Polychronous Modeling, Analysis, Verification and Simulation for Timed Software Architectures

Huafeng Yu[a], Yue Ma[a], Thierry Gautier[a], Loïc Besnard[b], Paul Le Guernic[a], Jean-Pierre Talpin[a]

[a]*INRIA Rennes - Bretagne Atlantique, 263 Avenue Général Leclerc, 35042, Rennes, France*
[b]*IRISA/CNRS, 263 Avenue Général Leclerc, 35042, Rennes, France*

Abstract

High-level modeling languages and standards, such as Simulink, SysML, MARTE and AADL (Architecture Analysis & Design Language), are increasingly adopted in the design of embedded systems so that system-level analysis, verification and validation (V&V) and architecture exploration are carried out as early as possible. This paper presents our main contribution in this aim by considering embedded systems architectural modeling in AADL and functional modeling in Simulink; an original clock-based timing analysis and validation of the overall system is achieved via a formal polychronous/multi-clock model of computation. In order to avoid semantics ambiguities of AADL and Simulink, their features related to real-time and logical time properties are first studied. We then endue them with a semantics in the polychronous model of computation. We use this model of computation to jointly analyze the non-functional real-time and logical-time properties of the system (by means of logical and affine clock relations). Our approach demonstrates, through several case-studies conducted with Airbus and C-S Toulouse in the European projects CESAR and OPEES, how to cope with the system-level timing verification and validation of high-level AADL and Simulink components in the framework of Polychrony, a synchronous modeling framework dedicated to the design of safety-critical embedded systems.

Keywords:
Polychrony, Signal, AADL, Simulink, model-driven engineering, timing modeling, formal analysis and verification

1. Introduction

Due to the fast development of hardware and software in recent years, embedded systems are featured by their high-performance and their dramatically increased complexity. At the same time, the overall cost and time to market are required to be reduced as much as possible, while improving their reliability.

Raising level of abstraction and early-phase validation in system design allow for an effective development in response to the previous issues, and they have received great attention from the system-level design community. High-level standardized modeling languages, such as Simulink [1], Systems Modeling Language (SysML) [2], Architecture Analysis & Design Language (AADL) [3], the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [4], AUTOSAR (AUTomotive Open System ARchitecture) [5], and EAST-ADL [6] are gradually adopted for system specification. In particular, AADL permits the fast yet expressive modeling of a system, including software architecture, execution platform, and their binding. Early-phase analysis and validation, e.g., schedulability, reliability, and allocation, can be rapidly performed based on the AADL models [7], [8], [9], [10], [11], [12].

Although AADL provides a fast design entry, there still exists critical challenges, such as unambiguous semantics, timing analysis, formal verification, co-modeling, distribution and simulation. In particular, timing analysis of high-level and possible heterogeneous models, system architecture/behavior co-modeling, formal verification of the system correctness, and simulating the mapping of concurrent software components onto target parallel architecture, are still difficult to be addressed. To address these issues, both expressive formal models and complete tool chains are required, based on which the previously mentioned verification and validation are enabled.

In this paper, we propose, based on previous contributions [12], [13], [14], the formal yet expressive timed modeling of AADL to support early-phase analysis, validation, architecture exploration, semantic-preserving transformation, and co-modeling with Simulink. In our approach, the polychronous model of computation (MoC) Polychrony [15] is adopted as the common and back-end formal model. This model enables the following formal techniques:

- static analysis, including determinism identification and deadlock detection [16];
- profiling-based analysis of real-time characteristics of a system [17];
- logical clock calculus to analyze the relations between multiple dependent or independent clocks. Furthermore, the resulting clock relations, together with data dependency, are used to check causality problems [16];
- affine clock calculus dedicated to analyzing affine relations between clocks [18];
- real-time scheduling and allocation through the SynDEx tool [19];
- co-simulation of AADL specifications and demonstration using the VCD (Value Change Dump) technique [20].

To implement these tasks, the high-level AADL and Simulink models and their associated timing properties and constraints are transformed into polychronous models [15], [21], via SSME models (Signal Syntax Model under Eclipse) [22]. Polychrony [22] is an Eclipse toolset, allowing AADL model transformation to the Signal language [21] through an SSME meta-model as well as supporting the previously mentioned analysis. Two case studies, developed in the framework of the OPEES project [23] and the CESAR project [24], are used in this paper to show the effectiveness of our contribution to system-level modeling, transformation and analysis.

Outline. Some related work are first summarized in Section 2. Section 3 briefly introduces AADL via a tutorial case study. Section 4 gives a short description to Polychrony and the Signal language. Section 5 presents our main contribution to AADL modeling, transformation and timing analysis, and case studies in Section 6 are then used to illustrate our approach. Finally, conclusion is drawn in Section 7.

2. Related work

AADL provides an efficient support to model and analyze complex embedded systems. In order to perform formal validation and analysis on AADL models, formal models and frameworks are widely used in the process [25, 10, 26, 27, 7, 8, 11, 28]. We follow the similar approach, but in comparison, we concentrate on the formal timing analysis of the system and co-modeling, which include clock calculus, affine clock relations, profiling, and architecture exploration. A more detailed comparison is given in the next.

The AADL2Fiacre project [25] aims at transforming models such as AADL into the Fiacre intermediate model, which is compiled in order to be used in model checkers, e.g., CADP [29] and TINA [30]. In comparison, both behavioral, architectural and timing aspects of embedded system are considered in our framework, and our verification and analysis tools are, either directly used on our polychronous model, or seamlessly connected to the model semantically.

AADL2BIP [10] allows the simulation of AADL models, as well as application of specific verification techniques: state exploration or component-based deadlock detection. This work takes into account threads, processes and processors as well as AADL behavior annex (BA), but does not include the AADL temporal properties and communication protocols.

The main objectives of AADL2Sync [26], [27] are to perform simulation and validation that consider both system architecture and functional aspects. Various asynchronous aspects of AADL, e.g., task execution time and clock drifts, are taken into account. In addition, task schedulers are generated for tasks and shared resources scheduling. Compared to our work, the main differences are: we use polychronous/multiple clocks to handle non-determinism, not "oracles" with the master clock in AADL2Sync, thus non-determinism is transparent to users; we do not use the "activation condition" in SCADE [31] for component activation, as the polychronous clock mechanism is necessary for our formal clock analysis; the modeling of both input and output frozen is implemented in our approach, while only output delay is considered in AADL2Sync; the scheduling of share resources, such as FIFO (First In First Out), is addressed by the compiler in our approach, unlike by the imported schedulers in AADL2Sync.

In the Compass Approach [8], the SLIM (System-Level Integrated Modeling) language is used to describe nominal hardware and software operations, as well as fault-related modeling and handling. Bounded SAT-based and symbolic model checking can be performed on the SLIM language. In comparison, we work does not consider fault and probabilistic models. However, we take advantage of a logical clock model, which is distinguishing. It is also possible to perform symbolic model checking and formal timing analysis techniques on our polychronous model.

The AADL2Maude tool[11] concentrates on the ambiguity of certain AADL semantics and thus introduces a real-time rewriting logic semantics, for a behavioral sub-set of AADL. Simulation and model checking based on LTL[32] are enabled in AADL2Maude. In comparison, our work considers both behavioral and architectural AADL specification.

The Ocarina tool [28] aims at providing a rapid prototyping for distributed embedded systems specified in AADL. It also provides scheduling analysis capabilities via Cheddar [7], connection with Petri Net based formal verification tools, and code generation for Ada. Actually, our work considers both semantic aspects and architectural aspects of AADL, as well as timing analysis, performance analysis and distribution.

The Cheddar project [7] mainly focuses on scheduling issues of AADL specifications. Cheddar is therefore complementary to our framework in dealing with AADL scheduling.

In addition, there are similar works in the automotive domain, such as [33, 34], outcomes from the TIMMO-2-USE project [35]. They mainly consider AUTOSAR (AUTomotive Open System ARchitecture) [5] and its complement EAST-ADL [6]. A language, called TADL2 [35], has been proposed to deal with timing characteristics and analysis. In comparison, we concentrate on a clock-based timing modeling and analysis, rather than event chain constraints and probabilistic models. Event chain constraints are mainly addressed by controller synthesis in POLYCHRONY [36].

Previously presented related work mainly focuses on pure AADL specification and its variants. Little research work has been done to deal with both temporal analysis and system co-modeling and simulation, for instance, consider architectural specification in AADL and functional specification in other languages or formalisms, such as Simulink and FSMs (Finite State Machines). Our work mainly addresses these aspects, which distinguishes from the previously mentioned related work.

3. Introduction to AADL

AADL is a Society of Automotive Engineers (SAE) standard dedicated to modeling embedded real-time system architectures. As an architecture description language, based on a component modeling approach, AADL describes the structure of systems as an assembly of software components allocated on execution platform components together with timing semantics.

3.1. Architecture

In AADL, three distinct component categories are provided: 1) software application components which include process, thread, thread group, subprogram, and data components, 2) execution platform components that model the hardware part of a system including (virtual) processor, memory, device, and (virtual) bus components, and 3) composite component that is a system which models a component containing execution platform, application software or other composite components.

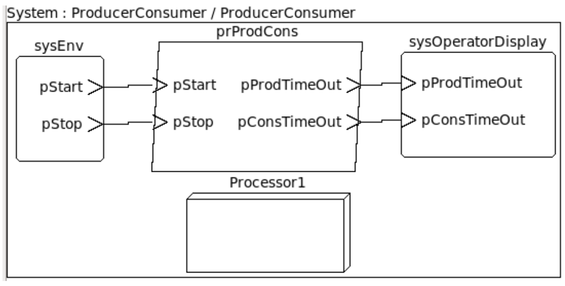

Figure 1: The *Producer-Consumer* case study in AADL (*system* level)

In the following, we use an OPEES tutorial case study, called *Producer-Consumer*, to illustrate progressively these AADL models. The system (in Figure 1) is in charge of producing and consuming data which is communicated through a shared data resource. It includes several functions allowing the producer and consumer to communicate and to access data:

- *sysEnv* system models the environment raising events to start/stop the process *prProdCons*.
- *sysOperatorDisplay* system signals when a timeout occurred on data production or consumption.

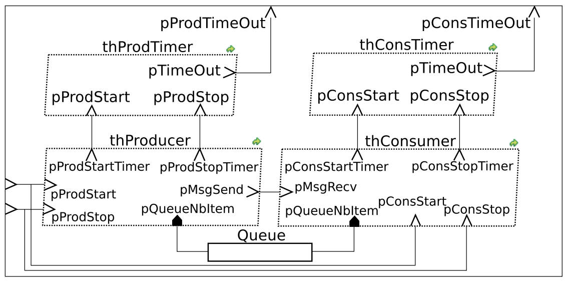

Figure 2: The *Producer-Consumer* case study in AADL (*process* level: the *prProdCons* process in AADL)

- *prProdCons* process communicates with the previous two systems. It contains four threads: *thProducer*, *thConsumer*, *thProdTimer* and *thConsTimer* (Figure 2). A data *Queue* is shared by threads *thProducer* and *thConsumer*: *thProducer* produces data in it, which in turn are consumed by *thConsumer*. The timer *thProdTimer* (resp.

thConsTimer) manages timer services for *thProducer* (resp. *thConsumer*). It permits to start, stop the timer and send a timeout event (*pTimeOut*) when the timer has expired.

- Processor *Processor1* is responsible for the executing of the process *prProdCons*.

Each component has a type, which represents the functional interface of the component and 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 components.

3.2. Properties

AADL properties provide various information about model elements of an AADL specification. For example, a property *Dispatch_Protocol* is used to provide the dispatch type of a thread. Property associations in component declarations assign a particular property value, e.g., *Periodic*, to a particular property, e.g., *Dispatch_Protocol*, for a particular component, e.g., *thProducer thread*. In this paper, we are mainly interested in two types of properties:

- timing properties, such as *Input_Time* (resp. *Output_Time*) of ports, that assure an *input-compute-output* model of thread execution. We will analyze the timing semantics and associated timing properties in Section 5.1. For example, the following timing properties are assigned to the thread *thProducer*:

```
thread thProducer
  properties
    Dispatch_Protocol => Periodic;
    Period => 4 ms;
    Deadline => 4 ms;
    Compute_Execution_Time => 2 ms;
end thProducer;
```

- the binding properties assign hardware platforms to the execution of application components. For example, the following property *Actual_Processor_Binding* specifies that process *pProdCons* is bound to processor *Processor1*:

```
Actual_Processor_Binding =>
        Processor1 applies to pProdCons;
```

3.3. Behavior

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

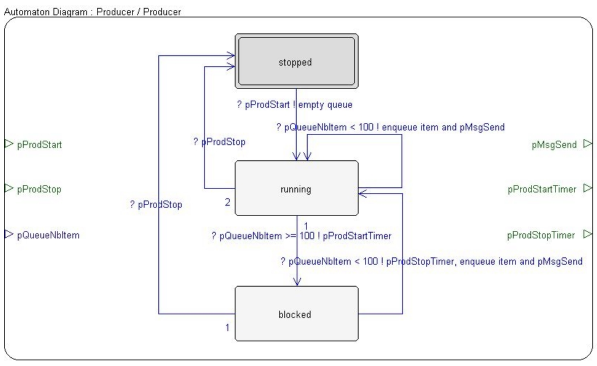
Automaton Diagram : Producer / Producer
stopped
running
blocked
? pProdStart ! empty queue
? pQueueNbItem < 100 ! enqueue item and pMsgSend
? pProdStop
? pProdStop
2
1
1
? pQueueNbItem >= 100 ! pProdStartTimer
? pQueueNbItem < 100 ! pProdStopTimer, enqueue item and pMsgSend
pProdStart
pProdStop
pQueueNbItem
pMsgSend
pProdStartTimer
pProdStopTimer

Figure 3: The behavior of AADL *Producer thread*

We take the *thProducer thread* as an example. Three states, *stopped*, *running*, and *blocked*, are specified together with the transitions between them (Figure 3). From the initial state *stopped*, when a *pProdStart* event is received, the shared *data Queue* is emptied, and the transition enters into the *running* state. Priorities are assigned to determine the transition to be taken, if more than one transition out of a state evaluates its condition to true. The higher the priority number is, the higher the priority of the transition is.

A brief introduction has been given in this section. In the next section, the Signal language and its associated environment Polychrony is presented. They provide the formal model support to AADL in our approach.

4. Introduction to Signal and Polychrony

Synchronous languages are dedicated to the design of synchronous reactive systems [37]. Their mathematical basis favors the trusted design of safety critical real-time systems. Among these languages, the Signal language stands out for its capability to describe circuits and systems with multi-clock [15], and to support *refinement* [38], i.e., the ability to assist and support system design from the early stages of requirement specification, to the later stages of synthesis and deployment. These characteristics make the Signal semantics much closer to AADL semantics than other pure synchronous or asynchronous models, and simplify the system validation.

4.1. The Signal language

Signal is a declarative language expressed within the polychronous MoC. It handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as x, and implicitly indexed by discrete time. At any instant, a *signal* may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. The set of instants where a *signal* x is present represents its *clock*, noted $\hat{x}$. Two *signals* are said to be synchronous if they are both present (or absent) at each instant.

Signal is the interface language of a design environment, called Polychrony [15], which provides a formal framework for the trustworthy system design, as well as simulation for deterministic specifications.

4.2. Basic constructs and Operators

Signal relies on a handful of primitive constructs, which can be combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for structuring [39]. Since the semantics of Signal is not the main topic of this paper, we will use a rudimentary increasing counter example [15] to illustrate the primitive operators and constructs of Signal.

The *Count* process[1] accepts an input *reset* signal and delivers an integer output signal *val*. It allows input signal *reset* and output *val* to have independent clocks.

```
process Count=
  (? event reset; ! integer val;)
  (| counter := val $1 init 0               %e1%
   | val := (0 when reset) default (counter + 1)
   |) where
       integer counter;
     end;
```

The local variable counter stores the previous value of the signal val (equation e1); its is initial value is 0. When reset occurs, the signal val is reset to 0 (expression (0 when reset)). Otherwise, val is accumulated by 1 (expression (counter+1)).

count event	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	...
reset		T					T				T		...
val	1	0	1	2	3	4	0	1	2	3	0	0	...
counter	0	1	0	1	2	3	4	0	1	2	3	0	...

Figure 4: Execution trace of Signal process *count*

Figure 4 presents a trace of the Count process execution. The activity of Count is governed by the clock of its output val, which differs from that of its input reset. If the signal val is required by the environment, then either reset is absent and Count increments *val*, or reset is present and Count sets val to 0.

[1]Process in this font is used to indicate a SIGNAL process when it is used standalone, compared to an AADL process.

4.3. Techniques for formal analysis and simulation

In this paper, techniques including clock calculus, affine clock relations, profiling, verification, scheduling and distribution are used to deal with AADL temporal properties. Only a brief description of these techniques is provided, but more details can be found in the corresponding references.

- **Clock calculus** [40], in the compilation of Signal programs, enables static analysis, code generation, formal verification, code distribution, etc. The main objective of clock calculus is to explore clock relations based on the clock *domination* relation. According to this relation, clock trees are built, which are used to identify the timing dependency and determinism in the system.
- **Affine clock relations** [18] yield an expressive calculus for the specification and analysis of time-triggered systems. A particular case of affine relations is the case of affine sampling relation expressed as $\hat{y} = \{d \cdot t + \phi \mid t \in \hat{x}\}$ ($\hat{y}$ is the clock of y) of a reference discrete time $\hat{x}$ (d, t, ϕ are integers): $\hat{y}$ is a subsampling of positive phase ϕ and strictly positive period d on $\hat{x}$. In Polychrony, the affine clock calculus implements synchronizability rules based on properties of affine relations, against which synchronization constraints can be assessed.
- **Profiling** is also adopted in Polychrony for the performance evaluation of Signal programs [41, 42]. In the framework of Polychrony, profiling refers to timing analysis through associating *date* and *duration* information with Signal programs. Transformation of Signal processes, more precisely temporal morphism, preserves semantic properties. The resulting Signal process can be composed with the original process for the co-simulation. The latter exhibits the timing behavior with regard to previously mentioned temporal properties.
- **Formal verification**, such as model checking, is performed in the framework of Polychrony. We use the Sigali tool [43] as the model checker. Sigali relies on an implementation of polynomials by Ternary Decision Diagram (TDD) (for three valued logic) in the same spirit of Binary Decision Diagram (BDD) [44]. However, the paths in the data structures of TDD are labeled by values in {1, 0, -1}. Properties, such as invariance, reachability and attractivity can be checked by Sigali. In addition, algorithms for computing state predicates are also available in the tool.
- **VCD-based demonstration.** In addition to profiling, another simulation has also been carried out.

It aims at the visualization of value change during the execution of programs via VCD. VCD files are usually generated by EDA (Electronic Design Automation) logic simulation tools, and the four-value VCD format has been defined in IEEE Standard [20].

◦ **Real-time scheduling and distribution** have been explored at the high level, i.e., at the AADL thread level. Furthermore a more detailed yet low level scheduling and distribution can be explored at the generated SIGNAL level with the SYNDEX [19] tool. A low level scheduling and distribution considers more detailed execution information thus enables a more precise simulation. In our framework, a translation from SIGNAL to SYNDEX has been developed. The final SYNDEX code considers both functional and architectural specification from Simulink and AADL respectively.

The AADL and SIGNAL languages have been briefly presented in the last sections. In the next section, our main contribution on AADL modeling and analysis will be presented in detail.

5. AADL modeling and analysis framework

The AADL time model allows the specification of both logical and chronometric clocks in the system, and these clocks can be periodic, aperiodic, sporadic, etc. In addition, each component can be associated with timing properties, which indicate their expected real-time characteristics. All these timing information are to be verified before implementation. In general, they can be checked by schedulability analysis or simulation at runtime, on an informal basis. We propose to perform formal timing analysis via a different yet efficient approach based on the polychronous MoC, allowing to identify and analyze determinism, temporal causality and synchronization problems.

Figure 5 presents our proposed approach. High-level architectural and behavioral models, such as AADL and Simulink, are transformed into the formal polychronous model. Scheduler synthesis and system integration are considered in this polychronous model. Timing analysis, formal verification, simulation and architecture exploration are then performed based on the polychronous model. In order to bridge between the high-level models with the polychronous model, we first present the timing analysis of typical AADL execution models. Then a brief comparison of AADL timing model with that of POLYCHRONY is given, which aids the understanding of our approach. Then, our proposed AADL time model in POLYCHRONY is presented, followed by the corresponding

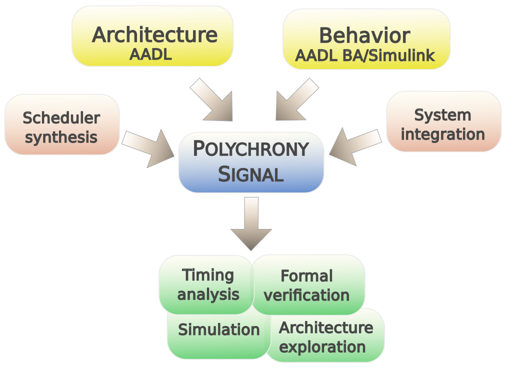

Figure 5: our proposed approach

model transformation. Scheduler synthesis and system integration are briefly described considering the resulting polychronous models. An implemented tool chain is finally illustrated to support our work.

5.1. AADL timing execution model analysis

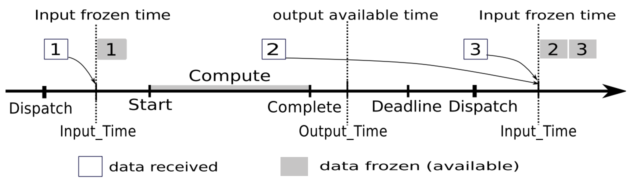

Figure 6: illustration of the execution time modeling for an AADL *thread*. *Input_Time* and *Output_Time* events indicate the *input frozen time* and *output available time* respectively.

Threads are the main components that have an execution timing semantics. Threads are dispatched, i.e., their execution is initiated either periodically or by the arrival of data or events on ports, or by the arrival of subprogram calls from other *threads*, depending on the *thread* type.

Three event ports are predeclared: *dispatch*, *complete* and *error*. A *thread* is activated to perform the computation at *start* time, and has to be finished before *deadline* (Figure 6). A *complete* event is sent at the end of the execution.

Input-compute-output model. The inputs received from other components are frozen at a specified point (*Input_Time*), by default the *dispatch* of a *thread*. As a result, the computation performed by a thread is not affected by a new arrival input until an explicit request for input, by default the next *dispatch*. Similarly, the output is made available to other components at a specified point of time (*Output_Time*), by default at *complete* (resp. *deadline*) time for out port if the associated port connection is *immediate* (resp. *delayed*) communication.

From the designers' perspective of an AADL specification, the ports of a thread have special timing semantics: the content of incoming ports are frozen at a specified *Input_Time*, which means that the content of the port that is accessible to the recipient does not change during the execution of a dispatch event though the sender may send new values. For example, the two data values 2 and 3 (in Figure 6) arriving after the first *Input_Time* will not be processed until the next *Input_Time*.

5.2. *Comparison of AADL and* POLYCHRONY *time models*

Due to the different timing semantics, modeling embedded systems specified in AADL in POLYCHRONY raises some difficulties. Thanks to SIGNAL's derived operations and associated tools, we can bridge their semantic gaps.

- ◦ AADL takes into account execution latency and communication delay, which are defined on chronometric clocks. Conversely, the synchronous semantics of POLYCHRONY only considers atomic *instantaneous* actions: instantaneous execution on logical clock. Possible solutions to bridge between these different time models have been presented in [12], where additional discrete events are added to model latency and delay. In this way, the AADL time domain is mapped onto SIGNAL logical clocks and formal logical clock analysis becomes possible.
- ◦ The multi-clock feature of SIGNAL allows to model systems with several clocks, where each component holds its own activation clock, as well as single-clocked systems, in a uniform way. This feature suits well for component-based architecture design in AADL, because different components, in general, have different execution/activation rates. POLYCHRONY provides powerful tools, such as logical clock calculus and affine clock calculus, to analyze the relations of multi-clocked systems.
- ◦ Periodic dispatch of threads is supported in AADL, where a dispatch request is issued in time intervals specified by the *Period* property. Periodic clocks can be modeled in SIGNAL using affine clock relations. Thus, synchronizability analysis can be carried out between multi-period threads.
- ◦ AADL supports communications associated with queues (for event and event data ports), in FIFO order by default. SIGNAL's bounded FIFO is used for message exchanges between several entities. Two types of FIFO are defined in our SIGNAL library: *fifo_reset* and *fifo*, with or without configuration resetting.
- ◦ Data can be shared between the components in AADL, so that it can be read or written by different components at different time instants. It is possible in SIGNAL to have several expressions associated with one signal by partial definitions [39]. The clock calculus can compute sufficient conditions to guarantee that the overall definition is consistent and total.
- ◦ Both AADL and POLYCHRONY support Globally Asynchronous Locally Synchronous (GALS) design [45], which provides a compromise between synchronous/asynchronous design in face to the increasing system complexity. In addition, the modularity feature of POLYCHRONY directly supports the component-based design approach of AADL, allowing modular development of complex systems. This feature, together with the relational feature of the polychronous MoC, makes it possible to build a system incrementally, in the same way as an AADL system.
- ◦ POLYCHRONY supports behavior modeling through SIGNAL automata extension, similar to the behavior annex, an extension of AADL via transition systems. Clocks are considered in SIGNAL automata, in consequence, temporal properties of the AADL component behavior can also be addressed.

5.3. *AADL time model in* POLYCHRONY

The key idea for modeling the 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`, that introduce delay and well suited synchronizations [12]. As a consequence, various properties result in explicit synchronization signals.

We introduce a "memory" `processes` $o = f_m(i, b)$ that repeats the input signal i on the instants of boolean signal b. The result o contains values of i when i is present and b is true, and the last value of i when i is absent and b is true:

$$o = f_m(i, b) \overset{def}{\equiv} \forall t > 0 :$$

$$o_t = \begin{cases} i_t & \text{if } i_t \neq \perp, \text{ and } b_t = true \\ i_{pred(t)} & \text{if } i_t = \perp, \text{ and } b_t = true, \\ & pred(t) = max\{k < t \mid o_k \neq \perp\} \\ \perp & \text{otherwise} \end{cases}$$

Input freezing. Let $f(x)$ represent the result of the behavior f of a given in port to its input signal x, e.g., f can be a FIFO to represent queued event or event data port. A port $y = f(x)$ gives the available output y from the current received input x. It defines an elementary `process` such that:

$$y = f(x) \overset{def}{\equiv} \forall t > 0 :$$
$$(x_t \neq \perp \Rightarrow f(x_t) \neq \perp) \wedge (y_t = f(x_t)) \wedge (f(\perp) = \perp)$$

x is frozen at t is a function that takes an input x, a frozen time event t, and produces a new signal z at time t. It is noted as $x \blacktriangleright t$:

$$z = x \blacktriangleright t \overset{def}{\equiv} z = f_m(f(x), t)$$

Thread activation. We use $th(z_1, z_2, \dots)$ to represent the original computation of thread th with the frozen inputs $z_1, z_2, \dots$ The thread th is activated to perform computation at "start", which is denoted as $th'(z_1, z_2, \dots, start)$, where its inputs $z_1, z_2, \dots$ are memorized at start. It is defined as follows:

$$th'(z_1, z_2, \dots, start) \overset{def}{\equiv} th(z'_1, z'_2, \dots)$$
$$\text{where } \forall z'_i = f_m(z_i, start)$$

Output sending. Similar to the in port, $g(y)$ represents the behavior of an out port. The *send* function defines a `process` such that the generated output of $g(y)$ is hold and sent out at time t. This is noted as $y \triangleright t$:

$$w = y \triangleright t \overset{def}{\equiv} w = f_m(g(y), t)$$

5.4. Transformation principles

In this section, we describe the detailed transformation process from a high level description in AADL to a Signal description. The timing semantics and properties are processed during the transformation.

5.4.1. Thread

Threads are the main executable and schedulable AADL components. A thread models a concurrent task or an active object, i.e., a schedulable unit that can execute concurrently with other threads. An AADL periodic thread is implemented as a Signal process composed of its behavior which represents the transition system that is specified in AADL behavior annex, properties, ports, subcomponents (if data subcomponents or subprogram subcomponents exist) and connections (Figure 7).

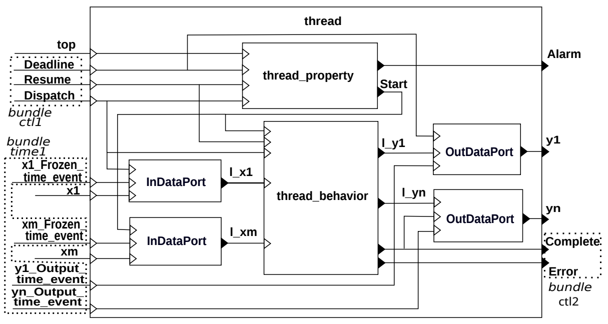

Figure 7: a Signal model of AADL *thread*

Some additional timing signals are added to the Signal processes:

- An input *bundle* signal *ctl1* (a *bundle* represents a polychronous tuple of signals) contains event signals provided by the scheduler, *Dispatch*, *Resume* and *Deadline*, which are implicit predeclared ports in AADL or signals added for simulation.
- An input *bundle* signal *time1* that provides the clock of the frozen time (resp. Output time) for the input (resp. output) ports, e.g., *xi_Frozen_time_event* and *y1_Output_time_event*.
- An output signal *ctl2* for the events *Error* and *Complete* (predeclared ports in AADL).
- An output signal *Alarm* that triggers an event when the properties are not satisfied.

Computing latency and communication delay, allowing to produce data of the same logical instants at different implementation instants, is taken into account in the thread. Those instants are precisely defined in the port and thread properties. Therefore, the ports of a thread are implemented as Signal processes instead of simply input/output signals.

The AADL port is a logical connection point for the directional exchange of data, event or both, between components. As mentioned in the execution semantics in Section 5.1, the thread ports have special timing semantics: the in (resp. out) port is frozen (resp. sent out) at *Input_Time* (resp. *Output_Time*).

In this section, we present the modeling of event data ports (the event ports can be implemented by counters, and the data ports modeling can be found in [12]). Incoming data events, which may be buffered in event data ports with queues, may trigger thread dispatches or mode transitions, or they may simply be queued for processing by the recipient. The default port queue size of a data event port 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 FIFO order.

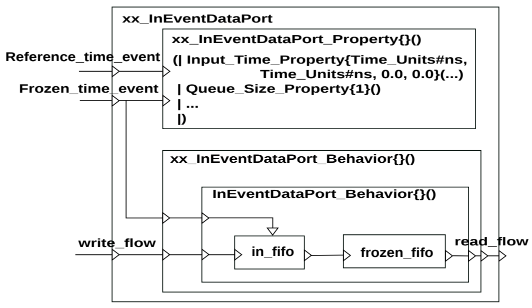
xx_InEventDataPort
xx_InEventDataPort_Property{}()
(| Input_Time_Property{Time_Units#ns, Time_Units#ns, 0.0, 0.0}(...)
| Queue_Size_Property{1}()
| ...
|)
Reference_time_event
Frozen_time_event
xx_InEventDataPort_Behavior{}()
InEventDataPort_Behavior{}()
in_fifo
frozen_fifo
write_flow
read_flow

Figure 8: a Signal model of AADL *in event data port*

In event data port. Two FIFOs are provided: *in_fifo* for storing the received data, and *frozen_fifo* for storing the frozen data (Figure 8) . The actual items of the *in_fifo* are frozen at *Input_Time*.

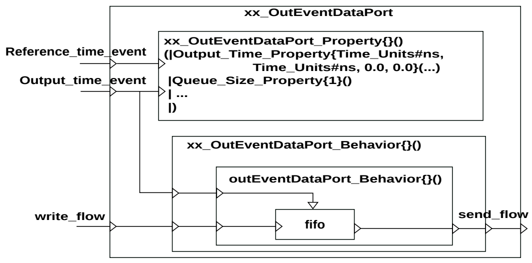
xx_OutEventDataPort
xx_OutEventDataPort_Property{}()
(|Output_Time_Property{Time_Units#ns, Time_Units#ns, 0.0, 0.0}(...)
|Queue_Size_Property{1}()
| ...
|)
Reference_time_event
Output_time_event
xx_OutEventDataPort_Behavior{}()
outEventDataPort_Behavior{}()
fifo
write_flow
send_flow

Figure 9: a Signal model of AADL *out event data port*

Out event data port. For an out event data port (e.g., *pProdStartTimer*), the values are stored in a *fifo*, and sent out (presented as Send) at *Output_Time* (Figure 9).

The `processes` *InEventDataPort_Behavior()* and *OutEventDataPort_Behavior()* that model the queuing of in/out data events are predefined in the AADL2SIGNAL library. The actual *Input_Time* (*Output_Time*) signals are generated by scheduler according to the values of timing properties.

5.4.2. Shared data

Components can share access to data subcomponents, where the data act as a critical region; mutual exclusion access clocks are thus required to assure only one access at a time. Therefore, in contrast with other categories of components, e.g., thread, which are translated into different instances of Signal processes, the shared data are represented as a single Signal FIFO instance that can be read/written by different components at different time instants. Depending on the type of access that is associated with data (i.e., *read_only*, *write_only* or *read_write*), a clock at which a thread reads (*x_r*), writes (*x_w*) or resets (*x_reset*) the data *x* is provided if the thread requires access to the data *x*. These clocks are declared as shared variables, so that they can be accessed by different Signal processes in different time instances.

- To write a value *vx* into the fifo, a partial definition of *x_w* is provided (in the behavior part): *x_w* ::= *vx when ew*1. (*ew*1 is the clock when a component writes value *vx* into data *x*.)
- To read a data, one can use: *z* := *x_r when er*1. (*er*1 is the clock when a component reads data from *x_r*.)
- The read clock of data *x* is the union of all such events er_i (equation (1) blow), similarly for the write and reset clocks; the clock of count *cnt* is at least the union of read, write and reset clock (equation (4)).

```
x_r ^= er1 ^+ er2 ^+ ...                   (1)
x_w ^= ew1 ^+ ew2 ^+ ...                   (2)
x_reset ^= ereset1 ^+ ereset2 ^+ ...       (3)
x_cnt ^= x_r ^+ x_w ^+ x_reset             (4)
```

`^=` and `^+` are two Signal operators to specify clock relations (not values) [16]: *synchronized with* and *clock union* respectively.

5.4.3. Process and processor

For their execution, AADL Processes, that supports the dispatch protocol required by the containing threads. This protocol is provided by *Actual_Processor_Binding* property:

```
Actual_Processor_Binding =>
     Processor applies to Process;
```

The AADL processes bound to this processor are implemented as Signal subprocesses of the Signal process that represents the processor (Figure 10).

The **interface** contains the original inputs/outputs of these AADL processes. Internal inputs/outputs between these `processes` are defined as internal local signals.

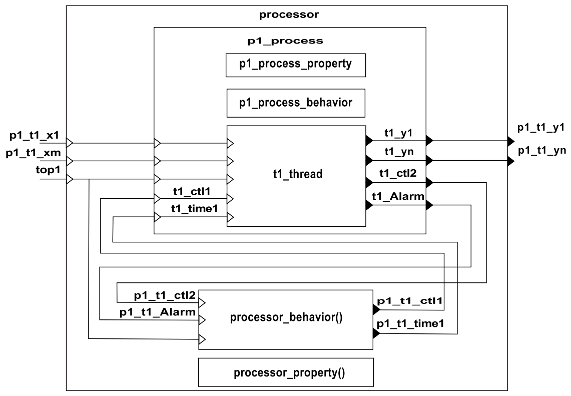

Figure 10: a SIGNAL model of AADL *processsor*

The **body** is composed of the (SIGNAL) processes, *xx_Processor_property()*, and *xx_Processor_behavior()* `subprocess`. The latter connects to an external scheduler, which generates all the scheduling signals for each thread, e.g., *p1_t1_ctl1*, *p2_t2_time1*, ...

Different processors which are responsible for executing threads may have different execution speeds. Therefore, an event input signal *top1* is added to allow the simulation of the application.

5.5. Scheduler synthesis

An AADL model is not complete and executable if the thread-level scheduling is not resolved. Thus a scheduler is expected to be integrated so that a complete model is used for the following validation, distribution and simulation. Traditional AADL scheduling tools, such as Cheddar [7], do not completely satisfy our demands because: 1) logical and chronometric clocks are easily transformed into each other according to the different context for formal and/or real-time analysis; 2) input/output frozen events and other more events are also considered in the scheduling; 3) static and periodic scheduling is expected for the purpose of predictability and formal verification. For instance, the scheduling, based on affine clock systems [18], is easily and seamlessly connected to POLYCHRONY for formal analysis. Affine clock relations yield an expressive calculus for the specification and the analysis of time-triggered systems. More details are found in Section 4.3.

Following the previous ideas, a static scheduler synthesis process is proposed as a complementary of our AADL modeling. It has the following subprocesses: 1) *calculate hyper-period* from the periods of all the threads according to the least common multiple principle; 2) *perform the scheduling* based on the hyper-period. More precisely, discrete events of each thread, such as dispatch, input/output frozen time, start and complete, are allocated in the hyper-period on condition that all their timing properties are satisfied. Affine clock relations of these events are ensured during the calculation. This process yields single-processor-oriented schedulers that are static and non-preemptive.

5.6. Behavior modeling in Simulink/Gene-Auto

The behavior of the AADL components can be specified using AADL behavior annex or other languages, such as Simulink/Stateflow [1] in the Matlab family. The latter is based on dataflow models and state machines, which are common models of computation adopted in the system design of avionics, automotive applications, etc. A typical Simulink model is defined by a set of interconnected blocks, which model entities in a system, such as sensors, actuators, and logical operations. The library of Simulink includes function blocks that can be linked and edited in order to model the dynamics of the system. Gene-Auto is a framework for code generation from a safe subset of Simulink and Stateflow models for safety critical embedded systems [46]. This safe subset is also adopted in our work. In this paper, Simulink is used for short to indicate the subset of Simulink and/or Stateflow languages that is adopted by Gene-Auto. In addition, only discrete time of Simulink is taken into account in the behavior modeling.

5.7. A complete tool chain

From the high-integrity systems point of view, the use of automatic code generation in the development process is highly profitable. We propose a tool chain (Figure 11) for modeling, timing analysis and verification of the AADL models in the polychronous MoC.

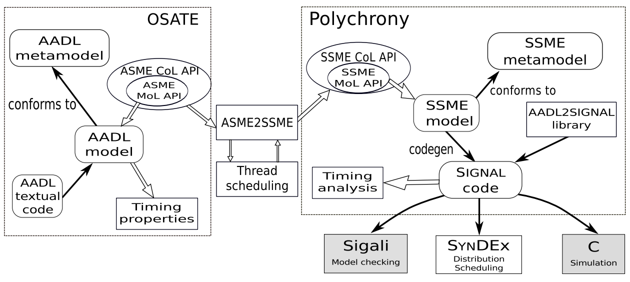

Figure 11: A global view of AADL to SIGNAL tool chain

The AADL model, which conforms to the AADL metamodel, is captured as AADL textual code in the OSATE toolkit [47]. The timing properties provide detailed timing specifications related to the AADL model.

A model transformation tool chain **ASME2SSME** performs analyses on the ASME models (AADL Syntax Model under Eclipse) and generate SIGNAL models (in SSME). This tool is implemented in Java, as an Eclipse plugin[2], and takes as input an AADL model (.aadl) and generates an SSME model (.ssme). The SSME model can be transformed to SIGNAL textual code within POLYCHRONY.

An AADL2SIGNAL library provides common SIGNAL processes reducing significantly the transformation cost. The timing properties represented as SIGNAL clocks are calculated and analyzed in the compilation of SIGNAL programs. After that, the executable code is generated for simulation. Associated tools, such as Sigali [43] and SYNDEX [19], can be used for further verification and validation.

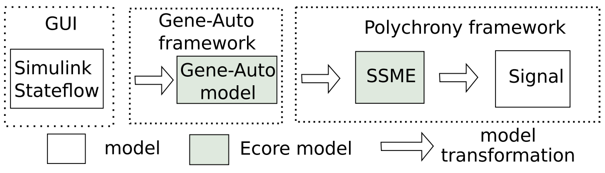

Figure 12: A global view of the functional model transformation chain

In addition to ASME2SSME, there is another model transformation chain from functional models in Simulink to SIGNAL. It is divided into several steps. The first step involves the transformation of Simulink models to Ecore based Gene-Auto models [46]. These models are then translated into SIGNAL via the SSME metamodel. The whole chain from high-level models to executable code is illustrated in Figure 12.

Scalability of this tool chain is considered in the following aspects: 1) in the framework of Eclipse EMF, the tool chain defines a CoL (Concept high Level) API to access the MoL (Model low Level) API. In this way, model transformations are independent of different low-level metamodels and heterogeneous models are easily integrated into the tool chain; 2) most AADL components are considered in order to handle large-sized systems, such as thread, process, subprogram, (shared) data, processor, bus, system, port, parameter, data access, subprogram access, and subcomponent; 3) in the framework of POLYCHRONY, analysis, verification, simulation, profiling techniques are considered as independent functions connected to the POLYCHRONY core, i.e., other functionality can be also integrated if necessary; 4) several thousand clocks can be handled by the clock calculus; 5) more than ten case studies have been tested, and there is no special size limitation on transformation. Limitation exists only in some formal validation techniques, such as model checking with the state space explosion issue. The clock calculus aids to reduce variable number and optimize clock relations, by removing unused variables and combining equivalent variables, in the clock synthesis step. In addition, static analysis, such as dependency analysis is performed in POLYCHRONY before model checking. Therefore, the system to be model-checked in our approach is optimized while preserving semantics equivalence. In addition, a simple but efficient mechanism of traceability has been implemented in the tool chain, i.e., the names of high level models are either preserved as names or preserved in annotations in the model transformation and code generation.

In the next section, two case studies will be presented to illustrate our approach and the transformation.

6. Case studies

Two case studies, Producer-Consumer [14] and SDSCS (Simplified Doors and Slides Control System) [28] have been developed in two European projects OPEES [23] and CESAR [24] respectively. Our industrial parteners from the avionic domain, Airbus[48] and C-S Toulouse[49], provided the specification of these two applications. The high-level modeling, transformation, analysis, verification and simulation have been carried out in our team, i.e., INRIA Espresso team[50]. With the first case study, translations of AADL thread and shared data are demonstrated. The second example illustrates how we address system-level co-design of avionic applications.

6.1. Case study: Producer-Consumer

In this section, we illustrate the translation process from a high level description in AADL to a synchronous description using the *Producer-Consumer* case study introduced previously in Section 3. The timing semantics and properties are processed during the transformation.

The SIGNAL process resulting from the system implementation in Figure 1 is given here in Figure 13: an instance of a SIGNAL process model of the processor *Processor1* communicates with two SIGNAL process instances that represent the systems *sysEnv* and *sysOperatorDisplay*. The links between these processes represent the component connections. Behavior (*ProducerConsumer_others_System_behavior()*) and property

[2]It defines a CoL (Concept high Level) API to access the MoL (Model low Level) API, which is provided by OSATE (resp. POLYCHRONY) as a set of classes providing access to attributes, for making the transformation independent of low level metamodel.

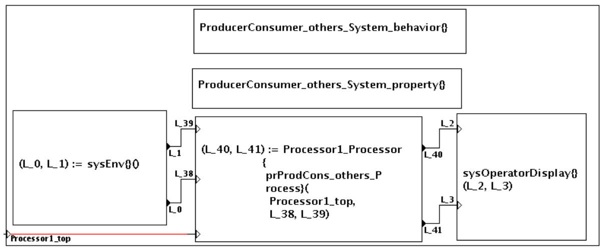
ProducerConsumer_others_System_behavior()
ProducerConsumer_others_System_property()
(L_0, L_1) := sysEnv()()
(L_40, L_41) := Processor1_Processor { prProdCons_others_Process}(Processor1_top, L_38, L_39)
sysOperatorDisplay()(L_2, L_3)
L_39
L_1
L_38
L_0
L_2
L_40
L_3
L_41
Processor1_top

Figure 13: the SIGNAL model of the AADL *system*: *ProducerConsumer*(presented in Figure 1)

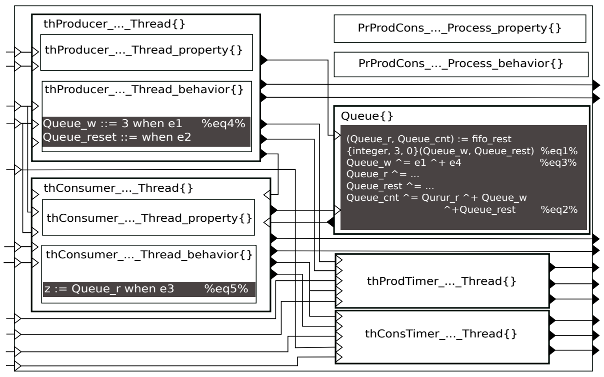
thProducer_..._Thread{}
thProducer_..._Thread_property{}
thProducer_..._Thread_behavior{}
Queue_w ::= 3 when e1 %eq4%
Queue_reset ::= when e2
thConsumer_..._Thread{}
thConsumer_..._Thread_property{}
thConsumer_..._Thread_behavior{}
z := Queue_r when e3 %eq5%
PrProdCons_..._Process_property{}
PrProdCons_..._Process_behavior{}
Queue{}
(Queue_r, Queue_cnt) := fifo_rest
{integer, 3, 0}(Queue_w, Queue_rest) %eq1%
Queue_w ^= e1 ^+ e4 %eq3%
Queue_r ^= ...
Queue_rest ^= ...
Queue_cnt ^= Qurur_r ^+ Queue_w
^+Queue_rest %eq2%
thProdTimer_..._Thread{}
thConsTimer_..._Thread{}

Figure 14: a SIGNAL model of AADL *shared data*: Queue in the Producer-Consumer case study. The Queue is connected with other threads in the AADL specification. This figure also illustrates the translation result of the *prProdCons* process in AADL (presented in Figure 2).

(*ProducerConsumer_others_System_property()*) subprocesses are added. In Figure 13, L_X is input/output of corresponding component, e.g., L_38 and L_39 are *pStart* and *pStop* of the *prProdCons* process (AADL) in Figure 1. Processor1_top is the default activation clock of the processor.

6.1.1. Architecture modeling

AADL processes (e.g., *pProdCons*) will be bound to a processor (e.g., *Processor1*) for their execution, that supports the dispatch protocol required by the containing threads. This protocol is provided by *Actual_Processor_Binding* property:

```
Actual_Processor_Binding =>
          Processor1 applies to pProdCons;
```

The processes bound to this processor are implemented as SIGNAL `subprocesses` of the `process` that represents the processor. The translation of AADL process and processor has been presented in Section 5.4.3. Besides the behavior and property `processes` (explained in Section 5.4.1), the corresponding SIGNAL `process` of AADL process *prProdCons* is also composed of the SIGNAL `processes` of subcomponents: the `processes` that represent the interpretation of corresponding threads, e.g., *thProducer_others_Thread()*, and a `process` *fifo_reset()* (and associated clock synchronizations) that models the shared data *Queue* (Figure 14).

The data *Queue* in the *prProdCons* process (presented in Figure 2) which is shared by threads *thProducer* and *thConsumer* is represented as a FIFO process instance *fifo_reset()* (equation *eq*1 in Figure 14). Translation of AADL shared data has been presented in Section 5.4.2. The values to be read or written in the FIFO (*Queue_r, Queue_w, Queue_reset*) are declared as shared variables, so that they can be accessed by different threads. To write (or reset) the data into the FIFO, a partial definition (such as equation *eq*4 in Figure 14) is provided (*e*1 is a time instant at which the thread writes data). The write (or read, reset) clock of data *Queue* is the union of all such events (equation *eq*3), and the clock of counter *Queue_cnt* that returns the current number of values is at least the union of read, write and reset clocks (equation *eq*2). The FIFO *type* (*integer*), *size* (3) and *initial value* (0) are provided by associated data properties.

The data access connection is not modeled explicitly, since the read/write access to the FIFO already implicitly indicates their connections.

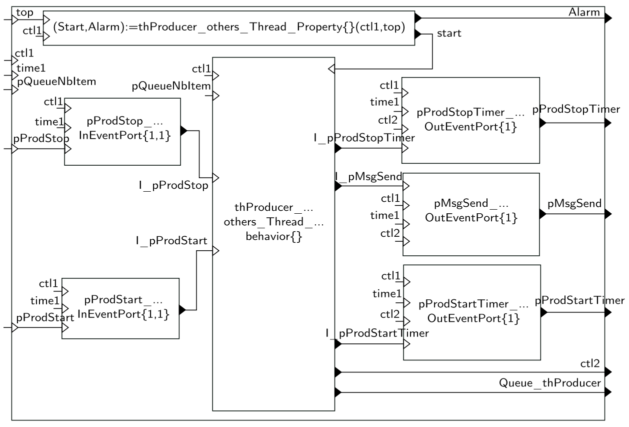
top
ctl1
(Start,Alarm):=thProducer_others_Thread_Property{}(ctl1,top)
Alarm
start
ctl1
time1
pQueueNbItem
pProdStop
pProdStop_... InEventPort{1,1}
l_pProdStop
l_pProdStart
thProducer_... others_Thread_... behavior{}
pProdStart
pProdStart_... InEventPort{1,1}
pProdStopTimer_... OutEventPort{1}
l_pProdStopTimer
pProdStopTimer
l_pMsgSend
pMsgSend_... OutEventPort{1}
pMsgSend
pProdStartTimer_... OutEventPort{1}
l_pProdStartTimer
pProdStartTimer
ctl2
Queue_thProducer

Figure 15: the SIGNAL model of the AADL *thread*: thProducer (in Figure 2)

In Figure 15, we give only the translation of one thread (*thProducer*) of the *Producer-Consumer* application, since all the threads of the application have the same structure and are built on the same AADL components. Translation of AADL thread has been presented in Section 5.4.1 and Figure 7. An

AADL periodic thread *thProducer* is implemented as a Signal process composed of its behavior (*thProducer_Thread_behavior()*) which represents the transition system that is specified in AADL behavior annex, properties (*thProducer_Thread_property()*), ports (e.g., *pProdStart_InEventPort()*), subcomponents (if data subcomponents or subprogram subcomponents exist) and connections.

The timing properties are not specified explicitly in this example, hence the default values are used.

6.1.2. Formal analysis and simulation

Once the AADL specification is modeled with the polychronous MoC and then translated into the Signal language, Polychrony can be used to formally analyze and verify the AADL model, which includes: static analysis, simulation, model-checking, etc. For instance, clock calculus can be used, in the compiling stage, to verify the determinism of the AADL model. An example is given here, if the priority properties are not completely specified on the transitions in the automata of the *producer thread*, the Signal compiler, via the clock calculus, detects a clock constraint: the specification is not deterministic. This constraint is found before any execution/simulation of the program.

In the case study, all the *threads* are periodically dispatched, periodic Signal clocks are required for the simulation. Periodic clocks can be specified in Signal using *affine* clock relations. In the application, the periods of the four *threads* (*Producer, Consumer, ProducerTimer, ConsumerTimer*) are respectively $4ms$, $6ms$, $8ms$ and $8ms$. We use the $affine\_sampling()$ Signal library *process* [39] to calculate the scheduling: first 24 ms, which is the least common multiple of the periods, is calculated. And the following scheduling is a repetition of the scheduling of the first 24 ms).

In addition to the previous formal clock analysis, we can also use simulation-oriented analysis, internal tool such as profiling, and external tools such as Cheddar and SynDEx. Profiling can be used for performance evaluation, once a specific hardware architecture is chosen and the corresponding temporal specification of the Signal program is defined on this architecture. Cheddar and SynDEx can also be connected to obtain static schedulers, which are used in the simulation considering real-time characteristics.

6.2. The SDSCS case study

SDSCS is a generic simplified version of the system that allows managing doors on Airbus 350 series aircrafts. It is a safety-critical system as incorrect door closing or opening during flight may lead to fatal crashes. In addition to the reliable system design and validation, high-level modeling and component-based development are also expected for fast and efficient design. SDSCS has been chosen for the demonstration of capabilities developed in the CESAR project. Compared to the first case study, SDSCS demonstrates how we handle heterogeneous high-level co-modeling, i.e., AADL and Simulink, of avionic applications.

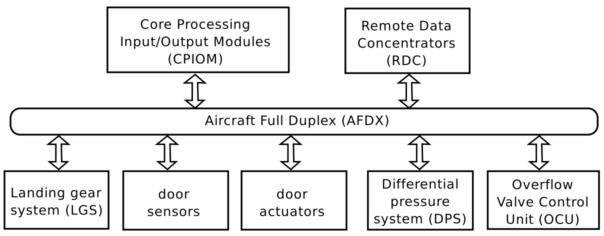
Core Processing Input/Output Modules (CPIOM)
Remote Data Concentrators (RDC)
Aircraft Full Duplex (AFDX)
Landing gear system (LGS)
door sensors
door actuators
Differential pressure system (DPS)
Overflow Valve Control Unit (OCU)

Figure 16: a simple illustration of the SDSCS system architecture.

In this case study, only the management of passenger doors is considered. Each passenger door has a software handler to achieve the following tasks: monitor door status via door sensors; control flight lock actuators; manage the residual pressure of the cabin by controlling the outflow valves, visual status indication and an aural warning, etc.

SDSCS is equipped with other hardware components, such as processing units, communication link, and concentrators as well as sensors and actuators. The SDSCS is implemented on the IMA (Integrated Modular Avionics) platform, in which CPIOMs (Core Processing Input/Output Modules) and RDCs (Remote Data Concentrators) are connected via the AFDX (Aircraft Full DupleX) network (Figure 16). Sensors and actuators are also connected to RDCs via AFDX. CPIOMs receive sensor readings via RDCs and communicate with other systems via AFDX.

6.2.1. Architecture modeling in AADL

Figure 17 shows an overview of the SDSCS modeled specified in AADL. The whole system is presented as an AADL *system*. The two doors, *door1* and *door2*, are modeled as *subsystems*. They are controlled by two AADL processes *doors_process1* and *doors_process2* respectively. These processes are bound to two *processors*: *CPIOM1* and *CPIOM2*, to perform the computation independently. *Sensors* and *Actuators*, such as *LGS*, *DPS*, *OCU*, etc., are modeled as AADL *devices* that interface with external environment of the system. All the communication between the *devices* and *processors* is through the *bus*: *AFDX1*. SDSCS has three

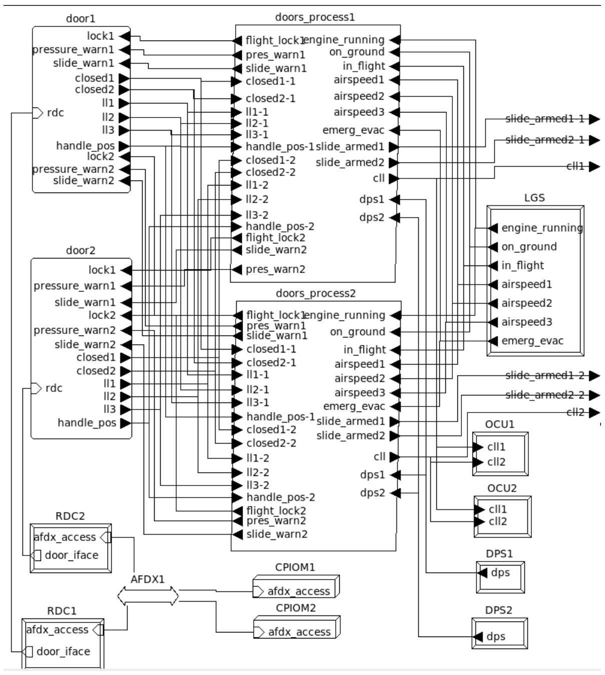

Figure 17: an overview of the SDSCS system architecture modeling in AADL

threads to manage doors: *door_handler1*, *door_handler2*, and *doors_mix*. These threads are implemented by Simulink models. In addition, each *processor* runs one *doors_process*. These two components are placed into one ARINC *partition*. Each *processor* is associated with an ARINC *partition_level_OS*, which is responsible for scheduling all the *processes* in the same *partition*. In this example, all the threads and devices are periodic, and share the same periodicity.

6.2.2. Behavior modeling in Simulink/Gene-Auto

The behavioral aspects of SDSCS have been modeled in Simulink and Stateflow (shown in Figure 18). Section 5.6 has briefly presented the Simulink and Stateflow modeling in SIGNAL. Sensors, such as *flight_status*, *dps*, and *door_io_in*, are connected to four Simulink blocks, each of which implements a SDSCS task as mentioned previously. Three blocks, *slide_warn_ctrl*, *pres_warn_ctrl*, and *closed_locked _and_latched* are associated with simple logic to determine actuator status from sensor readings. The fourth block, *flight_lock_ctrl* is associated with a state machine (specified in Stateflow), which decides the status of flight lock actuators.

6.2.3. Composing models: system integration

In addition to the high-level Simulink and AADL models, additional models are also needed in the SD-

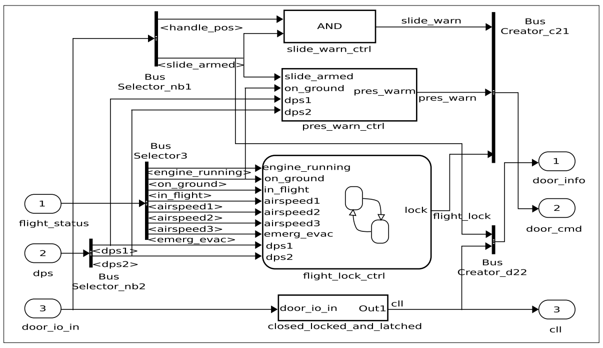

Figure 18: the door handler component of the SDSCS modeled in Simulink

SCS for the complete simulation. They include a scheduler model, an allocation model and an environment model.

- Scheduler models are generated for each AADL process in the case study, which has been presented in Section 5.5.
- In this case study, the allocation of functionality onto architecture is specified in the AADL model. In the AADL to SIGNAL transformation, all the threads that are mapped on to the same processor (CPIOM in this example) are placed in the same partition. The generated SIGNAL programs (from AADL threads) are annotated with allocation information. All the SIGNAL processes translated from the same partition have the same SIGNAL pragma[3] `RunOn i` [16] (`i` is the ID of a processor), which enables the distribution of these `processes` onto the same processor `i`.
- Sensors and actuators are the media between SDSCS and its environment. The environment model is responsible to provide sensor readings to SDSCS according to the estimated aircraft status in flight or on ground. The model includes functions to handle how the environment reacts to planes. The environment modeling is carried out directly in POLYCHRONY.

Figure 19 illustrates all the required models for a closed SDSCS system simulation. Once all these models are obtained, the composition of these models is possible at the SIGNAL process level, thanks to the synchronous composition operator [15]. All the models, such as system behavior, hardware architecture, envi-

[3]A pragma in SIGNAL is an annotation that is associated with a SIGNAL 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.

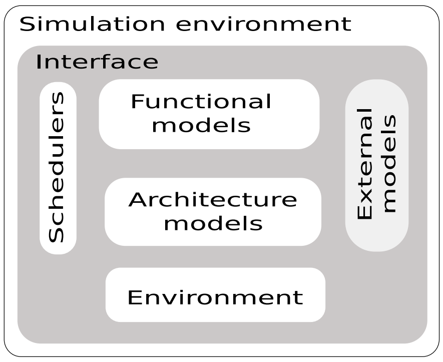

Figure 19: an implementation of SDSCS system integration, via Signal processes representing functional, archecture, scheduler, and environment models, in the framework of Polychrony.

ronment, and schedulers, are expressed by Signal processes, thus a composition of these processes achieves system integration. The integrated system is then used for C or Java code generation via the Signal compiler for closed system simulation purpose.

6.2.4. Simulation

In addition to heterogeneous system specification, another advantage of our approach, compared to other similar projects, is to benefit from simulation and validation techniques and tools associated with Polychrony (presented in Section 4.3) in the same framework. Three different techniques are chosen for this case study, including profiling as an example of formal timing analysis; VCD demonstration as an example of simulation; and scheduling and distribution as an example of architecture exploration.

Profiling. Figure 20 illustrates a schema of the co-simulation that has been carried out successfully. *SDSCS* is the original Signal program, whose inputs are provided by *Inputs*. *T(SDSCS)* is the temporal homomorphism of *SDSCS* with regard to specified *Temporal properties* and a parameterization of *Library of cost functions*. *Date* provides date signals to *T(SDSCS)* according to *I*. The input signals are synchronized to their corresponding date signals. Control values of *SDSCS*, which decide specific traces of execution, are sent to *T(SDSCS)* so that they have the same execution traces. Date signals of inputs and outputs of *T(SDSCS)* are finally sent to *Observer* in order to obtain the simulation result *V*.

VCD demonstration. In our simulation, simulation traces of SDSCS are recorded in VCD format. The

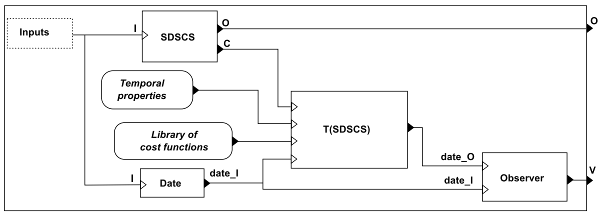

Figure 20: the co-simulation of Signal programs with regard to its temporal behavior

VCD files are then used for the visualization of simulation results through graphical VCD viewers, such as GTKWave [51]. Figure 21 shows a visualization result of the simulation. In this figure, the change of signal values with regard to the fastest clock is shown.

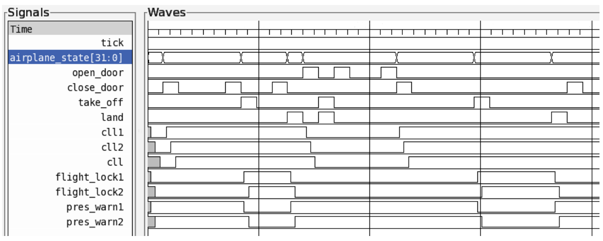

Figure 21: the simulation is illustrated by a VCD viewer: GTKWave

Real-time scheduling and distribution. Figure 22 illustrates the partial adequation results (i.e., a scheduling table) in SynDEx: algorithm (translated from the Simulink model) is mapped onto the architecture (translated from the AADL model) considering scheduling and distribution. There are five columns in the figures, which represents the five architectural components. The lines in the columns represents the computation or communication allocated on the corresponding architectural components. In this case study, the algorithm has more than 150 nodes and the architecture has 5 nodes in SynDEx. The adaquation takes about 15 minutes 35 seconds in average. With this tool chain, we can easily change the configuration of the execution platform. For example, the number of the processing units can be changed, the type of processing units and communication media can be easily changed. The influence of these changes are finally shown in SynDEx. Hence, our approach provides a fast yet efficient architecture exploration for the design of distributed real-time and embedded systems.

6.3. Discussions

The case studies have been presented in order to exhibit the modeling, formal analysis, simulation, archi-

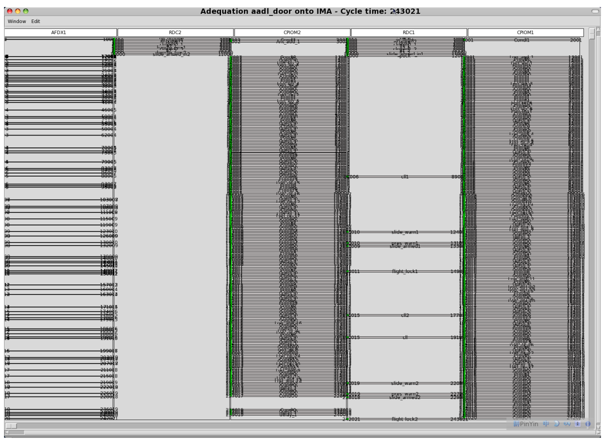

Figure 22: the result of the adaquation of algorithm and architecture in SynDEx: a partial view of a resulting scheduling table. The algorithm is distributed and scheduled onto the architecture. The first column represents the bus AFDX1. The second and fourth column represent RDC1 and RDC2. The third and fifth column represent CPIOM1 and CPIOM2. The black lines in these columns indicate the processing time on corresponding devices.

tecture exploration capabilities, provided by our Polychrony framework, for timed architectures in AADL. The first case study is mainly used to demonstrate the AADL-Signal translation, for instance, translations of thread and shared data. The second case study mainly presents the co-modeling, system integration and architecture exploration aspects. These case studies demonstrate the feasibility of a system-level codesign with different formal methods and techniques in the same framework, which is not always easy and obvious.

The two case studies show that all the analysis, simulation, verification and architecture exploration techniques are performed in the same Polychrony framework, based on the common polychronous model of computation. This ensures the semantic equivalence of the verified models while changing from one technique to another, which distinguishes from other work. The advantage of our approach is validation or simulation results are compatible and reusable from one technique to another. For example, the schedulers generated from one technique (affine-clock systems) can be directly used for VCD simulation and Sigali-based model-checking or controller synthesis [43, 36], without semantics domain change. However, only brief results have been presented for the formal techniques as how to use these techniques are not in the scope of this paper. Short descriptions and references have been given in Section 4.3 if readers are interested and expect to have more information.

Currently, the SDSCS case study, a simplified version of a real system, has not been certified with standards, such as DO178B and DO254 in our development. But it is interesting to consider them in the future. The redundant components in SDSCS, such as *doors_process1* and *doors_process2* can be isolated and allocated onto separate processing units, for example *CPIOM1* and *CPIOM2*, i.e., this isolation is considered at the allocation stage. Furthermore, each AADL process has its own scheduler running on the corresponding processing unit to enhance the isolation.

7. Conclusion

In this paper, we have presented the modeling and formal timing analysis of AADL components and their Simulink-based functional behaviors in a polychronous model of computation. The goal of our approach is to benefit both from: the high-level, domain-specific language AADL and Simulink for the system-level design, and the Polychrony toolset, based on the synchronous language Signal, for timing modeling, analysis and validation.

To bridge the gap between AADL and Polychrony, we have proposed a polychronous modeling of AADL, supporting different component types and timing properties, as well as behavior modeling in Simulink. We then used existing formal methods of Polychrony, including clock calculus, affine clocks and profiling to perform timing analysis. Scheduling, formal validation, simulation and distribution are also carried out, at high level and early phase, with reduced design cost and system complexity. Two case studies have been presented to demonstrate our approach.

Our approach shows advantages compared to Lustre/SCADE and Maude in the modeling of multi-clock systems, particularly multi-processor/distributed architectures. Users are not required to find/build the fastest clock in the system in our approach, which is not always obvious in a complex system. However, for simulation purpose, a fastest clock can be synthesized in an automatic way in Polychrony if necessary.

Despite the apparent complexity of the process and notations, but thanks to model engineering techniques and availability of integrated tool and technology platforms through initiatives like CESAR and OPEES, this approach is contributing towards the dissemination and use of formal verification techniques in industry.

The first perspective of our work is to work on a proposal towards the standardization of a synchronous timing annex for AADL, as a result of our successful AADL modeling and analysis framework. Another perspective is involved in the modeling of modes and behavior annex in AADL. Signal automata have been proposed to easily handle modes as well as AADL behavior annex. The third perspective is bidirectional translation between Signal and SynDEx, 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] MathWorks, The MathWorks: Simulink, http://www.mathworks.com/products/simulink/.

[2] Systems Modeling Language (SysML), http://www.sysml.org/specs.

[3] SAE Aerospace (Society of Automotive Engineers), Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL) , SAE AS5506A.

[4] Object Management Group (OMG), The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, http://www.omg.org/spec/MARTE/1.1/PDF (June 2011).

[5] AUTOSAR (AUTomotive Open System ARchitecture), http://www.autosar.org/.

[6] EAST-ADL, http://www.east-adl.info.

[7] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Scheduling and memory requirements analysis with AADL, in: ACM SIGAda international conference on ADA (SigAda'05), 2005.

[8] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, M. Roveri, Safety, Dependability, and Performance Analysis of Extended AADL Models, The Computer Journal 54 (2011) 754–775.

[9] P. Feiler, J. Hansson, Flow Latency Analysis with the Architecture Analysis and Design Language (AADL), Tech. rep., Carnegie Mellon University (2007).

[10] M. Chkouri, A. Robert, M. Bozga, J. Sifakis, Models in Software Engineering, Springer-Verlag, 2009, Ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems, pp. 5–19.

[11] P. Ölveczky, A. Boronat, J. Meseguer, Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude, in: J. Hatcliff, E. Zucca (Eds.), Formal Techniques for Distributed Systems, Vol. 6117, Springer, 2010.

[12] Y. Ma, H. Yu, T. Gautier, J.-P. Talpin, L. Besnard, P. Le Guernic, System Synthesis from AADL using Polychrony, in: Electronic System Level Synthesis Conference, 2011.
URL http://hal.inria.fr/inria-00594943

[13] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, O. Laurent, System-level Cosimulation of Integrated Avionics Using Polychrony, in: ACM Symposium on Applied Computing (SAC'11), 2011.
URL http://hal.inria.fr/inria-00536907/en/

[14] Y. Ma, H. Yu, T. Gautier, L. Besnard, P. Le Guernic, J.-P. Talpin, M. Heitz, Toward Polychronous Analysis and Validation for Timed Software Architectures in AADL, in: Design, Automation & Test in Europe (DATE'13), Grenoble, France, 2013, to appear.

[15] P. Le Guernic, J.-P. Talpin, J.-C. L. Lann, Polychrony for System Design, Journal for Circuits, Systems and Computers 12 (2002) 261–304.

[16] L. Besnard, T. Gautier, P. Le Guernic, J.-P. Talpin, Compilation of polychronous data flow equations, in: S. Shukla, J.-P. Talpin (Eds.), Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools, 2010.

[17] A. Kountouris, P. Le Guernic, Profiling of Signal Programs and its Application in the Timing Evaluation of Design Implementations, in: IEE Colloquium on the Hardware-Software Cosynthesis for Reconfigurable, 1996.

[18] I. M. Smarandache, T. Gautier, P. Le Guernic, Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints, in: World Congress on Formal Methods, 1999.

[19] Y. Sorel, SynDEx: System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems, ERCIM News 59 (2004) 68–69.

[20] IEEE, IEEE Standard for VHSIC Hardware Description Language (VHDL), IEEE Std 1364 -2005 (2006).

[21] L. Besnard, T. Gautier, P. Le Guernic, J.-P. Talpin, Compilation of polychronous data flow equations, Correct-by-construction embedded software design.

[22] Espresso, Polychrony/SSME, ESPRESSO, INRIA, http://www.irisa.fr/espresso/Polychrony/.

[23] OPEES Project, OPEES Project, http://www.opees.org/.

[24] Cost-efficient methods and processes for safety relevant embedded systems (CESAR project), http://www.cesarproject.eu/.

[25] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, Fiacre: an Intermediate Language for Model Verification in the Topcased Environment, in: Embedded Real-Time Software and Systems (ERTS'08), 2008.
URL http://hal.inria.fr/inria-00262442/en/

[26] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens, Virtual Execution of AADL Models via a Translation into Synchronous Programs, in: ACM & IEEE international conference on Embedded software (EMSOFT), 2007.

[27] E. Jahier, N. Halbwachs, P. Raymond, Synchronous Modeling and Validation of Priority Inheritance Schedulers, in: Fundamental Approaches to Software Engineering (FASE'09), 2009.

[28] J. Hugues, B. Zalila, L. Pautet, F. Kordon, From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite, ACM Transactions in Embedded Computing Systems (TECS).

[29] H. Garavel, F. Lang, R. Mateescu, W. Serve, CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes, in: Int. Conf. On Computer Aided Verification (CAV'07), 2007.

[30] B. Berthomieu, P.-O. Ribet, F. Vernadat, The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets, International Journal of Production Research 42 (14) (2004) 2741–2756.

[31] Esterel Technologies, SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/.

[32] A. Pnueli, The Temporal Logic of Programs, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), 1977, pp. 46–57.

[33] M.-A. Peraldi-Frati, A. Goknil, J. DeAntoni, J. Nordlander, A Timing Model for Specifying Multi Clock Automotive Systems. The Timing Augmented Description Language V2., in: 17th IEEE Conf. on Engineering of Complex Computer Systems (ICECCS), 2012.

[34] S. Quinton, R. Ernst, D. Bertrand, P. Meumeu Yomsi, Challenges and New Trends in Probabilistic Timing Analysis, in: Design, Automation, and Test in Europe (DATE), Dresden, Germany, 2012.

[35] TIMMO-2-USE Project, TADL2 deliverable, http://www.timmo-2-use.org/.

[36] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand,

P. Le Guernic, Polychronous Controller Synthesis from MARTE CCSL Timing Specifications, in: ACM/IEEE Ninth International Conference on Formal Methods and Models for Codesign (MEMOCODE'11), Cambridge, UK, 2011.

[37] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The Synchronous Languages Twelve Years Later, Proceedings of the IEEE.

[38] J.-P. Talpin, P. Le Guernic, S. Shukla, F. Doucet, R. Gupta, Formal Refinement Checking in a System-level Design Methodology, Fundamenta Informaticae 62 (2) (2004) 243–273.

[39] L. Besnard, T. Gautier, P. Le Guernic, SIGNAL V4-Inria Version: Reference manual, `http://www.irisa.fr/espresso/Polychrony/document/V4_def.pdf`.

[40] P. Amagbegnon, L. Besnard, P. Le Guernic, Implementation of the Data-flow Synchronous Language Signal, in: Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'95), ACM, 1995, pp. 163–173.

[41] A. Kountouris, P. Le Guernic, Profiling of Signal Programs and its application in the timing evaluation of design implementations, in: Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, HP Labs, Bristol, UK, 1996.

[42] A. Gamatié, T. Gautier, L. Besnard, Modeling of Avionics Applications and Performance Evaluation Techniques using the Synchronous Language SIGNAL, in: SLAP'03, Elsevier Science B.V., 2003.

[43] H. Marchand, M. L. Borgne, Synthesis of Discrete-event Controllers Based on the Signal Environment, in: Discrete Event Dynamic System: Theory and Applications, 2000, pp. 325–346.

[44] R. Bryant, Graph-Based Algorithms for Boolean Function Manipulations, IEEE Transaction on Computers C-45 (8) (1992) 677–691.

[45] M. C. Daniel, Globally-Asynchronous Locally-Synchronous Systems, Ph.D. thesis, Stanford University (October 1984).

[46] A. Toom, T. Naks, M. Pantel, M. Gandriau, I. Wati, Gene-Auto: An Automatic Code Generator for a Safe Subset of SimuLink/StateFlow and Scicos, in: European Conference on Embedded Real-Time Software (ERTS'08), 2008.

[47] OSATE, OSATE V2 Project, `http://gforge.enseeiht.fr/projects/osate2/`.

[48] Airbus, `http://www.airbus.com/`.

[49] Communications & Systems (CS), `http://www.c-s.fr/`.

[50] INRIA Espresso team, `http://www.irisa.fr/espresso/home_html`.

[51] GTKWave, `http://gtkwave.sourceforge.net/`.