Front. Comput. Sci., 2013, 7(5): 627–649
DOI 10.1007/s11704-013-2307-z

RESEARCH ARTICLE

Exploring system architectures in AADL via POLYCHRONY and SYNDEX

Huafeng YU[1], Yue MA[1], Thierry GAUTIER (✉)[1], Loïc BESNARD[2]
Jean-Pierre TALPIN[1], Paul Le GUERNIC[1], Yves SOREL[3]

1 INRIA Rennes - Bretagne Atlantique, 263, av. du Général Leclerc, Rennes 35042, France
2 IRISA/CNRS, 263, av. du Général Leclerc, Rennes 35042, France
3 INRIA Paris - Rocquencourt, Domaine de Voluceau, BP 105, Le Chesnay Cedex 78153, France

Abstract Architecture analysis & design language (AADL) has been increasingly adopted in the design of embedded systems, and corresponding scheduling and formal verification have been well studied. However, little work takes code distribution and architecture exploration into account, particularly considering clock constraints, for distributed multi-processor systems. In this paper, we present an overview of our approach to handle these concerns, together with the associated toolchain, AADL-POLYCHRONY-SYNDEX. First, in order to avoid semantic ambiguities of AADL, the polychronous/multiclock semantics of AADL, based on a polychronous model of computation, is considered. Clock synthesis is then carried out in POLYCHRONY, which bridges the gap between the polychronous semantics and the synchronous semantics of SYNDEX. The same timing semantics is always preserved in order to ensure the correctness of the transformations between different formalisms. Code distribution and corresponding scheduling is carried out on the obtained SYNDEX model in the last step, which enables the exploration of architectures originally specified in AADL. Our contribution provides a fast yet efficient architecture exploration approach for the design of distributed real-time and embedded systems. An avionic case study is used here to illustrate our approach.

Received February 15, 2013; accepted June 1, 2013

E-mail: Thierry.Gautier@inria.fr

Keywords POLYCHRONY, SIGNAL, AADL, SYNDEX, architecture exploration, modeling, timing analysis, scheduling, distribution

1 Introduction

The architecture analysis & design language (AADL) [1] is gradually adopted for high-level system co-modeling in embedded systems due to issues of system complexity, time to market, verification and validation, etc. It permits the fast yet expressive modeling of a system [2]. Early-phase analysis and validation can be rapidly performed [3–9]. AADL provides a fast design entry, however, there are still critical challenges, such as unambiguous semantics, architecture exploration, code distribution, timing analysis, or co-simulation. To address these issues, expressive formal models and complete toolchains are required.

Synchronous languages are dedicated to the trusted design of synchronous reactive embedded systems [10]. Thanks to their mathematical definition and the software tools that are founded on these definitions, synchronous languages are good candidates to address the above issues of AADL. Among these languages, the SIGNAL language stands out as it enables to describe systems with multiclock relations [11], and to support *refinement* [12]. POLYCHRONY is a toolset based on the SIGNAL language, dedicated to multiclock synchronous program transformation and verification.

Various tools exist to help the implementation of ap-

plications onto distributed platforms composed of processors, specific integrated circuits, and communication media all together connected. Among them, SynDEx is a system level computer aided design (CAD) software, based on the algorithm-architecture adequation (AAA) methodology [13]. It aids the designer to search, manually and/or automatically, for an optimized implementation of an embedded control application onto a distributed platform architecture, while satifying real-time constraints. SynDEx provides interfaces to application description languages such as Signal.

The AADL language is designed to be used with analysis tools. Our aim is to make Polychrony and SynDEx such tools in an integrated methodology.

In this paper, an overview of our approach is presented, including the formal timing modeling, timing analysis, clock synthesis, architecture exploration as well as the associated toolchain, called AADL-Polychrony-SynDEx. First, two polychronous/synchronous formalisms [10], Polychrony/Signal [14,15] and SynDEx [16], are adopted to provide support for formal timing modeling based on a polychronous model of computation (MoC) [11]. We revisit the timing semantics of AADL as multi-clocked, so that AADL components are modeled in this polychronous MoC. In this way, users are not suffered to find and/or build the fastest clock of the system, which distinguishes from [4,16–18]. According to this principle, AADL models are transformed into Signal models. To bridge the gap between the polychronous semantics of Signal and synchronous semantics of SynDEx, clock synthesis in Polychrony [14], the design environment dedicated to Signal, is applied. The translation from Signal to SynDEx is integrated in Polychrony. Finally, SynDEx models are used to perform distribution, scheduling, and architecture exploration.

Main advantages of our approach are following ones: (1) a formal model is adopted to connect the three formalisms, and it helps to preserve the semantic coherence and correct code generation in the transformations; (2) the formal model and methods used in the transformations are transparent to AADL practitioners, and it is fast and efficient to have illustrative results for architecture exploration; (3) it provides the possibility for one of the three formalisms to take advantage of the functionalities provided by the other ones.

An AADL-Polychrony-SynDEx toolchain has been developed, which includes automatic model transformations between the three formalisms, considering both semantic and syntactic aspects. A tutorial avionic case study is used in this paper to show the effectiveness of our contribution. This compact yet typical and general case study has been developed in the framework of the CESAR project [19].

An overview of the tool architecture is given in Section 2. The three formalisms, Polychrony/Signal, AADL and SynDEx, are introduced in Section 3. Section 4 and Section 5 present our original contribution: the timing modeling, translations between the different formalisms, clock synthesis, architecture exploration. The whole work is illustrated by a case study in Section 6. Some related works are summarized in Section 7, and a conclusion is drawn in Section 8.

2 The scenario

Our proposed approach for architecture exploration of AADL high-level models is illustrated in Fig. 1. Three stages are presented in the design process, which include: high-level modeling in AADL, considering both architectural and behavioral aspects; model transformations, timing analysis and clock synthesis using Polychrony; and architecture exploration with the aid of SynDEx. The Polychrony design environment [14], associated with the Signal language, provides a formal framework for back-end semantic-preserving transformations, scheduling, code generation, formal analysis and verification, architecture exploitation, and distribution [20]. The internal representation used for these purposes is a data control graph (DCG), composed of a clock hierarchy and a conditioned precedence graph (see [20] for more details). The Polychrony toolset is an open-source modeling framework, DO-330 qualified (as verification tool, criteria 3); it is being integrated in the PolarSys toolset (Polarsys is an industry working group of the eclipse foundation [21]).

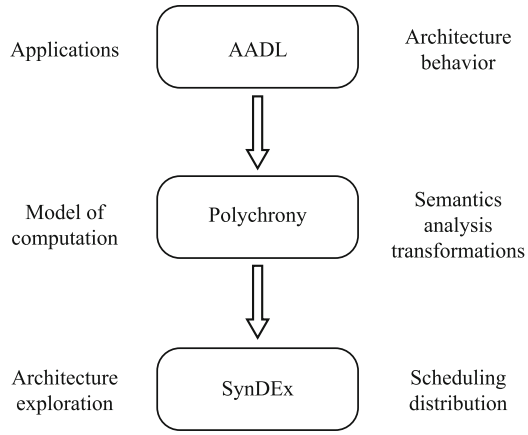


Fig. 1 The AADL-Polychrony-SynDEx approach

The inherent formal model, associated transformations and tools used in this approach are transparent to system designers, i.e., AADL practitioners. The results of architecture ex-

ploration is graphically illustrated. This approach is thus fast and efficient to perform architecture exploration from a high-level modeling perspective.

The polychronous model is adopted to bridge between the three formalisms, i.e., AADL, Signal and SynDEx. This model helps to preserve the timing semantic coherence and correct code generation in the transformations between the different formalisms.

In line with our approach, a complete toolchain (see Fig. 2) for modeling, timing analysis, scheduling, and distribution of AADL models via Polychrony and SynDEx has been developed in the eclipse modeling framework (EMF) [22]. An AADL model with timing properties, which conforms to the AADL metamodel, is created in the OSATE toolkit [23]. A model transformation **ASME2SSME** allows one to perform analysis on ASME models (AADL syntax model under eclipse) and generate corresponding Signal SSME (Signal syntax model under eclipse) models. ASME2SSME uses high-level APIs (in the figure, CoL stands for "concept oriented level"). Those high-level APIs are defined in terms of low-level APIs that provide access to the model implementation (in the figure, MoL stands for "model oriented level"). The Signal code, capturing both functional and architectural aspects of the original application, is then generated from the SSME models. The Signal compiler (from the Polychrony toolset) is used for analysis and transformations. It is used in particular to generate the SynDEx code. Finally, the latter is fed into the SynDEx tool for architecture exploration.

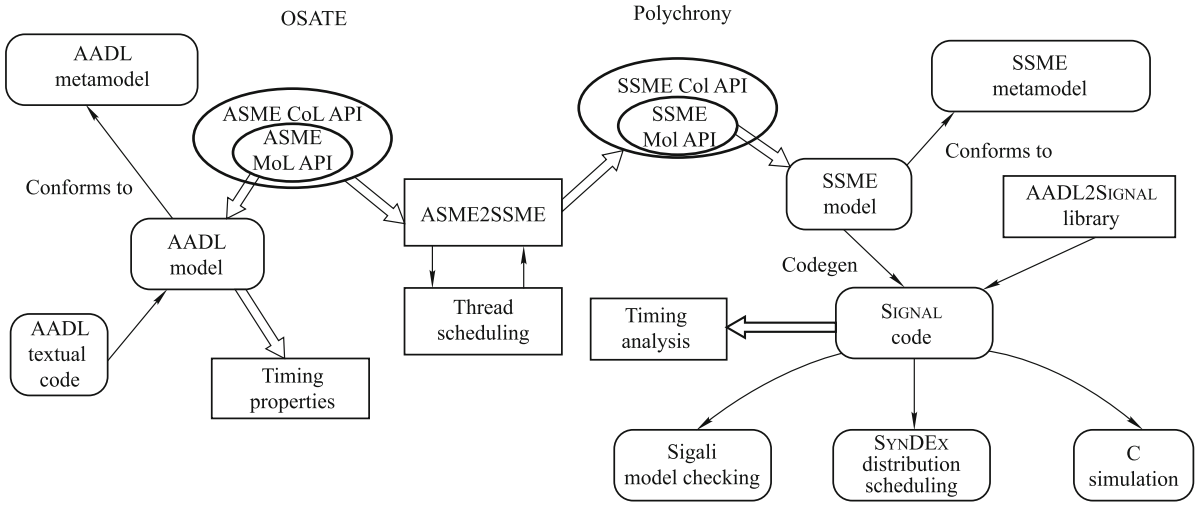


Fig. 2 The AADL-Polychrony-SynDEx toolchain

3 The casting

3.1 The polychronous model and the Signal kernel language

The semantics of Signal is defined over a polychronous Model of Computation. A Signal process defines a set of (partially) synchronized signals as the composition of equations. A signal x is a finite $((\exists n \in \mathbb{N})(x = (x_t)_{t\in\mathbb{N}, t\leqslant n})$ or infinite $(x = (x_t)_{t\in\mathbb{N}})$ sequence of typed values in the data domain D_x; the indices in the sequence represent logical discrete time instants. At each instant t, a signal is either present and holds a value v in D_x, absent and virtually holds an extra value denoted #, or completed and never holds any actual or virtual value for all instants s such that $t \leqslant s$. The set of instants at which a signal x is present is represented by its clock $\hat{x}$. Two signals are synchronous iff they have the same clock. Clock constraints result from implicit constraints over signals and explicit constraints over clocks.

The semantics of the full language is deduced from the semantics of a kernel language, and from the Signal definition of the extended features. A Signal kernel language process is either a kernel equation $x := f(x_1, x_2, \ldots, x_n)$ where f is a kernel function, or the composition $P|Q$ of two kernel processes P and Q, or the binding P/x of the signal variable x to the kernel process P. In this Section, we give a sketch of its functional part using data-flow models.

3.1.1 Semantic domains

For a set of values (a type) $\mathbb{D}_1$ we define its extended set $\mathbb{D}_{1\#} = \mathbb{D}_1 \cup \{\#\}$, where $\# \notin \mathbb{D}_1$ is a special symbol used to denote the absence of a signal value. The semantics of Signal is de-

fined as least fixed point in domains: for a data domain $\mathbb{D}_1$, we consider the poset $(\mathbb{D}_{1\#} \cup \{\bullet, \perp\}, \leqslant)$ where:

- $(\mathbb{D}_{1\#}, \leqslant)$ is flat (i.e., $x \leqslant y \Rightarrow x = y$);
- $\bullet$ that denotes the presence of a signal is the infimum of $(\mathbb{D}_1 \cup \{\bullet\}, \leqslant)$, $\bullet$ and # are not comparable;
- $\perp$ that denotes the absence of information is the infimum of $(\mathbb{D}_{1\#} \cup \{\bullet, \perp\}, \leqslant)$.

We denote by $\mathbb{D}^\infty = \mathbb{D}^* \cup \mathbb{D}^\omega$ the set of finite ($\mathbb{D}^*$) and infinite ($\mathbb{D}^\omega$) sequences of "values" in $\mathbb{D}_\#$. The empty sequence is denoted by ϵ. All n-ary signal functions $f : \mathbb{D}_1^\infty \times \mathbb{D}_2^\infty \times \ldots \times \mathbb{D}_n^\infty \to \mathbb{D}_{n+1}^\infty$ are defined using the following conventions: $s_1, s_2, \ldots, s_n$ are (possibly empty) signals in $\mathbb{D}_i^\infty$, $v_1, v_2, \ldots, v_n$ are values in $\mathbb{D}_i$ (cannot be #), $x_1, x_2, \ldots, x_n$ are values in $\mathbb{D}_{i\#}$. As usual, $|s|$ is the length of s, $s_1.s_2$ is the concatenation of s_1 and s_2 (equal to s_1 if $s_1 \in \mathbb{D}^\omega$).

Given a non empty finite set of signal variables A, a function $b : A \to \mathbb{D}^\infty$ that associates a sequence $b(x)$ with each variable of A is named a behavior on A. The length $|b|$ of a behavior b on A is the length of the smallest sequence $b(a)$. An event on A is a behavior $b : A \to \mathbb{D}_\#$. For a behavior b on a set of signal variables A, an integer $i \leqslant |b|$, $b(i)$ denotes the event e on A such that $e(a) = (b(a))(i)$ for all $a \in A$. An event e on A is said to be empty iff $e(a) = \#$ for all $a \in A$. The concatenation of signals is coordinatewise extended to tuples of signals. Two behaviors b_1, b_2 are stretch-equivalent iff they only differ on non-final empty events (see [11] for more details).

3.1.2 The SIGNAL kernel functions

A SIGNAL kernel function is a n-ary (with $n > 0$) function f that is total, strict and continuous over domains [24] (w.r.t. prefix order) and that satisfies the following general rules:

- stretching: $f(\#.s_1, \#.s_2, \ldots, \#.s_n) = \#.f(s_1, s_2, \ldots, s_n)$
- termination: $((\exists i \in 1, n)(s_i = \epsilon)) \Rightarrow f(s_1, s_2, \ldots, s_n) = \epsilon$

A n-ary function f is synchronous iff it satisfies: $\forall v_1, v_2, \ldots, v_n \in \mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_n$, $x_1, x_2, \ldots, x_n \in \mathbb{D}_{1\#}, \mathbb{D}_{2\#}, \ldots, \mathbb{D}_{n\#}$, $s_1, s_2, \ldots, s_n \in \mathbb{D}_1^\infty, \mathbb{D}_2^\infty, \ldots, \mathbb{D}_n^\infty$,

$$\begin{cases} - ((\exists i, j \in 1, n)(\exists v \in \mathbb{D}_j)(x_i = \# \wedge x_j = v)) \Rightarrow \\ \quad (f(x_1.s_1, x_2.s_2, \ldots, x_n.s_n) = \epsilon) \\ \text{and} \\ - (\exists v \in \mathbb{D}, s \in \mathbb{D}^\infty) f(v_1.s_1, v_2.s_2, \ldots, v_n.s_n) = v.s \end{cases}$$

- Definition of stepwise extension kernel functions

Given $n > 0$ and a n-ary total function $f : \mathbb{D}_1 \times \mathbb{D}_2 \times \cdots \times \mathbb{D}_n \to \mathbb{D}_{n+1}$, the stepwise extension of f denoted F is the kernel synchronous function that satisfies:

$$- F(v_1.s_1, v_2.s_2, \ldots, v_n.s_n) = f(v_1, v_2, \ldots, v_n).F(s_1, s_2, \ldots, s_n)$$

Usual infix notation is used for standard operators such as =, and, +, etc.

- Definition of previous value kernel function

delay: $\mathbb{D}_i \times \mathbb{D}_i^\infty \to \mathbb{D}_i^\infty$ is the kernel synchronous (state-) function that satisfies:

- $\text{delay}(v_1, v_2.s) = v_1.\text{delay}(v_2, s)$

The infix notation of $\text{delay}(v_1, s)$ is s \$ init v_1.

- Definition of prioritized merge kernel function

default: $\mathbb{D}_i^\infty \times \mathbb{D}_i^\infty \to \mathbb{D}_i^\infty$ is the kernel function that satisfies:

- $\text{default}(v.s_1, x.s_2) = v.\text{default}(s_1, s_2)$
- $\text{default}(\#.s_1, x.s_2) = x.\text{default}(s_1, s_2)$

The infix notation of $\text{default}(s_1, s_2)$ is s_1 default s_2.

- Definition of Boolean sampling kernel function

Let $\mathbb{B} = \{\text{ff}, \text{tt}\}$ denote the set of Boolean values.
when: $\mathbb{D}_i^\infty \times \mathbb{B}^\infty \to \mathbb{D}_i^\infty$ is the kernel function that satisfies:

- for $b \in \{\#, \text{ff}\}$, $\text{when}(x.s_1, b.s_2) = \#.\text{when}(s_1, s_2)$
- $\text{when}(x.s_1, tt.s_2) = x.\text{when}(s_1, s_2)$

The infix notation of $\text{when}(s_1, s_2)$ is s_1 when s_2. For a Boolean signal s, when s is the unary notation of when $(\hat{s}, s)$ that returns tt iff s is tt, # otherwise.

3.1.3 Deterministic process

An equation is a pair (x, E) denoted $x := E$. An equation $x := E$ associates with the variable x the sequence resulting from the evaluation of the signal function f denoted by E (defined as a composition of kernel functions). If $A = \{x_1, x_2, \ldots, x_n\}$ ($x \notin A$) is the set of the free variables in E, the equation $x := E$ denotes a process on A, i.e., a set of behaviors on $A \cup \{x\}$; a process is closed by stretch-equivalence (thanks to the stretching rule).

The parallel composition of processes $P_1|P_2$ defined on kernel equations is a process P equal to a network of strict continuous functions interconnected on the basis of signal names, as usual in equations. When P satisfies the Kahn con-

ditions (no cycle, no double definition...), P is a strict continuous function or Kahn process network (KPN) [25], defined as least upper bound satisfying the equations. This function satisfies the "stretching" and the "termination" rules. It may be or not synchronous. It is stretch-closed. The binding P/x is a projection that makes local the variable x. In the "intersection" semantics of SIGNAL [11], a process is the set of infinite behaviors accepted by the above "KPN" semantics.

3.1.4 Non deterministic process

A process with feedback or local variables may be non deterministic. The semantics of a non deterministic process can be defined using Plotkin's power-domain construction [26]. The input free equation $x := x$ `$ init 0` is a typical example of a non deterministic process: x holds a sequence of constant value 0 separated by any number of # (stretch-equivalence).

3.2 The full SIGNAL language

3.2.1 Process models

A process model is a structuration feature illustrated by the counter modulo process (Listing 1): at each step, `nb` is the number modulo n (static parameter) of the current occurrence of `val`.

Listing 1 Definition of the SIGNAL process counter Modulo

```
process counterModulo = {type tau; integer n}
        (? tau val ! integer nb)
    (| nb := (0 when reset) default plus_un
     | reset := pre_nb = n − 1
     | plus_un := pre_nb+(1 when ^val)
     | pre_nb := nb $ init (n − 1)
     |)
  where boolean reset;
        integer pre_nb, plus_un;
end
```

A process model is made of a name, a list of static parameters (formal type `tau` and constant value n), a list of signal inputs (the signal `val` of which occurrences are counted), a list of signal outputs (the signal `nb` that holds the number of each `val` occurrence modulo n), a body that contains a composition of equations and a list of local variables. An occurrence (such as

`counterModulo{event,60}(sec_event,sec_count)`

in equation form, or

`sec_count:=counterModulo{event,60}(sec_event)`

in function call form) of an instance of `counterModulo` is replaced by the body of `counterModulo` with required substitutions.

3.2.2 From non deterministic to deterministic process

The operator `var` (see [20]) is a typical example of an operator that is not a function. It is a *derived* operator, the definition of which is that of preMemoryCell (Listing 2).

Listing 2 Definition of the SIGNAL process preMemoryCell

```
process preMemoryCell = {type tau; tau val}
        (? tau input ! tau output)
    (| mem := input default mem $1
     | output := mem when ^output
     |)
  where tau mem init val;
end
```

The process model preMemoryCell is a mix of data-driven and demand-driven operator: `output := mem when output` (" " is the operator that returns tt when its argument is present) outputs the current value when it is (implicitly) required. Thus `preMemoryCell` is not a function but a relation. It is a polychronous process that has two independent clocks, `input` and `output`. The clock `mem` is an upper bound of the clocks `input` and `output`.

Including the relation `preMemoryCell` (equivalent to the operator `var`) in a context for which the clock of the output is added as an input signal allows to build a deterministic process memoryCell (Listing 3).

Listing 3 Definition of the SIGNAL process memoryCell

```
process memoryCell = {type tau; tau val}
        (? tau input; event clk_output ! tau output)
    (| mem := var input
     | output := mem when clk_output
     | mem ^= clk_output ^+ input
     |)
  where tau mem init val;
end
```

The expression `mem = clk_output + input` means that the clock of `mem` is the least upper bound of the clock `clk_output` and that of the `input` signal. The operators `=` and `+` denote respectively a relation and a function over clocks derived form SIGNAL kernel features.

Another example of a non deterministic process is the equation x `:= 0` that defines the constant signal x as being equal to # or equal to 0. This equation is a shortcut for x `:=` x `$ init 0`.

A last example of non deterministic process is the equation x `::=` E that defines x to be equal to E when E is present and undefined when E is #. This equation is a shortcut for x `:=` E `default` x. A signal x can be consistently defined

by several equations $x ::= E_1, E_2, \ldots, x ::= E_n$ in a process, provided that for every pair of equations $x ::= E_i$, $x ::= E_j$, when E_i and E_j are both present, they hold the same value. If $E_1, E_2, \cdots, E_n$ do not recursively refer to x and if they denote functions, then $(|x ::= E_1| \ldots |x ::= E_n|)$ is a deterministic process. These partial definitions are very useful in automata where the function that computes the value of a signal often depends on current state. The states being exclusive, the consistency property is satisfied.

3.2.3 Delay sensitivity

Deterministic processes are defined over data types extended with # symbol. The presence (or absence) of the occurrence of a signal can be tested in the context of a global program with a global clock by interface functions such as "present" in Esterel [10]. In the context of a distributed system, when an occurrence of a signal is "arrived", the signal has not necessarily to be considered as present in the logical instant. Conversely, when the action associated with a logical time is started, some of the required occurrences may not be arrival. Not only because of the delay sensitivity, but also because of computation scheduling: for instance, when a process P_1 computes a signal x that is used in another process P_2 to compute a signal y that is used in P_1 at the same instant. Processes are usually delay sensitive. In particular, a reactive process with more than one input is delay sensitive. Delay insensitive processes are flow functions, i.e., functions on pure signals that do not contain explicit occurrences of #. Clearly, stepwise extension functions and delay functions are delay insensitive. To get a delay insensitive *sampling*, one can associate with an equation `output := input when sample` a Boolean clock parameterization (`C_input, C_sample`) of the *sampling* equation as in flowSampling (Listing 4).

Listing 4 Definition of delay insensitive sampling

```
process flowSampling = {type tau}
      (? boolean C_input, C_sample;
         tau input; boolean sample; ! tau output)
   (| output := input when sample
    | (| C_input ^= C_sample
      | input ^= when C_input
      | sample ^= when C_sample
      |)
    |)
```

The process model flowSampling denotes an endochronous process: it has a master clock (a tick) `C_input = C_sample`, and the other clocks are (recursively) functions of this master clock and the input signal values. One can similarly define an endochronous extension of a `default` equation. A composition P of endochronous processes that satisfies the Kahn conditions is a strict continuous (flow) function and then P is delay insensitive. Note that a delay insensitive process may not be endochronous. Usually a KPN is not endochronous (it may require unbounded fifos to be implemented in a single thread).

Thanks to the properties of the SIGNAL composition (commutativity, associativity, idempotence), the possibility of moving inner bindings out of a composition, and the properties of a powerful clock algebra, the so-called clock calculus can arrange a process P as a composition of endochronous components, adding required Boolean inputs as clock parameters.

3.2.4 Dependences

The equations of a SIGNAL process induce guarded data-flow dependences. For instance in the equation z `:=` x `default` y, x precedes $z (x \rightarrow z)$ and `y` precedes z when x is not present $((y \rightarrow z)$ when $(\hat{}y \hat{}\text{-} \hat{}x))$. Input/output guarded dependences of a process are computed in a path algebra. They are written by POLYCHRONY as abstract interface properties of black box behaviors. Dependences can be made explicit by the user to enforce scheduling.

3.2.5 Specification

A process that has no output and that does not call external functions is a property process. For instance $a = b$ denotes the equality of clocks of a and b, $(a \rightarrow b)$ when c can be written to specify that an input a precedes (is read before) an other input b when the Boolean input c is tt (and then c implicitly precedes a and b).

Such a property process can be used as a constraint in the body of a process or as assumed or asserted properties in its interface.

3.2.6 SIGNAL open features

An external process (a function, a subprogram, a method, ...) can be abstracted (referred to) as a specification of its interface.

A pragma is an annotation that is associated with a process for specific purpose. Pragmas are sorted by name related to an action (for instance code generation), a tool (for instance a model checker), etc. The set of pragma classes is open.

3.3 AADL

AADL [1] is a society of automotive engineers (SAE) stan-

dard dedicated to design and analyze the architecture of performance-critical real-time systems. AADL describes the structure of an embedded application as an assembly of software components allocated on execution platform. The component interactions and the dynamic behavior can be described. The AADL standard specifies software and hardware components from an external perspective: AADL uses black box interfaces to describe non functional properties such as timing requirements or abstractions, fault and error behaviors, time and space partitioning, and safety and certification properties. Thus each component has a type, which represents the functional interface of the component and its externally observable attributes. Each type may be associated with zero, one or more implementation(s) that describe the contents of the component, as well as the connections between them.

In AADL, three main distinct component categories are provided:

- a software component is a process, a thread, a subprogram, or a data component;
- execution components model the hardware part of a system, including (virtual) processor, memory, device, and (virtual) bus components;
- composite component, named system, contains execution platform, application software or other composite components.

In this presentation we ignore other components such as thread groups that do not introduce new aspects in our translation scheme.

AADL provides mechanisms of exchange and control of data: message passing, event passing, synchronized access to shared components, thread scheduling protocols (periodic, aperiodic, sporadic, background), timing requirements, remote procedure calls. AADL components interact exclusively through defined interfaces. A component interface consists of directional flow through: data ports for unqueued data, event data ports for queued message data, event ports for asynchronous events (triggers for the dispatch of an aperiodic thread, initiators of mode switches, alarm communications), synchronous subprogram calls, explicit access to data components. Interactions among components are specified explicitly. For example, data communication (immediate or delayed) among components is specified through connection declarations.

Properties are specified to provide more information about model elements. For example, application components have properties that specify timing requirements such as period, worst-case execution time, deadlines, space requirements, arrival rates, and characteristics of data and event streams. In addition, properties identify the following elements: source code and data that implement the application component being modeled, constraints for binding threads to processors, source code, and data onto memory.

The behavior annex provides an extension to AADL core standard so that behavior specifications can be attached to AADL components. The behavior is described with a state transition system equipped with guards and actions.

Adapting the terminology of [1], we name complete application model an instance of a root system in which all components are recursively instantiated. A complete application model is bound if each thread is bound to a processor, each source text, data component and port is bound to a memory, each connection is bound to a bus. In this paper we are interested in complete application models for software architecture and bound complete application models. So a weak attention will be given to types and other abstraction or partial features.

In the following, an industrial case study of simplified doors and slides control system (SDSCS) in an avionic generic pilot application, proposed by Airbus in the frame of CESAR project, is used to illustrate the basic components of an AADL model. In this case study (see Fig. 3), a typical safety critical system takes charge of the management of passenger doors. It includes different components modeling hardware and software, and allowing them to communicate and control the doors.

In the system door_manager, two subsystems, $door_1$ and $door_2$, are managed by two processes, $doors\_process_1$ and $doors\_process_2$ (see Fig. 3). The processor CPIOM1 (resp. CPIOM2) is responsible for scheduling and executing threads in process doors_process1 (resp. doors_process2). The devices, e.g., LGS, DPS, etc., interface with external environment of the system. All the communication between the devices and processors is through the bus AFDX1.

3.4 SYNDEX

SYNDEX [27] is a system level CAD software based on the algorithm-architecture adequation (AAA) methodology [13] which allows the designer to optimize the implementation of embedded control applications onto distributed platforms, while satifying real-time constraints. This is a freeware distributed free of charge [28]. Using graph models, the AAA methodology allows the designer to perform a functional specification that we call “algorithm”, a non-functional

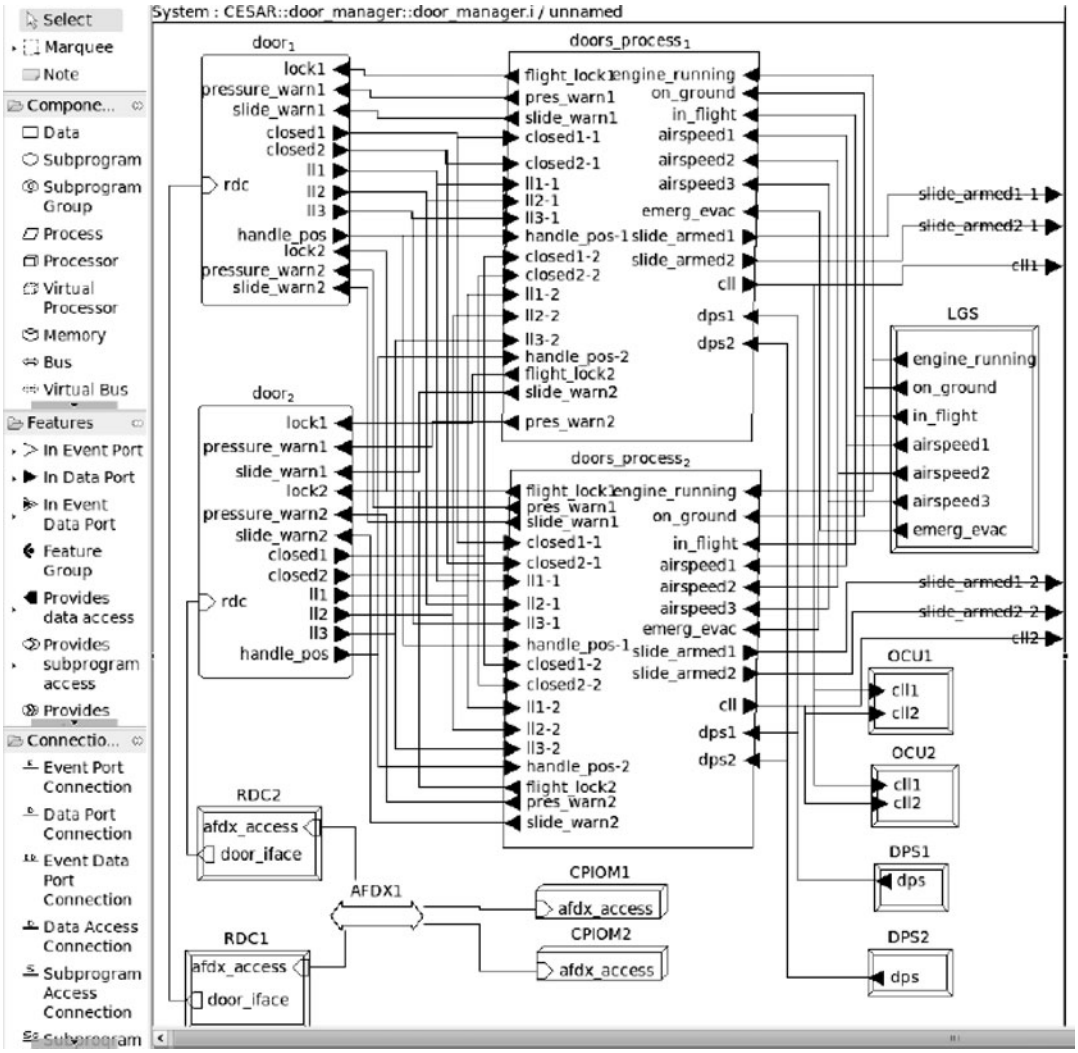


Fig. 3 An overview of the SDSCS case study in AADL

specification describing the real-time characteristics (worst case execution time (WCET), worst case communication time (WCCT), deadline, period, end-to-end latency, etc.) and the distributed platform (processors, specific integrated circuits and communication media) that we call "architecture". Consequently, all the possible implementations of a given algorithm onto a given architecture are described in terms of graph transformations. An implementation consists in distributing and scheduling a given algorithm onto a given architecture. "Adequation" amounts to choose one optimized implementation among all the possible ones while taking into account the real-time characteristics. This is achieved automatically through off-line multiprocessor real-time schedulability analyses combined with optimization heuristics. The result of the adequation is a scheduling table. Finally, from the scheduling table our graph models allow to generate automatically, as an ultimate graph transformation, two types of codes: dedicated distributed real-time executives, or configuration of standard distributed real-time executives, e.g., Linux, Linux/RTAI, Windows/RTX, OSEK, etc., for processors, and structural VHDL for specific integrated circuits. In this paper we focus only on the specifications and on the optimized implementation. See [29] for details about the code generation.

3.4.1 Algorithm model

If we want to use efficiently multiprocessor architectures offering some available parallelism, algorithms should be specified with at least as much potential parallelism as the available parallelism of the architecture. Moreover, since we want to be able to compare the implementation of an algorithm onto different architectures, the algorithm graph must be specified independently of any architecture graph.

Our algorithm model is an extension of the well-known

data-flow model from Dennis [30]. It is a directed acyclic hyper-graph (DAG) that we call "conditioned factorized data dependence graph", whose vertices are "operations" and hyper-edges are directed "data or control dependences" between operations. Hyper-edges are used to model data diffusion. The data dependences define a partial order on the execution of the operations, called "potential operation-parallelism". Actually, two operations which are not data-dependent may be executed, in any order, on a unique processor or in parallel on two different processors. Each operation may be in turn described as a graph to allow hierarchical specification of algorithms. Therefore, a graph of operations is also an operation. Operations which are the leaves of the hierarchy are said "atomic" in the sense that it is not possible to distribute this kind of operation on more than one processor. The basic data-flow model was extended in three directions, firstly infinite (resp. finite) repetitions in order to take into account the reactive aspect [31] of real-time systems (resp. "potential data-parallelism" similar to loop or iteration in imperative languages `for` $i = 1$ `to` n `do...`), secondly "state" when data dependences are necessary between repetitions introducing cycles which must be avoided by specific vertices called "delays" (similar to z^{-n} in automatic control), thirdly "conditioning" of an operation by a control dependence similar to conditional control structure in imperative languages (`if...then...else...`). A conditioned operation is a hierarchical vertex which contains several sub-dataflow graphs. According to the value of its specific input called "condition" only one of the possible sub-dataflow graphs will be executed during the considered reaction. A finitely repeated operation is also a hierarchical vertex which contains N times the same sub-dataflow graph. Conditioned operations may contain repeated operations, and vice-versa.

We denote by $G_{\text{al}} = (O, D)$ the conditioned factorized data dependence graph where O is the set of vertices (operations) and $D \subseteq O \times O$ is the set of data dependences defining a partial order on the execution of the operations. As mentioned before, this graph is repeated infinitely such that every repetition corresponds to a reaction of the real-time system. During the functional specification, we do not consider non-functional specification, i.e., physical time used to define timing characteristics and architecture. Each reaction defines a logical time and according to the synchronous language principles presented previously, we do not care about physical time taken by the execution of operations which will be considered further on in the implementation model. Thus, every data dependence $d \in D$ is associated with a signal which is an infinite sequence of events taking values in a domain $\mathbb{D}$. But we do not have the absent event (#) like in the Signal language and consequently all the signals have the same clock. This is the reason why SynDEx is considered "mono-clock".

3.4.2 Architecture model

An architecture is composed of processors, possibly of different types, specific integrated circuits performing a unique function, and point-to-point or multi-point communication media. All these components are together connected. We propose an architecture model which is at an intermediate level between high level models neglecting details of the architecture and accurate low level models. The complexity of this model is sufficient to enable optimizations while it is not too fine leading to combinatorial explosions.

Our architecture model is an oriented graph, denoted by $G_{\text{ar}} = (V, E)$ where V is the set of vertices, and E the set of edges. V corresponds to four kinds of finite state machines (FSM) called operator (V_{O}), communicator (V_{C}), memory (V_{M}) and Bus/Mux/Demux (BMD) with or without arbiter (V_{B}): $V = V_{\text{O}} \cup V_{\text{C}} \cup V_{\text{M}} \cup V_{\text{B}}$, and $V_{\text{O}} \cap V_{\text{C}} \cap V_{\text{M}} \cap V_{\text{B}} = \varnothing$. Each edge $s \in E$ represents a connection between an input and an output of FSMs.

There are two types of memory vertices, random access memory (RAM, $S_{\text{RAM}} \in S_{\text{M}}$) and sequential access memory (SAM, $S_{\text{SAM}} \in S_{\text{M}}$). RAM memories are used to store operations of the algorithm graph, in this case we call them RAM_{P} (program memory). When RAM memories store only data we call them RAM_{D} (data memory). We call them RAM_{DP} when they store both program and data. A RAM may be shared (i.e., connected to several operators and/or communicators), then it may be used for data communications. SAM memories are always shared since they are only used to communicate data using the message passing paradigm. In a SAM, data must be read in the same order as it has been written (as FIFO), all access is then said to be synchronized whereas in a RAM it is not synchronized since it is possible to read data independently of the order of the write operation. The synchronized, or not, properties are exploited during the code generation. SAM may be point-to-point or multi-point (bus), supporting or not broadcasting. Each memory is characterized by its size and its access bandwidth.

Each operator sequentially executes a finite subset of the algorithm operations stored in a RAM_{P} (or RAM_{DP}) which must be connected to the operator. The WCET of an operation depends on the operator and the memory characteristics. An operation executed by an operator reads input data stored in a connected RAM_{D} (or RAM_{DP}) and produces output data

which is written in the RAM_D (or RAM_{DP}) connected to it.

Each communicator sequentially executes communication operations stored in their connected RAM_P (or RAM_{DP}). These operations transfer data from one memory (SAM, RAM_P, RAM_{DP}) connected to the communicator into another memory connected to the same communicator. The WCCT of a communication depends on the size of data to be transmitted but also depends on the available bandwidth computed from all parameters of the edge.

BMD vertices are used to model the bus, the multiplexer and the de-multiplexer of an architecture. When a single sequencer (operator or communicator) requires to access more than one memory, a BMD vertex must be inserted between the sequencer and each memory. When a memory is shared by several sequencer vertices, a BMD including an arbiter (a BMDA) must be inserted between all sequencers and the shared memory.

In our model, a processor or a specific integrated circuit corresponds to an architecture subgraph made of one operator and optionally one or several communicator(s) and BMD(s). The architecture model is detailed in [32] as well as the set of connection rules for building a valid architecture graph. With respect to this model, the only difference between a processor and a specific integrated circuit is that the latter is only able to execute a unique operation whereas a processor may execute, sequentially, several operations. Therefore, only a unique operation may be distributed on a specific integrated circuit.

3.4.3 Implementation model

Given a pair of algorithm and architecture graphs, we transform the algorithm graph according to the architecture graph in order to obtain an implementation graph. This transformation corresponds to a distribution and a scheduling of the algorithm graph.

A distribution, also called allocation or partitioning, is modeled by the relation dist applied to a pair of algorithm and architecture graphs. This produces a distributed algorithm graph G'_{al} such that: $(G_{al}, G_{ar}) \stackrel{dist}{\longrightarrow} (G_{al})'(.)$. The distribution is obtained in three main steps. Firstly, a spatial allocation of the operations onto the operators, leading to inter-operator edges (data dependence between two operations belonging to two different operators). Secondly, each of these edges is replaced by a linear subgraph (sequence of edges and vertices). The new vertices of this subgraph are communication operations which are allocated to the communicators belonging to the different processors forming the route the data dependence has been allocated to. A route is a path in the architecture graph. Thirdly, for each operation allocated to an operator, new "memory allocation vertices" are added and allocated to one of the RAM memories connected to the operator.

A scheduling is modeled by the relation sched applied to a pair (G'_{al}, G_{ar}) so that $(G'_{al}, G_{ar}) \stackrel{sched}{\longrightarrow} (G''_{al})(.)$. For each operator (resp. communicator) the scheduling is a temporal allocation of the operations (resp. communication operations) allocated to this sequencer. This amounts to create new "precedence edges" without data transfer ($e_p \in E_p$) in order to transform the partial order associated with the operations allocated to an operator (resp. communication operations allocated to a communicator). This is necessary because operators (resp. communicators), which are FSMs, require a total execution order between the operations (resp. communication operations) allocated to them. This order must be compatible with the precedences involved by the data dependences of the algorithm graph.

From a pair of algorithm and architecture graphs we get the set of all the possible implementations by composing the two previous relations: $(G_{al}, G_{ar}) \stackrel{dist\ o\ sched}{\longrightarrow} (G''_{al})(.)$ where $G''_{al} = (O \cup V_{alloc_P} \cup V_{alloc_{IO}} \cup V_{alloc_D} \cup V_C \cup V_i, D \cup E_p)$. The partial order of the algorithm graph G_{al} has been transformed in an other partial order, thanks to the added precedence edges E_p, according to the available parallelism of the architecture graph. In [32] we prove that by construction the partial order of G''_{al} includes the partial order of G_{al}.

For a given pair of algorithm and architecture graphs, there is a large but finite number of possible implementations, among which we need to select an optimized one, i.e., which satisfies real-time constraints and minimizes some cost function. This optimization problem, as most resource allocation problems, is equivalent to a "bin packing problem" known to be NP-hard, and then its set of solutions is tremendously huge for realistic applications. This is the reason why we use the term "optimized implementation" rather than "optimal implementation". Consequently, we need to use a heuristic since exact algorithm, giving the optimal solution, should take too much time. The heuristic we chose is a fast and efficient "greedy list scheduling" algorithm, with a cost function based on the "critical path" and on the "schedule flexibility" of the implementation graph. It takes into account the execution WCET of operations and WCCT of communications. This heuristic can be improved by using "local searches" based on "backtracking" whereas greedy heuristics do not backtrack.

As mentioned before, real-time characteristics are associ-

ated with every operation. Some characteristics are dependent of the architecture like the WCET and the WCCT but other ones are not. A period and a deadline are characteristics associated with every operation, which are independent of the architecture. An operation temporally characterized like that is usually called a "real-time task" in the real-time scheduling community, as presented for example in the Liu & Layland's seminal paper [33]. Consequently, we consider data-dependent task systems according to the algorithm graph.

The proposed heuristic is composed of three main steps:

1) Performs an unrolling of the algorithm graph by duplicating every operation according to the ratio between its period and the least common multiple (LCM) of all the operation periods.

2) Performs a multiprocessor real-time schedulability analysis assuming the inter-processor communications have no cost. This analysis is based on schedulability conditions that are different depending on whether the task system is preemptive or non preemptive.

3) Performs the actual scheduling by computing for every processor the start times of every operation that has been allocated to this processor, for computation as well as communication operations. Therefore, the communication costs are, now, taken into account. During this last step the response time of every operation, distributed and scheduled on a processor, is minimized according to a cost function which takes into account its schedule flexibility and the increase of the critical path when an operation has to receive data because it is data-dependent with an other operation distributed and scheduled on a different processor, inducing an inter-processor communication cost. The cost function is detailed in [34]. Moreover, the cost of the communication is also minimized by choosing the shortest routes of the architecture graph.

We propose two versions of this heuristic. The first one considers non-preemptive tasks [35] and thus is best suited for safety critical applications but has a worse schedulability ratio than the other version which considers preemptive tasks [36]. Finally, the off-line scheduling heuristic produces a scheduling table that we call further on in the paper "adequation result". This table is used to generate the real-time embedded distributed code.

4 From AADL to Signal and clock synthesis

An AADL model describes the architecture and execution characteristics of an application system in terms of its constituent software and execution platform components and their interactions. An AADL feature description is mostly made of: i) properties that abstract the behavior of those components as well as their non functionnal characteristics; ii) interconnections; and iii) scheduling. The polychronous semantics of Signal and its constraint programming style make it closer to AADL timing semantics and architectural style than other pure synchronous or asynchronous models.

Such characteristics depend on the hardware executing the software, where the timing properties and execution binding properties are associated. In this section, we mainly handle timing and binding properties of AADL components with regard to our polychronous MoC. Syntactic aspects in the transformation are only briefly described. The timing modeling of AADL applications in the framework of Polychrony is based on these properties.

4.1 AADL to Signal translation principles

The transformation from AADL to Signal is a recursive function applied to an AADL component and its features.

A package, which represents the root of an AADL specification, is transformed into a Signal module, the root of a Signal program. Both allow to describe an application in a modular way.

Each AADL component and its interface is translated into a Signal process and its corresponding interface. Assembly of AADL components, considering connections and binding, is represented by a parallel composition of corresponding translated Signal processes.

- Data components are translated into signals that are present at every instant of the overall application. Signal provides standard data types (Boolean, integer, char, string, ...), array, record and union (bundle) types. It also provides extern type declarations allowing to define abstract constants and standard operators (=) for those types.
- AADL data ports are considered as event data ports that are present on appropriate Dispatch. We consider an event port as an event data port that can be tt or #.
- AADL (remaining event data) ports are translated into Signal processes that contain the required fifos (or memory cell), synchronizations, temporal and non functional properties, and possible local behavior. A "port" process is connected to its owning thread and to its environment owner (as described by the port connections).

– AADL processes and threads are translated into SIGNAL processes.
– AADL subprograms are considered as threads with specific Dispatch and input/output synchronization. When such a subprogram does not contain subprogram calls, it is declared as an extern SIGNAL process.
– AADL properties are translated into SIGNAL property processes (a set of constraints without output).

Each SIGNAL process resulting from translation following the above rules contains:

– An interface consisting of input/output signals translated from the features (ports) provided by the AADL component type.
– Additional control signals depending on the component category (for instance Dispatch and Deadline for a thread).
– A body made of the composition of a process that defines the behavior of the AADL entity and a process that defines its non functional properties.
– A declaration area for processes resulting of the AADL local subcomponents translation and the above structuring processes.

4.2 Timing properties of components

AADL supports an input-compute-output model of communication and execution for threads and port-based communication (see Fig. 4). The inputs of a thread received from other components are frozen at a specified point, represented by Input_Time property (by default the Dispatch time), during thread execution, and made available to the thread for access. From that point on, its content is not affected by the arrival of new values for the remainder of the current execution until an explicit request for input, e.g., the two new arrival values 2 and 3 (see Fig. 4) will not be processed until the next Input_Time. Similarly, the output is made available to other components at time specified by Output_Time property (for data ports by default at Complete or Deadline time depending on the associated port connection communication type).

The key idea for modeling AADL computing latency and communication delay in SIGNAL is to keep the ideal view of instantaneous computations and communications, moving computing latency and communication delays to specific *memory* processes, which introduces delay and well suited synchronizations [37].

4.3 Example: translation of an in event data port

Event data ports are intended to message transmission. An event data port can have a queue associated with it. The default port queue size is 1 and can be changed by explicitly declaring a Queue_Size property association for the port. Queues will be serviced according to the Queue_Processing_Protocol, by default in a first in first out order (fifo). For an in event data port, the items are frozen at Input_Time, and a number of items (depending on the Dequeue_Protocol property) are dequeued and made available to the receiving application through the port variable (implemented as constraint in the property part). This mechanism induces that the content of the port which is accessible to the application does not change during the execution of a dispatch even though the sender may send new values, and assures an input-compute-output model of thread execution.

To take into account the different events used to speficy the semantics of the port, the translation of an in event data port *xx*, with fifo as Queue_Processing_Protocol, is implemented as an instance of SIGNAL process model *xx*_InEventDataPort, composed of *xx*_InEventDataPort_Behavior( ) and *xx*_InEventDataPort_Property( ) subprocesses (see Fig. 5).

The process *xx*_InEventDataPort_Behavior( ) calls the InEventDataPort_Behavior( ) process model, which is defined in a SIGNAL library. Two fifos are used: in_fifo that stores the receiving in event data (write_flow), and frozen_fifo that is accessible by the thread (through read_flow). At Frozen_time_event (depending on the Input_Time, it is provided by the scheduling part), the actual items of the in_fifo are frozen: certain items are moved to frozen_fifo. The inputs arrived after the Frozen_time_event will be available at the

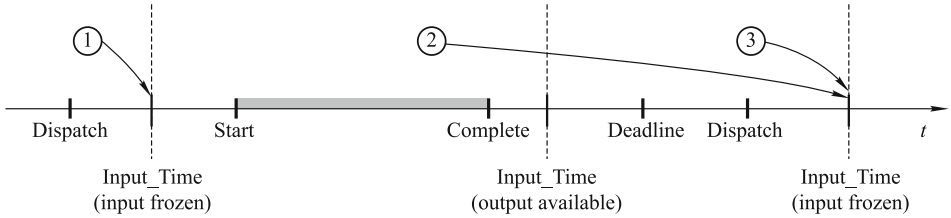


Fig. 4 A time model of the execution of a thread

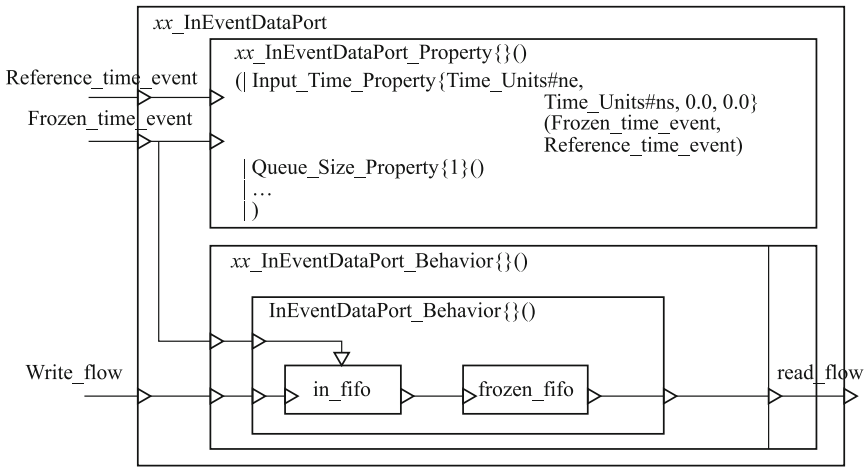


Fig. 5 In event data port translation

next occurrence of Frozen_time_event. The signal Reference_time_event is the event sent when the port is read.

The properties associated with the port are implemented as SIGNAL processes composed in the body of *xx*_InEventDataPort_Property( ): the property values are provided as parameters, and a SIGNAL process (called Input_time_property) is defined in a library to verify whether the constraints are satisfied using the input signals Frozen_time_event and Reference_time_event (in a simulation for example).

4.4 Example: translation of a thread

To further illustrate the principles of the translation, consider (Listing 5) a thread (*T* implemented by *T*.RS) with an input event data port called iport and an output event port called oport. A property Input_Time is assigned to the port iport: Input_Time=>(Time =>Start;Offset=>10 ms..15 ms;). It specifies that the input is frozen at some amount of execution time from the beginning of execution (Start). The time is within the specified time range (10 ms..15 ms).

Listing 5 Example of an AADL thread

```
thread T
features
  iport: in event data port Integer
    { Input_Time=>((Time =>Start; Offset=>10 ms..15 ms;));};
  oport: out event port;
end T;
```

The translation in SIGNAL is given in Listing 6. The interface contains the input/output signals that represent the features provided by the thread declaration, and also some added control signals (top, ctl1, time1, ctl2, alarm). These added signals are provided by the scheduler (input ones) or sent to it (output ones). The body is composed of processes that represent the translation of the properties associated with the thread (*T*_RS_Thread_property( )) and the behavior of the thread (*T*_RS_Thread_behavior( ), which may be described, for example, by some transition system), and the ports with their timing semantics (iport_InEventDataPort( ), oport_OutEventPort( )). The translation of the iport follows the principles described above. The translation of an output event port is quite similar. The properties of a port (default ones or specified ones) are also translated (iport_InEventDataPort_Property( ) and oport_OutEventPort_Property( )): the port queue processing (size, protocol, overflow), and the input/output timing (Input time, Output time) are explicited.

Restrictions. The current version of the AADL to SIGNAL translator does not yet implement all the AADL components. Among non implemented components we distinguish:

- Expected ones: mode, behavior annex, bus access.
- Components that will not be implemented: thread group, subprogram group, virtual processor, virtual bus, flow.
- Other ones that are partly translated (translated with some restrictions on properties such as Dequeue_ Protocol restricted to one item, Queue_Processing_ Protocol restricted to fifo).

4.5 Processor and its affine-scheduling

An AADL model is not complete and executable if the processor-level scheduling is not resolved. A scheduler is therefore required to be integrated so that a complete model is used for the subsequent validation, distribution and

Listing 6 Translation of the *T* thread in SIGNAL

```
process T_RS_Thread =
  ( ? integer iport;
      CTL1 ctl1;
      T_TIME_EVENT time1;
      event top;
    ! boolean oport;
      CTL2 ctl2;
      boolean Alarm; )

  ( % thread behavior %
  | (l_oport,ctl2) := T_RS_Thread_behavior{ }
       (l_iport,ctl1.Dispatch,Start,ctl1.Resume)
    % thread properties %
  | (Start,Alarm) := T_RS_Thread_property{ }(ctl1,top)
   % iport translation %
  | l_iport := iport_InEventDataPort{Integer_INIT,1,1}
      (time1.iport_Frozen_time_event,iport,ctl1.Dispatch)
    % oport translation %
  | oport := oport_OutEventPort{1}
      (time1.oport_Output_time_event,l_oport,ctl2.Complete)
  |)
where
  process iport_InEventDataPort = {...} (...)
  (| read_flow := iport_InEventDataPort_behavior
      {def_value,size,dequeue_number,"iport"}
          (write_flow,Frozen_time_event)
  | iport_InEventDataPort_Property{ }
          (Frozen_time_event,Reference_time_event)
  |)
  where
    type msg_type =Integer;
    process iport_InEventDataPort_behavior = {...} (...)
     (| read_flow := InEventDataPort_Behavior {...}(...) |)
     ;
    process iport_InEventDataPort_Property = {...} (...)
     (| Input_Time_property{Time_Units#ms,Time_Units#ms,
           10,15}(Frozen_time_event,Reference_time_event)
      | Overflow_Handling_Protocol_property
                {Overflow_Handling_Protocol#DropOldest}( )
      | Queue_Size_property{1}( )
      | Queue_Processing_Protocol_property
                {Queue_Processing_Protocol#Fifo}( )
      | Dequeued_Items_property{1}( )
      | Dequeue_Protocol_property{Dequeue_Protocol#OneItem}( )
      |)
    end;
    process oport_OutEventPort = ...
      (| sent_flow := oport_OutEventPort_behavior
           {size,"oport"}(write_flow,Output_time_event)
       | oport_OutEventPort_Property{ }(Output_time_event,
                                        Reference_time_event)
       |)
     where
       type msg_type = boolean;
       process oport_OutEventPort_behavior = {...} (...)
        (| sent_flow := OutEventPort_Behavior{size,
             port_name}(write_flow,Output_time_event) |)
        ;
       process oport_OutEventPort_Property = ...
        (| Output_Time_property{Time_Units#ns,Time_Units#ns,
             0,0}(Output_time_event,Reference_time_event)
        | Queue_Size_property{1}( )
        | Queue_Processing_Protocol_property
                        {Queue_Processing_Protocol#Fifo}( )
        | Overflow_Handling_Protocol_property
                {Overflow_Handling_Protocol#DropOldest}( )
        |)
       end;
end;
```

simulation. A scheduling based on affine clock systems [38] is thus developed for each AADL processor. A particular case of affine relations is affine sampling relation, expressed as $y = \{d \cdot t + \phi \mid t \in x\}$, of a reference discrete time x (d, t, ϕ are integers): y is a subsampling of positive phase ϕ and strictly positive period d on x. Affine clock relations yield an expressive calculus for the specification and the analysis of time-triggered systems. The scheduling based on the affine clocks can be easily and seamlessly connected to POLYCHRONY for formal analysis.

4.6 Binding

For a complete system specification, the application component instances must be executed by the appropriate execution platform components. How to combine the components of a system to produce a physical system implementation is called binding in AADL.

A process is bound to the processor specified by the Actual_Processor_Binding property. Support for process/threads execution may be embedded in the processor hardware, or it may require software that implements processor functionality. Such software must be bound to a memory component that is accessible to the processor via the Actual_Memory_Binding property. The interactions among these execution platform components are enabled through a bus via the Actual_Connection_Binding property.

Binding properties are declared in the system implementation that contains in its containment hierarchy both the components to be bound and the execution platform components that are the target of the binding. This binding information, as well as timing properties, is reflected in the generated SIGNAL program by SIGNAL property processes representing these properties. In a further step, the SIGNAL program is analyzed and useful properties are kept as specific pragmas.

4.7 Timing analysis and clock synthesis

The previous translation from AADL to SIGNAL concentrates on the timing modeling of software architectures in AADL. This modeling enables a formal timing analysis and clock synthesis, based on the polychronous MoC.

4.7.1 Timing analysis

Timing analysis in this paper mainly refers to analyzing clock relations based on clock hierarchy. The clock hierarchy of a process is a component of its data control graph (DCG). The DCG is made of a multigraph G and a clock system Σ.

- Description of G:
  - A node n in G represents a polychronous process process(n); it has input signals input(n) and output signals output(n) which are those of process(n); it has a clock clock(n) which is either a Boolean variable or a clock formula acting as a guard.
  - A directed edge (n_1, n_2, a, h) links a source node n_1 to a target node n_2; it is labeled by the name a of the signal that is sent by n_1 and received by n_2; it has a clock h. The signal a is transmitted when h is tt. Two nodes n_1 and n_2 may be linked by several edges.
- Description of Σ:
  - Given X the set of signal names and $B \subset X$ the set of Boolean signal names, an elementary formula is either $\hat{0}$ that stands for #, $\hat{x}$ that denotes the clock of a signal x, $[b]$ that denotes the clock representing the tt occurrences of a Boolean signal b, or $[-b]$ that denotes the clock representing the ff occurrences of a Boolean signal b. The set of elementary formulas is denoted EF.
  - The set of clock formulas CF is the smallest set that satisfies:
    - EF $\subset$ CF
    - for all formulas $f_1, f_2 \in$ CF, $f_1 \widehat{+} f_2$ and $f_1 \widehat{*} f_2$ belong to CF.
  - A clock equation is a class of equivalent clock formulas.
  - Σ is a forest (set of trees) of clock equations.
  - The node n of a tree is a clock equation; it contains the list of signals signal(n) the clock of which is equal to this class.
  - A tree is en endochronous process.
  - Each subtree S of a tree T is itself an endochronous process; its clock is a ("non recursive") function of the signals signal(n) where n is a node in T.

The following DCG levels are distinguished (see Fig. 6) in the compiling stages of SIGNAL programs:

- Every SIGNAL process is represented by a general DCG as described above, the **DCGPoly**; its clock system Σ can be normalized using:
  - elementary Boolean to event transformations such as E `when` (a `and` b) $\rightarrow$ (E `when` a) `when` b,
  - transformations based on algebraic properties such as transitivity of partial order.
- When the forest Σ of a process P is a tree, P is an endochronous process; its DCG belongs to the **DCGEndo** level. One can reduce a consistent forest to a tree by:
  - building a triangular system of equations equivalent to the original one, and
  - adding supplementary required parameters to the system as it is done to build `memoryCell` from `preMemoryCell` (Listings 2, 3).
- An endochronous process P may be free of event type signals and binary event expressions ($\widehat{+}$, $\widehat{*}$). It can still use signals that are not always defined. Then P is said to be a Boolean clocked process and its DCG belongs to the **DCGBool** level. It is always possible to transform a usual endochronous process into a Boolean clocked process that is equivalent: an event signal `h` can be associated with a (possibly new) Boolean signal C such that each occurrence of h is replaced by `true when` C provided that C is equal to # or ff when h is #.
- A Boolean clocked process in which all Boolean clock signals are synchronous is named flat. Its associated DCG belongs to the **DCGFlat** level. The clock of these Boolean clock signals is the root of the clock hierarchy. The clock hierarchy has at most two levels: subclocks of the root represent the instants at which Boolean clocks are tt: Boolean clocks are used as guards for the processes associated with the nodes of the DCG. It is al-

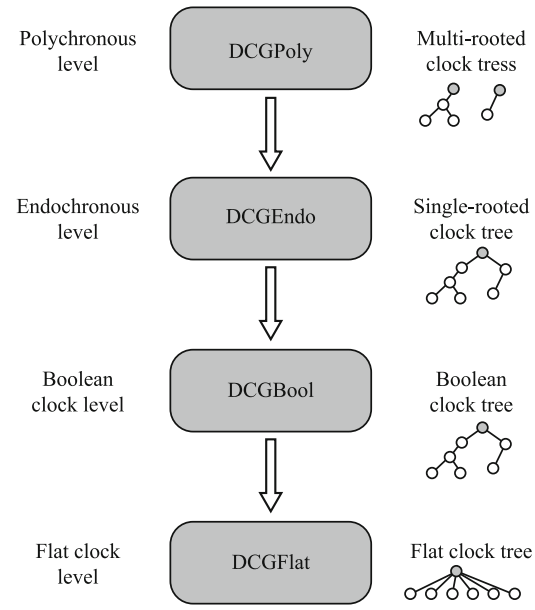


Fig. 6 Clock hierarchy transformations in the timing analysis and synthesis in SIGNAL

ways possible to transform a Boolean clocked process into an equivalent flat one. Quartz [39] and SynDEx belong to this class of processes.

4.7.2 Clock synthesis

In a clock tree with only one root, the simulation clock (the fastest clock in the system) is based on the root clock (i.e., the tick). In this case, the system is endochronous and it is possible to build the unique deterministic behavior in the code generation. But if there is no common root for all the trees, i.e., there is no fastest clock, the system is polychronous, and non deterministic concurrency is thus introduced. In an AADL multi-processor specification, it is generally hard to find the fastest clock, as each component may have its own activation clock or frequency. Code generation considering the deterministic behavior is therefore difficult, even impossible. To tackle this issue, independent clocks, particularly the root clocks in different trees, are required to be synchronized. This synchronization, called endochronization, can be performed in an ad-hoc way by the compiler, or in a specific way in a manual manner. More details can be found in [20]. A more general clock synchronization method via controller synthesis is also possible [40]. Endochronization leads to the transformation from the DCGPoly level to the DCGEndo level.

5 From Signal to SynDEx and architecture exploration

Both SynDEx and Signal belong to the family of synchronous/polychronous languages. However, there are still differences to consider for the translation. First, Signal is based on a polychronous MoC, whereas SynDEx relies on the synchronous MoC. Clock synthesis is therefore required to endochronize multi-clocked specifications before the translation (presented in Section 4). Secondly, system representations of Signal and SynDEx, based on graphs, are at different abstraction levels. Transformation between these levels is thus required. The DCGFlat level is appropriate for translating Signal programs to SynDEx algorithm graphs. At this level, all the clocks have been expanded up to the most frequent clock, and state variables are also defined at this most frequent clock. Other signals can have a "don't care" interpretation at the instants at which they are #. This corresponds to the SynDEx interpretation. Execution platform and real-time characteristics, preserved in Signal pragmas, are translated into SynDEx architecture and constraints.

5.1 Syntactic translation rules

The correspondence between Signal and SynDEx representations is defined in two aspects:

- The first one is related to the algorithm [41], including clock hierarchy, nodes and dependences. Clock hierarchy plays the semantic role in Signal and serves as a structural backbone in the translation: each clock in the hierarchy is associated with a SynDEx algorithm graph. This algorithm combines the computations to be performed according to this clock, which is represented by a SynDEx `condition`. The translations of nodes and dependences are mainly syntactic, which will be detailed later.
- The second one is related to architecture, allocation constraints, temporal information: these information are preserved in pragmas of the original Signal programs and will be finally translated into SynDEx architecture and *constraints* (presented in next subsection).

As this translation includes large syntactic details, only some key rules are briefly presented.

In the DCGFlat representation of a Signal program, the clock hierarchy, which is a tree, contains at most two levels: the root of the tree (the tick), and the children level, which comprises less frequent clocks defined by the tt values of the Boolean clock signals. Let us denote these Booleans by $b_1, b_2, \ldots, b_n$, and the clock $[b_i]$ denotes the instants at which b_i is tt. These clocks are subclocks of the clock *tick*. Since during the Signal compilation, each clock is associated with the subgraph containing the nodes that are defined at this clock, the translation strategy is the following: whenever a graph is associated with a clock, this graph is translated as an algorithm in SynDEx.

5.1.1 Translation of the clock hierarchy

Any graph g, associated with a clock that is a child of the tick, is translated into a SynDEx algorithm (which is a graph of operations), with a condition defined from the clock $[b_i]$ of this graph. The translation of g can be sketched as follows:

```
def algorithm P_bi:
 ?  ...
 !  ...
conditions:  bi = 1
```

The top level graph P contains the nodes associated with the tick, as well as references to each of the graphs associated

with the subclocks of *tick*. The translation of P is a SYNDEX algorithm P with `conditions: true`.

The input and output signals (called ports in SYNDEX) are translated from signals communicating between clock graphs. Interface signals of a whole program are translated into sensors and actuators in SYNDEX.

5.1.2 Translation of nodes

A SYNDEX algorithm is mainly composed of four elements: ports, conditions, references, and dependences. Ports and conditions are generated along with the translation of the clock hierarchy, references and dependences are generated according to the attached nodes of the considered clock. In DCGFlat, the nodes can be categorized into:

- **Constants** The constants are explicitly represented by references to constant vertices in SYNDEX.
- **Equations** A general form of equations is considered: $X := F(Y_1, Y_2, \ldots, Y_n)$, where F represents SIGNAL n-ary operators. Only elementary expressions are considered as it is always possible to translate other expressions into a composition of elementary definitions. SYNDEX defines libraries to provide basic algorithm (atomic operation) declarations for SIGNAL elementary operators. Let $X := F(Y_1, Y_2, \ldots, Y_n)$ an operation of type T, a corresponding SYNDEX algorithm named F with type T is declared in a predefined library. For example, the predefined algorithm corresponding to the SIGNAL `default` for integer type (as it is represented in the DCG) is defined as follows (dependences are discussed below):

```
def algorithm default:
?  int i1; ?  bool i2; ?  int i3;
!  int o;
conditions:  i2 = 1;
references:
dependences:  strong_precedence_data i1-> o;
conditions:  i2 = 0;
references:
dependences:  strong_precedence_data i3-> o;
description:  "x := (i1 when i2) default
                   (i3 when (not i2))"
```

  When translating such an equation, only a reference to F associated with T is used, T/F.

  Note that there is no need of predefined algorithm representing the `when` operator. Indeed, in the DCG representation, for a process $X := E$ when b, b represents the clock of X and the corresponding node is associated with the subclock $[b]$. This subclock is taken into account when translating the clock hierarchy, thus, in the algorithm corresponding to this subclock, $X := E$ `when` b may be considered as $X := E$.
- **Memorizations** The translation of a memorization $ZX := X\$$ is similar to the translation of the equations, except that it is a reference to a *memory*.
- **Partial definitions** Let us consider the following partial definitions: $X ::= E_1$ `when` $b_1, \ldots, X ::= E_n$ `when` b_n. In the DCG representation, $b_1, b_2, \ldots, b_n$ are clocks that are exclusive. When translating such partial definitions, the original graph is rewritten as follows:

```
X := (E1 when b1) default ...
                  default (En when bn)
```

- **Process calls** The called process `Q` is defined explicitly as a SYNDEX algorithm. A call to the process Q is translated into a reference to the algorithm Q.
- **External calls** The translation of an external process is similar to the translation of any process, except that the body of the algorithm is empty since the process is not defined in SIGNAL.

5.1.3 Translation of dependences

After the references are declared for each node, dependences between them are set. The dependences (the edges of the SYNDEX graph) are built either from data-flow connections of the SIGNAL graph or from explicit precedence relations.

For each data-flow connection d from the mth output of a node p_k to the nth input of a node p_l, the translation associates a `strong precedence data` with the corresponding translated vertices in SYNDEX: `strong_precedence_data p_k.o_m -> p_l.i_n;` (an example of such a dependence has been given for the predefined `default`).

Another kind of dependence is used to express *precedence* relations. A precedence $E_i \rightarrow E_j$ in the DCG is translated directly as a corresponding `precedence` in SYNDEX.

5.2 Translating non-functional aspects

In the ASME2SSME transformation, execution platform, timing properties and binding specified in AADL are preserved in the pragma part of SIGNAL programs. In the translation from SIGNAL to SYNDEX, the information of execution platform, related to processing and communication units, are taken from the pragmas and translated into an architecture graph in SYNDEX. For example, the architecture of the SDSCS case study is translated as follows (extract):

```
def architecture IMA:
```

```
operators:
  CPIOM CPIOM1;
  RDC RDC1;
medias:
  AFDX AFDX1 no_broadcast;
connections:
  CPIOM1.io AFDX1;
  RDC1.io AFDX1;
```

CPIOM and RDC are types of computing units and AFDX is the type of bus. The computing units are connected via the bus.

Timing properties, such as period, deadline, computation time, are also taken from pragmas and finally set to corresponding SynDEx nodes. Here is an example from SDSCS, where the computing time of AIRSPEED and CLL on RDC is 1 000 units:

```
extra_durations_operator RDC:
  AIRSPEED = 1 000;
  CLL = 1 000;
```

The binding information of software and hardware are translated into groups and allocation constraints in SynDEx, i.e., all the software components allocated on the same processing unit are put into the same group, and then an allocation constraint is specified in SynDEx so that this group is allocated on the same processing unit. In the following example, on_ground and in_flight are declared in group g_rdc1, which is allocated onto the computing units, RDC1 or RDC2.

```
operation_group g_rdc1:
  [\\aadl_door\on_ground,attach_all]
  [\\aadl_door\in_flight,attach_all]
absolute constraint:
  g_rdc1 on IMA.RDC1 IMA.RDC2;
```

5.3 Architecture exploration with SynDEx

SynDEx allows the designer to explore manually and/or automatically the design space solutions using schedulability analyses and optimization heuristics for distributing (allocating) and scheduling the application functions (operations) onto the processors while satifying real-time constraints and minimizing computing resources. The exploration results (multiprocessor scheduling table) show the real-time behavior of the functions allocated onto components of the architecture, i.e., processors, specific integrated circuits, and communication media. This approach conforms to the typical architecture exploration process [42, 43]. In addition, code is automatically generated as a dedicated real-time executive, or as a configuration file for a resident real-time operating system such as Linux, Linux/RTAI, Windows/RTX, Osek [44].

6 An avionic case study

SDSCS is a generic simplified version of the system that manages passenger doors on Airbus series aircrafts. As a safety-critical system, in addition to the fulfillment of safety objectives, high-level modeling and component-based development are also expected for fast and efficient design.

In this example, each passenger door has a software handler, which achieves the following tasks: (1) monitor door status via door sensors; (2) control flight lock via actuators; (3) manage the residual pressure of the cabin; (4) inhibit cabin pressurization if any external door is not closed, latched and locked. The four tasks are implemented with simple logic that determines the status of monitors, actuators, etc., according to the sensor readings. In addition to sensors and actuators, SDSCS is equipped with other hardware components, e.g., CPIOMs (Core Processing Input/Output Modules) and RDCs (Remote Data Concentrators) are connected via the

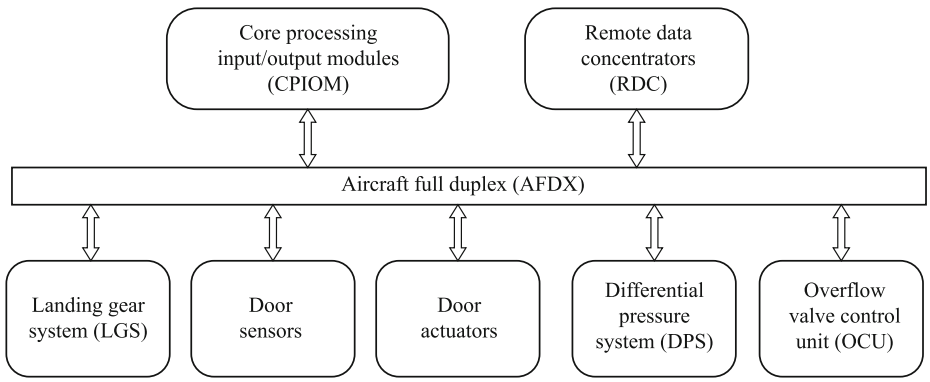


Fig. 7 A simple example of the SDSCS architecture

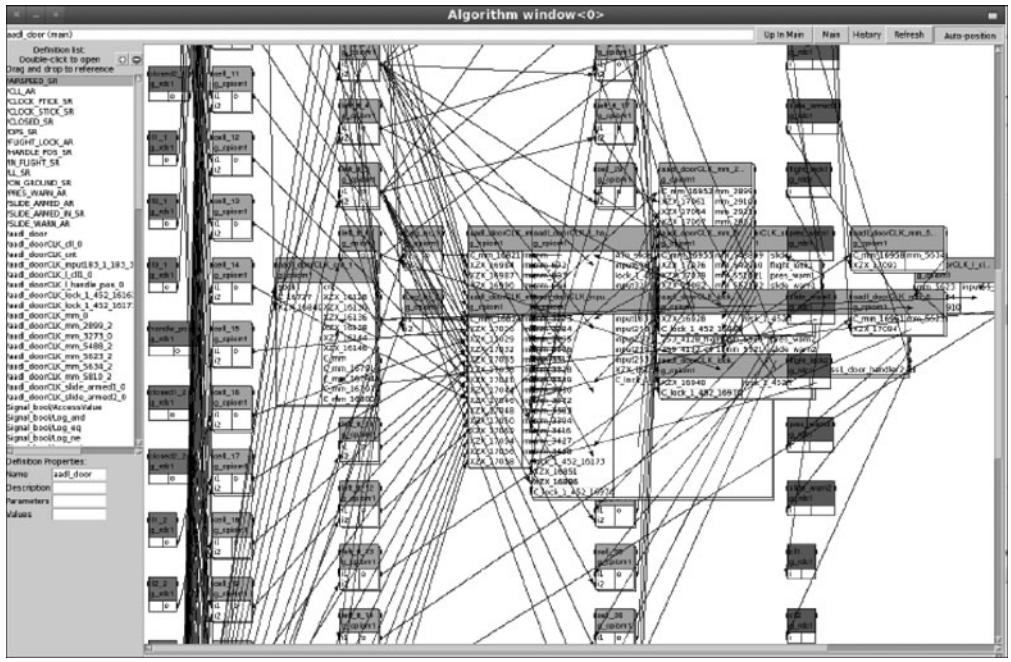


Fig. 8 A partial view of the SYNDEX algorithm graph

AFDX (Aircraft Full DupleX) network (see Fig. 7). Sensors and actuators are also connected to RDCs and communicate with other systems via AFDX.

An overview of SDSCS modeled in AADL is shown in Fig. 3. The whole system is presented as an AADL system. The two doors, modeled as subsystems, are controlled by two processes doors_process$_1$ and doors_process$_2$. Each process contains three threads to perform the management of doors. All the threads and devices are periodic and share the same period. The process doors_process$_1$ (resp. doors_process$_2$) is executed on a processor CPIOM1 (resp. CPIOM2). The bus AFDX connects the processors, CPIOM1 and CPIOM2, and devices that model the sensors and actuators, such as DPS, OCU, etc.

The case study was successfully transformed with the toolchain described in Fig. 2. From a .aadl file (AADL textual file), we first get.aaxl (AADL model file) with OSATE. Aided by high-level and low-level APIs, ASME2SSME is able to read the AADL models. While performing the model transformation towards the SSME model, a thread-level scheduler is also integrated. The SSME model is transformed into SIGNAL code, from which static analysis can be performed. The SIGNAL code can be transformed into .sdx file (SYNDEX code), as well as other format for model checking [45] and simulation [46]. From the SYNDEX tool, the .sdx file is read, and the adequation (optimized multiprocessor real-time scheduling) can be carried out. Figure 8 illustrates a partial view of the SYNDEX algorithm graph describing a partial order on the execution of the functions (vertices of the graph) through data dependences (edges of the graph). This algorithm graph is translated from the AADL functional part of SDSCS. Figure 9 shows the SYNDEX architecture graph, which is translated from the AADL architectural part of SDSCS. The architecture is composed of two processing units (CPIOM1 and CPIOM2), two concentrators (RDC1 and RDC2) and a communication media (AFDX1) of AFDX bus type, that are together connected.

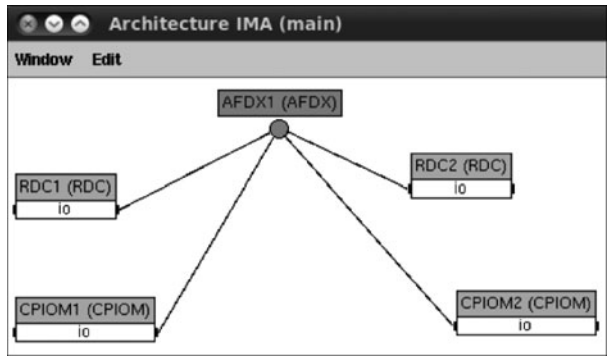


Fig. 9 The SYNDEX architecture graph

Figure 10 illustrates a partial view of the adequation results (scheduling table) obtained with SYNDEX: the algorithm (see Fig. 8) is distributed and scheduled onto the architecture (see Fig. 9). There are five columns in the figure, which represent the five components of the architecture. From left, the first column represents the bus AFDX1. The second and fourth

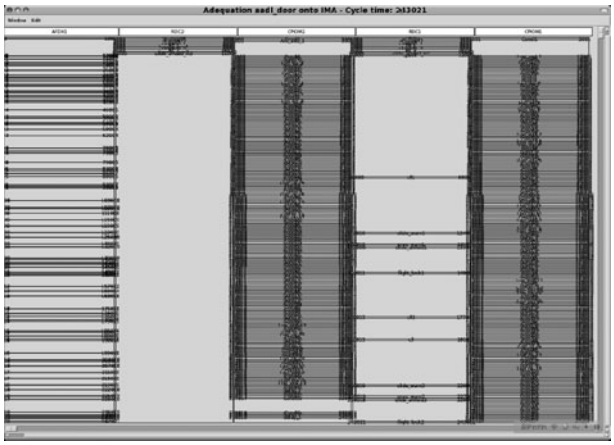


Fig. 10 A partial view of the SynDEx adequation result: a scheduling table

columns represent RDC1 and RDC2 concentrators. The third and fifth columns represent CPIOM1 and CPIOM2 processors. For every column, the start time and the complete time of every operation (function) and every communication operation are given and materialized by a horizontal line. Thus, the processing time, flowing from top to bottom, of every operation is represented by a vertical box. In this case study, the algorithm has more than 150 nodes and the architecture has five nodes in SynDEx. The adequation takes about 15 minutes 35 seconds in average. With this toolchain, it is easy to change the configuration of the execution platform and binding. For example, the number of processing units can be changed, as well as the type of processing units and communication media. The influence of these changes has finally a repercussion in SynDEx in order to be exploited. In addition, generation of dedicated distributed real-time executives with optional real-time performance measurement is also possible for different processors and communication media [29]. The names of AADL components are always kept in all the transformations in order to enable traceability. Hence, our approach provides a fast yet efficient architecture exploration for the design of distributed real-time and embedded systems.

7 Related work

Many contributions on schedulability analysis and scheduling tools exist for AADL specifications, such as [3], [47], and [48]. The main reason to choose SynDEx in our work is that SynDEx and Signal have similar synchronous semantics; this proximity reduces the difficulties in the translation. In addition, the scheduler generated by SynDEx can be integrated, in a manual manner currently, into the original Signal programs for formal verification purpose.

AADL has been connected to many formal models for analysis and validation. The AADL2Sync project [17] and the compass approach [4] provide complete toolchains from modeling to validation, but they are generally based on the synchronous semantics, which is not close to AADL timing semantics. AADL2BIP [6] allows simulation of AADL models, as well as application of particular verification techniques, i.e., state exploration and component-based deadlock detection. The AADL2Fiacre project [49] mainly concentrates on model transformation and code generation, in other words, formal analysis and verification are performed externally with other tools and models. The previous projects do not address code distribution and real-time scheduling in a general sense. The Ocarina project [7] considers code generation for distributed real-time and embedded systems, but using formal models and semantics is not reported in the work.

8 Conclusion

In this paper, we present our proposed approach to address architecture exploration based on AADL, Polychrony, and SynDEx. Software architecture is specified in AADL at a high level of abstraction; Polychrony provides the polychronous model of computation, formal analysis, clock synthesis, and transformations to bridge between AADL and SynDEx; SynDEx is finally used for multiprocessor real-time schedulability analyses and resource optimizations, both allowing architecture explorations. With a reduced design cost of embedded systems, our approach and its associated toolchain make it possible to perform architecture exploration at the earliest design stage. An avionic case study is used to illustrate our approach.

In a longer term, as a perspective, the translation between Signal and SynDEx is expected to be bidirectional, i.e., a particular distribution and scheduling determined by SynDEx is automatically synthesized in the Signal programs for the purpose of formal verification, performance evaluation, and other analyses.

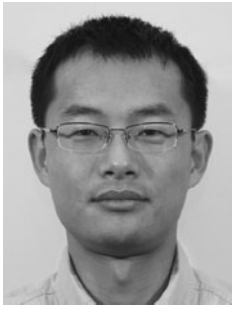
References

1. SAE (Society of Automotive Engineers) Aerospace. Aerospace Standard AS5506A: architecture analysis and design language (AADL). SAE AS5506A, 2009
2. Feiler P, Gluch D. Model-based engineering with AADL. Addison Wesley Professional, September 2012
3. Singhoff F, Legrand J, Nana L, Marcé L. Scheduling and memory requirements analysis with AADL. Ada Letters. 2005, 1–10
4. Bozzano M, Cimatti A, Katoen J P, Nguyen V, Noll T, Roveri M. Safety, dependability, and performance analysis of extended AADL

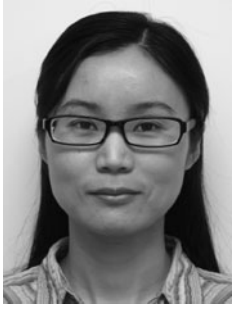
models. The Computer Journal, 2011, 54(5): 754–775

5. Feiler P, Hansson J. Flow latency analysis with the architecture analysis and design language (AADL). Technical Report, CMU, 2007
6. Chkouri M, Robert A, Bozga M, Sifakis J. Models in software engineering. Translating AADL into BIP-Application to the Verification of Real-Time Systems. Springer-Verlag, 2009
7. Hugues J, Zalila B, Pautet L, Kordon F. From the Prototype to the final embedded system using the ocarina AADL tool suite. ACM Transactions in Embedded Computing Systems (TECS), 2008, 7(4): 42:1–42:25
8. Yang Z, Hu K, Ma D, Pi L. Towards a formal semantics for AADL behavior annex. In: Proceedings of the 2009. Conference on Design, Automation and Test in Europe. 2009, 1166–1171
9. Ma Y, Yu H, Gautier T, Le Guernic P, Talpin J P, Besnard L, Heitz M. Toward polychronous analysis and validation for timed software architectures in aadl. In: Proceedings of the 2013 Conference on Design, Automation and Test in Europe. 2013, 1173–1178
10. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages twelve years later. Proceedings of the IEEE, 2003, 9(1): 64–83
11. Le Guernic P, Talpin J P, Le Lann J C. Polychrony for system design. Journal for Circuits, Systems and Computers, 2002, 12: 261–304
12. Talpin J P, Le Guernic P, Shukla S, Doucet F, Gupta R. Formal refinement checking in a system-level design methodology. Fundamenta Informaticae, 2004, 62(2): 243–273
13. Sorel Y. Massively parallel computing systems with real time constraints: the "algorithm architecture adequation" methodology. In: Proceedings of the 1st International Conference on Massively Parallel Computing Systems. 1994, 44–53
14. The polychrony toolset. http://www.irisa.fr/espresso/Polychrony/
15. Gamatié A. Designing embedded systems with the SIGNAL programming language. Springer, 2010
16. Sorel Y. SynDEx: system-level CAD software for optimizing distributed real-time embedded systems. ERCIM News, 2004, 59: 68–69
17. Jahier E, Halbwachs N, Raymond P. Synchronous modeling and validation of priority inheritance schedulers. In: Fundamental Approaches to Software Engineering, Springer, 2009, 140–154
18. Girault A. A survey of automatic distribution method for synchronous programs. In: Maraninchi F, Pouzet M, Roy V, eds, Proceedings of the 2005 International Workshop on Synchronous Languages, Applications and Programs, ENTCS. April 2005
19. Cost-efficient methods and processes for safety relevant embedded systems (CESAR project). http://www.cesarproject.eu/
20. Besnard L, Gautier T, Le Guernic P, Talpin J P. Compilation of polychronous data flow equations. In: Shukla S, Talpin J P, eds, Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction, Springer, 2010, 1–40
21. An industry working group focusing on open source tools for the development of embedded Systems. http://polarsys.org/
22. Eclipse modeling framework project (EMF). http://www.eclipse.org/modeling/emf/
23. OSATE V2 project. http://gforge.enseeiht.fr/projects/osate2/
24. Abramsky S, Jung A. Domain theory. In: Abramsky S, Gabbay D, Maibaum T, eds, Handbook of Logic in Computer Science, volume 3, 1–168. Oxford University Press, 1994
25. Kahn G. The semantics of a simple language for parallel programming. Information Procesing, 1974, 471–475
26. Plotkin G. A powerdomain construction. SIAM Journal on Computing, 1976, 5: 452–487
27. Sorel Y. Syndex: system-level cad software for optimizing distributed real-time embedded systems. Journal ERCIM News, 2004, 59: 68–69
28. The syndex software. http://www.syndex.org
29. Grandpierre T, Sorel Y. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In: Proceedings of the 1st ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'03). 2003, 123–132
30. Dennis J. First version of a dataflow procedure language. In: Lecture notes in computer science, volume 19, 362–376. Springer-Verlag, 1974
31. Harel D, Pnueli A. On the development of reactive systems. In: Apt K, ed, Logics and Models of Concurrent Systems. Springer Verlag, New York, 1985
32. Grandpierre T. Modèisation d'architectures parallèes hétérogènes pour la génération automatique d'exéutifs distribués temps réel optimisés. PhD thesis, Universitéde Paris Sud, Spéialité éectronique, 2000
33. Liu C, Layland J. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of ACM, 1973, 14(2): 46–61
34. Grandpierre T, Lavarenne C, Sorel Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of the 7th International Workshop on Hardware/Software Co design, CODES'99. 1999, 74–78
35. Kermia O, Sorel Y. A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In: Proceedings of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07. September 2007, 1–6
36. Ndoye F, Sorel Y. Safety critical multiprocessor real-time scheduling with exact preemption cost. In: Proceedings of the 8th International Conference on Systems, ICONS'13. January, 2013, 127–136
37. Ma Y, Yu H, Gautier T, Talpin J P, Besnard L, Le Guernic P. System synthesis from AADL using polychrony. In: Proceedings of the 2011 Electronic System Level Synthesis Conference. 2011, 1–6
38. Smarandache I, Gautier T, Le Guernic P. Validation of mixed Signal-Alpha real-time systems through affine calculus on clock synchronisation constraints. In: Proceedings of the 1999 World Congress on Formal Methods. 1999, 1364–1383
39. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin J P. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. Design Automation for Embedded Systems, 2012, 1–35
40. Yu H, Talpin J P, Besnard L, Gautier T, Marchand H, Le Guernic P. Polychronous controller synthesis from MARTE CCSL timing specifications. In: Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'11). 2011, 21–30
41. Pan Q, Gautier T, Besnard L, Sorel Y. Signal to SynDEx: translations between synchronous formalisms. 2003
42. Pimentel A, Erbas C, Polstra S. A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Transactions on Computers, 2006, 55(2): 99–112
43. Gries M. Methods for evaluating and covering the design space during early design development. Integration, the VLSI Journal, 2004, 38(2): 131–183
44. Osek. http://www.osek-vdx.org/

45. Ma Y. Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of compotation. PhD thesis, University of Rennes 1, 2010
46. Yu H, Ma Y, Glouche Y, Talpin J P, Besnard L, Gautier T, Guernic L P, Toom A, Laurent O. System-level co-simulation of integrated avionics using polychrony. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11). 2011, 354–359
47. Sokolsky O, Lee I, Clarke D. Schedulability analysis of AADL models. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing. 2006, 179
48. Gui S, Luo L, Li Y, Wang L. Formal schedulability analysis and simulation for AADL. In: Proceedings of the 2008 International Conference on Embedded Software and Systems (ICESS). 2008, 429–435
49. Berthomieu B, Bodeveix J P, Farail P, Filali M, Garavel H, Gaufillet P, Lang F, Vernadat F. Fiacre: an intermediate language for model verification in the topcased environment. In: Proceedings of the 2008 International Conference of Embedded Real Time Software. 2008



Huafeng Yu has been an expert research engineer within INRIA Rennes, France. His work is involved in timing analysis, formal verification, simulation, and synthesis of MARTE-based timed systems, AADL, and Simulink in the framework of several European projects, such as CESAR and OPEES. He completed his Master's study in Systems and Software at Université Joseph Fourier Grenoble 1 (France) in 2005. He received his PhD in Computer Science from Université des Sciences et Technologies de Lille (France) in 2008. He is now working in Toyota InfoTechnology Center USA as a senior researcher. His main research interests include model-based systems engineering, automotive and aerospace engineering, embedded systems design, formal methods, and synchronous languages.



Yue Ma has been a post-doc fellow in IRISA/INRIA Rennes, France. She works on the modeling, temporal analysis, formal verification and simulation of globally asynchronous locally synchronous systems, especially AADL using polychrony in the framework of European TopCased, CESAR and OPEES projects. She received her PhD in Computer Science from University of Rennes 1 (France) in 2010. She is now working in itemis France as a senior software architect. Her research interests include software engineering, embedded systems design, synchronous programming, AADL modeling and analysis, automotive engineering, such as AUTOSAR and EAST-ADL.

Thierry Gautier is a researcher with INRIA. He received the graduate degree from the Institut National des Sciences Appliqu'ees, Rennes, France, in 1980, and the PhD in Computer Science from Université de Rennes 1 in 1984. He is one of the designers of the signal language, the polychronous model of computation and the Polychrony toolset. His main research interests lie in the safe design of complex embedded systems, including formal modeling, formal validation, and transformations of models to target architectures.

Loïc Besnard is currently a senior engineer at CNRS, France. He received his PhD in Computer Science from Université de Rennes, France (1992). His research interests include the software reliability for the design of embedded systems: modeling, temporal analysis, formal verification, simulation, and synthesis of embedded systems. He is involved in the development of the polychony toolset based on the synchronous language signal.

Jean-Pierre Talpin is a senior researcher with INRIA and leads the project-team who develops the open-source polychrony environment. He received his PhD from Université Paris VI Pierre et Marie Curie in 1993. He then was a research associate with the European Computer-Industry Research Centre in Munich before to join INRIA in 1995. Jean-Pierre edited two books with Elsevier and Springer, guest-edited more than ten special issues of ACM and IEEE scientific journals, and authored more than 20 journal articles and book chapters and 60 conference papers. He received the 2004 ACM Award for the most influential POPL paper, for his 2nd conference paper with Mads Tofte, and the 2012 LICS Test of Time Award, for his 1st conference paper with Pierre Jouvelot.

Paul Le Guernic graduated from Institut National des Sciences Appliquées de Rennes in 1974. He performed his Thèse de troisième cycle in Computer Science in 1976. From 1978 to 1984 he had a research position at INRIA. He is the Directeur de Recherche in this institute since 1985. He has been the head of the "Programming Environment for Real Time Applications" group, which has defined and developed the signal language. His main current research interests include the development of theories, tools, and methods, for the design of real-time embedded heterogeneous systems. He is one of the architects of the polychrony toolset.

Yves Sorel is a research director at INRIA (National Institute for Research in Computer Science and Control), and scientific leader of the Rocquencourt's team AOSTE (models and methods of analysis and optimization for systems with real-time and embedding constraints). He is a member of the Steering Committee of the Competitivity Cluster OCDS/SYSTEM@TIC Paris-Region. His main research topic concerns the analysis and the optimization of distributed real-time embedded systems. He is also the founder of the AAA methodology (Algorithm Architecture Adequation) and of SynDEx, a system level CAD software downloadable free of charge at: www.syndex.org. More details, including a publication list, can be found at: www-rocq.inria.fr/ sorel/work.