Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony

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

[a] *INRIA Rennes - Bretagne Atlantique, 263 avenue du Général Leclerc, 35042 Rennes, France*
[b] *IRISA/CNRS, 263 avenue du Général Leclerc, 35042 Rennes, France*
[c] *Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France*
[d] *TOYOTA ITC USA, 465 N Bernardo Avenue, Mountain View, CA 94043, USA*
[e] *Itemis France SAS, 198 avenue de Verdun, 92130 Issy-les-Moulineaux, France*

Abstract

High-level modelling languages and standards, such as Simulink, UML, SysML, MARTE and AADL (Architecture Analysis & Design Language), meet increasing adoption in the design of embedded systems in order to carry out system-level analysis, verification and validation (V&V) and architecture exploration, as early as possible. These analysis, V&V, architecture exploration techniques rely on mathematical foundations and formal methods in order to avoid semantics ambiguities in the design of safety-critical systems.

In order to support integration validation, it is necessary to define a formal framework of virtual prototyping to integrate, verify, exercise and analyse the application code generated by modelling tools as early as possible and virtually integrate it with simulators of third-party middleware and hardware. Such a virtual prototyping platform makes it possible to validate the expected behaviour of the final application software and check that the resulting system indeed meets the specified performance requirements before the actual hardware even actually exists.

In this paper, we present the definition, development and case-study validation of such a comprehensive framework, based on the synchronous paradigm and the polychronous model of computation and communication of its supportive open-source toolset: Polychrony. A longer-term aim of our work is to equip the AADL standard with an architecture-centric framework allowing for synchronous modelling, verification and synthesis of embedded software.

Keywords: Embedded systems, software architectures, formal methods, model-based design, timed behavioural modelling, affine scheduling, AADL, Signal.

1. Introduction

Modern design of embedded systems consists of the integration of hardware and software concurrently engineered by engineering teams with of different skills, with specific tasks, using heterogeneous tools. To design the sole software functions of an aircraft, for instance, they may for instance use a variety of tools as heterogeneous as Scade [1], Matlab [2], Ptolemy [3] or Rhapsody [4].

In an embedded system design flow, from top to bottom of Figure 1, the same heterogeneity characterises design objectives (bottom). It may range from that of mapping the functional design on specific hardware architectures to that of virtual prototyping, simulation, checking hardware/software compatibility, performance evaluation, or energy footprint estimation. Co-modelling encompasses a variety of engineering activities at the crossing between functional and physical views of system design. It is typically the system archi-

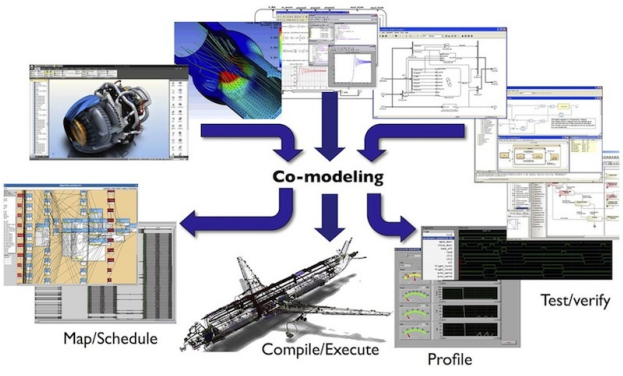

Figure 1: Co-modelling heterogeneous systems

tect, for instance, who explores possible ways to map software functions on a target hardware, in order to evaluate it according to different metrics such as speed, throughput, consumption.

Still, system validation does not stop with the formal verification and automated synthesis of the system's software from a high-level model. Code automatically generated by synthesis tools is usually integrated to middleware which has not. It is usually interfaced to hardware through drivers, third-party APIs, external libraries, and runs on an uncertified operating system.

Integration poses numerous problems, from ill-typed usage of libraries to insufficient performances, etc. It is hence necessary to verify conformance between expected properties that were formally demonstrated at the model-level with respect to these offered by the execution platform to form the actual application software.

Some of the performance issues discovered during validation may require architectural changes as drastic as actually implementing part of the software in hardware (e.g. using an FPGA). More generally, it is sometimes necessary to optimise architecture choices made at the modelling level and reiterate design validation.

In order to support integration validation, it is essential to define a virtual prototyping framework that makes it possible to integrate, verify, exercise and analyse the application code generated by modelling tools and integrated with third-party middleware. Such a virtual prototyping platform makes it possible to validate the expected behaviour of the final application software and check that the resulting system indeed meets the specified performance requirements.

Goals

Our contribution in this aim of virtual prototyping embedded architectures is illustrated, Figure 2, by a case-study we conducted with Airbus in [5]. We co-modelled the doors management system (DMS) of the Airbus A350 aircraft by using the AADL standard [6] to describe its architecture and by using Simulink **??** to specify its embedded software. The aim of the case study was to demonstrate how to perform simulation, scheduling analysis, verification and code-generation, by solely considering the information provided by these combined, system-level, AADL and Simulink specifications.

To this end, we implemented a systematic translations of the AADL standard and of Simulink diagrams in the multi-clocked synchronous semantics of the Signal data-flow language [7]. We use its Eclipse environment, Polychrony on Polarsys [8], as a pivot semantic model, in order to synthesise simulation code from imported Simulink and AADL models.

The aim of the present article is, first, to summarise all the effort conducted in this case study. In conclusion, or as a lesson learned from that experimental work, the article proposes, second, a reworked semantic model of Polychrony in terms of constrained automata, section 4, and a formal definition of the scheduling theory framework [9, 10, 11], section 7, that we elaborated to best suit our experimental framework.

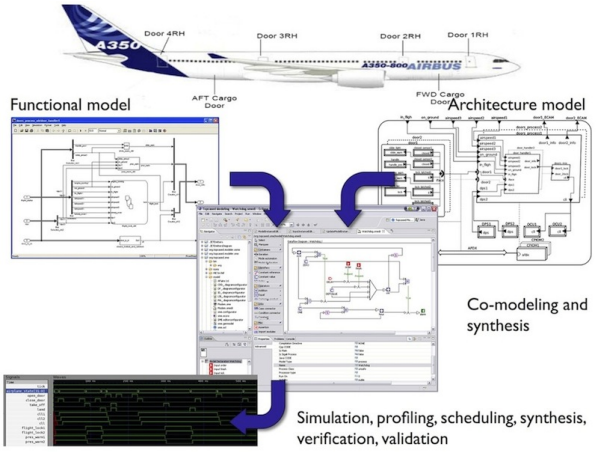

Figure 2: Co-modelling with synchrony

After review of the related works, Section 2, and to this end, Sections 5, 3 and 4 give a brief overview of the AADL core concepts, of the data-flow language Signal and of the model of constrained automata which we use to represent timed behaviours in the AADL.

The principle of our compositional semantic translation of the AADL specification into the polychronous model of computation and communication is then outlined in Section 6. In Section 7, we address our core issue to define a proper scheduling policy of AADL threads in our framework using a model of affine clocks.

Section 8 presents one of the case studies we performed to validate and experiment with our approach. Section 9 concludes our presentation by offering perspectives toward a formal specification of the AADL.

2. Related Approaches

The formal analysis, verification, architecture exploration techniques available in the AADL rely on formal models to avoid semantics ambiguities for the design of safety-critical systems [12, 13, 14, 15, 16, 7]. As a result, many related approaches have contributed to

allowing the formal specification, analysis and verification of AADL models and its annexes, hence implicitly or explicitly proposing a formal semantics or a model of computation and communication for the AADL.

The analysis language REAL [17] allows to define structural properties of AADL models that can be inductively checked by visiting the objects of a model under verification. Authors in [18] present an extension of this language, called LUTE, which further uses PSL (Property Specification Language) to check behavioural properties of AADL models, as well as a framework for assume-guarantee reasoning between composed AADL models.

The COMPASS project has also proposed a framework for formal verification and validation of AADL models and its error annex [13]. It puts the emphasis on capturing multiple aspects of nominal and faulty, timed and hybrid behaviours of models. Formal verification is supported by the nuSMV tool. Similarly, the FIACRE framework [19] uses executable specifications and the TINA model checker to check structural and behavioural properties of AADL models.

RAMSES [20], on the other hand, presents the implementation of the AADL behavioural annex. The behavioural annex supports the specification of automata and sequences of actions to model the behaviour of AADL programs and threads. Its OSATE implementation proceeds by model refinement and can be plugged in with Eclipse-compliant backend tools for analysis or verification. For instance, the RAMSES tools uses OSATE and Ocarina to generate C code for OSs complying the ARINC-653 standard.

Synchronous modelling is central in [16], which presents a formal real-time rewriting logic semantics for a behavioural subset of the AADL. This semantics can be directly executed in Real-Time Maude and provides a synchronous AADL simulator. It also allows to model-check LTL properties in that framework.

Similarly, Yang et al. [21] define a formal semantics for an implicitly synchronous subset of the AADL, which includes periodic threads and data port communications. Its operational semantics is formalised as a timed transition system. This framework is used to prove semantics preservation through model transformations from AADL models to the target verification formalism of timed abstract state machine (TASM).

Our proposal carries along the same goal and fundamental framework of the related work: to annex the core AADL with formal semantics frameworks to express executable behaviours and temporal properties, by taking advantage of model reduction possibilities offered thanks to a synchronous hypothesis, of close correspondence with the actual semantics of the AADL.

Yet, we endeavour in an effort of structuring and using them together within the framework of a more expressive multi-rate or multi-clocked, synchronous, model of computation and communication: polychrony. Polychrony would allow us to gain abstraction from the direct specification of executable, synchronous, specification in the AADL, yet offer services to automate the synthesis of such, locally synchronous, executable specification, together with global asynchrony, when or where ever needed.

CCSL [22], the clock constraint specification language of the UML profile MARTE [23], relates very much to the effort carried out in the present document. CCSL is an annotation framework to making explicit timing annotation to MARTE objects in an effort to disambiguate its semantics and possible variations. CCSL actually provides a clock calculus of greater expressivity than polychrony, allowing for unbounded, asynchronous, causal properties between clocks to be expressed such as the inf and sup clock operations.

While CCSL is essentially isolated as an annex of the MARTE standard for specifying annotations, our approach is instead to build upon the semantics of the behaviour and constraint annexes in order to implement a synchronous design methodology in the AADL, and specify it within a polychronous model of computation and communication.

3. An overview of Polychrony

The Polychrony toolset [24, 25] is an open-source modelling framework, DO-330 qualified (as verification tool) and integrated in the Eclipse Industrial Working Group Polarsys [26]. It is based on the Signal language [27], dedicated to multiclock synchronous program transformation and verification.

Signal is a polychronous data-flow language that allows the specification of multi-clocked systems. Signal handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical clock indicating the set of instants when the signal is present, noted $\hat{x}$. At a given instant, a signal may be present where it holds a value, or absent (denoted by #). In Signal, a process (written P or Q) consists of the synchronous composition (noted $P|Q$) of equations over signals x, y, z, written $x := y\, f\, z$ or $x := f(y, z)$. The process P/x restricts the lexical scope of the signal x to the process P. An equation $x := y\, f\, z$ defines the output signal x by the result of the application of operator f to its input signals y and z.

$$P, Q ::= x := y\, f\, z \mid P|Q \mid P/x$$

Semantic domains. For a set of values (a type) $\mathbb{D}_1$ we define its extended set $\mathbb{D}_{1\#} = \mathbb{D}_1 \cup \{\#\}$, where $\# \notin \mathbb{D}_1$ is a special symbol used to denote the absence of an occurrence in the signal. $\mathbb{D}_{1\#}$ is flat. We denote by $D^\infty = D^* \cup D^\omega$ the set of finite and infinite sequences of "values" in $\mathbb{D}_\#$. ϵ denotes the empty sequence. All signal functions $f : \mathbb{D}_1^\infty \times ... \times \mathbb{D}_n^\infty \to \mathbb{D}_{n+1}^\infty$ are defined using the following conventions: $s_1, ..., s_n$ are signals, $v_1, ..., v_n$ are values in D_i (cannot be #), $x_1, ..., x_n$ are values in $D_{i_\#}$, $s_1.s_n$ is the concatenation of s_1 and s_n. Signal functions are total, strict and continuous functions over domains [28] (wrt prefix order) that satisfy the following general rules:

- $f(\#.s_1, ..., \#.s_n) = \#.f(s_1, ..., s_n)$
- $f(s_1, ..., s_n) = \epsilon$ when for some i $s_i = \epsilon$

A function is *synchronous* iff it satisfies:

- $f(x_1.s_1, ..., x_n.s_n) = \epsilon$ when $x_i = \#$ and $x_j \neq \#$ for some i, j

Stepwise extension. Given $n > 0$ and an n-ary total function $f : \mathbb{D}_1 \times ... \times \mathbb{D}_n \to \mathbb{D}_{n+1}$ the *stepwise extension* of f denoted F is the synchronous function that satisfies:

- $F(v_1.s_1, ..., v_n.s_n) = f(v_1, ..., v_n).F(s_1, ..., s_n)$

Previous value. The *delay*: $\mathbb{D}_i \times \mathbb{D}_i^\infty \to \mathbb{D}_i^\infty$ is the synchronous (state-)function that satisfies

- $delay(v, v_{-1}.s) = v.delay(v_{-1}, s)$

Deterministic merge. The *default* $\mathbb{D}_i^\infty \times \mathbb{D}_i^\infty \to \mathbb{D}_i^\infty$ signal function is recursively defined by

- for $v \in \mathbb{D}_i$, $default(v.s_1, x.s_2) = v.default(s_1, s_2)$
- $default(\#.s_1, x.s_2) = x.default(s_1, s_2)$

Boolean sampling. The *when* $\mathbb{D}_i^\infty \times \mathbb{B}^\infty \to \mathbb{D}_i^\infty$ signal function is recursively defined by

- for $b \in \mathbb{B}_\#, b \neq tt, when(x.s_1, b.s_2) = \#.when(s_1, s_2)$
- $when(x.s_1, tt.s_2) = x.when(s_1, s_2)$

A network of strict, continuous signal functions that satisfies the Kahn conditions is a strict, continuous signal function or Kahn Process Network [29].

Non-deterministic processes. A process with feedback or local variables may be (asynchronously) non-deterministic. One typical example is the *var* operator in Listing 1 (the notation ^ returns true when its argument is present). The semantics of non-deterministic processes can be defined using Plotkin's power-domain construction [30].

```
process var = {type t; t val} (? t in ! t out)
    (| mem := in default mem$ init val
     | out := mem when ^out
     |)
where t mem
end
```

Listing 1: Definition of the Signal process cell

4. A model of constrained automata in Polychrony

To support a semantic translation of AADL behavioural annexes (see Section 5) in a polychronous model of computation, we consider a model of automata that comprises transition systems to express explicit reactions together with constraints in the form of boolean formula over time to represent implicit or expected timing requirements.

The implementation of such an automaton amounts to composing its explicit transition system with that of the controller synthesised from its specified constraints. It is supported by the Polychrony toolset and offers an expressive capability similar to those of the Esterel [31] and Signal [27] synchronous programming languages.

The fundamental difference between synchronous automata and the asynchronous style of the current AADL behavioural annex [20] is that, in a synchronous automaton, transitions can be labelled by guards defined by a conjunction of events. To simply overcome this limitation, we will write $a \hat{} * b$ for the conjunction of occurrences of events a and b.

Constrained automata are reactive synchronous automata which manipulate timing events and are subject to constraints. These constraints formulate safety or temporal requirements. Would a transition of an automaton possibly violate such constraints during runtime, then its possible state transition should be inhibited and instead stutter or raise an error. Figure 3 depicts a constrained automaton manipulating events a and b.

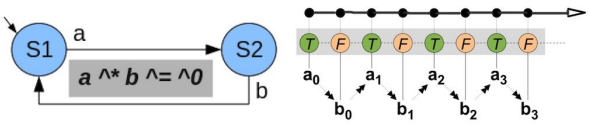

Figure 3: An alternating automaton controlling its input flow.

The automaton specifies the alternation of two input event streams a and b, depicted by the trace on the top-right of the figure. Its reactive behaviour, depicted by the top-left automaton, keeps track of alternation between a and b by switching between states s_1 and s_2. It is yet a partial specification of possible synchronous transitions over the vocabulary of events $\{a, b\}$: it does not yet make the cases of a, b in s_1 or s_2 explicit. This is done by superimposing it with the requirement that $a \hat{} * b = 0$, which imposes a and b to never occur simultaneously (literally, a and b occur 0 time) during a transition. With that constraint in place, the automaton behaves as an asynchronous one. Finally, the absence of reflexive transitions specify that b (respectively a) cannot occur alone in state s_1 (respectively s_2). A reactive extension of this automaton allows b (respectively a) to

occur in state s_1 (respectively s_2). But the reflexive transition must be fired only when b (respectively a) occurs alone. This is denoted by the event expression $b\hat{} - a$ (respectively $a\hat{} - b$).

The combination of a synchronous automaton and of a temporal constraint yields the hybrid structure of timed automata depicted in Listing 2. It supports an algebraic definition, presented next. Using the Polychrony toolset, we are currently implementing transformation and synthesis techniques which allow to synthesise an imperative program (or, equivalently, a complete synchronous automaton) that is able to both satisfy the intended semantics of the event automaton, but also enforces the expressed constraint formulas. In addition, these formula can themselves be expressed as automata abstracted by regular expressions on events (event formula), e.g., $(a; b)^*$ to mean "always a followed by b".

```
thread alternate
features
   a,b:  in event port;
   c:    out event port;
end alternate;

thread implementation alternate
annex behaviour_specification {**
   states
      s1: initial state;
      s2: state;
   transitions
      t1: s1-[on dispatch a]->s2;
      t2: s2-[on dispatch b]->s1 { c! };
**};

annex timing_specification {**
   constraints
      never a ^* b;
      -- i.e. always (a^-b)^+(b^-a)*
      -- i.e. regexp ((a^-b)^+(b^-a))*
**};
end alternate;
```

Listing 2: Separation of structure, behaviour and time concerns

4.1. Constrained automata

We first consider a countable set of Boolean *signal variables* of which V denotes a possibly empty finite subset. S is a non empty finite set of states; states and signal variables are disjoint sets. In the reminder, the symbol $\hat{}$ prefixes the so called clock of a variable (e.g. $\hat{}x$), of a state, of an operator. The term variable is used for signal variable. We first consider a Boolean algebra $\phi(V, S) = (\mathbf{F}_{V,S}, \hat{}*, \hat{}+, \hat{}\neg_V, \hat{}\mathbf{0}, \hat{}\mathbf{1}_V)$ consisting of

- infimum $\hat{}+$, supremum $\hat{}*$, complement $\hat{}\neg_V$, minimum $\hat{}\mathbf{0}$ and maximum $\hat{}\mathbf{1}_V$
- Expressions $\forall f, g \in \mathbf{F}_{V,S}, f\hat{} * g, f\hat{} + g, \hat{}\neg_V f \in \mathbf{F}_{V,S}$ built from constants: $\hat{}\mathbf{0}, \hat{}\mathbf{1}_V \in \mathbf{F}_{V,S}$ and atoms: $\forall x \in V \cup S$, $\hat{}x, [x], [-x] \in \mathbf{F}_{V,S}$.

Parentheses and affix notation are used. Our Boolean algebra supports the following formal properties.

- $\forall x \in (V \cup S)$, $(\hat{}x = [x]\hat{} + [-x]) \wedge ([x]\hat{} * [-x] = \hat{}\mathbf{0})$
- $\hat{}\Sigma_{x\in V}(\hat{}x) = \hat{}\mathbf{1}_V$, $(\forall s \in S)(\hat{}s = \hat{}\mathbf{1}_V)$
- $\forall s_1, s_2 \in S$, $[s_1]\hat{} * [s_2] = \hat{}\mathbf{0}) \vee (s_1 = s_2)$

Then, a *constrained automaton* $A = (S_A, s_0, \rightarrow_A, V_A, T_A, \mathbf{C}_A)$ defined on such formulas is, up to isomorphism, composed of

- S_A a non empty set of states and s_0 the initial state
- $\rightarrow_A \subset S_A^2$ a transition relation
- V_A a set of signal variables
- We denote by $\mathbf{F}_{A,S}$ its set of formulas on $\phi(V_A, S_A)$
- $T_A : (\rightarrow_A) \rightarrow F_{A,S}$ assigns a formula to transitions. However, note that, $\forall s, s_1, s_2 \in S_A$, $[s]$ shall not occur in $T_A(s_1, s_2)$ *(since* $[s_1]$ *is true and for any other state* $[s]$ *is false)*
- $\mathbf{C}_A$ is the constraint of A, a null formula of $\mathbf{F}_{A,S}$.
  - A formula f in $\mathbf{F}_{A,S}$ is null in A iff $f\hat{} * \mathbf{C}_A = f$.
  - If $\mathbf{C}_A$ is $\hat{}\mathbf{0}$, the automaton is constraint free.
  - If $\mathbf{C}_A$ is $\hat{}\mathbf{1}_{V_A}$ all formulas in A are null.

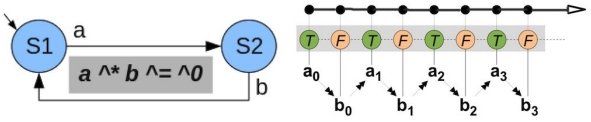

Figure 4: The alternating automaton is decomposed into states $S = \{s_1, s_2\}$, variables $V = \{a, b\}$, transitions labelled by $T = \{(s_1, s_2) \mapsto a, (s_2, s_1) \mapsto b\}$ and constraint $C : (a\hat{} * b) = \hat{}\mathbf{0}$. Its control clock is $\hat{}\mathbf{1} = a\hat{} + b$. In state s_1, the trigger is $T(s_1) = a$, the null clock $C(s_1) = C\hat{} * \hat{}\mathbf{1} = C$ so that the automaton can only accept a.

$\mathbf{1}_A$ denotes the supremum $\mathbf{1}_{V_A}$ of an automaton A, for a state s in A, $\mathbf{C}_A(s) = \mathbf{C}_A\hat{} * [s]$ and $\mathbf{C}_A(s)$ the null clock of a state s. It is defined as the simplified positive Shannon cofactor (for atom $[s]$) of $\mathbf{C}_A\hat{} * [s]$: occurrences of $[s]$ (resp. $[-s]$) are replaced by $\hat{}\mathbf{1}_A$ (resp. $\hat{}0$) and, if t is not s, occurrences of $[t]$ (resp. $[-t]$) are replaced by $\hat{}0$ (resp. $\hat{}\mathbf{1}_A$). When $h = \mathbf{1}_A\hat{} - (\mathbf{C}_A(s)\hat{} + \mathcal{T}(s))$ is not null, there is a (stuttering) step $h : s \rightarrow_A s$.

4.2. Regular expressions

We define the algebra of regular expressions which will be used to abstract constrained automata or represent there null formula [32]. We consider a Kleene algebra $(A, +, ., *, 0, 1)$, i.e., an idempotent semiring $(A, +, ., 0, 1)$, an idempotent commutative monoid

$(A,+,0)$ and a monoid $(A,.,1)$ s.t. $a.0 = 0.a = 0$, $a.(b+c) = a.b+a.c$ and $(a+b).c = a.c+b.c$. A supports a natural, monotonic, partial order $(a \leq b)$ iff $(a+b=b)$ and star definition satisfying $1+aa* \leq a*$, $1+a*a \leq a*$, $b+ax \leq x \Rightarrow a*b \leq x$ and $b+xa \leq x \Rightarrow ba* \leq x$. Our objective is to represent events and event formulas as regular expressions (extended) with counting. We compare with the related property specification language PSL [33].

The words $S \in W_A$ of an automaton A are generated from values, operators and formula. Values h are event formula (in place of $\{h\}$) and neither the empty set $\mathbf{0}$ nor $\mathbf{1} = \{\epsilon\}$ have PSL representation. Both 0 and 1 should remain implicit, as part of the event algebra, with no explicit syntax. Operators of concatenation $S_1.S_2$, or $S_1;S_2$ in PSL; union $S_1 + S_2$, $S_1|S_2$ in PSL; star $S*$; positive $S+ = S;S*$; option $S? = 1 + S$; fusion $S : T$, synchronous product $S|T$, interleaving and subsets as well as the usual reduction rules. Finally, counters [34] of the form $S[n]$ are inductively defined by $S[0] = 1$ and $S[m+1] = S;S[m]$

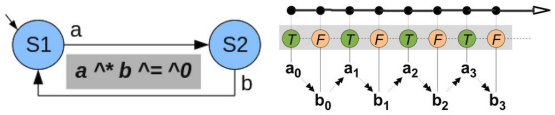

Figure 5: The constraint of the alternating automaton $C = (a\hat{}* b)$ can equivalently be expressed as the regular event expression $((a\hat{}-b)+(b\hat{}-a))*$. The alternating automaton could itself be alternatively expressed by the composition of two regular event expressions consisting of the negation of the constraint $(a\hat{}* b)*$ and of its transitions $(a.b)*$, which yields $((a\hat{}-b).(b\hat{}-a))*$.

5. An overview of the AADL

AADL [6] is a Society of Automotive Engineers (SAE) standard dedicated to modelling embedded real-time system architectures. As an architecture description language, based on a component modelling approach, AADL describes the structure of systems as an assembly of software components allocated on execution platform components together with timing constraints.

Architecture

In AADL, three distinct families of components 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 such as execution platforms, application softwares.

We illustrate the AADL structure by using a generic producer-consumer design pattern *Producer-Consumer* of common use for avionic systems. The system diagram, Figure 6, is in charge of producing and consuming data which is communicated through a shared data resource.

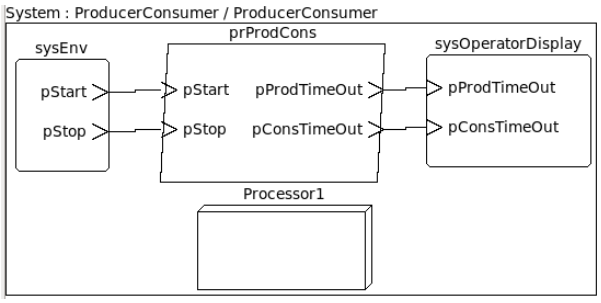

Figure 6: System-level view of a *producer-consumer* model in AADL

The AADL components communicate via data, event, and event data ports. It contains 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.
- *prProdCons* process communicates with the previous two systems. It contains four threads: *thProducer*, *thConsumer*, *thProdTimer* and *thConsTimer* (Figure 7).

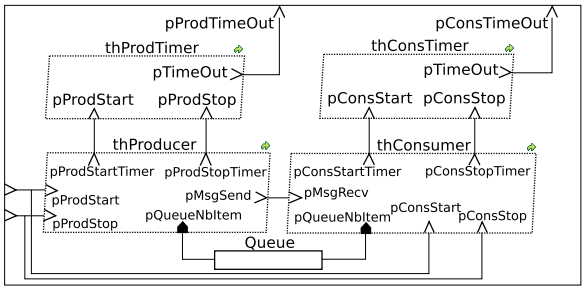

Figure 7: Process-level view of a *producer-consumer* model in AADL

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 executing threads 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.

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 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. 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;

Actual_Processor_Binding =>
   Processor1 applies to prProdCons;
```

Listing 3: Real-time and binding properties of the *thProducer* thread

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

In general, 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. Three event ports are predeclared: *dispatch*, *complete* and *error* (Figure 8). 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. 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 data values 2 and 3 (in Figure 8) 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 `Output_Time`, by default at *complete* (resp. *deadline*) time for out port if the associated port connection is immediate (resp. delayed) communication.

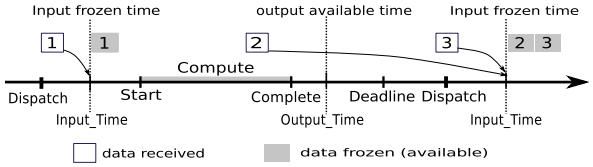

Figure 8: Execution time model for an AADL thread

Behaviour

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

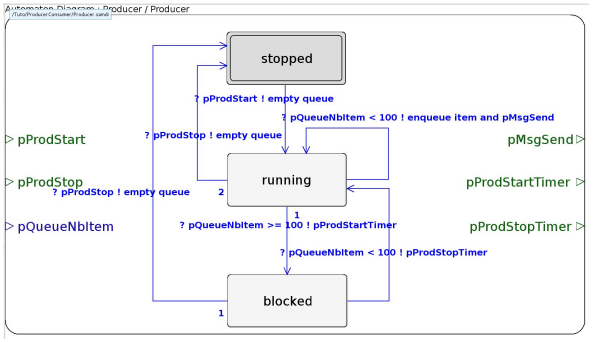

Figure 9: The behaviour of AADL *Producer* `thread`

We take the *thProducer* thread as an example. Three states, *stopped*, *running*, and *blocked*, are specified together with the transitions between them (Figure 9). 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.

6. Semantic model of the AADL in Polychrony

The key idea for modelling computing latency and communication delay in Signal is to keep the ideal view of instantaneous computations and communications of synchrony while delegating computing or simulation

of latency and delays to specific "memory" processes, that introduce delay with well-suited synchronisations to timed signals.

Input freezing. For an in event data (or event) port, the items are frozen at `Input_Time`. Two FIFOs are used: the first one, *in_fifo*, that stores the in event data, and another one, *frozen_fifo* (limited to one place in the current implementation). At the frozen time, the items of the *in_fifo* are frozen and some items (one in the current implementation) are popped up from the *in_fifo* to the *frozen_fifo* and made available at the execution.

Thread activation (Figure 10). An activation condition "*Start*" is introduced into the process *P* which models a thread. With the process *IM()*, the input *i* is resynchronised with *Start* before entering the synchronous program *P*.

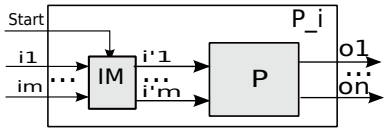

Figure 10: Activation condition.

Output sending (Figure 11). To the process *P* modelling a thread behaviour, we associate a process *P_o*, the output o' of which is the value of the output *o* of *P* delayed until its output time occurs, represented by the input event signal *OutEvent*. An additional output memory process *OM()* is introduced to hold the values, which includes a delay process *AT()* for each output.

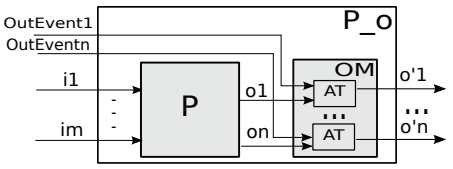

Figure 11: Modelling a time consuming task.

The semantic transformation from the AADL to Signal is recursive. A package, which represents the root of an AADL specification, is transformed into a Signal module, the root of a Signal program, allowing to describe an application in a modular way. An AADL component implementation consists of:

- an identifier of the component implementation,
- a set of features (such as ports) of the component type that define a functional interface,
- subcomponents, subprogram call sequences,
- port connections, which are an explicit relation between features (ports) that enables the directional exchange of data or event among its features (ports) and subcomponents,
- properties and a transition system specifying the functional behaviour.

The rest of the transformation proceeds modularly by an inductive translation of the AADL concepts of a given model or source text. Each component implementation is translated into a Signal process composed of the following subprocesses:

- an interface consisting of input/output signals translated from the features (ports) provided by the component type,
- additional control signals, that may also be added depending on the component category (*Dispatch* and *Deadline* for a thread),
- a body, itself composed of subcomponents, subprogram call sequences, connections, component properties and a transition system.

6.1. Threads

Threads are the main executable and schedulable AADL components. An AADL thread encapsulates a functionality that contains an interface (a set of ports), a set of connections and a set of timing properties. The behavioural specification of the thread consists of subcomponents that may contain subprograms and data and is functionally represented by an annex that defines its transition system over a set of local states.

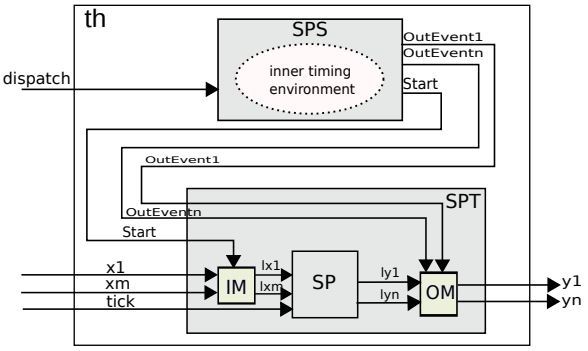

Figure 12: Translation of an AADL thread in Signal

The semantic translation of a thread in Signal has the data-flow structure depicted in Figure 12 and consists of the following steps.

- An AADL thread *th* is translated to a Signal process *SP*, which contains a representation of its transition system and of the associated subcomponents *Su* and connections *C*. Process *SP* has the same input/output flows as thread *th*, plus an additional *tick* to interface it with the scheduler (Figure 12) and receive dispatch. Each input (resp. output) port $p_i \in P$ corresponds to an input (resp. output) signal.

- The AADL standard specifies that the input signals of a thread are transmitted from the sending threads at the output time of the ports. They are consequently translated by a Signal process *SPT* which plays the role of timing semantics interface between the AADL environment and the synchronous model of the thread's behaviour. In *SPT*, two memory components *IM()* and *OM()* are added. The main functions of *SPT* are to memorise signals and activate threads.
- The execution of a thread is characterised by real-time features such as dispatch time and completion time. To model them, a template process *SPS* is added to records all the temporal properties associated with the thread, and when it receives the scheduling signals (e.g., *dispatch*) from the thread scheduler, it starts to calculate the timing signals (*Start, Completion, Deadline...*) for activating and completing the *SPT* process.

6.2. Processor and Scheduling

In the AADL, a processor is an abstraction of a hardware execution unit and the software or middleware responsible for executing and scheduling threads. The property `Scheduling_Protocol` defines the way the processor will be shared between the threads of the application. Possible scheduling protocols include: `Rate_Monotonic`, `Deadline_Monotonic`, `Earliest_Deadline_First`, `Least_Laxity_First`, `Highest_Priority_First` or user defined scheduling policies.

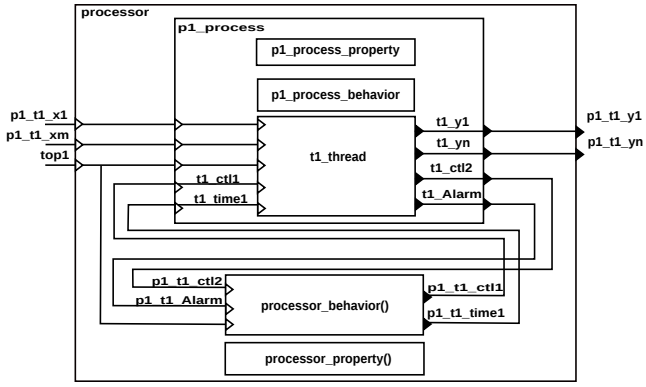

Figure 13: Translation of an AADL processor in Signal

An AADL processor is represented as a Signal process (Figure 13). The interface contains the inputs/outputs of the AADL processes bound to this processor. The body is composed of the Signal processes representing these AADL processes (*p1_process*), and subprocesses representing *processor_behavior* and *processor_property*. The *processor_behavior* subprocess implements a scheduler (or the interface to a scheduler), that provides scheduling information to the threads, in accordance with the recorded timing constraints and the scheduling policy.

6.3. Connections

Port connections are explicitly declared relations between ports (groups) that enable directional exchange of data and events between components. There are several combinations of event/data port connections. In this section, we focus on modelling data port connections. The AADL provides three types of data port communication mechanisms between threads: *immediate*, *delayed* and *sampled* connections. For an immediate or delayed data port connection, both communicating periodic threads must be synchronised and dispatched simultaneously. The transmission time of a connection is specified by a `Latency` property. The rest of the transformation proceeds modularly by an inductive translation of AADL concepts by corresponding source code patterns.

Sampled (Figure 14). A *Connection* process receives values from an output port when *OutEvent* is present. Since the communication takes a duration represented by *Latency*, an event *LatencyEvent* is introduced to specify the time at which the value has been sent to the destination. The received values cannot be used immediately: they are not available until input time, specified by `Input_Time` property and represented by an event *InEvent* in Signal. Hence, another memory *AT()* process is added.

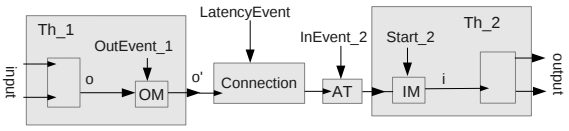

Figure 14: AADL sampled data connection in Signal

Delayed (Figure 15). The value from the sending thread is transmitted at its deadline (*OutEvent* = `Deadline`), and is available to the receiving thread at its next dispatch (*InEvent* = `Dispatch`).

Immediate (Figure 16). Data transmission is initiated when the source thread completes and enters the suspended state. The output is transmitted to the receiving thread at the complete time of the sending thread (*OutEvent* = `Completion`) and available on the receiver

side when *InEvent* = `Start`. The scheduler will dispatch the sender thread first, and ensure the receiver starts after the completion of the sender.

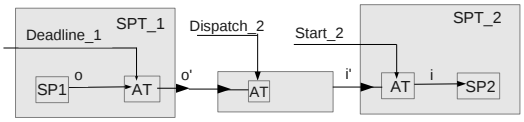

Figure 15: AADL delayed data connection in Signal

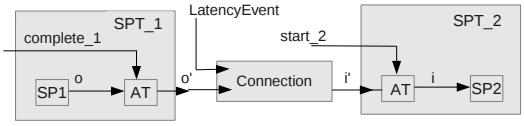

Figure 16: AADL immediate data connection in Signal

6.4. Binding

A complete AADL system specification includes both software application and execution platform. The allocation of functionality onto architecture is specified, for example, the property `Actual_processor_binding` declares the processor binding property along with the AADL components and functionality to which it applies. In the corresponding Signal programs, the threads (Signal processes) bound to the same processor are placed in a same process *M*, and they are controlled by the scheduler *MS* generated for the processor. The generated Signal programs are annotated with allocation information. The function *Binding(th)* provides the processor to which a thread *th* is bound.

In [35], we describe a distributed simulation technique using MPI [36] as runtime environment and the distributed code generator of the Polychrony toolset [24] in order to generate a concurrent executive in which every individual Signal process corresponding to an AADL processor will be assigned to a given runtime thread. In the future, we plan to use similar multi-processor real-time scheduler synthesis technique as these available in SynDEx [37].

6.5. System

The system is the top-level component of an AADL model. A system is organised into a hierarchy that represents the integrated software and hardware of a dedicated application. A system specifies the runtime architecture of an operational physical system. It consists of ports declaring its interface, port connections among components, properties among which additional allocation directives are provided, and a set of threads.

An AADL system is transformed into a Signal process which consists of the composition of containers including the executable *thread* processes, their connections and related properties, as well as a global scheduler to activate each processor scheduler and ports.

7. AADL scheduling

Each thread translated from AADL is associated with a timing environment (Figure 17), which records timing constraints associated with the thread and computes the timing control signals (i.e., *Start*, *OutEvent*, etc.), once it is activated (by *dispatch* event) by the scheduler.

Figure 17 shows a composition of two threads *P* and *Q* presented in Signal. Each component is activated by its corresponding activation event *Start*. Once *SPS* is triggered by the signal *dispatch*, it generates activation clock *Start* to activate the component, and computes the output available clock *OutEvent* that satisfies the specified clock properties. The following section shows how the timing environment *SPS* can be generated from an affine analysis proposed in [38, 9, 10]. Once the different timing semantics are bridged, now we can model the AADL threads and other components in Polychrony.

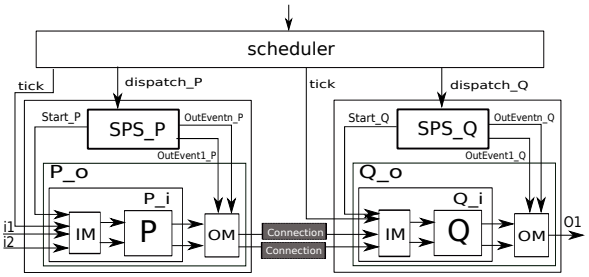

Figure 17: Modelling asynchronous composition of threads.

An AADL specification generally consists of a set of threads with various dispatch protocols and timing requirements that must execute on a given architecture according to some scheduling policy. Schedulability analysis is a critical step in the design of any real-time system. It predicts whether the timing requirements will be satisfied by the system or not. This section does not present a standard schedulability analysis but a parametric one; i.e. timing and scheduling parameters (e.g. periods, deadlines, priorities) that ensure schedulability and optimize the system's performance are rather synthesised in accordance with the AADL specification. Furthermore, buffering parameters (i.e. capacities of connections) are also computed to ensure overflow and underflow-free communications. In this section, we will

hence try to adapt well-known standard (EDF and fixed-priority) schedulability tests to our needs to synthesise the timing environment of each thread.

In the following, we present the parametric schedulability analysis of a restricted but yet expressive task model that corresponds to a subset of AADL specifications. But, we will first present the modelling of threads and communications between them.

7.1. System model

Systems considered in this section consist of a set $\mathcal{P}$ of N threads to be scheduled on $M \geq 1$ homogeneous processors according to either the (partitioned) earliest-deadline first (EDF) policy or the (partitioned) fixed-priority policy. Each thread p_i has the following timing properties: a periodic `Dispatch_Protocol` with a `Period` π_i, a `First_Dispatch_Time` r_i, a `Compute_Execution_Time` property C_i, and a hard `Compute_Deadline` d_i which can be equal to the period (i.e. implicit deadline) or a fraction of it; $d_i = \beta_i \pi_i$ (i.e. a constrained deadline).

In case of fixed-priority scheduling, each thread p_i has a distinguished `Priority` ω_i with 1 being the highest priority. In case of partitioned multiprocessor scheduling, each thread p_i is permanently allocated to processor number ψ_i (AADL property `Actual_Processor_Binding`). Apart from the worst-case execution times, the designer is *free to not* specify values of the other timing and scheduling parameters; and hence lets the parametric schedulability analysis compute the appropriate parameters. Many optimization problems can be considered such as throughput maximisation or buffer storage capacities minimisation.

Interactions among threads are restricted to one-to-one port connections which are feature connections whose source and destination are limited to ports and data access. There are two kinds of port connections: message queuing and data port connections.

Message queuing connections. They are connections whose destination ports are either event ports or event data ports. They can be modelled as flow-preserving FIFO queues; i.e. all sent messages are consumed, without loss or duplication, in the same order of arrival. These connections establish some scheduling precedences between communicating threads.

A thread will block (or raise an underflow exception) if frozen inputs do not contain the necessary number of messages for its execution. Similarly, a thread will block (or raise an overflow exception) when it attempts to send messages on saturated connections. The AADL standard supports `Overflow_Handling_Protocol` to handle overloaded queues; our scheduling approach will however statically check and ensure that overflow and underflow exceptions will not occur at run-time.

Besides `Input_Time` and `Output_Time` properties of ports, `Input_Rate` and `Output_Rate` properties allow specifying the number per dispatch at which input and output are expected to occur at the port; they describe therefore the production and consumption rates. We will only consider an analysable class of rates called ultimately periodic rates. Furthermore, the (possibly unspecified) number of initial messages on the connection e is denoted by $\theta(e)$; while the (possibly unspecified) `Queue_Size` is denoted by $\delta(e)$.

Data port connections. They are connections whose destination ports are data ports. They have a special non-flow-preserving semantics. They can be either sampled, immediate, or delayed. In a sampled connection, the receiver samples the output of the sender at time specified by the `Input_Time` property. Hence, the sampling occurs non-deterministically due to concurrency and preemption. Sampled connections do not hence impose scheduling precedences.

For delayed connections, the `Input_Time` is assumed to be the dispatch time and `Output_Time` is assumed to be the deadline. Delayed connections do not also impose any scheduling constraint. For immediate connections, the `Input_Time` is assumed to be the start time; they however impose scheduling precedences since the receiving thread must be delayed if its dispatch occurs at the same time or after the dispatch of the sending thread and before execution completion of the sending thread.

7.2. Scheduling approach

Scheduling of such systems consists of two (not necessarily separated) steps: the construction of an abstract periodic schedule and its concretisation.

Abstract schedules. An abstract schedule consists of all the scheduling constraints deduced from the user-provided scheduling parameters or from the interactions between threads. Scheduling constraints are mainly affine relations that relate dispatch clocks of threads, precedences between dispatches (i.e. jobs) of threads, priority assignments (in case of fixed-priority scheduling), and partitions (in case of multiprocessor scheduling). An abstract schedule must be checked to detect inconsistency and must be appropriately refined to complete absent information.

Symbolic schedulability analysis. It is the process of computing the timing and scheduling properties of each thread so that all threads meet their deadlines when scheduled using a specific scheduling policy on a given architecture. Of course, the computed parameters must respect the abstract schedule.

If all the timing and scheduling parameters are user-provided, then the symbolic schedulability analysis problem reduces to the standard real-time schedulability analysis. Two priority-driven scheduling policies are considered in this section: EDF and fixed-priority scheduling.

7.3. Abstract schedules

Many scheduling constraints must be satisfied to produce an overflow/underflow-free schedule that conforms to the AADL specification.

7.3.1. Precedences

An immediate data port connection may cause the receiver to be delayed until the completion of the sender. The AADL standard suggests using synchronisation protocols to implement such scheduling precedences. However, synchronisation and resource sharing protocols result in a much complicated and pessimistic schedulability analysis. We adopt a different strategy to implement precedence constraints by exploiting the priority-driven scheduling policy.

Let us assume that there is an immediate data port connection between threads p_i and p_k. The j^{th} dispatch of a thread p is denoted by $p[j]$. If job $p_k[j']$ of the receiver thread is dispatched at the same time as job $p_i[j]$ or after the dispatch of $p_i[j]$ and before its completion, then we add a scheduling constraint saying that job $p_i[j]$ must precede job $p_k[j']$.

This scheduling constraint is encoded as follows. For EDF, the absolute deadline of job $p_i[j]$ (denoted by $D_i[j]$) will be adjusted to be less than the deadline of job $p_k[j']$ and the two threads are allocated to the same processor (i.e. $\psi_i = \psi_k$); hence, job $p_k[j']$ will never preempt or execute in parallel with job $p_i[j]$. For the fixed-priority scheduling policy, thread p_i must have a higher priority than thread p_k (i.e. $\omega_i < \omega_k$) and the two threads must be allocated to the same processor.

Adjusting deadlines and priorities to impose precedences is not a new idea. It has been first introduced by Chetto and Blazewicz [39, 40] and used in [41] to encode more general AADL communication patterns. We use this technique in a parametric context where timing properties are not known a priori.

7.3.2. Affine relations

Affine relations relate the dispatch clocks of threads. Let $\hat{p}_i$ be the dispatch clock of thread p_i. The relative positioning of ticks of two periodic dispatch clocks $\hat{p}_i$ and $\hat{p}_k$ can be described by an affine relation [42] which has three parameters $n, d \in \mathbb{N}_{>0}$ and $\varphi \in \mathbb{Z}$ (Figure 18).

We say then that threads p_i and p_k are (n, φ, d)-affine-related. In case φ is positive (resp. negative), clock $\hat{p}_i$ is obtained by counting each n^{th} instant on a referential abstract clock $\hat{c}$ starting from the first (resp. $(-\varphi + 1)^{th}$) instant; while clock $\hat{p}_k$ is obtained by counting each d^{th} instant on $\hat{c}$ starting from the $(\varphi+1)^{th}$ (resp. first) instant. Affine relations are obtained as follows:

• If p_i and p_k have known periods and first dispatch times, then they are (n, φ, d)-affine related such that

$$n\pi_k = d\pi_i \quad \wedge \quad r_k - r_i = \frac{\varphi}{n}\pi_i \tag{1}$$

• If p_i and p_k communicate with each other through a message queuing connection, then the affine relation between them must exclude (without using synchronisation protocols) potential overflow and underflow exceptions as described in the sequel.

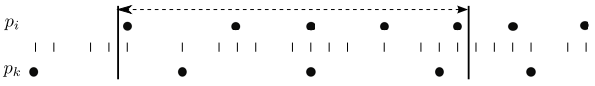

Figure 18: A $(3, -4, 5)-$affine relation. There is a positioning pattern which consists of 5 activations of p_i and 3 activations of p_k.

Overflow and underflow analysis. Let e be a message queuing connection between the sender p_i and the receiver p_k. Rate functions $x, y : \mathbb{N}_{>0} \longrightarrow \mathbb{N}$ denote the production and consumption rates, respectively. For instance, $x(j)$ represents the number of messages produced by $p_i[j]$. The cumulative function of a rate function x is the function $X : \mathbb{N} \longrightarrow \mathbb{N}$ such that $X(j) = \sum_{l=1}^{j} x(l)$. Function x is ultimately periodic if and only if $\exists j_0, \pi \in \mathbb{N}_{>0} : \forall j \geq j_0 : x(j + \pi) = x(j)$. One important property of such class of functions is that $\lim_{j\to\infty} \frac{X(j)}{j}$ is a constant $\tilde{x} \in \mathbb{Q}_{>0}$. Ultimately periodic rates are more expressive than the constant rates of the SDF model [43] and the periodic rates of the CSDF model [44].

By the `Input_Time` of job $p_k[j']$, the number of accumulated messages on the input port must be greater or equal to $y(j')$ in order to exclude underflow exceptions. Therefore, if $X^*(j')$ denotes the total number of produced messages before the `Input_Time` of $p_k[j']$, we must have that

$$\forall j' \in \mathbb{N}_{>0}, \theta(e) + X^*(j') - Y(j') \geq 0 \tag{2}$$

By the `Output_Time` of $p_i[j]$, there must be at least $x(j)$ empty entries in the queue in order to

avoid overflow exceptions. Therefore, if $Y^*(j)$ denotes the total number of consumed messages before the `Output_Time`, we must have that

$$\forall j \in \mathbb{N}_{>0}, \theta(e) + X(j) - Y^*(j) \leq \delta(e) \quad (3)$$

The values of $X^*(j')$ and $Y^*(j)$ depend on the affine relation between p_i and p_k, `Input_Time` and `Output_Time` properties, and the scheduling policy. If they cannot be exactly predicted due to, for example, potential preemption, then safe bounds are required.

Figure 19 gives an intuition on how these values are computed. In the four cases, deadlines are equal to periods and the `Output_Time` is equal to the completion time. The `Input_Time` is equal to the dispatch time except in case (d) where it is equal to the start time. Thread p_i and p_k are allocated to the same processors except in case (d).

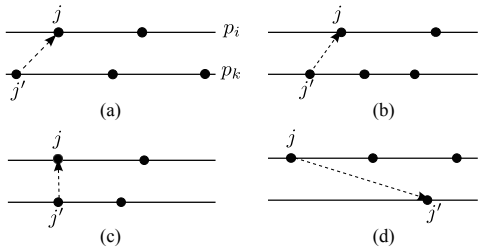

Figure 19: Precedences in the overflow and underflow analysis

• **Case (a)**- EDF scheduling: Since $D_k[j'] < D_i[j] < D_k[j'+1]$, job $p_i[j]$ will start after completion of $p_k[j']$ and end before the start of $p_k[j'+1]$. Hence, $Y^*(j) = Y(j')$. By the `Input_Time` of $p_k[j'+1]$, it is not certain that $p_i[j]$ has completed or not; hence a safe bound of $X^*(j'+1)$ is $X(j-1)$.

• **Case (b)**- EDF scheduling: It is likely that $p_k[j'+1]$ preempts $p_i[j]$ which implies that $Y^*(j) = Y(j'+1)$. But, if the time interval between the dispatches of $p_i[j]$ and $p_k[j'+1]$ is sufficient to complete $p_i[j]$, then a safe bound of $Y^*(j)$ is $Y(j')$.

• **Case (c)**- RM scheduling: Thread p_k has a higher priority than thread p_i. So, $p_k[j']$ will complete before $p_i[j]$ starts. However, job $p_k[j'+1]$ may or may not preempt job $p_i[j]$. Hence, a safe bound of $Y^*(j)$ is $Y(j')$.

• **Case (d)**- Fixed-priority scheduling with $\omega_i < \omega_k$ and $\psi_i \neq \psi_k$: Even if p_i has a higher priority than p_k, job $p_i[j+1]$ can run in parallel with job $p_k[j']$. Hence, a safe bound of $X^*(j')$ is $X(j)$.

In case of ultimately periodic rates, Equations 2 and 3 can have solutions only if (boundedness criterion)

$$\frac{n}{d} = \frac{\tilde{x}}{\tilde{y}} \quad (4)$$

Consistency analysis. The previous analysis constructs each affine relation independently of the others. However, the obtained set of affine relations may be inconsistent. Let the graph of affine relations be the undirected graph where nodes represent threads and edges represent affine relations. An affine relation can be reversed; i.e. saying that p_i and p_k are (n, φ, d)-affine-related is equivalent to saying that p_k and p_i are $(d, -\varphi, n)$-affine-related. Proposition 1 [38] ensures consistency.

Proposition 1. *The set of affine relations is consistent if for every fundamental cycle* $p_1 \xrightarrow{(n_1,\varphi_1,d_1)} p_2 \rightarrow \cdots \rightarrow p_m \xrightarrow{(n_m,\varphi_m,d_m)} p_1$ *in the graph of affine relations, we have*

$$\prod_{i=1}^{m} n_i = \prod_{i=1}^{m} d_i \quad (5) \qquad \sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} d_j\right)\left(\prod_{j=i+1}^{m} n_j\right)\varphi_i = 0 \quad (6)$$

Equations 1 to 6 are used to compute all the necessary affine relations. In [38], we have presented an integer linear formulation of the problem by considering safe linear approximations of Equations 2 and 3. In the rest of this section, we will consider the case where the graph of affine relations is weakly connected. Hence, according to Equation 1, all periods and deadlines of threads can be expressed as $\forall p_i \in \mathcal{P} : \pi_i = \alpha_i T, d_i = \beta_i T$. Furthermore, T must be a multiple of some integer B so that the equation may have integer solutions.

7.4. symbolic schedulability analysis

This step concerns the process of synthesizing timing and scheduling parameters of threads that respect the abstract schedule, ensure schedulability, and optimise the system's performance.

7.4.1. Priority assignments

In case of fixed-priority scheduling, priorities must be assigned to threads so that constraints resulted from the precedences encoding are respected together with the user-provided priorities. The deadline-monotonic (DM) priority ordering policy [45] is popular. It assigns the highest priority to the task with the shortest deadline. When deadlines are equal to periods, DM priority ordering is equivalent to rate-monotonic (RM) priority ordering. The DM priority assignment policy aims at maximising the processor utilisation ($U = \sum_{p_i \in \mathcal{P}} \frac{C_i}{\pi_i}$) and hence the throughput. However, if minimising queue sizes is a more important requirement, then other policies (see [11] for examples) could achieve better buffer storage capacities than the DM priority assignment since priorities affect queue size computation.

7.4.2. Multiprocessor partitioning

The major advantage of partitioned scheduling techniques is that, once an allocation of threads to processors has been achieved, it is possible to apply real-time symbolic schedulability analyses for uniprocessor systems (described in the next section).

Partitioning the hard tasks on M identical processors to optimise some criterion is somehow equivalent to the bin-packing problem and hence NP-hard. Therefore, heuristics are attractive solutions. They consist in sorting tasks by some criteria (e.g. according to their priorities or deadlines) and then assigning each task to a processor according to some conditions [46].

Let $(V_i)_{i=1,M}$ be the set of initially empty M partitions. Threads are allocated one by one according to the chosen order. Let U^i be the processor utilisation obtained by the symbolic schedulability analysis when thread p_k is assigned to partition V_i.

One allocation strategy is to assign thread p_k to the partition that gives the maximum U^i. This best fit strategy aims at maximising the throughput. Strategies that minimise buffering requirements can be designed since the allocation of threads affects the queue sizes computation.

7.4.3. Parametric schedulability analysis

Assuming that the graph of affine relations is connected, we can put $\forall p_i \in \mathcal{P} : \pi_i = \alpha_i T, d_i = \beta_i T$, and $U = \frac{\sigma}{T}$ such that $T^l \leq T = kB \leq T^u$. The bounds on T (i.e. T^l and T^u) are deduced either from the user-provided bounds on the frequencies of threads or from the constraint $C_i \leq d_i$. Regardless of the scheduling policy, a periodic task set is infeasible on one processor if its utilisation U is greater than one.

Hence, a necessary but not sufficient condition for schedule feasibility is that $T \geq \sigma$. Taking $T^l = \max\{T^l, \sigma\}$, we have that $T = \left\lceil \frac{T^l}{B} \right\rceil B + kB$ for $k \in \mathbb{N}$. The enumerative solution for throughput maximisation consists in checking the schedulability of the task system for each T in an increasing order, starting from the minimum value (i.e. $k = 0$) and until reaching an appropriate value or exceeding the upper bound T^u. However, this time consuming approach can be improved, as we show next.

Fixed-priority scheduling. A sufficient schedulability test of DM scheduling of constrained-deadline periodic task systems is that $\sum_{p_i \in \mathcal{P}} \frac{C_i}{d_i} \leq N(\sqrt[N]{2} - 1)$. The value of the appropriate T can be easily deduced from that equation. The most accurate analysis of the fixed-priority scheduling policy (regardless of the priority assignment policy) is based on worst-case response times [47]. The worst-case response time R_i of a periodic task $p_i \in \mathcal{P}$ is given by

$$R_i = C_i + \sum_{\omega_k < \omega_i} \left\lceil \frac{R_i}{\pi_k} \right\rceil C_k \tag{7}$$

Equation 7 can be solved with a recurrence approach starting with $R_i^0 = C_i$ and until $R_i^{n+1} = R_i^n$. If $R_i \leq d_i$ for all hard periodic tasks then all these tasks will meet their deadlines. Searching for the minimum value of $T \in [T^l, T^u]$ that satisfies $\forall p_i \in \mathcal{P} : R_i \leq d_i$ can be speedup by noting that:

1) If $R_i \leq d_i$ for some T, then $R_i \leq d_i$ for all $T' > T$.

2) *Reducing the search space*: We have that

$$\frac{C_i - \sum_{\omega_k < \omega_i} C_k(1 - U_k)}{1 - \sum_{\omega_k < \omega_i} U_k} = R_i^l \leq R_i \leq R_i^u = \frac{C_i + \sum_{\omega_k < \omega_i} C_k(1 - U_k)}{1 - \sum_{\omega_k < \omega_i} U_k}$$

Hence, if $\exists p_i \in \mathcal{P} : R_i^l > d_i$, then the task system is infeasible. Dually, if $\forall p_i \in \mathcal{P} : R_i^u \leq d_i$, then the task set is schedulable. Solving these linear second degree inequalities should reduce enormously the search space.

3) In the case of multiprocessor scheduling, threads are ordered according to their priorities. Assigning task p_k to a partition V_i will not affect the worst-case response time of the threads already assigned to V_i. Hence, it is sufficient to satisfy the new condition $R_k \leq d_k$.

EDF scheduling. A sufficient EDF schedulability test of constrained-deadline periodic task systems is that $\sum_{p_i \in \mathcal{P}} \frac{C_i}{d_i} \leq 1$. The most accurate analysis is based on the notion of processor demand [48]. A periodic task set is schedulable if $U \leq 1$ and $\forall l \leq L : h(l) \leq l$. The feasibility bound L is equal to the length of the synchronous busy period given by the solution of the following recurrence:

$$w^0 = \sum_{p_i \in \mathcal{P}} C_i \qquad w^{m+1} = \sum_{p_i \in \mathcal{P}} \left\lceil \frac{w^m}{\pi_i} \right\rceil C_i \tag{8}$$

The processor demand function $h(l)$ calculates the maximum processor demand of all jobs which have their arrival times and deadlines in a contiguous interval of length l. So, $h(l)$ is given by

$$h(l) = \sum_{p_i \in \mathcal{P}} \max\{0, 1 + \left\lfloor \frac{l - d_i}{\pi_i} \right\rfloor\} C_i$$

It is necessary to check condition $h(l) \leq l$ only for the set of absolute deadlines which are less than L. This set can be large and a technique like the Quick convergence Processor demand analysis (QPA) [49] is needed.

Searching for the minimum value of $T \in [T^l, T^u]$ that satisfies the EDF schedulability can be speedup by incorporating the search for the minimum T in the QPA algorithm. Algorithm 1 represents our symbolic QPA algorithm. Let $L(T)$ denote the value of L for a given T. Starting from $T = T_{\min}$, SQPA checks points in the interval $[0, L(T_{\min})]$ in a backward manner. This first iteration leads to either $h(l) \leq \min\{d\}$ or a deadline miss, i.e. $h(l) > l$. In the first case, the task system is schedulable and the algorithm returns $T_{\min}$. In the second case, T is increased and the verification continues without rechecking the previous points thanks to the following observations:
1) If $h(l) \leq l$ holds for a given T, then it holds for any $T' > T$.
2) If $T < T'$, then $L(T) \geq L(T')$.

$T = T_{\min}$ (i.e. $k = 0$); $l = \max\{d | d \leq L(T)\}$;
while $h(l) > \min\{d\}$ **do**
  if $h(l) < l$ **then** $l = h(l)$ **else if** $l == h(l)$ **then**
  $l = \max\{d | d < l\}$ **else**
    $prev = h(l)$; increase T;
    if $T > T^u$ **then return** task set not
    schedulable $l = \min\{prev, \max\{d | d \leq L(T)\}\}$
  end
end
return T;

Algorithm 1: SQPA algorithm

The performance evaluation of these two symbolic schedulability tests together with many other tests can be found in [11]. The objective of this section was to show that standard schedulability tests can be adapted to *synthesise* the timing and scheduling characteristics of AADL threads. However, we have only yet analysed a small subset of AADL specifications and much more properties should be considered in futur work (e.g. jitters, offsets, aperiodic dispatches, etc.).

8. Case studies

In the frame of the collaborative projects Topcased [50], OpenEmbeDD [51], Cesar [52], Opees [53], we performed a series of case studies whose results are reported in earlier publications [7, 54, 5, 35]. In this section, we will focus on the probably most representative of the type of application under consideration: a simplified design of a slides and doors control system (SDSCS). The *SDSCS* is a generic simplified version of the system that allows managing slides and doors on the Airbus 350 series aircrafts chosen for the demonstration of capabilities developed in the CESAR project [52].

Architecture

The *SDSCS* is implemented on an 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 20). Sensors and actuators are also connected to RDCs via AFDX. CPIOMs receive sensor readings via RDCs and communicate also with other systems via AFDX.

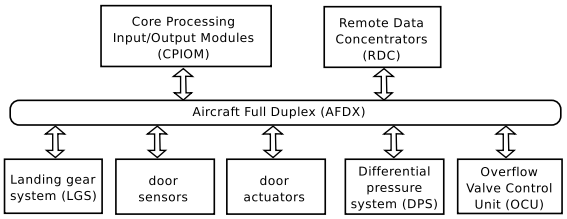

Figure 20: A simple illustration of the SDSCS system architecture

Figure 21 gives an overview of the *SDSCS* system model in the AADL. The two doors, modelled as subsystems, are controlled by two processes *doors_process1* and *doors_process2*. Each process contains three threads to perform the management of doors. All the threads and devices are periodic and share the same period. The process *doors_process1* (resp. *doors_process2*) is executed on a processor *CPIOM1* (resp. *CPIOM2*). The bus *AFDX* connects the processors, *CPIOM1* and *CPIOM2*, and devices that model the sensors and actuators, such as *DPS*, *OCU*, etc.

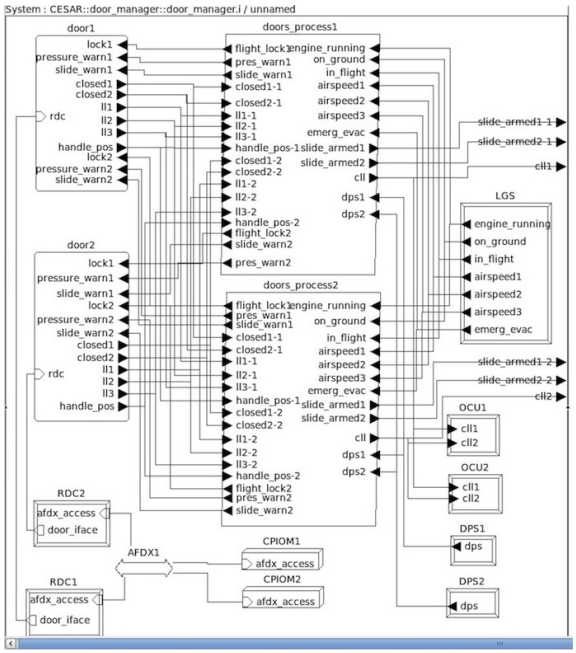

Figure 21: The SDSCS system architecture model in the AADL

The embedded software driver of the SDSCS is responsible for the following tasks:

- monitor door status via door sensors;
- control flight lock actuators;
- manage the residual pressure of the cabin;
- inhibit cabin pressurisation if any external door is not closed, latched and locked.

Functions

The behaviour of the AADL components is specified using Simulink/Stateflow [55]. The latter is based on data-flow models and state machines, which are common models of computation adopted in the system design of avionic or automotive applications. 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 [56] is a framework for code generation from a safe subset of Simulink and Stateflow models for safety critical embedded systems. This safe subset was adopted in our work and we used GeneAuto to filter the original Simulink models of the SDSCS into a safe Gene-Auto model that would be easy to interpret in the data-flow model language Signal.

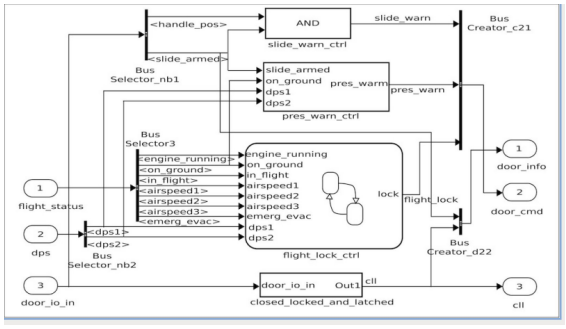

Figure 22: A Simulink door handler in the SDSCS model

The behavioural aspects of SDSCS are modelled in Simulink and Stateflow (Figure 22). Sensors, such as *flight_status*, *dps*, and *door_io_in*, are connected to four Simulink blocks, each of which implements an SDSCS task. Three blocks, *slide_warn_ctrl*, *pres_warn_ctrl*, and *closed_locked_and_latched*, are associated with simple logic to determine actuators status from sensors readings. The fourth block, *flight_lock_ctrl*, is associated with a state machine (specified in Stateflow), which decides the status of flight lock actuators. Each Simulink block is associated with a specific activation clock. At each tick of this clock, the block produces new outputs from available inputs. A global activation clock synchronises that of Simulink blocks. From this point of view, our Simulink model is synchronous, and each of its blocks is thus modelled as a synchronous Signal process after pre-processing using GeneAuto.

Integration

The integration of the SDSCS system, specified in the AADL, with its functional specifications, implemented using Simulink, requires further the specification of a few more components in order to perform simulation, as well as analysis and synthesis from the original models to, e.g., generate a scheduler or allocate the generated code onto the simulated architecture, as well as a model of the system's environment.

The synthesis of an AADL thread-level scheduler is required to integrate threads onto virtual processors. It must be generated from the timing properties specified with the AADL thread models. The allocation of threads onto virtual processors is specified in the AADL model. It is translated into Signal compilation directives (called pragmas) to perform distributed code generation from allocation directives so as to synthesise one executable program for each (virtual) processing unit. Finally, sensors and actuators interface the SDSCS with its environment. Therefore, a hand-written environment model is responsible for providing sensor readings to the SDSCS according to (an abstraction of) the aircraft status.

Once these models are obtained, they can be composed with those generated from the AADL and Simulink models in order to build a simulator. In the process, the clock calculus of the Polychrony toolset is of great help to synthesise the hierarchical control over the activation condition and dispatch of all functional blocks that are composed, and in a manner most generally transparent to the user. All the models, such as system behaviour, hardware architecture, environment, and schedulers, expressed by Signal processes, are composed together to build the complete system. The integrated system is then used for C or Java code generation via the Signal compiler for simulation purpose.

Simulation

The simplest form of simulation can easily be performed by visualising value changes during the execution of programs via VCD. VCD files are generally generated by EDA (Electronic Design Automation) logic simulation tools, and they adopt an ASCII-based file format, whose simplicity and compact structure allows a wide spread in application simulation. The simple and yet compact structure of the VCD format has allowed its

use to become ubiquitous and to spread into non-Verilog tools such as the VHDL simulator GHDL and various kernel tracers. In our simulation, traces are recorded in VCD format. The VCD files are then used for the visualisation of simulation results through graphical VCD viewers, such as GTKWave.

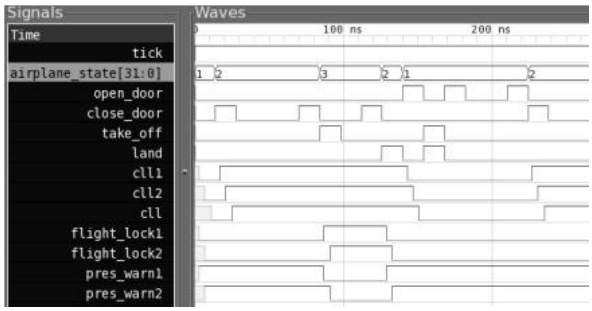

Figure 23: The simulation is illustrated by a VCD viewer: GTKWave

Figure 23 shows the result of a simulation. In this figure, the change of signal values with regard to the fastest clock is shown. Step by step, interactive simulation through the generated VCD interface code is also possible, allowing for the user to interact with the simulator directly and inject particular usage scenarios.

Profiling

In addition to heterogeneous system specification, another advantage of our approach, compared to similar projects, is to benefit from simulation and validation tools associated with Polychrony in the same framework. The Polychrony toolset adopts various analysis and validation techniques: static analysis, simulation, model-checking, and co-simulation or profiling [57].

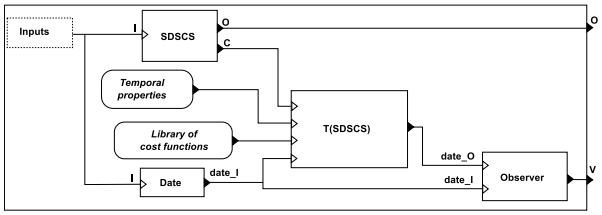

Figure 24: Co-simulation of SDSCS and its temporal behaviour

Figure 24 illustrates the co-simulation of the *SDSCS*. *T(SDSCS)* is the temporal homomorphism of *SDSCS* with regard to specified *Temporal properties* and a parameterisation of *Library of cost functions*. *Date* provides date signals to *T(SDSCS)* according to inputs *I*. The input signals are synchronised 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. Inputs-outputs of *T(SDSCS)* are finally sent to *Observer* in order to obtain the simulation results *V*.

Distributed real-time scheduling

As well as other formats used for model checking and controller synthesis, the Signal code can be transformed into SynDEx code (.sdx file) for real-time scheduler synthesis. From the SynDEx tool, the .sdx file is read, and the "adequation" can be carried out. Figure 25 illustrates the SynDEx algorithm graph describing a partial order on the execution of the functions.

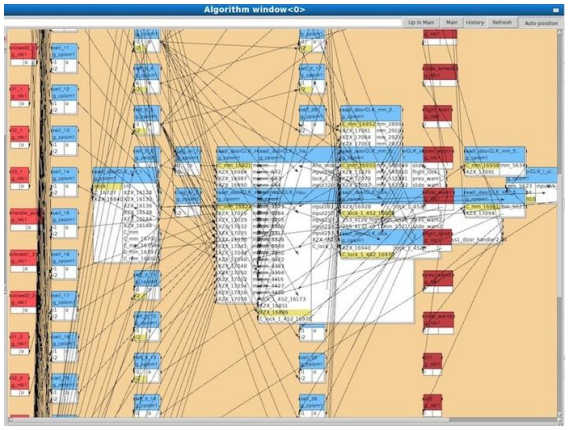

Figure 25: Algorithm graph obtained from the SDSCS

This algorithm is translated from the AADL functional part of SDSCS. Figure 26 shows the SynDEx architecture graph, which is translated from the AADL architectural part of SDSCS. The two processing units (CPIOM1 and CPIOM2), two concentrators (RDC1 and RDC2) and the communication media (AFDX1) can be easily found in the figure.

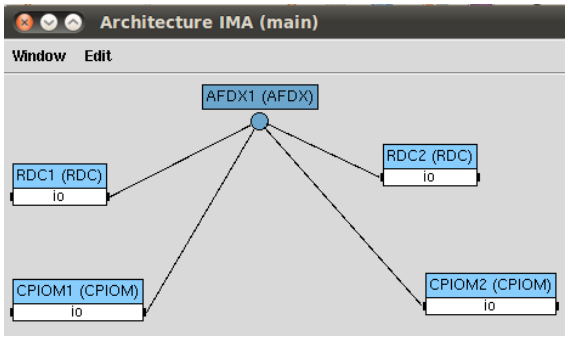

Figure 26: SynDEx architecture graph obtained from the SDSCS

In this case study, the algorithm had more than 150 nodes and the architecture had 5 nodes in SynDEx, so that the adequation algorithm took about 15 minutes 35 seconds in average. With this toolchain, it is easy to change the configuration of the execution platform and

binding. For example, the number of processing units can be changed, the type of processing units and communication media can be changed. The names of AADL components are always kept in all the transformations in order to enable traceability. Hence, our approach provides a fast yet efficient architecture exploration for the design of distributed real-time and embedded systems.

9. Conclusion

In this paper, we have presented the definition, development and case-study validation of a comprehensive framework for the analysis, verification, validation of the AADL and its behavioural annex, allowing for both simulation and real-time code generation. Our framework is based on the synchronous paradigm and the polychronous model of computation and communication and its supportive open-source toolset: Polychrony. A longer-term aim of our work is to equip the AADL standard with a framework allowing for synchronous modelling, verification and synthesis of architecture-focused embedded software.

Based on the formal timing modelling of AADL presented in this paper, logical clock constraints are also considered to be associated with AADL core, behaviour annex, mode, etc. These timing constraints are, for the logical ones, resolved by using the controller synthesis framework of the Polychrony toolset. They are, for the numerical ones, solved by using efficient affine scheduling techniques we have developed.

Our approach, based on controller and scheduler synthesis, very much differs from those based on direct code generation techniques available from synchronous languages as well as interpretation and non-deterministic constraint resolution techniques implemented in CCSL [58], the timing annex of MARTE. We have shown that logical clock constraints could be considered as the control objectives of a system design and they could be enforced by the synthesised controller. We have developed efficient algorithms to abstract and manipulate numerical timing constraints through affine equations to perform thread-level scheduler synthesis. Polychronous controller synthesis further offers the additional advantage to enable the simulation of sporadic or asynchronous clocks.

We believe that the work presented in this paper provides the first comprehensive framework of timed modelling, analysis, verification, and synthesis of AADL concepts, including software, architecture, behaviour annex and mode, yet in a way compatible with related work on the AADL in verification, synthesis, and requirement engineering. It is also a framework for implementation of synthesis of reactive and control systems. Our approach also offers a partial solution to the formal and systematic implementation of Globally Asynchronous Locally Synchronous (GALS) architecture at a high-level of abstraction.

The work reported in this paper has span over several collaborative projects to develop a polychronous framework for the AADL and study its pertinence through several industry-scale case-studies. Is has inspired new research with the definition and implementation of a model of constrained automata, based on the polychronous model of computation and communication. It is being pursued with an ongoing proposal towards the standardization of a synchronous timing annex for the AADL.

[1] SCADE Suite, `http://www.esterel-technologies.com/products/scade-suite`.

[2] Matlab, `http://www.mathworks.com/products/matlab`.

[3] J. Buck, S. Ha, E. A. Lee, D. G. Messerchmitt, Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, in: Readings in hardware/software co-design, Kluwer Academic Publishers, 2002, pp. 527–543.

[4] Rational rhapsody family, `http://www-142.ibm.com/software/products/us/en/ratirhapfami`.

[5] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, O. Laurent, System-level Co-simulation of Integrated Avionics Using Polychrony, in: ACM Symposium on Applied Computing (SAC'11), 2011.

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

[7] 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.

[8] Polychrony on polarsys, `www.polarsys.org/projects/polarsys.pop`.

[9] A. Bouakaz, J.-P. Talpin, Buffer minimization in earliest-deadline first scheduling of data-flow graphs, SIGPLAN Notice 48 (5) (2013) 133–142.

[10] A. Bouakaz, J.-P. Talpin, Design of Safety-Critical Java Level 1 Applications Using Affine Abstract Clocks, in: Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems, 2013, pp. 58–67.

[11] A. Bouakaz, Real-Time Scheduling of Dataflow Graphs, Ph.D. thesis, University of Rennes 1 (2013).

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] O. Gilles, J. Hugues, Expressing and enforcing user-defined constraints of aadl models, in: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS '10, IEEE Computer Society, 2010, pp. 337–342.

[18] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, L. Sha, Compositional verification of architectural models, in: Proceedings of the 4th international conference on NASA Formal Methods, NFM'12, Springer-Verlag, 2012, pp. 126–140.

[19] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, F. Vernadat, Formal Verification of AADL models with Fiacre and Tina, in: ERTSS 2010 - Embedded Real-Time Software and Systems, 2010, pp. 1–9.

[20] G. Lasnier, L. Pautet, J. Hugues, L. Wrage, An Implementation of the Behavior Annex in the AADL toolset OSATE2, in: IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2011.

[21] Z. Yang, K. Hu, J.-P. Bodeveix, L. Pi, D. Ma, J.-P. Talpin, Two formal semantics for a subset of the AADL, in: Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'11), 2011.

[22] F. Mallet, J. Deantoni, C. André, R. De Simone, The Clock Constraint Specification Language for building timed causality models, Innovations in Systems and Software Engineering 6 (1-2) (2010) 99–106.

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

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

[25] The polychrony/sme toolset, `http://www.irisa.fr/espresso/Polychrony`.

[26] Polarsys - open source tools for the development of embedded systems, `http://polarsys.org`.

[27] P. Le Guernic, J.-P. Talpin, J.-C. L. Lann, Polychrony for system design, JOURNAL FOR CIRCUITS, SYSTEMS AND COMPUTERS 12 (2002) 261–304.

[28] S. Abramsky, A. Jung, Domain theory, in: S. Abramsky, D. M. Gabbay, T. S. E. Maibaum (Eds.), Handbook of Logic in Computer Science, Vol. 3, Clarendon Press, 1994, pp. 1–168.

[29] G. Kahn, The semantics of simple language for parallel programming, in: IFIP Congress, 1974, pp. 471–475.

[30] P. Knijnenburg, Algebraic domains, chain completion and the plotkin powerdomain construction, Tech. Rep. RUU-CS-93-03, Utrecht University (1993).

[31] D. Potop-Butucaru, S. Edwards, G. Berry, Compiling Esterel, Springer, 2007.

[32] D. Kozen, A completeness theorem for kleene algebras and the algebra of regular events, in: Logic in Computer Science, 1991, pp. 214–225.

[33] Ieee standard for property specification language (psl), IEEE Std 1850-2005 (2005) 1–143.

[34] W. Gelade, M. Gyssens, W. Martens, Regular expressions with counting: Weak versus strong determinism, SIAM Journal of Computing 41 (1) (2012) 160–190.

[35] Y. Ma, J.-P. Talpin, S. Shukla, T. Gautier, "distributed simulation of aadl specifications in a polychronous model of computation", in: IEEE International Conference on Embedded Software and Systems (ICESS'09), 2009.

[36] The message-passing interface (mpi) standard, `https://www.mcs.anl.gov/research/projects/mpi`.

[37] Syndex: System-level cad software for distributed real-time systems, `http://syndex.org`.

[38] A. Bouakaz, J.-P. Talpin, J. Vitek, Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications, in: Proceedings of the 12th International Conference on Application of Concurrency to System Design, IEEE Computer Society, Washington, DC, USA, 2012.

[39] H. Chetto, M. Silly, T. Bouchentouf, Dynamic scheduling of Real-Time Tasks under Precedence Constraints, Real-Time Syst. 2 (3) (1990) 181–194.

[40] J. Blazewicz, Scheduling Dependant Tasks with Different Arrival Times to Meet Deadlines, Gelende H. Beilner (eds), Modeling and Performance of Computer Systems.

[41] J. Forget, F. Boniol, E. Grolleau, D. Lesens, C. Pagetti, Scheduling Dependent Periodic Tasks without Synchronization Mechanisms, in: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, 2010, pp. 301–310.

[42] I. M. Smarandache, P. Le Guernic, Affine Transformations in Signal and Their Application in the Specification and Validation of Real-Time Systems, in: Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software: Transformation-Based Reactive Systems Development, ARTS'97, Springer-Verlag, London, UK, 1997, pp. 233–247.

[43] E. A. Lee, D. G. Messerchmitt, Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing, IEEE Trans. Comput. 36 (1987) 24–35.

[44] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete, Cycle-Static Dataflow, IEEE Transactions on Signal Processing 44 (1996) 397–408.

[45] J. Y. T. Leung, J. Whitehead, On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks, Performance Evaluation 2 (4) (1982) 237–250.

[46] R. I. Davis, A. Burns, A Survey of Hard Real-Time Scheduling for Multiprocessor Systems, ACM Comput. Surv. 43 (4) (2011) 35:1–35:44.

[47] N. C. Audsley, A. Burns, M. F. Richardson, A. J. Wellings, Hard Real-Time Scheduling: the Deadline-Monotonic Approach, in: Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software, 1991, pp. 133–137.

[48] I. Ripoll, A. Crespo, A. K. Mok, Improvement in Feasibility Testing for Real-Time Tasks, Real-Time Syst. 11 (1) (1996) 19–39.

[49] F. Zhang, A. Burns, Schedulability Analysis for Real-Time Systems with EDF Scheduling, IEEE Transactions on Computers 58 (2009) 1250–1258.

[50] The topcased project, `http://www.topcased.org`.

[51] The openembedd project, `http://openembedd.org`.

[52] The artemisia project cesar, `http://www.cesarproject.eu`.

[53] The itea2 project opees, `http://www.opees.org`.

[54] 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.

[55] Simulink, `http://www.mathworks.com/products/simulink`.

[56] 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.

[57] 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.

[58] Timesquare, t^2 in a nutshell, `http://timesquare.inria.fr`.