# Toward polychronous analysis and validation for timed software architectures in AADL

*Abstract*—
**High-level architecture modeling languages, such as Architecture Analysis & Design Language (AADL), are gradually adopted in the design of embedded systems so that design choice verification, architecture exploration, and system property checking are carried out as early as possible. This paper presents our recent contributions to cope with clock-based timing analysis and validation of software architectures specified in AADL. In order to avoid semantics ambiguities of AADL, we mainly consider the AADL features related to real-time and logical time properties. We endue them with a semantics in the polychronous model of computation; this semantics is quickly reviewed. The semantics enables timing analysis, formal verification and simulation. In addition, thread-level scheduling, based on affine clock relations is also briefly presented here. A tutorial case study is finally adopted to illustrate our overall contribution.**

***Keywords*-AADL; MDE; Polychrony; timing analysis**

## I. Introduction

High-level standardized modeling languages, such as Architecture Analysis & Design Language (AADL) [1], the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [2], and Systems Modeling Language (SysML) [3], are gradually adopted for system modeling and specification due to issues of system complexity, time to market, validation, etc. Without necessarily having the physical implementation of a system, these languages, particularly AADL, permit the fast yet expressive modeling of a system, including software architecture, execution platform, and their binding. Early-phase analysis and validation can be therefore rapidly performed [4], [5], [6], [7], [8], [9].

Although AADL provides a fast design entry, there are still some critical challenges, such as unambiguous semantics, timing analysis, formal verification and co-simulation. To address these issues, expressive formal models and complete tool chains are required, based on which the previously mentioned verification and validation are enabled.

In our proposed approach, we first analyze the timing semantics of AADL, from which the formal polychronous/multiclock semantics is derived thanks to the multiclock nature of AADL specifications. Thus users are not suffered to find and/or build the fastest clock in the system. This distinguishes from [10], [5], where synchronous semantics is a prerequisite. This polychronous semantics is then expressed via a polychronous model of computation (MoC) [11] covering both AADL software, execution platform, and their binding. In addition, AADL thread-level scheduling is also explored and integrated according to affine clock relations [12]. With the scheduler synthesis, the translated AADL model is complete and executable, and can be used for the following analysis and validation.

Polychrony [13], a software environment dedicated to the trustworthy design of synchronous/polychronous embedded systems, provides the back-end semantic-preserving transformation, scheduling, code generation, formal analysis and verification, architecture exploitation, and distribution [14]. More precisely, the following concrete techniques are considered in our work: 1) static analysis, including determinism identification and deadlock detection; 2) profiling-based analysis of real-time characteristics of a system [15]; 3) affine clock calculus to analyze the affine relations between clocks [12]; 4) real-time scheduling and allocation through the Syndex tool [16]; 5) co-simulation of AADL specifications and demonstration using the VCD technique [17].

An automatic tool chain has been developed to support our work. A tutorial case study, developed in the framework of the OPEES project [18], is adopted in this paper to show the effectiveness of our contribution.

**Outline.** Section II briefly introduces AADL via the case study. Section III gives a short introduction to the polychronous MoC. Section IV presents our main contribution, and exemplifying it with the case study in Section V. Some related works are summarized in Section VI, and conclusion is drawn in Section VII.

## II. Introduction to AADL

AADL is the Society of Automotive Engineers (SAE) standard dedicated to modeling embedded real-time system architectures. 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. Three distinct component categories are provided in AADL: software application components (process, thread, thread group, subprogram, and data), execution platform components ((virtual) processor, memory, device, and (virtual) bus), and composite component (system).

In the following, an OPEES tutorial case study will be used, called *Producer-Consumer*, to illustrate progressively these AADL models. In this case study (in Fig. 1), a typical generic component takes charge of producing and consuming data through a shared data resource. It is implemented by different components, allowing the producer and consumer to communicate and to access data.

The system is composed of a process *prProdCons* (in Fig. 1) that communicates with two subsystems: *sysEnv* (models the environment) and *sysOperatorDisplay* (informs when a

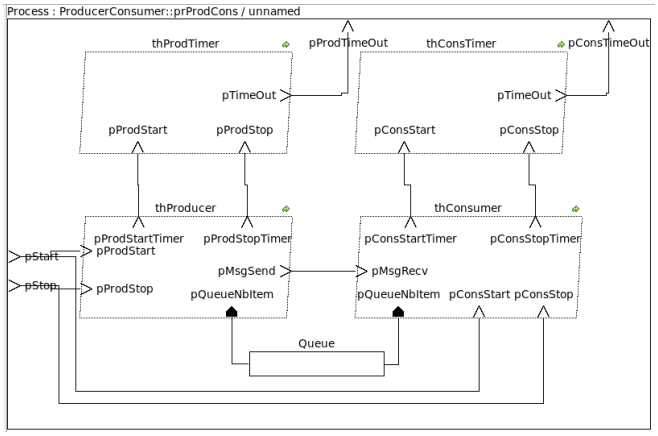


Fig. 1. AADL *Producer-Consumer example* (process level)

timeout occurred on data production or consumption). The *prProdCons* process is executed on a processor *Processor1*. It contains four threads: *thProducer*, *thConsumer*, *thProdTimer* and *thConsTimer*. Thread *thProducer* produces shared data in *Queue*, which in turn is consumed by thread *thConsumer*. The timer *thProdTimer* (resp. *thConsTimer*) manages timer services for *thProducer* (resp. *thConsumer*). It permits to start, stop timer and send a timeout event (*pTimeOut*) when the timer has expired.

Properties are specified to provide more information about model elements. We are interested in 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 IV-A.

## III. The Polychronous Model of Computation

Synchronous languages are dedicated to the design of synchronous reactive systems [19]. 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 relations [11], and to support *refinement* [20]. The multiclock/polychronous semantics of SIGNAL makes it more approximate to AADL semantics than other pure synchronous or asynchronous models, and thus simplify the system modeling, transformation, and validation.

The polychronous MoC of SIGNAL handles unbounded series of values in the domain $D_x$, where $x = (x_t)_{t\in\mathbb{N}}$, called *signals*, implicitly indexed by discrete time. At any instant, a *signal* is either present and holds a value $v$ in $D_x$; or absent and holds an extra value, denoted by $\perp$. The set of instants when a *signal* $x$ is present represents its *clock*. Two *signals* are said to be synchronous if they are both present (or absent) at the same instants (they have the same clock). Operations on signals include: step-wise functions, delay, sampling, deterministic merging, etc. More details can be found in [14]. SIGNAL is associated with the Polychrony design environment [13], which provides a formal framework for the trustworthy system design. From a polychronous MoC, Polychrony automatically synthesizes the fastest simulation clock and can make the non-determinism caused by multiclock transparent to users.

## IV. AADL Modeling and Analysis Framework

The AADL time model allows the specification of both logical and chronometric clocks in the system. In addition, each component can be associated with timing properties, which indicate their expected real-time characteristics. In general, this timing information is 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.

### A. AADL *timing execution model*

The thread component and its polychronous execution timing semantics is mainly involved and presented here. A thread is dispatched either periodically, or by the arrival of data or events on ports, or by the arrival of subprogram calls, depending on the thread type. Several timing properties can thus be assigned to a thread, e.g.:

```
Dispatch_Protocol => Periodic;
Period => 4 ms;
Deadline => 4 ms;
```

Three event ports are predeclared: *dispatch*, *complete* and *error* (Fig. 2). A thread is activated to perform the computation at *start* event, and has to be finished before deadline. A *complete* event is sent at the end of the execution.

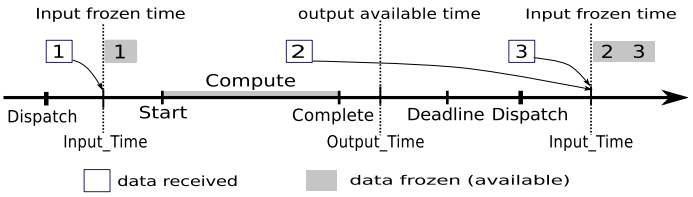


Fig. 2. Execution time of a thread

**Input-compute-output model.** The received inputs are frozen at a specified point (*Input_Time*), by default the *dispatch* time, which means that the content of the port that is accessible to the recipient does not change during the execution of a dispatch even though the sender may send new values. For example, the two values 2 and 3 (in Fig. 2) arriving after the first *Input_Time* will not be processed until the next *Input_Time*. As a result, the performed computation is not affected by a new arrival input until an explicit request for input. Similarly, the output is made available to other components at a specified point of *Output_Time*, by default at *complete* (resp. *deadline*) time for out port if the associated port connection is immediate (resp. delayed) communication.

### B. AADL *vs Polychrony time models*

Due to the different timing semantics, modeling embedded systems specified in AADL with POLYCHRONY raises some difficulties:

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 [9], where additional discrete events are added to model latency and delay.

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 the component-based architecture design in AADL.

Periodic clocks can be modeled in SIGNAL using affine clock relations. Thus, synchronizability analysis can be carried out between multi-period threads.

Data can be shared, read or written by different components at different time instants in AADL. It is possible in SIGNAL to have several expressions associated with one signal by partial definitions [14]. The clock calculus can compute sufficient conditions to guarantee that the overall definition is consistent and total.

### C. 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 introduces delay and well suited synchronizations [9].

A "memory" process $o = f_m(i, b)$ 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 value of previous $i$ when $i$ is absent and $b$ is true:

$$o = f_m(i,b) \stackrel{def}{\equiv}$$

$$\forall t > 0: \; o_t = \begin{cases} i_t & \text{if } i_t \neq \perp, \text{ and } b_t = true \\ i_{pred(t)} & \text{if } i_t = \perp, \text{ and } b_t = true, \\ & pred(t) = max\{k < t \mid o_k \neq \perp\} \\ \perp & \text{otherwise} \end{cases}$$

**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 currently received input $x$. It defines an elementary process such that:

$y = f(x) \stackrel{def}{\equiv} \forall t > 0: \; (x_t \neq \perp \Leftrightarrow y_t \neq \perp) \wedge (y_t = f(x_t))$.

*$x$ 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 $x \blacktriangleright t$:

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

**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:

$$th'(z_1, z_2, \dots, start) \stackrel{def}{\equiv} th(z_1', z_2', \dots)$$
where $z_i' = f_m(z_i, start)$

**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 $y \rhd t$: $w = y \rhd t \stackrel{def}{\equiv} w = f_m(g(y), t)$.

### D. Thread-level scheduler synthesis

An AADL model is not complete and executable if the thread-level scheduling is not resolved. Some scheduling tools, such as Cheddar [4], are well connected to AADL for schedulability analysis, scheduler synthesis and simulation inside these tools. However, they do not completely satisfy our demands for the following reasons: 1) logical and chronometric clocks are easily transformed into each other for formal and real-time analysis; 2) more events, such as input/output frozen events are also involved in the analysis; 3) static and periodic scheduling rather than stochastic/dynamic scheduling is expected due to predictability and formal verification; 4) the scheduling is easily and seamlessly connected to affine clock systems [12] so that formal analysis can be performed in Polychrony. Affine clock relations yield an expressive calculus for the specification and the analysis of time-triggered systems. A particular case of affine relations is the case of affine sampling relation expressed as $y = \{d \cdot t + \phi \mid t \in x\}$ of a reference discrete time $x$ ($d, t, \phi$ are integers): $y$ is a subsampling of positive phase $\phi$ and strictly positive period $d$ on $x$.

We therefore propose a static scheduler synthesis process including 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, and valid schedules are calculated according to a static, non-preemptive, and single-processor scheduling policy. 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. In the calculation process, different scheduling policies are considered, such as EDF and RM; 3) *export schedules to* SIGNAL *affine clocks in a direct way*.

### E. A complete and automatic tool chain

A tool chain for modeling, scheduling, timing analysis, and verification of AADL models in the polychronous MoC has been developed in the framework of Eclipse. The AADL model with timing properties, which conforms to the AADL metamodel, is captured in the OSATE toolkit [21]. A model transformation **ASME2SSME** allows to perform analysis on ASME models (AADL Syntax Model under Eclipse) and generate corresponding SIGNAL SSME models (SIGNAL Syntax Model under Eclipse). An AADL2SIGNAL library provides

common SIGNAL processes reducing significantly the transformation complexity and cost. With the SIGNAL and binary code generated from the SSME model, analysis and validation is carried out in the framework of Polychrony.

This tool chain is *scalable* in three 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) in the framework of Polychrony, analysis, verification, simulation, profiling techniques are considered as independent functions connected to the Polychrony core; 3) 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. In addition, several thousand clocks can be handled by the clock calculus. 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.

## V. A CASE STUDY

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 above. The timing semantics and properties are processed during the transformation.

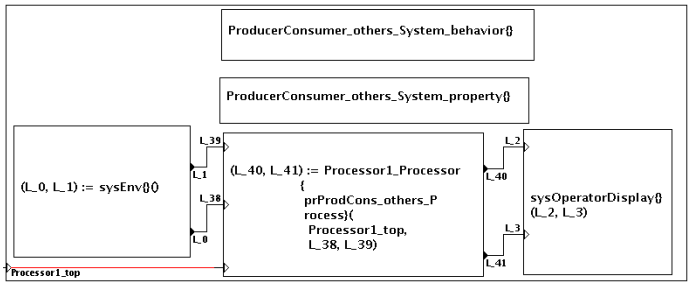


Fig. 3. *ProducerConsumer* system modeling in SIGNAL

The SIGNAL process resulting from the system implementation is given in Fig. 3: an instance of a SIGNAL process model of the processor *Processor1* communicates with two process instances that represent the systems *sysEnv* and *sysOperatorDisplay*. Subprocesses that represent system behavior (*ProducerConsumer_others_System_behavior()*) and property (*ProducerConsumer_others_System_property()*) are added.

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

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

The processes bound to this processor are implemented as SIGNAL subprocesses of the SIGNAL process that represents the *processor*.

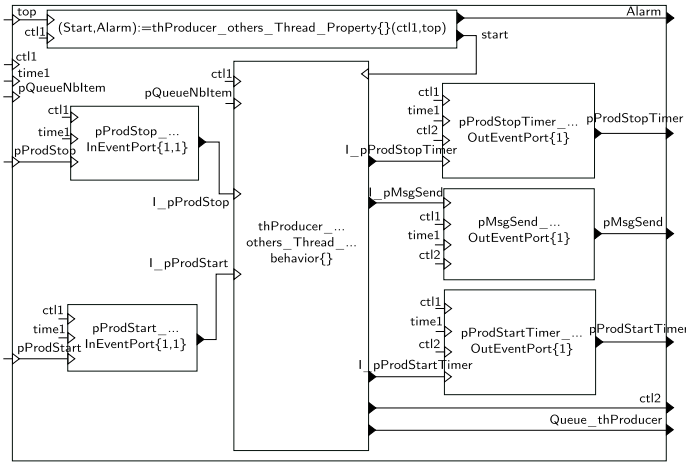


Fig. 4. *Producer* thread modeling in SIGNAL

### A. Thread

An AADL periodic thread is implemented as a SIGNAL process composed of its behavior, properties, ports, subcomponents (if data or subprogram subcomponents exist) and connections. Some additional timing signals are added (Fig. 4):

- An input *bundle* signal *ctl1* (a *bundle* represents a polychronous tuple of signals) contains event signals, *Dispatch*, *Resume* and *Deadline*, which are implicit predeclared ports or added simulation signals.
- An input *bundle* signal *time1* that provides the clock of the frozen time and output time for the event ports, e.g., *pProdStart_Frozen_time_event*.
- An output bundle 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 port is a logical connection point for the directional exchange of data/events between components. A thread port has special timing semantics: the in (resp. out) port is frozen (resp. sent out) at *Input_Time* (resp. *Output_Time*). Incoming events (the event data ports are similar, and the data ports modeling can be found in [9]) may be buffered in *event ports* with queues. The queue size can be explicitly declared by *Queue_Size* property, by default it is 1. Queues will be serviced according to the *Queue_Processing_Protocol*, by default in a First In First Out order (FIFO).

*In event port:* two FIFOs are provided: *in_fifo* for storing the received events, and *frozen_fifo* for storing the frozen events. The actual items of the *in_fifo* are frozen (presented as Frozen in Fig. 5) at *Input_Time* (presented as Frozen_time_event).

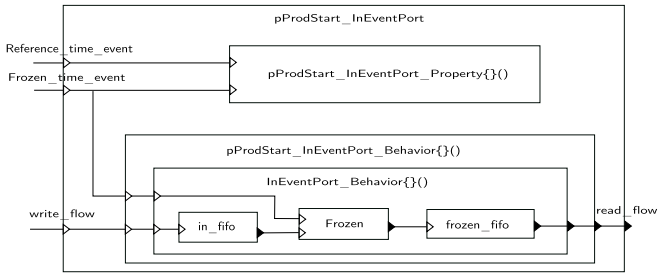


Fig. 5. In event port *pProdStart* modeling in SIGNAL

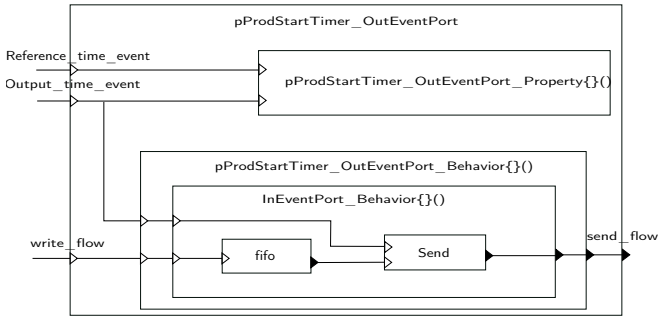


Fig. 6. Out event port *pProdStartTimer* modeling in SIGNAL

*Out event port:* for an out event port (e.g., *pProdStartTimer*), the values are stored in a *fifo*, and sent out (represented as Send) at *Output_Time* (Fig. 6).

### B. Shared data

Components can have shared access to data subcomponents, where the data act as a critical region and mutual exclusion access clocks are 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 is represented as a single 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*), a clock at which a thread reads, writes or resets the data is provided if the thread requires access to this data.

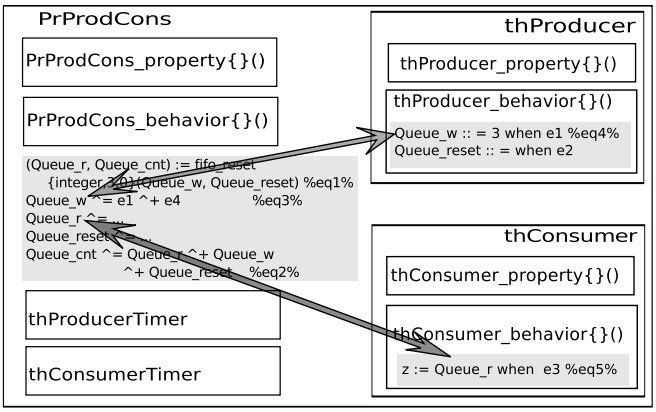


Fig. 7. AADL data *Queue* modeling in SIGNAL

The data *Queue* in the *prProdCons* process which is shared by threads *thProducer* and *thConsumer* is represented as a FIFO process instance *fifo_reset()* (equation $eq1$ in Fig. 7). 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) a data into the FIFO, a partial definition (such as equation $eq4$) is provided ($e1$ is a time instant at which the thread writes data).

### C. Formal analysis and simulation

Based on the polychronous MoC, an AADL specification is translated into the SIGNAL language. POLYCHRONY is used to formally analyze and verify the corresponding model, which includes: static analysis, simulation, performance analysis, etc. We only give a brief description here. Clock calculus has been applied, in the compiling stage, to analyze clock relations and identify the determinism in the AADL model. For example, the automaton of the *thProducer* thread has been checked: without correct priority properties specified on the transitions, the automaton is found to be non-deterministic. Other static analyses are also available, such as deadlock detection and model checking, which will not be illustrated here.

In the case study, all the threads are periodically dispatched, e.g., the periods of the four threads (*thProducer, thConsumer, thProducerTimer, thConsumerTimer*) are $4ms$, $6ms$, $8ms$ and $8ms$ respectively. A thread-level scheduler is first built considering SIGNAL affine clocks, which implements synchronizability rules based on properties of affine relations, against which synchronization constraints can be assessed. The generated valid schedules (see a simulation example in Fig. 8) are then seamlessly translated into SIGNAL for validation and simulation purposes. Our approach to verify scheduled models makes the main difference compared to other AADL scheduling tools like Cheddar.

Profiling has been 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 [15]. In addition, code distribution can also be implemented considering a distributed architecture [14]. Syndex has equally been connected, taking into account software and hardware and their binding, to perform low-level static scheduling, considering real-time, architectural, and allocation characteristics [16].

## VI. RELATED WORK

AADL has been connected to many formal models for analysis and validation. The AADL2Fiacre project [22] and the Ocarina project [23] mainly focus on model transformation and code generation, in other words, formal analysis and verification are performed externally with other tools and models. The AADL2Sync project [10] and the Compass Approach [5] provide complete tool chains from modeling to validation, but they are generally based on the synchronous semantics, which is not approximate to AADL timing semantics. We consider neither error model in [5] nor a more complete

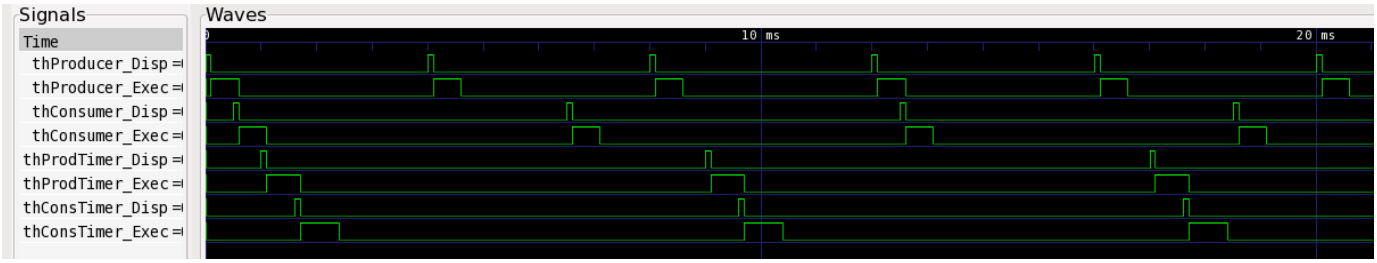


Fig. 8. An automatically generated static scheduling for *Producer-Consumer* system

scheduling of shared resources like in [10]. However, connections to allocation and code distribution are not reported in these works. AADL2Maude [8] introduces a real-time rewriting logic semantics only for a behavioral subset of AADL. AADL2BIP [7] allows simulation of AADL models, as well as application of particular verification techniques, i.e., state exploration and component-based deadlock detection. In comparison of all these projects, we provide a more natural and closed timing modeling with regard to AADL multiclock timing semantics, as well as a rich connection to various formal methods for verification and validation, to support system-level codesign.

## VII. Conclusion

In this paper, we present a polychronous semantics of AADL that considers both software and execution platform of a system, as well as timing properties of AADL components. The goal of our approach is to benefit both from the high-level, domain-specific language AADL for the system-level design, and the Polychrony toolset, based on the synchronous language SIGNAL, for timing analysis and validation. A tutorial case study was presented and used to demonstrate our approach. A perspective of our work is related to modes in AADL. SIGNAL automata have been proposed to easily handle modes as well as AADL behavior annex.

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 OPEES, this approach is contributing towards the dissemination and use of formal verification techniques in industry.

## References


[1] SAE Aerospace (Society of Automotive Engineers), "Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL) ," *SAE AS5506A*, 2009.

[2] 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.

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

[4] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and memory requirements analysis with AADL," in *ACM SIGAda international conference on ADA (SigAda'05)*, 2005. [Online]. Available: http://doi.acm.org/10.1145/1103846.1103847

[5] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, and M. Roveri, "Safety, Dependability, and Performance Analysis of Extended AADL Models," *The Computer Journal*, vol. 54, no. 5, pp. 754–775, 2011.

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

[7] M. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Models in Software Engineering," M. R. Chaudron, Ed. Springer-Verlag, 2009, ch. Translating AADL into BIP - Application to the Verification of Real-Time Systems, pp. 5–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01648-6_2

[8] P. Ölveczky, A. Boronat, and J. Meseguer, "Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude," in *Formal Techniques for Distributed Systems*, J. Hatcliff and E. Zucca, Eds. Springer, 2010, vol. 6117.

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

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

[11] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for System Design," *Journal for Circuits, Systems and Computers*, vol. 12, pp. 261–304, 2002.

[12] I. M. Smarandache, T. Gautier, and 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.

[13] "The Polychrony Toolset," http://www.irisa.fr/espresso/Polychrony/.

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

[15] A. Kountouris and 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.

[16] Y. Sorel, "SynDEx: System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems," *ERCIM News*, vol. 59, pp. 68–69, 2004.

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

[18] "Open Platform for the Engineering of Embedded Systems (OPEES Project)," http://www.opees.org/.

[19] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," *Proceedings of the IEEE*, 2003.

[20] J.-P. Talpin, P. Le Guernic, S. Shukla, F. Doucet, and R. Gupta, "Formal Refinement Checking in a System-level Design Methodology," *Fundamenta Informaticae*, vol. 62, no. 2, pp. 243–273, 2004.

[21] "OSATE V2 Project," http://gforge.enseeiht.fr/projects/osate2/.

[22] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat, "Fiacre: an Intermediate Language for Model Verification in the Topcased Environment," in *ERTS'08*, 2008. [Online]. Available: http://hal.inria.fr/inria-00262442/en/

[23] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite," *ACM Transactions in Embedded Computing Systems (TECS)*, 2008.