA ESPRESSO Project. **The Synchronous Programming Language SIGNAL, A Tutorial**, September 2004

[10] J. Hudak, P. Feiler, **The SAE Architecture Analysis & Design Language (AADL) Standard: A Language Summary**, AADL Standard Document, 2006

[11] Airlines Electronic Engineering Committee. **Avionics Application Software Standard Interface**. Arinc Specifcation 653, January 1997

[12] A. Gamatié, T. Gautier. **Synchronous Modeling of Modular Avionics Architectures using the SIGNAL Language**, Technical Report, IRISA, December 2002

[13] L. Besnard, T. Gautier, P. Le Guernic. **SIGNAL V4-INRIA version: Reference Manual**, IRISA, June 2006

[14] A. Gamatié, and T. Gautier, **Synchronous modeling of avionics applications using the Signal language**. In Proc of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'2003), May 2003, IEEE Press

[15] A. Gamatié, T. Gautier, P. Le Guernic and J.-P. Talpin, **Polychronous Design of Embedded Real-Time Applications**, ACM Transactions on Software Engineering and Methodology (TOSEM), April 2007

[16] P. Dissaux, **AADL model transformations**, Proc DASIA 2005 Conference in Edinburgh, UK, 2005

[17] A. Bertolino, R. Mirandola, **Modeling and Analysis of Non-functional Properties in Component-Based Systems**, Electronic Notes in Theoretical Computer Science 82(6), 2003

[18] F. Singhoff, J. Legrand, L. Nana, L. Marcé, **Cheddar: a Flexible Real Time Scheduling Framework**, Proc of the ACM SIGAda International Conference, Atlanta, US, 2004.

[19] J. Hugues, B. Zalila, L. Pautet, **Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina**, Proc of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), Brazil, 2007

[20] A. Ledeczi, M. Maroti, and P. Volgyesi, **The Generic Modeling Environment**, Technical Report, Vanderbilt University, 2002. Available from http://www.isis.vanderbilt.edu/projects/gme/GME2000Overview.pdf

[21] AADL2SYNC project, http://www-verimag.imag.fr/~synchron/index.php?page=aadl2sync

[22] P. Caspi, C. Mazuet, and N. R. Paligot, **About the design of distributed control systems, the quasi-synchronous approach**, In SAFECOMP'01, LNCS 2187, 2001. 1, 2

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399