Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL)

Loïc Besnard, Etienne Borde, Pierre Dissaux, Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin

Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL)

Loïc Besnard, Etienne Borde, Pierre Dissaux, Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin (corresponding author)

TECHNICAL REPORT

N° 446

01/03/2014

Project-Team TEA

ISSN 0249-6399

Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL)

Loïc Besnard[1], Etienne Borde[2], Pierre Dissaux[3], Thierry Gautier[4], Paul Le Guernic[4], Jean-Pierre Talpin[45]
Project-Team TEA

Technical Report N° 446 — Avril 2014 —21 pages.

Abstract: The aim of this document is to provide an analysis of the SAE standard AADL (AS5506) and submit recommendations to equip the standard with a synchronous model of computation and communication (MoCC). Our goal is to provide a framework that best fits the semantic and expressive capability of the AADL, and is designed in a way that requires as few conceptual, semantic, or syntactic extensions as possible, on either the standard or its existing annexes.
Our approach consists of the definition of an algebraic framework in which time is formally defined from implicit or specified AADL concepts, such as events. Starting from these concepts, that constitute the synchronous core of the AADL, we define a formal design methodology to use the AADL in a way that supports formal analysis, verification and synthesis of timed properties.
By putting forward synchrony and timing, we intend to define time starting from software and hardware events that incur synchronisation in an architecture specification. Synchronisation indeed is the fundamental artefact from which time can be sensed, in either software or hardware.
Synchrony relates to that fundamental concept as a model of computation and communication, applicable to both software or hardware design. It puts emphasis on logical time, abstracted through synchronisation points, in order to break down computations into zero-time reactions and regard communications as instantaneous.
While abstracting real time, synchronous logical time provides an algebraic framework in which both event-driven and time-triggered execution policies can be specified. Bridging the gap between system-level, logical, synchronous specifications and time-triggered, distributed, and dynamically scheduled real-time applications necessitates a refinement-based design methodology, which we additionally intend to outline, to support the applicability of the proposed concepts in system design.
To support the formal presentation of our MoCC, we define a algebra of automata consisting of transition systems and logical timing constraints. We consider the behaviour annex (BA) as the mean to implement this model, together with the constraint annex (CA), as a mean to represent abstractions of behaviour annexes using clock constraints and regular expressions.

Key-words: embedded system design, formal semantics and analysis, synchronous programming, SAE standard AADL

[1] CNRS

[2] Telecom ParisTech

[3] ELLIDISS

[4] INRIA

[5] Corresponding author jean-pierre.talpin@inria.fr

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Contents

1 Introduction **3**

2 Related Work **4**

3 A framework of constrained automata **6**

4 Synchronous specifications in the AADL **7**

5 A synchronous extension of the behavioural annex **8**

6 Constraints as abstracted behaviours **12**

7 Regular constraints **14**

8 Processes as abstract threads **16**

9 Conclusion **18**

A A framework of constrained automata **21**

1 Introduction

The purpose of this document is to provide an analysis of the SAE standard AADL (AS5506) and submit recommendations to equip it with a synchronous model of computation and communication (MoCC). Our goal is to provide a framework that best fits the semantic and expressive capability of the AADL, and is designed in a way that requires as few conceptual, semantic, or syntactic extensions as possible, on either the standard or its existing annexes.

Our approach consists of the definition of an algebraic framework in which time is formally defined from implicit or specified AADL concepts, such as events. Starting from these concepts, that constitute the synchronous core of the AADL, we define a formal design methodology to use the AADL in a way that supports formal analysis, verification and synthesis of timed properties.

By putting forward synchrony and timing, we intend to define time starting from software and hardware events that incur synchronisation in an architecture specification. Synchronisation indeed is the fundamental artefact from which time can be sensed, in either software or hardware.

Synchrony relates to that fundamental concept as a model of computation and communication, applicable to both software or hardware design. It puts emphasis on logical time, abstracted through synchronisation points, in order to break down computations into zero-time reactions and regard communications as instantaneous.

While abstracting real time, synchronous logical time provides an algebraic framework in which both event-driven and time-triggered execution policies can be specified. Bridging the gap between system-level, logical, synchronous specifications and time-triggered, distributed, and dynamically scheduled real-time applications necessitates a refinement-based design methodology, which we additionally intend to outline, to support the applicability of the proposed concepts in system design.

To support the formal presentation of our MoCC, we define a algebra of automata consisting of transition systems and logical timing constraints. We consider the behaviour annex (BA) as the mean to implement this model, together with the constraint annex (CA), as a mean to represent abstractions of behaviour annexes using clock constraints and regular expressions.

Outline

The next section, Section 2, presents the related work. Section 3 gives a brief outline of the model of computation and communication (MoCC) under consideration whose formal definition is given in Appendix A. Our analysis starts Section 4 by the identification of the core AADL artefacts from which time can be sensed and on which our model will operate. *Compound* (synchronous) events are defined from core thread and ports events and property fields.

Based on that notion, Section 5, recommends the minimal updates to infuse the AADL behaviour annex with a synchronous semantics. In the aim of equipping this MoCC with refinement-based design capabilities, Section 6 defines abstraction of behaviour annexes automata in terms of clock constraints. Section A extends them to regular expressions, in the spirit of the constraint annex, and Section 8 outlines its use within processes seen as abstract threads.

The present recommendations aim at exploiting all existing concepts of the AADL core, its behaviour annex, and its forthcoming constraint annex, in order to express a synchronous model of computations and communications. Its implementation reduces to the update of a limited number of concepts in the behavioural annex and provides a synchronous design methodology for the AADL. Tooling this methodology requires using a pivot model of computations and communications, namely the framework of constrained automata, which we formally define in Appendix A. Section 9 concludes

our presentation by offering perspectives.

2 Related Work

Many related works have contributed to the formal specification, analysis and verification of AADL models and its annexes, hence implicitly or explicitly proposing a formal semantics of the AADL in the model of computation and communication of the verification framework considered.

The analysis language REAL [2] allows to define structural properties on AADL models that are checked inductively visiting the object of a model under verification. [4] presents an extension of this language called LUTE which further uses PSL (Property Specification Language) to check behavioural properties of models as well as a contract framework called AGREE for assume-guarantee reasoning between composed AADL model elements.

The COMPASS project has also proposed a framework for formal verification and validation of AADL models and its error annex [3]. 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 [5] uses executable specifications and the TINA model checker to check structural and behavioural properties of AADL models.

RAMSES, on the other hand [6], presents the implementation of the AADL behaviour annex. The behaviour annex supports the specification of automata and sequences of actions to model the behaviour of AADL programs and threads. Its implementation OSATE 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 to generate C code for OSs complying the ARINC-653 standard.

Synchronous modelling is central in [7], 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 (as well as LTL model-checking). It is implemented by the AADL2 MAUDE using OSATE.

Similarly, Yang et al.[8] 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 semantic 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 computations and communications: 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, the clock constraint specification language of the UML profile MARTE [10], 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 semantic and possible variations.

CCSL actually provides a clock calculus of greater expressivity than polychrony, allowing for the expression of unbounded, asynchronous, causal properties between clocks (e.g. inf and sup).

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

Our previous work demonstrated that the all concepts and artefact of the AADL core could, as specified in its normative documents, be given an interpretation in the polychronous model of computation and communication [11, 12, 13, 14, 15], by mean of its import and simulation in the Eclipse project POP's toolset[1].

Why synchrony and not just reactive programming ?

The synchronous hypothesis is based on a very simple pragmatical and realistic principle: if the actual duration required to process an atomic action A at time t is δ_A, and if the result must be available in a delay Δ_A, then one can consider that, instead of being active during δ_A units of time and sleeping during $(\Delta_A - \delta_A)$ units, the actor (to avoid confusion with AADL process we improperly use actor instead of process/thread,...) is active during 0 unit of time and sleeping during (Δ_A) units.

The key concepts for this level of the design are partial order of (data) events, and equivalence relation over events (logical synchronisation). One shall however refrain from further simplifying the synchronous hypothesis by, e.g., considering delayed communications or computations, strictly periodic reactions, etc.

Delayed communications can introduce unsuitable variations caused by the influence of architectural choices on the algorithm. For instance, if a function F is computed by the composition $x = f(a)$, $y = g(x)$ and $b = h(y)$ then, depending on the mapping of functions f, g and h in one, two or three threads, one may get either $b_t = h(g(f(a_t)))$ or $b_t = h(g(f(a_{t-1})))$ or $b_t = h(g(f(a_{t-2})))$, for t the index of signals a and b.

When designing or verifying the behaviour of a specific component, a modular approach consists in viewing the other components of the architecture as a part of the environment. This abstraction principle makes the design modular and compositional allowing to consider the environment as a standard system (or set of systems). The counter-part is that non-deterministic specifications must be possible, which is in fact compatible with synchronous approach. Non-determinism is not suitable in an embedded system but it is necessary to its refinement-based or component-based design, which starts from system abstractions that are partially defined by the composition of elementary blocks and an abstraction of the system's environment.

If logical delay must be specified, e.g., to avoid a causal loop between two communicating threads, one can simply add a one place FIFO (a pre in Lustre). This FIFO can itself be considered as a specific actor, like a connector in a coordination language. One first advantage of this approach is to provide a uniform vision of communication between actors (including these connectors) in which communication takes 0 time.

More generally, compositional design dictates to start an architecture-focused system design from the composition of a set of components, each with their own clocks, the total of distinct clocks, e.g. harmonics, being fixed later in the design. Hence the necessity to initially cope with possible non-determinism and the benefits of a multi-rate model of computations and communications: a poly-

[1] Polarsys Industry Working Group, Eclipse project POP
`http://www.polarsys.org/projects/polarsys.pop`

chronous model, which allows to cope with it and provide model refinement techniques to reach the goal of a globally deterministic design.

3 A framework of constrained automata

To support the formal presentation of our proposed extensions to the AADL, we define 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 and Signal synchronous programming languages.

The fundamental difference between synchronous automata and asynchronous automata is that, in a synchronous automaton, transitions can be triggered by guards defined by a conjunction of events. Such a conjunction of occurrences of events a and b is written $a\widehat{*}b$. 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.

Listing 1 depicts a constrained automaton manipulating two events a and b. The automaton specifies the alternation of two input event streams a and b. Its reactive behaviour, depicted by the 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 specify the case of simultaneous events a, b in s_1 or s_2. This is done by superimposing it with the requirement or constraint that a and b should never occur simultaneously. With that constraint in place, the automaton behaves as a constrained asynchronous one (event interleaving).

```
thread alternate
   features
      a,b:  in event port;
   constraints
      never a and b;
   end alternate;

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

Listing 1: A controlled automaton in the AADL behavioural annex

The combination of a synchronous automaton and of a temporal constraint yields the hybrid structure of timed automaton depicted Listing 1. It supports an algebraic definition, presented in

Appendix A. This definition relies on the model of computation and communication (MoCC) of Polychrony in order to define a framework of constrained automata capable of expressing both the behavioural and constraint annexes of the AADL.

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 automaton, but also enforces the expressed constraint formula.

In addition, and as we shall see, these constraints 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", etc. Our plan is to use the behavioural and constraint annex of the AADL much in the flavour of the program depicted in Listing 1 to separately specify explicit reactive behaviour using automata and refine these specifications (using, e.g., controller synthesis) to enforce satisfaction of implicit timing constraints and temporal requirements.

4 Synchronous specifications in the AADL

The behavioural annex of the AADL defines all the needed artefacts to define logically timed synchronous constraints. Therefore, our recommendation for the AADL rests on the behaviour annex AS5506/2 as a foundation [1]. It can be defined as an extension of the behaviour annex or, alternatively, as an update or "erratum" to it.

Our approach is to regard the behaviour annex as the semantic core of the AADL, define synchronous specifications inherited from (untouched) behavioural specifications, and perform the smallest upgrade possible to manifest synchrony in it. The behaviour annex defines a transition system (an extended automaton) by three sections:

- variables declarations;
- states declarations;
- transitions declarations;

States

The states of a transition system (transition system is written STS for short) can be a qualified *initial state*, a qualified *complete state*, that represents temporary suspension of execution and resumption based on external trigger conditions, an unqualified *execution state*, that represents intermediate computation state, or a qualified *final state*. The transitions that have an intermediate *execution state* as source state can be interpreted as immediate transitions.

The STS of a thread or a device (D.2 par. 2) has one initial state, one or more final state; it can have complete and execution states. The underlining principle is that all threads are finite. A synchronous interpretation of STSs raises two questions:

- the time lapse of an execution condition catching a previously raised timeout (D3 par. 18)
- the transition from an execution state to another that can send value to, e.g., a port (D3 par. 20)

The STS of a subprogram has one initial state and one final state; it can have execution states. The STS of another component has one initial state, one or more complete states and one final state. As for threads, an embedded system is usually assumed not to terminate.

Transitions

Transitions are made of two parts: a state transition condition and an action. The state transition conditions fall into two categories (D.2 par. 4-8)

- an *execution condition* models a behaviour on input values from ports, shared data, parameters, and behaviour variable values
- a *dispatch condition* affects the execution of a thread on external triggers. Those include:
  - subprogram call to the STS of a subprogram
  - the arrival of events and event data on ports of a non periodic thread to the STS of the thread and the hybrid state automaton defined in the AADL core standard [AS5506A 5.4.1]
  - the transmission request on an outgoing port to the STS of a virtual bus or bus
  - time out

  One, several, or all dequeued elements are made available to the current action of the behaviour_Specification (D.2 par. 7.9).

Roadmap

Accordingly, Section 5 uses the behaviour annex to define the notions of *compound* events (combinations of events) and *actions* (operations performed during a transition). To avoid any confusion with the term "event", used in the AADL core and the behaviour annex, we will use the term *compound* to denote a compound or generalised dispatch condition provided by a core AADL specification or a behaviour annex.

To render the refinement-based design methodology underlying the design of the AADL, Section 6 associates synchronous behavioural annexes with the definition of logically timed constraints to abstract them. Taking advantage of previous work and the ongoing definition of the constraint annex, we choose to use regular expression to represent abstractions of synchronous behaviour annexes. Regular expressions over compound events will represent requirements or invariant to be formally satisfied by behaviour annex, or to be enforced using controller synthesis.

5 A synchronous extension of the behavioural annex

Based on the above analysis, we identify some of the extensions to STSs that would be suited toward synchronous extensions, annotations, annexes of AADL behaviour specifications. We propose to describe an STS (transition system, or extended automaton) using the following three sections:

- variables declarations;
- states declarations;
- transitions declarations;

completed by constraints expressed as regular expressions over compound events to express observers, invariants, or guarded actions.

States

In a synchronous annex, threads should be allowed to run forever. Usually, the system scheduler is such a "thread"; it usually has a final state reached when the whole system is halted, but this very

much differs from the final state of a subprogram. For instance the “Sender behaviour Specification” (D.4, Listing 2), is better interpreted as the description of a session of the actual sender sub-system, which would iterate such sessions.

```
thread implementation sender.v1
   annex behaviour_specification {**
      states
         st: initial complete state;
         sf: complete final state;
         s1, s2: state;
      transitions
         st-[on dispatch timeout]->st {d!(1)};
         st-[on dispatch a]->s1;
         s1-[a=1]->sf;
         s1-[a=0]->st;
         sf-[on dispatch timeout]->st {d!(0)};
         sf-[on dispatch a]->s2;
         s2-[a=0]->st;
         s2-[a=1]->sf;
   **};
end sender.v1;
```

Listing 2: specification of a sender protocol element in the behavioural annex, Section D.4, p26-27

Transitions

The AADL property that“dispatch does not depend on the input value” corresponds the the kind of causal constraint found in synchronous languages like Lustre or Signal in which the availability of a value along a signal depends on the availability/presence of its clock (e.g. $\widehat{x} \rightarrow x$ means that the clock of x precedes the signal x).

By applying the same principle to the AADL (e.g. status of a queue and value, ...), one can unambiguously specifiy the schedules of dispatch and read actions on ports a and timeout, all using the same triggering transitions, as depicted in Listing 3.

With this extension, and provided a simple causal analysis to reconstruct a graph of causal relations between triggers and values, the explicit specification of numerous intermediate transitions can be avoided, as well as some of the guarding conditions.

We note `a?(1)` for reading the value available along a's port queue and testing it equal to 1 (operationally i.e. `a?(v) and v=1)`). It is not clear whether $a = 1$ is well-typed and operationally equivalent to that.

```
thread implementation sender.v2
   annex behaviour_specification {**
      states
         st: initial complete state;
         sf: complete final state;
      transitions
         st-[on dispatch timeout]->st {d!(1)};
         st-[on dispatch a and a?(1)]->sf;
         st-[on dispatch a and a?(0)]->st;
         sf-[on dispatch timeout]->st {d!(0)};
         sf-[on dispatch a and a?(0)]->st;
```

```
12          sf-[on dispatch a and a?(1)]->sf;
13    **};
14 end sender.v2;
```

Listing 3: Sender with generalised triggering expressions

Compounds events

A *compound* is an aggregated evaluation condition or trigger, as depicted in Listing 4.

A transition can also be labelled by a *transition* label which is an implicitly declared and valued event or *compound*. That compound occurs iff the transition is selected. It can alternatively be defined by an event that occurs in that state on the triggering condition of the transition.

Finally, the AADL conjunction "and" is extended to triggers and execution condition. Since the use of priority of the AADL allows to distinguish between "a and b" present and "a but not b" present, we therefore introduce the"andnot" operator in execution conditions.

As an example, Listing 4 outlines a variant of the sender that ensures deterministic behaviour of the transition system with priority given to timeout port dispatch.

```
1  thread implementation sender.v3
2     annex behaviour_specification {**
3        states
4           st: initial complete state;
5           sf: complete final state;
6        transitions
7           st-[on dispatch timeout andnot a]->st {d!(1)};
8           st-[on dispatch a and a?(1)]->sf;
9           st-[on dispatch a and a?(0)]->st;
10          sf-[on dispatch timeout andnot a]->st {d!(0)};
11          sf-[on dispatch a and a?(0)]->st;
12          sf-[on dispatch a and a?(1)]->sf;
13    **};
14 end sender.v3;
```

Listing 4: Sender with andnot expressions

This specification also disambiguates the case when a is dispatched "at the same time as" timeout, which is not taken care of in Listing 2, possibly resulting in a non-deterministic choice between `st-[on dispatch timeout]->st` and `st-[on dispatch a]->s1`. The opposite priority could be given to timeout by imposing `on dispatch a andnot timeout` in the other transitions, Listing 5.

```
1  thread implementation sender.v4
2     annex behaviour_specification {**
3        states
4           st: initial complete state;
5           sf: complete final state;
6        transitions
7           st-[on dispatch timeout]->st {d!(1)};
8           st-[on dispatch a and a?(1) andnot timeout]->sf;
9           st-[on dispatch a and a?(0) andnot timeout]->st;
10          sf-[on dispatch timeout]->st {d!(0)};
```

```
11          sf-[on dispatch a and a?(0) andnot timeout]->st;
12          sf-[on dispatch a and a?(1) andnot timeout]->sf;
13    **};
14 end sender.v4;
```

Listing 5: Sender with andnot expressions

Notice that the `andnot` specification applied to the port a or timeout is operationally equivalent to checking a's queue empty with `a'count=0`. Although testing absence is in absolute forbidden in a synchronous framework (it yields non stuttering-equivalent specifications) here it unambiguously refers to the context of the sender's execution clock, Listing 6.

```
1  thread implementation sender.v5
2     annex behaviour_specification {**
3        states
4           st: initial complete state;
5           sf: complete final state;
6        transitions
7           st-[on dispatch timeout and a'count=0]->st {d!(1)};
8           st-[on dispatch a and a?(1)]->sf;
9           st-[on dispatch a and a?(0)]->st;
10          sf-[on dispatch timeout and a'count=0]->st {d!(0)};
11          sf-[on dispatch a and a?(0)]->st;
12          sf-[on dispatch a and a?(1)]->sf;
13    **};
14 end sender.v5;
```

Listing 6: Sender testing a's queue empty

Listing 6 is a variant of the sender with the addition of conjunction. Similarly dispatch a implements checking a's queue non-empty at the time of the transition. The conjunction implies a causality relation between the dispatch and the read actions. The first transition means that, at the time the automaton executes, if event timeout has arrived along its FIFO queue (it is consumed by the dispatch) and that there is still no value present in the FIFO queue of port a, then value 1 is emitted along port d. Yet, deterministic priority is clearly given to a: if timeout has occurred but a is dispatched as well, then the thread transits into `sf` or `st` depending on the value available at a.

Syntax

A compound generalises the definition of `behavior_condition` defined in page 22 of the behaviour annex (AS5506/2). Hence, it can be empty or consist of the dispatch condition "on e" of some event e or an execute condition (e.g. a timeout). The item `dispatch_trigger` is defined page 25 of AS5506/2.

```
1 compound_condition ::= behavior_condition
2                      | compound_condition or compound_condition
3                      | compound_condition and compound_condition
4                      | compound_condition andnot dispatch_trigger
```

Listing 7: Syntax of compound events

Restrictions and consistency rules

- Behaviour actions `behavior_action` are restricted to computations, to reading the frozen part of ports and to emitting through ports.
- A behaviour action does not consume values, only the dispatch of a guard should. Hence, it may be suitable to specify how many values a dispatch freezes and consumes at a time. By default, one would write `on dispatch a*` to mean all available values.
- One may additionally want to use regular (counting) expressions to specify a particular range of values to freeze, e.g., `a[1,3]` to mean the three first/oldest values, or just a to mean the first value of the queue `a[1]`.

6 Constraints as abstracted behaviours

The purpose of a behavioural annex is to specify the operational function of a system, component, process in an AADL specification.

The purpose of a timing constraint is to abstract this behaviour over the logical time or temporal properties it implies or satisfies (just as a clock abstracts the computation of a signal's value in a synchronous language like Lustre or Signal).

In that context, a compound condition (a `behavior_condition` object) clearly belongs to the operational processing of port queues performed in behaviour annexes. Dispatching, reading, writing, querying a port queue or handling a timeout cannot directly be used as a constraint specification. These operations must be abstracted by the logical and temporal conditions they imply (just as in synchronous languages, reasoning on state transitions and computations of signal values are abstracted by instants).

As a result, a constraint should refer to a port identifier a as the abstraction of a port queue, indistinctive of its direction, input or output. It should refer an equality $a = v$ as the specification of its current value, indistinctive of its direction, sent or received. Conversely, it should refer as `constraint_condition andnot dispatch_trigger` for the absence of value on a given port.

Example

Listing 8 depicts a tentatively abstract specification of the sender protocol. It only checks the relative presence of a and d. This means that the sender must always have a present (it doesn't say if it's read or written) or, exclusively, d present and a absent. Yet, the automaton doesn't say if d can or cannot be read (by an automaton composed with the sender) when a is read. Hence, "a andnot d" might be over specified.

```
1 thread sender.v6
2    constraints
3       always (a andnot  d) or (d andnot  a);
4 end sender.v6;
```

Listing 8: Most abstract specification of the sender protocol

Listing 9 is a tentative refinement of the above abstraction. It uses two specific notations. The equation $a = 0$ means that the present value of a is equal to 0. The term `prev(a=0)` refers to the value of its sub-expression $a = 0$ from the very last time it was evaluated. Hence, it means that, if the

sender was previously in a state were a was evaluated to 0 then, either a should now be present or d set to the value 1 if it's not.

```
thread sender.v7
   constraints
      always prev(a=0) and (a or (d=1 andnot  a));
      always prev(a=1) and (a or (d=0 andnot  a));
end sender.v7;
```

Listing 9: A refinement of the sender constraints

Syntax

A `constraint_condition`, Listing 10, consists of a Boolean expression built from dispatch triggers `dispatch_trigger` to mean ports with dispatched values and refer to `constraint_condition andnot dispatch_trigger` for the absence thereof. It can be build from conjunction, disjunction, past and future tense of constraints. The `property_expression` refers to an AADL value expression. The term `always constraint_condition` means that `constraint_condition` should hold at all times (i.e. every time it is evaluated). The constraint `never constraint_condition` means that `constraint_condition` must be false any time it is evaluated.

```
constraint_condition ::=  dispatch_trigger [= property_expression]
                       | constraint_condition or constraint_condition
                       | constraint_condition and constraint_condition
                       | constraint_condition andnot dispatch_trigger
                       | prev constraint_condition
                       | next constraint_condition

behavior_constraints ::= always constraint_condition
                         | never constraint_condition
```

Listing 10: Syntax of constraint conditions

Restrictions and consistency rules

Notice that a constraint such as `never a and b` is a partial specification (i.e. a non-executable property). It is important, however, to possibly provide for an exception if the constraint is not satisfied at runtime. There are several ways to implement that:

- when an unexpected event occurs, it is placed in a fifo and will be taken into account at the next activation step of the handler thread (this semantics conforms to a data-flow synchronous semantic but not that of AADL).
- the unexpected event is ignored and lost (this corresponds to a broadcast-synchronous semantic)
- an error is implicitly raised
- an error is explicitly raised and handled in the automaton

Possible variants

The examples of Figures 8 and 9 define the constraint section as part of the AADL core specification of a thread (resp. process), as part, for instance, of its forthcoming v2.2 revision. It is worthwhile to consider possible variants:

- the integration of behaviour constraints in the forthcoming constraint annex of the AADL

```
thread sender.v6b
   annex constraint_specification {**
   constraints
      always (a andnot  d) or (d andnot  a);
   **};
end sender.v6b;
```

- the encapsulation of constraint sections into behaviour annexes

```
thread sender.v6c
   annex behaviour_specification {**
   constraints
      always (a andnot  d) or (d andnot  a);
   **};
end sender.v6c;
```

- the extension of the behaviour annex with ad-hoc, abstract, behaviour annexes

7 Regular constraints

Instead of using past or future tense, or a transition system, one may use a regular expression on constraint conditions to specify temporal properties. The use of regular expressions to specify behaviour abstraction is very common in system design with the IEEE standard PSL [24]. It is also very much reminiscent of the COSY specification language and of path expressions [26, 25].

Example

Listing 11 is a tentative refinement of the sender's constraints using regular expressions. From a equal to 0, the sender should either accept a or d equal to 1 when a is absent.

```
thread sender.v9
   constraints
      always {a=0; a or (d=1 andnot a)};
      always {a=1; a or (d=0 andnot a)};
end sender.v9;
```

Listing 11: Sender abstraction using regular expressions

Counting

Just as in PSL, counting expression may additionally be useful to relate events with time units and periodic behaviours. An example of the behaviour annex in which this may prove so is that of the

client protocol page 40 of AS5506/2, Listing 12.

```
thread a_client
   features
      pre : requires subprogram access long_computation;
      post : requires subprogram access send_result; properties
      Dispatch_Protocol => Periodic;
      Period => 200ms;
   annex behavior_specification {**
      variables x : result_type;
      states s : initial complete final state;
      transitions
         s -[ on dispatch ]-> s { pre!; computation(60ms); post!(x) };
   **};
end a_client;
```

Listing 12: Client-server protocol (AS5506/2, page 40)

Instead of giving an operational specification to the expected 60ms duration of the abstracted computational task, one may instead favor further abstracting it (from port dispatch and directions) and refer to $ms[60]$ as the count of milliseconds between pre and post processing, Listing 13. Additionally, one may specify that `pre` always occurs every $200ms$ to match the thread period feature.

```
thread a_client
   features
      pre : requires subprogram access long_computation;
      post : requires subprogram access send_result; properties
      Dispatch_Protocol => Periodic;
      Period => 200ms;
   constraint
      always {pre; ms[60]; post}
      always ms[200] and pre
end a_client;
```

Listing 13: A multi-clocked automaton with constraints

If several regular expressions are present in the constraint section of an annex, then the associated semantics should be the synchronous product of those regular expressions. Similarly, if several annexes are present in a component specification, their associated semantics should be the synchronous product of the declared behaviours. However, more compositions operators may be considered if necessary.

Example

The same principle can be applied to the sender protocol specification by, e.g., setting the timeout to $10ms$. While we may or may not constraint a thread's `timeout` signal directly, it might be possible to specify the minimum amount of time before port d is triggered as a result of the timeout dispatch, Listing 14. Terms a and d denote the "clock" of ports, the periods or instants at which they are dispatched or checked empty. The only departure from PSL is that absolute negation $!a$ (which in PSL is relative to the formula's evaluation "clock") is replaced by the relational "d andnot a" to check a absent in the context of d. The constraint means that it is always the case that either a occurs and not

d or that, after 10 milliseconds d occurs and not a.

```
thread sender.v11
      constraints
         always {ms[10]; d andnot  a} or (a andnot  d)
end sender.v11;
```

Listing 14: Timed abstraction of the sender protocol

A refinement of this clocked specification with a real-time constraint is defined in Listing 15. The timeout should trigger the emission of d when a is absent. But, since it is an implementation port object, we cannot mention it in a process or thread abstraction. So, we decompose the specification into two parts. One part consists of the implicit state transitions from a equal to 0 or 1 to either another a or d (equal to 1 or 0). The second part is the trigger of d, or the conditions for accepting d. Namely that 10 milliseconds have elapsed (relative to the context in which the constraint is checked) and (then) no a is present. A consequence of that is to define a clock sub-domain of 10ms within the clock that evaluates the process/thread/automaton transition.

```
thread sender.v12
   constraints
      always {a=0; a or d=1};
      always {a=1; a or d=0};
      always d and (ms[10] andnot a)
end sender.v12;
```

Listing 15: Real-time abstraction of the sender protocol

Syntax

The syntax of regular expression is listed in Listing 16. It corresponds to a Kleene algebra on compound conditions, in the spirit of the PSL specification language. We note `regexp?` for option, `regexp[n]` for counting, `regexp ; regexp` for concatenation or sequence, `regexp + regexp` for sum or choice, and `regexp*` for star or loop. The keyword `always` is equivalent to the star.

```
regexp ::= compound_condition
       | {regexp ; regexp}
       | regexp + regexp
       | regexp*
       | regexp?
       | regexp[n]
       | regexp : regexp
       | regexp || regexp
```

Listing 16: Syntax of regular expressions

8 Processes as abstract threads

The formal definition of synchronous automata and constraints expressed as regular expressions allow us to define a refinement-based design methodology from abstract components and processes specifi-

cations with properties and constraints (to explicit requirements) down to their implementations using systems and threads. We can exemplify this methodology by considering our running example of the sender, Listing 17. Its abstract specification as a process may consist of anything but concepts linked to its implementation such as timeouts or ports (hence dispatch). As a result, we can only say that the thread should alternate between reading a or emitting d when no a is available.

```
process sender.v13
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      always a or (d andnot a);
end sender.v13;
```

Listing 17: Abstract specification of the Sender

Listing 18 defines a possible state-full refinement of the above state-less property of the sender process. It says that in fact d will send 1 when the last a was 0, and conversely, 0 when 1.

```
process sender.v14
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      always (a=0);(a or (d=1 andnot a));
      always (a=1);(a or (d=0 andnot a));
end sender.v14;
```

Listing 18: Specification refinement for the Sender

The specification says to send d when a is empty once the process (or thread) is executed. Now, if we allow to predefined (hardware) events, such as ms to tick every milliseconds, then we can define this condition in a way lot closer to Listing 20.

```
process sender.v15
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      always (a=0);(a or d=1);
      always (a=1);(a or d=0);
      always d and (ms[10] andnot a)
end sender.v15;
```

Listing 19: Specification refinement for the Sender

Inheritance

A necessary complement to the above is to establish a mechanism to check the conformance of a thread implementation (an automaton) with respect to the constraints specified in its abstraction (thread features, process). One possible way of doing that is by using some explicit inheritance mechanism, to

"type" an automaton by its constraints, as in Listing 20.

```
thread implementation sender.v16
   inherits sender.v15;
```

Listing 20: Explicit refinement relation between the specification and implementation of the sender

Contracts

Another suitable extension is to make the refinement relation algebraically richer by introducing assume-guarantee reasoning on constraints by means of contracts. Timing contracts aim at the specification of constraints (guarantees) of scope restricted to a context (assumptions) and offer a compositional algebraic mean to compose and enforce them.

Listing 21 is a possible contract of the sender protocol. It makes each of the three transition scenarios explicit. Two define the next value of d depending on the current value of a. The last one says that d is triggered when no a occurs within 10ms.

```
process sender.v17
   constraints
      assume a=0 guarantee next (a or (d=1 andnot a));
      assume a=1 guarantee next (a or (d=0 andnot a));
      assume ms[10] andnot a guarantee d
end sender.v17;
```

Listing 21: Contract of the sender protocol

Restrictions and consistency rules

- A process may have a constraint section
- A process should only declare input-output connections, and not ports

9 Conclusion

By exploiting all existing concepts of the core AADL and behavioural annex, the present recommendations reduce to the specification of a synchronous, refinement-based, design methodology for the AADL. They relies on a model of computation and communication (MoCC) of Polychrony, presented in Appendix A, presented as a model of constrained automata, currently under implementation.

The outline of a complete design flow, outlined in section 8, would start from the elicitation of requirement specifications using implicit constraints expressed as regular expressions, the explicit specification of core reactive behaviours using BA, behavioural refinement using, e.g., a clock calculus or controller synthesis from the specified constraints and, finally, conformance checking between the specified requirements and behaviours and the synthesised models or programs.

Acknowledgments

We wish to thank all the members of the AADL committee for valuable discussions during previous meetings.

References

[1] ”SAE Architecture Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behavior Model Annex”. Report AS5506/2. SAE Aerospace, 2011.

[2] Expressing and enforcing user-defined constraints of AADL models. Olivier GILLES, Jerome HUGUES. IEEE ICECCS, 2010.

[3] Formal Verification and Validation of AADL Models. M.Bozzano, R.Cavada, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, X. Olive. ERTS, 2010.

[4] ”Compositional Verification of Architectural Models”. Darren Cofer, Andrew Gacek, Steven Miller, Michael Whalen. Springer NFM, 2012.

[5] ”Formal verification of AADL models with Fiacre and Tina”. B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, F. Vernadat. ERTS, 2010.

[6] ”Design Patterns for Rule-based Refinement of Safety Critical Embedded Systems Models”. Fabien Cadoret, Etienne Borde, Sbastien Gardoll and Laurent Pautet. International Conference on Engineering of Complex Computer Systems (ICECCS’12), Paris (FRANCE), 2012.

[7] Formal Semantics and Analysis of behavioural AADL Models in Real-Time Maude. Peter Csaba, Olveczky, Artur Boronat, Jose Meseguer, and Edgar Pek. LNCS FTDS 2010.

[8] ”Two formal semantics for a subset of the AADL”. Yang, Z., Hu, K., Bodeveix, J.-P., Pi, L., Ma, D.,Talpin, J.-P. UML&AADL workshop at the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’11) . IEEE, 2011.

[9] ”From AADL to timed abstract state machine: a certified model transformation”. Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, J.-P. Talpin. In Journal of System and Software. Elsevier, 2014.

[10] ”The clock constraint specification language for building timed causality models”. Frédéric Mallet, Julien DeAntoni, Charles André, Robert de Simone. Innovations in Systems and Software Engineering, 6(1):99-106. Springer, Mars 2010.

[11] ”Toward polychronous analysis and validation for timed software architectures in AADL” Y. Ma, H.Yu,T. Gautier, L. Besnard, P. Le Guernic, J.-P. Talpin and Maurice Heitz.Design Analysis and Test in Europe (DATE’13). IEEE, April 2013.

[12] ”System synthesis from AADL using Polychrony”. Y. Ma, H. Yu, T. Gautier, J.-P. Talpin, L. Besnard and P. Le Guernic. Electronic System Level Synthesis Conference (ESLSYN’11). IEEE, June 2011.

[13] ”System-level co-simulation of integrated avionics using polychrony”. Yu, H., Ma, Y., Glouche, Y., Talpin, J.-P., Besnard, L., Gautier, T., Le Guernic, P., Toom, A., and Laurent, O. ACM Symposium on Applied Computing (SAC’11). ACM, 2011.

[14] ”Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony”. L. Besnard, A. Bouakaz, T. Gautier, P. Le Guernic, Y. Ma, J.-P. Talpin, H. Yu. In Science of Computer Programming. Elsevier, 2014.

[15] ”Polychronous modeling, analysis, verification and simulation for timed software architectures”. H. Yu, Y. Ma, T. Gautier, L. Besnard, P. Le Guernic, J.-P. Talpin. In Journal of Systems Architecture. Elsevier, 2013.

[16] ”Polychronous controller synthesis from MARTE’s CCSL constraints”. Yu, H., Talpin, J.-P., Besnard, L., Gautier, T., Marchand, H., Le Guernic, P. ACM-IEEE Conference on Methods and Models for Codesign. IEEE, July 2011.

[17] ”Formal verification on compiler transformations on polychronous equations”. V. C. Ngo, J.-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. International Conference on Integrated Formal Methods. Springer, June 2012.

[18] ”Compositional design of isochronous systems” Talpin, J.-P., Ouy, J., Gautier,T., Besnard, L., Le Guernic, P. In Science of Computer Programming. Elsevier, 2011.

[19] ”Affine data-flow graphs for the synthesis of hard real-time applications”. A. Bouakaz, J.-P. Talpin, and J. Vitek. Application of Concurrency to System Design. IEEE Press, June 2012.

[20] ”A completeness theorem for Kleene algebras and the algebra of regular events”. Dexter Kozen. Infor. and Comput., 110(2):366-390, May 1994.

[21] ”Regular algebra and finite machines”. Conway, J.H. Chapman and Hall. ISBN 0- 412-10620-5. 1971.

[22] Two complete axiom systems for the algebra of regular events, Arto Salomaa. J. Assoc. Comput. Mach. 13:1 (January, 1966), 158169.

[23] ”Regular Expressions with Counting: Weak versus Strong Determinism”. Wouter Gelade, Marc Gyssens, and Wim Martens SIAM J. Comput., 41(1), 160190. (31 pages)

[24] ”IEEE Standard for Property Specification Language (PSL)” - 1850-2010. `http://standards.ieee.org/findstds/standard/1850-2010.html`

[25] ”COSY a system specification language based on paths and processes”. Lauer, P.E., Torrigiani, P.R., Shields, M.W. Acta Informatica v. 12(2). Springer, 1979.

[26] ”A Description of Path Expressions by Petri Nets”. Lauer, P. E. and Campbell, R. H. 2nd. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, 1975.

Appendix

A A framework of constrained automata

Boolean control algebra

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 "$\widehat{\ }$" denotes the clock of a variable (e.g. $\widehat{x}$), of a state, of an operator. The term variable is used for signal variable.

definition 1 *A* Boolean Control Algebra *is a Boolean Algebra* $\phi(V,S) = (\mathbf{F}_{V,S}, \widehat{*}, \widehat{+}, \widehat{\neg}_V, \widehat{\mathbf{0}}, \widehat{\mathbf{1}}_V)$ *that satisfies the Boolean Control Algebra properties defined below, where*

- $\widehat{+}, \widehat{*}$ *are notations for meet (infimum) and join (supremum) operations*
- $\widehat{\neg}_V$ *is notations for complement*
- $\widehat{\mathbf{0}}, \widehat{\mathbf{1}}_V$ *are notations for minimum and maximum*
- *the set of formulas* $\mathbf{F}_{V,S}$ *is the smallest set that satisfies*
  - *constants:* $\widehat{\mathbf{0}}, \widehat{\mathbf{1}}_V \in \mathbf{F}_{V,S}$
  - *atoms:* $(\forall x \in V \cup S)(\widehat{x}, [x], [-x] \in \mathbf{F}_{V,S})$
    $[x]$ *and* $[-x]$ *denote the clock of* x *sampled when* x *is true, resp. false*
  - *expressions:* $(\forall f, g \in \mathbf{F}_{V,S})(\widehat{*}fg, \widehat{+}fg, \widehat{\neg}_V f \in \mathbf{F}_{V,S})$
    Parentheses and affix notation are freely used.

Our Boolean control algebra supports the following formal properties.

- *sampling partition*: $(\forall x \in (V \cup S))((\widehat{x} = [x] \mathbin{\widehat{+}} [-x]) \wedge ([x] \mathbin{\widehat{*}} [-x] = \widehat{\mathbf{0}}))$
- *automaton clocking*: $\widehat{\Sigma}_{x \in V}(\widehat{x}) = \widehat{\mathbf{1}}_V$, $(\forall s \in S)(\widehat{s} = \widehat{\mathbf{1}}_V)$
- *state exclusiveness*: $(\forall s_1, s_2 \in S)([s_1] \widehat{*} [s_2] = \widehat{\mathbf{0}}) \vee (s_1 = s_2))$

Constrained automata

definition 2 *A* constrained automaton *A is a tuple* $A = (S_A, s_0, \rightarrow_A, V_A, T_A, \mathbf{C}_A)$ *where*

- S_A *is the non empty set of states and* s_0 *the initial state*
- $\rightarrow_A \subset S_A^2$ *is the transition relation*
- V_A *is the set of signal variables*
- *We denote by* $\mathbf{F}_{A,S}$ *the set of formulas in the Boolean Control Algebra* $\phi(V_A, S_A)$
- $T_A : (\rightarrow_A) \rightarrow F_{A,S}$ *is the function that assigns a formula to a transition. The formula is by definition the* trigger *of the transition. Since when the current state is* s $[s]$ *is true and for any other state* t $[t]$ *is false, we assume that*
  $\forall s, s_1, s_2 \in S_A$, $[s]$ *does not occur in* $T_A(s_1, s_2)$
- $\mathbf{C}_A$ *is the constraint of A.*
  It is a formula in $\mathbf{F}_{A,S}$ *that is (constrained to be) null.*
  - *A formula* f *in* $\mathbf{F}_{A,S}$ *is null in* A *iff* $f \widehat{*} \mathbf{C}_A = f$.
  - *If* $\mathbf{C}_A$ *is* $\widehat{\mathbf{0}}$*, the automaton is said constraint free.*
  - *If* $\mathbf{C}_A$ *is* $\widehat{\mathbf{1}}_{V_A}$ *all formulas in* A *are null.*

A constrained automaton is defined upto isomorphism.

Example

As a result of the above definition, the alternating automaton of Listing 22 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 $C : (a \widehat{*} b) = \widehat{\mathbf{0}}$. Its control clock is $\widehat{\mathbf{1}} = a \widehat{+} b$. In state s_1, the trigger is $T(s_1) = a$, the null clock $C(s_1) = C \widehat{*} \widehat{\mathbf{1}} = C$ so that the automaton can only accept a.

```
thread alternate
   features
      a,b:  in event port;
   constraints
      never a and b;
end alternate;

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

Listing 22: A controlled automaton in the AADL behavioural annex

Notations

- Boolean Control difference: $f \widehat{-} g$ is used to denote $f \widehat{*} (\neg_V g)$
- $\mathbf{1}_A$ denotes the supremum $\mathbf{1}_{V_A}$ of an automaton A, for a state s in A, $\mathbf{C}_A(s) = \mathbf{C}_A \widehat{*} [s]$.
- Labeled transitions are denoted by $h : s_1 \rightarrow_A s_2$ meaning that $(s_1, s_2) \in \rightarrow_A$ and $T_A(s_1, s_2) = h$
- The control clock of an automaton A is $\widehat{\mathbf{1}}_A$
- In $h : s_1 \rightarrow_A s_2$, h is the trigger of (s_1, s_2) and a trigger of s_1
- The trigger of a state s, $\mathbf{T}_A(s)$ is the upper bound of the triggers of s
- The null clock of a state s is $\mathbf{C}_A(s)$.
  It is defined as the simplified positive Shannon cofactor (for atom $[s]$) of $\mathbf{C}_A \widehat{*} [s]$.
  - occurrences of $[s]$ ($[-s]$) are replaced by $\widehat{\mathbf{1}}_A (\widehat{0})$
  - if t is not s, occurrences of $[t]$ ($[-t]$) are replaced by $\widehat{0} (\widehat{\mathbf{1}}_A)$
- The stuttering clock of a state s is $\tau(s) = \mathbf{1}_A \widehat{-} (\mathbf{C}_A(s) \widehat{+} \mathbf{T}_A(s))$; when $\tau(s)$ is not null, a silent implicit transition $s \rightarrow_A s$ is fired. We name step in s the labeled transition $\tau(s) : s \rightarrow_A s$

Properties

- a state s in a constrained automaton A is *deterministic* if the triggers of its transitions are mutually exclusive; formally s is *deterministic* iff
  $(\forall((s, s_1), (s, s_2)) \in \rightarrow_A \times \rightarrow_A)((s_1 = s_2) \vee (T_A((s, s_1)) \widehat{*} T_A((s, s_2)) \; is \; null))$
- a constrained automaton is deterministic iff all its states are deterministic

- a state s in a constrained automaton A is *reactive or total* if for all input configuration, represented by a control formula I there exists a trigger h, a state s_1 and a transition or a step $h : (s, s_1)$ such that $h\widehat{*}I$ is not null; formally s is *reactive* iff
$\tau(s) \widehat{+} (\Sigma_{(s,t)\in \rightarrow_A}(\mathbf{T}_A((s,t)))) = \widehat{\mathbf{1}}_A$
- a constrained automaton is reactive iff all its states are reactive (note that if $\mathbf{C}_A$ is not $\widehat{0}$ then A is not reactive)

Regular expressions

We define the algebra of regular expressions which will be used to abstract constrained automata or represent there null formula [20].

definition 3 *A Kleene algebra is structure* $(A, +, ., *, 0, 1)$ *satisfying, for all* $a, b, c \in A$*,*

- $(A, +, ., 0, 1)$ *is an idempotent semi-ring*
  - $(A, +, 0)$ *is an idempotent commutative monoid*
  - $(A, ., 1)$ *is a monoid*
  - $a.0 = 0.a = 0$
  - $a.(b + c) = a.b + a.c$
  - $(a + b).c = a.c + b.c$
- *Partial order* $(a \leq b)$ *iff* $(a + b = b)$
  - $a + a = a \Rightarrow a \leq a$
  - $a + b = b \wedge b + c = c \Rightarrow a + c = a + b + c = b + c = c$
  - $a + b = a \wedge a + b = b \Rightarrow a = b$
- *Star definition with natural partial order*
  - *(SK1):* $1 + aa^* \leq a^*$
  - *(SK2):* $1 + a^*a \leq a^*$
  - *(SK3):* $b + ax \leq x \Rightarrow a^*b \leq x$
  - *(SK4):* $b + xa \leq x \Rightarrow ba^* \leq x$
- *Monotonicity:* $\leq$ *is monotonic with respect to all Kleene operators*

Example

The constraint of the alternating automaton $C = (a\widehat{*}b)$ can equivalently be expressed as the regular event expression $((a\widehat{-}b) + (b\widehat{-}a))^*$

```
thread alternate
   features
      a,b:  in event port;
   constraints
      always (a andnot b) or (b andnot a);
end alternate;
```

Listing 23: A controlled automaton in the AADL behavioural annex

Notations

Our objective is to represent events and event formulas as regular expressions (extended) with counting. We therefore start with a comparison to the property specification language PSL [24]. The words $S \in W_A$ of an automaton A are generated from the following 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 $S_1; S_2$, union $S_1 + S_2$, star S^*, positive $S+ = S; S^*$, option $S? = 1 + S$, fusion $S : T$, synchronous product $S|T$, interleaving and subsets.
- Reduction
  - $0 + S = S + 0 = S$, $1; S = S; 1 = S$, $0; S = S; 0 = 0$, $S + S = S$
  - $S^*; S^* = S^{**} = (1 + S)^* = (1 + SS^*) = S^*$, $0^* = 1 + 0; 0^* = 1 + 0 = 1$

Counters [23] of the form $S[n]$ are inductively defined by $S[0] = 1$ and $S[m + 1] = S; S[m]$

- (SD1) $(\forall n \geq m) S[m..n] = S[m]; (1 + S)[n - m]$
- (SD2) $S[..n] = S[0..n] = S[0]; (1 + S)[n] = (1 + S)[n]$
- (SD3) $S[..] = S^*$
- (SD4) $S[m..] = S[m]; S[..] = S[m]; S^*$

Example

The alternating automaton of Listing 22 could itself be alternatively expressed by the composition of two regular event expression consisting of the negation of the constraint $(a \widehat{*} b)^*$ and of its transitions $(a; b)^*$, which yields $((a \widehat{-} b); (b \widehat{-} a))^*$.

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

      constraints
         always {(a andnot b);(b andnot a)}
   end alternate;
end alternate;
```

Listing 24: A controlled automaton in the AADL behavioural annex

Synchronous product

The global behaviour of a component such as a thread can be defined by the composition of features belonging to this component. The synchronous product $\Vdash$ is one of these composition operators that will be used in Synchronous AADL Annex. Given two constrained automata $A = (S_A, s_{A0}, \rightarrow_A, V_A, T_A, \mathbf{C}_A)$ and $A = (S_B, s_{B0}, \rightarrow_B, V_B, T_B, \mathbf{C}_B)$ their constrained synchronous product $A \Vdash B$ corresponds to the conjunction of the behaviours specified by each of them. $A \Vdash B$ is the constrained automaton $AB = (S_{AB}, s_{AB0}, \rightarrow_{AB}, V_{AB}, T_{AB}, \mathbf{C}_{AB})$ where

- $S_{AB} = S_A \times S_B$ is the set of states,

- $s_{AB0} = (s_{A0}, s_{B0})$ is the initial state,
- $\rightarrow_{AB} = \{((s_1, t_1), (s_2, t_2))/((s_1, s_2), (t_1, t_2)) \in \rightarrow_A \times \rightarrow_B\}$,
- $V_{AB} = V_A \cup V_B$ is the set of variables,
- $(\forall\ st = ((s_1, t_1), (s_2, t_2)) \in \rightarrow_{AB})\ (T_{AB}(st) = T_{AB}((s_1, t_1)) \widehat{*} T_{AB}((s_2, t_2)))$,
- $\mathbf{C}_{AB} = \mathbf{C}_A \ \widehat{+}\ \mathbf{C}_B$.

The synchronous product is associative (context-independent), commutative (order-independent) and has neutral element $\mathbf{1} = (\{s\}, s, \emptyset, \emptyset, \emptyset, \widehat{\mathbf{0}})$. Deterministic automata are idempotent.

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399