SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

AS5506B/SSA

2014-09-20

ARCHITECTURE ANALYSIS & DESIGN LANGUAGE (AADL) V2 SYNCHRONOUS SYSTEMS ANNEX (SSA)

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

TO PLACE A DOCUMENT ORDER:
Tel: 877-606-7323 (inside USA and Canada)
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org

SAE WEB ADDRESS: **http://www.sae.org**

TABLE OF CONTENTS

ANNEX DOCUMENT SSA – SYNCHRONOUS SYSTEMS ANNEX

SSA.1 *Scope*

(1) The Synchronous Systems Annex (SSA) defines an extension of the behavioural annex (BA) by means of errata to support synchronous composition. This extension is complemented with

- the specification of an annex sublanguage to formally specify software-timing constraints: an event or clock calculus,
- and behavioural abstractions, by means of regular expressions over events and data-flow networks.

(2) Just as behavioral annexes are associated with AADL thread elements, behavioural abstractions are associated with thread abstractions, represented by AADL processes.

(3) The purpose of the SSA is support system design with a development methodology reminiscent of synchronous programming languages (SC-Charts, SyncCharts, CCSL, Signal, SCADE). The aim of such a methodology is to automate the synthesis of behaviors and program threads from high-level specifications that formalise user requirements:

- Timing constraints, in the form of assertions and regular expressions
- Flow and connections, in the form of data-flow networks

Automated synthesis not only facilitates design but also guarantees the preservation of formal properties and requirements specified and verified on abstract architectures.

(4) To support the formal definition of our MoCC, we present a algebra of automata consisting of synchronous 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

(5) The SSA is build upon existing concepts of the AADL [1], its behaviour annex [2], and its forthcoming constraint annex, in order to express a synchronous model of computations and communications.

(6) Its specification reduces to the update of a limited number of concepts in the behavioural annex and provides a synchronous design methodology for the AADL. Consequently, the structure of the present document reads as follows.

- Section SSA.2 identifies the core AADL artefacts from which time can be sensed and on which our model will operate.
- Section SSA.3, defines an update of the AADL behaviour annex to support the proposed synchronous semantics. *Guards* are defined as generalised behaviour conditions from events defined by AADL core threads, ports and property fields.
- Section SSA.4 defines abstraction of behaviour annexes automata in terms of timing constraints.
- Section 105 extends them to regular expressions, in the spirit of the constraint annex, and Section SSA.6 outlines its use within processes seen as abstract threads.
- In appendix, Section SSA.7 summarises the formal semantics that defines the synchronous model of computation and communication (MoCC) of the SSA.

SSA.2 *Overview of the AADL behaviour annex*

(7) The behavioural annex of the AADL [2] provides the required artefacts to define logically timed synchronous autamata. Our approach is it as the semantic core of the AADL and define synchronous specifications inherited from behaviour annex specifications: the SSA rests on the behaviour annex AS5506/2 as a foundation [1], of which it defines an extension, an update, or "erratum".

(8) In the behaviour annex, the specification of a synchronous transition system (STS) comprises three sections: variables, states and transitions.

States

(9) The state of an STS can be qualified *initial*, to represent the default entry point, *complete*, to represent suspension of execution and resumption based on external trigger conditions, or *final*, to represent termination. Otherwise, it is an unqualified *execution state*, that represents intermediate computation state. The transitions that have an intermediate *execution state* as source state can be interpreted as immediate transitions.

(10) The STS of a thread or device (D.2 par. 2) has one initial state and 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)

(11) 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

(12) 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)

- 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
- an *execution condition* models a behaviour on input values from ports, shared data, parameters, and behaviour variable values

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

Example

(13) Listing (14) is an example of behavioural annex embedded in the specification of a thread [2, Section D.4, p26-27]. It defines an automaton with an initial (session) state `st`, final (session) state `sf` and (intermediate)

execution states `s1` and `s2`. The automaton is executed every $10ms$ and awaits dispatch from port `a`. If no data is available within the $10ms$ deadline, a `timeout` event is dispatched (by the scheduler), causing the STS to stutter and to signal its current state (or expectation) along port `d`: `1` in `st` and `0` in `sf`. If data is available along port `a`, it is dispatched to the thread which steps into an intermediate state: `s1` from `st`, `s2` from `sf`. The thread then finalises a transition to a complete state depending on the value of `a`: `1` to `sf` and `0` to `st`.

(14) *Specification of the sender control thread in a communication protocol*

```
thread sender
   features
      d: out event data port;
      a: in event data port;
   properties
         Dispatch_Protocol => Timed;
         Period => 10ms;
and sender;

thread implementation sender
   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]->sf {d!(0)};
         sf-[on dispatch a]->s2;
         s2-[a=0]->st;
         s2-[a=1]->sf;
   **};
end sender;
```

SSA.3 *A synchronous behavioural annex*

(15) The SSA describes an STS using the same three sections as the BA: variables, states and transitions; completed by constraints expressed as regular expressions to express observers and invariants. The combination of STS and of temporal constraints, presented Section SSA.4, yields the hybrid structure of time-constrained automata depicted Listing (91). It supports an algebraic definition [4], outlined Section SSA.7.

States

(16) The SSA has no intermediate or execution states, like `s1` and `s2` in Listing (14). In the SSA, time is measured from the discrete series of transitions occurring during the execution of an STS. Conversely, the time elapsed from the departing the source state of a transition, evaluating its guard, executing its actions, and entering its target state are abstracted to zero: all events and actions occurring within that lapse of time are assumed to be logically synchronous, as depicted below.

$$\overbrace{\underbrace{\texttt{st -[on dispatch a]-> s1 [a=1]-> sf } \{\ldots\} \texttt{ ...}}_{t_0}}^{\Delta t=10ms} \underbrace{\texttt{ sf -...-> st}}_{t_1}$$

(17) The SSA has no final state either. In the SSA, threads should be allowed to run forever. Usually, the system scheduler is such a "thread"; it indeed has a final state, but it is reached when the whole system is halted. This very much differs from the final state of a subprogram. For instance the "Sender behaviour Specification" (D.4, Listing (14)), is better interpreted as the description of a session of the actual sender sub-system, which would indefinitely iterate such sessions.

(18) As a result, in any STS of the SSA, one and only one state must be explicitly qualified as `initial` and all states are implicitly qualified as `complete`.

Transitions

(19) 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).

(20) By applying the same principle to the AADL (e.g. status of a queue and value, ...), one can unambiguously specify the schedules of dispatch and read actions on ports a and timeout, all using the same triggering transitions, as depicted in Listing (27).

(21) 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.

Guards

(22) The SSA defines the notion of *guard* as an aggregation of triggers and evaluation conditions, as depicted in Listing (27). Guard form logical formula constructed using the AADL conjunction `and`, disjunction `or`, applied to triggers and execution conditions. Since the use of priority of the AADL allows to distinguish between "a and b" present and "a but not b" present, we additionally introduce the `andnot` combinator applied to execution conditions.

(23) Notice that the BA [2, p22 and p25] allow the aggregation of dispatch conditions using logical formulas already, Listing (24). The SSA slightly extends its grammar to uniformly reason about time as the synchronisation of triggers, conditions and actions.

Logical dispatch conditions in the BA

(24)
```
dispatch_trigger_condition ::=
          dispatch_trigger_logical_expression
        | provides_subprogram_access_identifier
        | dispatch_relative_timeout_catch
        | stop
  dispatch_trigger_logical_expression ::=
        dispatch_conjunction { or dispatch_conjunction }*
  dispatch_conjunction ::=
        dispatch_trigger { and dispatch_trigger }*
```

(25) In the SSA, we explicitly write `a?(1)` for reading the first value available on the queue of a dispatched port a and for testing it equal to 1 (operationally i.e. `a?(v) and v=1`). For a data event port, this can alternatively be abbreviated as $a = 1$, as in Listing (14). To avoid cluttering guard expressions, we abbreviate the dispatch condition `on dispatch a` on a port `a` using the `@`sign as `@a`.

Example

(26) Listing (27) outlines a variant of the sender that ensures deterministic behaviour of the transition system with priority given to timeout port dispatch.

(27) *Sender with synchronous guard*

```
transitions
   st-[on dispatch (timeout andnot a)]->st {d!(1)};
   st-[on dispatch a and a?(1)]->sf;
   st-[on dispatch a and a?(0)]->st;
   sf-[on dispatch (timeout andnot a)]->sf {d!(0)};
   sf-[on dispatch a and a?(0)]->st;
   sf-[on dispatch a and a?(1)]->sf;
```

(28) The proposed aggregation supports the principle of associating a complete transition within the logical instant (e.g. t_1) within which `timeout`, `d` and not `a` occur simultaneously.

$$\overbrace{\underbrace{\texttt{sf -[on dispatch (timeout andnot a)]-> sf \{d!(0)\}}\ \ldots}_{t_1}}^{\Delta t=10ms}\ \underbrace{\texttt{sf -...-> st}}_{t_2}$$

Remarks

(29) The specification of Listing (27) disambiguates the possibility that a could be dispatched "at the same (logical) time as" the timeout, or that `a` could be dispatched even after the occurrence of the `timeout`, which Listing (14) does not account for, possibly resulting in a non-deterministic choice between transitions `st-[on dispatch timeout]->st` and `st-[on dispatch a]->s1`.

(30) The `andnot` specification applied to the port a or timeout is operationally equivalent to checking a's queue empty with `a'count=0`. Using the latter may in general result in a non stuttering specifications, which would have the capability to change state without reacting to an input event.

Syntax

(31) A guard generalises the definition of `behavior_condition` and `dispatch_trigger_condition` defined in page 22 and 25 of the behaviour annex (AS5506/2).

(32) *Syntax of guards in the SSA*

```
guard ::=  [on dispatch] dispatch_trigger
         | logical_value_expression
         | guard ( and | or ) guard
         | guard andnot dispatch_trigger
```

Restrictions

(33) A number of restrictions on the discriminated usage of guards and actions applies to the SSA.

(34) Restrictions on guards

1. Guards are only permitted to freeze ports upon dispatch, and to read and test values from frozen port ranges.
2. The dispatch action of a guard consumes an amount of values specified by the dispatch action. By default, `on dispatch a` freezes the first value in a queue, i.e. `a[1]`, `on dispatch a*` means that all the queue is frozen, and `a[m,n]` means that the mth to nth values are frozen.
3. Guards may not manipulate the queue of a port by, e.g., checking it empty (`a'count=0`, absence of event) or not (`a'count>0`, presence of an event) but instead rely on the logical abstraction offered by the `andnot` combinator.
4. The use of the combinators `and`, `or` and `andnot` o not synchronise their trigger and logical subexpressions in guard expressions. For example, `s0 -[a and b] -> s1` does not force the events `a` and `b` to occur simultaneously. It is an observer of their simultaneous occurrence relative to the parent observer: the dispatched automaton. However, the behaviour action `c := a and b` requires `a` and `b` to be synchronised to `c` with respect to the parent observer.

(35) Restrictions on behaviour actions

1. Behaviour actions `behavior_action` are only permitted to read frozen port values, to perform computations, and to emit values.
2. By default, an output port may only be used (sent to) once during the behaviour action of a given STS transition. One may however specify and implement how many values are being sent at all times (burst mode) just as for reading port queues.
3. Iterations in behaviour actions `behavior_action` should be allowed (oversampling) by defining a local (scoped) clock domain.
4. A behaviour action cannot perform a dispatch and therefore consume values.

SSA.4 *Constraints as behaviour abstractions*

(36) To render the refinement-based design methodology underlying the design of the AADL, the SSA associates synchronous behavioural annexes with the definition of logically timed constraints to abstract them.

- The purpose of a (synchronous) behavioural annex (of a thread) is to specify the operational function of a system, component, process in an AADL specification.
- The purpose of a constraint section (of a process or component) is to specify an abstraction of this behaviour with respect to its timing and temporal properties.

(37) Guard in behavioural annexes belong to the operational processing of port queues performed in behaviour annexes. Dispatching, reading, writing, querying a port queue or handling a timeout cannot be performed by a constraint. These operations must be abstracted with respect to the logical and temporal conditions they imply.

- As a result, a constraint should refer to a connection or port identifier a as the abstraction of a port queue, possibly indistinctive of its direction, input or output.
- Similarly, a constraint should refer to an equation $a = v$ as the specification of the current, reception or emission of a value of a port or variable a.
- Conversely, it should refer as `andnot` a for the absence of value along a.

Examples

(38) Listing (39) depicts an 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.

(39) *Most abstract specification of the sender protocol*

```
constraints
    (a andnot  d) or (d andnot  a);
```

(40) Listing (41) 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 `pre (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.

(41) *A refinement of the sender constraints*

```
constraints
    pre (a=0) and (a or (d=1 andnot  a));
    pre (a=1) and (a or (d=0 andnot  a));
```

Syntax

(42) A `constraint`, Listing (43), 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.

(43) *Syntax of constraint conditions*

```
constraint     ::=  guard | pre logical_value_condition

specification ::= constraints ( ([always] | never) constraint; )*
```

(44) The `logical_value_condition` refers to an AADL value expression. The term `always constraint` means that `constraint` should hold at all times (i.e. every time its implementation evaluates it). The constraint `never constraint` means that `constraint` must be false at all times.

Restrictions and consistency rules

(45) 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 it

- when an unexpected event occurs, it is placed in a queue 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

(46) In a guard expression of a constraint, we abstract `on dispatch a` to have the same meaning as `a` i.e. an occurrence of an event along a concrete port abstracted by `a` as well as, conversely, `a?(0)` and `a=0` to mean reading a value along a and testing it to be 0.

SSA.5 *Regular constraints*

(47) 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 [6].

Example

(48) Listing (49) is a 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.

(49) *Sender abstraction using regular expressions*

```
constraints
    {a=0; a or (d=1 andnot a)};
    {a=1; a or (d=0 andnot a)};
```

Counting

(50) Just as in PSL, counting expressions in SSA constraints relate events with time units and periodic behaviours. Its principle can be applied to the sender protocol specification by, e.g., considering its timeout of $10ms$. While one may or may not want to specify the thread's `timeout` signal directly, it is still possible to specify the minimum amount of time before port d is triggered as a result of the timeout dispatch, Listing (51).

(51) *Timed abstraction of the sender protocol*

```
constraints
    (a andnot d every ms[0..10[) or (d andnot a every ms[10..[)
```

(52) In Listing (51), the terms a and d denote port events, the periods or instants at which they are dispatched or checked empty. The constraint means that it is always the case that either a occurs and not d within $10ms$ and that, every $10ms$ after dispatch, d occurs as a result of the timeout, when a is absent. The keyword `every` explicitly times the port d by associating it with a period of time.

(53) A possible refinement of this specification is defined in Listing (54), decomposes in two parts. One part consists of the logical 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, it is that 10 milliseconds have elapsed (relative to the context in which the constraint is checked) and (then) no a is present.

(54) *Real-time abstraction of the sender protocol*

```
constraints
    {a=0; a xor d=1};
    {a=1; a xor d=0};
    d every ms[10]
```

(55) Notice that the time unit ms cannot be regarded as a logical event, as its meaning is not a port. Consequently, it shouldn't trigger transition either by, e.g., being made explicit in a guard expression. However, it is mandatory to introduce the `every` keyword to explicitly time a given event, whose occurrence may itself trigger a transition.

Syntax

(56) The syntax of regular expression is listed in Listing (57). It corresponds to a Kleene algebra on guards, 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.

(57) *Syntax of regular expressions*

```
regexp ::= guard
         | { regexp ; regexp }
```

```
| regexp + regexp
| regexp*
| regexp?
| regexp[n]
| regexp : regexp
| regexp || regexp
```

(58) By extension, we revise the syntax of constraints in Listing (43) to allow for a general logical and timed synchronisations between ports and regular expressions on port events, Listing (59). Unit keywords account for time units. A time expression consists of a time unit, period and phase. A regular constraint is regular expression `regexp` possibly synchronised to a guard (to trigger it by or to synchronize it to an event) or to a linear time specification.

(59) *Updated syntax of guards*

```
unit ::= h | m |Âăs | ms | ps

time ::= unit[integer]+integer

regular_constraint ::= regexp [ every (guard | time ) ]

specification ::= constraints ( regular_constraint; )*
```

Restrictions

(60) Semi-column `;` denotes the progression of time between the two consecutive transitions of its sub-expressions.

(61) Time units `h, m, s, ms, ps` are not ports. They cannot not trigger transitions or be used in guard expressions.

(62) 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.

(63) Similarly, if several annexes are present in a component specification, their associated semantics should be the synchronous product of the declared behaviours.

Example

(64) Another example of the behaviour annex in which this may prove so is that of the client protocol page 40 of AS5506/2, Listing (65). It specifies a thread of period $100ms$ to perform a computation consuming $60ms$.

(65) *Client-server protocol (AS5506/2, page 40)*

```
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;
```

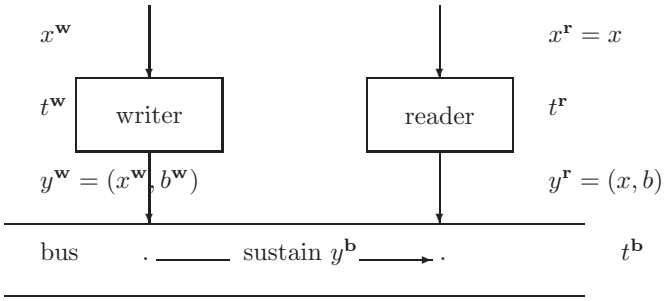
(66) One can simply abstract it by specifying the phase of `post` relative to the period of `pre` as follows.

```
constraint
    pre every ms[200]
    post every ms[200]+60
```

(67) The SSA is expressive enough to express common protocols and features used in system design, such as phase detection. Here `sunrise` is set to happen every time `light` event switches from false to true. Data event `light` is calculated every $10ms$ from `sensor_value` refreshed every $1ms$.

```
constraint
    sunrise every not (pre light) and light;
    sunset every (pre light) and not light;
    light and sensor_value >= threshold ;
    sensor_value every ms[1];
    light every ms[10];
```

(68) Notice that the sensor and light event can have independent rates. Harmonic (or not) multi-rate systems can equally be expressed using the SSA, together with their necessary down-sampling/up-sampling adapters. A good example is the LTTA protocol [7]. The LTTA is composed of three devices, a writer, a bus, and a reader. Each device is activated by its own, approximately periodic clock, subject to bounded jitter.



(69) At the nth clock tick $t^w(n)$, the writer generates the value $x^w(n)$ and an alternating flag $f^w(n)$.

```
process writer
   features
      xw : in event data port;
      cw : in event port;
      fw : out event data port;
   constraints
      fw every xw every cw;
      pre fw and not fw;

      cw every ms[InRate]
end writer;
```

(70) At any time t, the writer's output buffer contains the last value that was written into it. At time $t^b(n)$, the bus fetches the value (x^w, f^w) to store it in the input buffer of the reader, denoted by (x^b, f^b).

```
process bus
   features
      xw : in event data port;
      cw : in event port;
      cb : in event port;
      xb : out event data port;
      fb : out event data port;
      fwb : out event data port;
   connections
      xb -> writer.xw
      fb -> writer.fw
   constraints
      fb every xb every cb;

      cb every ms[BusRate]
end bus;
```

(71) At $t^r(n)$, the reader loads the input buffer (x^b, f^b) into the variable $(x(n), b(n)) = (x^b, f^b)(t^r(n))$.

```
process reader
   features
      xb : in event data port;
      fb : out event data port;
      cr : in event port;
      xr : in event data port;
      fr : out event data port;
   constraints
```

```
            xr every cr;
            (xr = xb) every (fb and not (pre fb));

            cr every ms[OutRate]
      end reader;
```

(72) A correct implementation of the LTTA amounts to maintaining the rate ratios of

$$\texttt{InRate} \geq \texttt{BusRate} \quad \text{and} \quad \lfloor \texttt{InRate/BusRate\verb} \rfloor \geq \texttt{OutRate/BusRate\verb}$$

between the reader, bus, and writer.

SSA.6 *Processes as abstract threads*

(73) 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 specifications 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 (74). 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.

(74) *Abstract specification of the Sender*

```
process sender_abstraction
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      a or (d andnot a);
end sender_abstraction;
```

(75) Listing (76) 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.

(76) *Specification refinement for the Sender*

```
process sender_refined_abstraction
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      { (a=0); (a or (d=1 andnot a)) };
      { (a=1); (a or (d=0 andnot a)) };
end sender_refined_abstraction;
```

(77) 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 (80).

(78) *Specification refinement for the Sender*

```
process sender_refinement
   features
      a: in data port boolean;
      d: out data port boolean;
   constraints
      { (a=0); (a or d=1) };
      { (a=1); (a or d=0) };
      d every ms[10]
end sender_refinement;
```

Inheritance

(79) 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 (80).

(80) *Explicit refinement relation between the specification and implementation of the sender*

```
thread implementation sender
   inherits sender_refinement;
```

Data-flow components

(81) To allow for unlimited multi-rate system design, the SSA comprises pre-defined process templates for the multi-clocked sampling and merge operators `when` and `default` reminiscent of the Signal data-flow language [8].

(82) Process `default` defines its output `z` by `x` every time it is available and by `y` otherwise. The control port `cz` defines the *clock* or pace of `z`. It is the union of these of `x` and `y`, by definition.

```
process default
   features
      x : in event data port;
      y : in event data port;
      z : out event data port;
   constraints
      cz every (x or y);
```

```
      z=x every cz and x;
      z=y every cz andnot x;
end default;
```

(83) Process `when` samples data from port `x` when and only when `y` is available and its value equal to true. Hence the rate `cz`, the conjunction of the paces of `x` and `y` and of the guard `y=true`

```
process when
   features
      x : in event data port;
      y : in event data port;
      z : out event data port;
   constraints
      cz every (x and y and y?(true));
      z=x every cz;
end when;
```

Acknowledgments

(84) This work is partly funded by Toyota InfoTechnology Center (ITC) and by INRIA D2T's standardisation support program. It is based on earlier recommendations to the SAE committee on AADL [3] and a formal semantics published in [4].

REFERENCES

[1] "SAE Architecture Analysis and Design Language (AADL) Annex Volume 2. Report AS5506/2. SAE Aerospace, 2011.

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

[3] "Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL)". L. Besnard, E. Borde, P. Dissaux, T. Gautier, P. Le Guernic, J.-P. Talpin. INRIA Technical Report n.446, 2014.

[4] "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.

[5] *CCSL: specifying clock constraints with UML/MARTE*, OMG, 2008. `http://www.omgmarte.org/node/66`.

[6] *IEEE Standard for Property Specification Language*. IEEE, 2005. `http://dx.doi.org/10.1109/IEEESTD.2005.97780`.

[7] *"A protocol for loosely time-triggered architectures*. Benveniste, A., Caspi, P., Le Guernic, P., Marchand, H., Talpin, J.-P., Tripakis, S. Embedded Software Conference. Lectures Notes in Computer Science. Springer Verlag, October 2002.

[8] *Programming Real-Time Applications with Signal*. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Proceedings of the IEEE, 79(9), September 1991.

APPENDIX

SSA.7 *Formal semantics of automata*

Control Algebra

(85) Let V a (possibly empty) countable set of signal variables and S a non empty finite set of state variables with $S \cap V = \emptyset$.
A *Boolean control algebra* $\Phi(V,S)$ is a tuple $(F_{V,S}, \widehat{*}, \widehat{+}, \widehat{-}, \widehat{\neg}_V, \widehat{0}, \widehat{1}_V)$, where,

- $\widehat{*}$, $\widehat{+}$ are notations for meet (infimum) and join (supremum) operations,
- $\widehat{0}$, $\widehat{1}_V$ are notations for minimum and maximum,
- and the set of control Boolean formulas $F_{V,S}$ is the smallest set that satisfies:
  - constants: $\widehat{0}, \widehat{1}_V \in F_{V,S}$;
  - atoms: $(\forall x \in V \cup S)(\widehat{x}, [x], [\neg x] \in F_{V,S})$;
  - unary expression (clock complementary): $(\forall f \in F_{V,S})(\widehat{\neg}_V f \in F_{V,S})$;
  - binary expressions: $(\forall f, g \in F_{V,S})(\widehat{+}fg, \widehat{*}fg, \widehat{-}fg \in F_{V,S})$.

  Parentheses and infix notations can be used in the formulas.
  The formula $\widehat{x}$ designates the *clock* of a variable x.

(86) $(F_{V,S}, \widehat{*}, \widehat{+}, \widehat{\neg}_V, \widehat{0}, \widehat{1}_V)$ is a Boolean algebra. The following supplementary axioms are also considered

- difference: $f \mathbin{\widehat{-}} g = f \mathbin{\widehat{*}} \widehat{\neg}_V g$;
- sampling partition: $(\forall x \in V \cup S)((\widehat{x} = [x] \mathbin{\widehat{+}} [\neg x]) \wedge ([x] \mathbin{\widehat{*}} [\neg x] = \widehat{0}))$ ($[x]$ and $[\neg x]$ respectively denote the clock of x sampled when x is tt and when x is ff);
- automaton clock: for V non empty, $\sum_{x \in V}([x] \mathbin{\widehat{+}} [\neg x]) = \widehat{1}_V$, and $(\forall s \in S)(\widehat{s} = \widehat{1}_V)$;
- null clock: $\widehat{1}_\emptyset = \widehat{0}$;
- state exclusiveness: $(\forall s1, s2 \in S)([s1] \mathbin{\widehat{*}} [s2] = \widehat{0}) \vee (s1 = s2)$.

Constrained Automata

(87) For an automaton A with V_A its set of signal variables and S_A its set of states, we denote by $F_{A,S}$ the set of *normal form* formulas in the Boolean control algebra $\Phi(V_A, S_A)$. A *polychronous constrained automaton* A is an epsilon-free automaton defined up to isomorphism (over states) as a tuple $A = (S_A, s_0, \twoheadrightarrow_A, V_A, T_A, C_A)$ where

- S_A is the non empty finite set of states;
- s_0 is the initial state;
- $\twoheadrightarrow_A \subset S_A \times S_A$ is the transition relation;
- V_A is the, possibly empty, finite set of signal variables;
- $T_A : (\twoheadrightarrow_A) \rightarrow F_{A,S}$ is the function that assigns a formula to a transition;
- C_A is the *constraint* of A: it is a formula in $F_{A,S}$ that is (constrained to be) *null* (a formula f in $F_{A,S}$ is *null* in A iff $f \mathbin{\widehat{*}} C_A = f$).

(88) A transition has an associated formula in the Boolean control algebra that represents the trigger of the transition.

(89) If the formula C_A is $\widehat{0}$, then the automaton is *constraint-free*; if C_A is $\widehat{1}_{V_A}$, all formulas are null.

Example

(90) Listing (91) 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).

(91) *A controlled automaton in the AADL behavioural annex*

```
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;
```

(92) The alternating automaton of Listing (**??**) 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. Since a, b are events, $[\neg a]$, $[\neg b]$ should be null.
$A_{alternate} = (S_{next\_commute} : \{S_1, S_2\},\ s_0 : S_1,\ \twoheadrightarrow_{next\_commute}: \{(S_1, S_2), (S_2, S_1)\},\ V_{next\_commute} : \{a, b\},\ T_{next\_commute} : (S_1, S_2) \mapsto a,\ (S_2, S_1) \mapsto b,\ C_{next\_commute} : a \widehat{*} b \widehat{+} [\neg a] \widehat{+} [\neg b])$

Notations

(93) For an automaton A and V_A its set of variables, 1_A denotes the supremum $\widehat{1}_{V_A}$; and for a state s in A, $C_A(s)$ is the normal form of $[s] \widehat{*} C_A$ ($C_A(s)$ represents the *null clock* of a state s).

(94) There exists at most one labeled transition from a given source s_1 to a given target s_2; two potential labeled transitions $h_1 : s_1 \twoheadrightarrow_A s_2$ and $h_2 : s_1 \twoheadrightarrow_A s_2$ are represented by $(h_1 \widehat{+} h_2) : s_1 \twoheadrightarrow_A s_2$.

(95) An automaton with an empty set of transitions is $\mathbb{O}_V = (\{s\}, s, \emptyset, V, \emptyset, \widehat{1}_V)$, which blocks all occurrences of all variables of V.

(96) The automaton with an empty set of variables is $\mathbb{I} = \mathbb{I}_\emptyset = (\{s\}, s, \emptyset, \emptyset, \emptyset, \widehat{0})$; it is equal to $\mathbb{O}_\emptyset = (\{s\}, s, \emptyset, \emptyset, \emptyset, \widehat{1}_\emptyset)$.

(97) The *control clock* of an automaton A is 1_A $(= \sum_{x \in V_A}(\widehat{x}))$, the supremum of the clocks of its variables.

(98) In $h : s_1 \twoheadrightarrow_A s_2$, h is *the trigger* of (s_1, s_2) and *a trigger* of s_1. The *trigger* of a state s, $trigger_A(s)$, is the upper bound of the triggers of s.

(99) The *stuttering clock* of a state is the clock difference between the control clock of the automaton and the trigger of the state (plus the null clock of the state): $\tau(s) = 1_A \mathbin{\widehat{-}} (C_A(s) \mathbin{\widehat{+}} trigger_A(s))$.

(100) When the stuttering clock $\tau(s)$ of a state s is not null, there is a silent implicit transition $\tau(s) : s \twoheadrightarrow_A s$ named *step*.

Properties

(101) A state t is *n-reachable* in A iff s_0 and t are not null and i/ either $n = 0$ and $t = s_0$, ii/ or $n > 0$ and either t is $(n-1)$-reachable in A, or $(\exists s\ (n-1)$-reachable in $A)(\exists h)(h \mathbin{\widehat{*}} [s]$ not null$)(h : s \twoheadrightarrow_A t)$.

(102) A state t is *reachable* in A iff it is $|S_A|$-reachable in A.

(103) A state s is *deterministic* if the triggers of its transitions are mutually exclusive: formally, s is deterministic iff $(\forall((s, s_1), (s, s_2)) \in \twoheadrightarrow_A \times \twoheadrightarrow_A)((s_1 = s_2) \vee (T_A((s, s_1)) \mathbin{\widehat{*}} T_A((s, s_2)) = \widehat{0}))$.

(104) An automaton is *deterministic* iff all its reachable states are deterministic.

(105) A state s is *total* (or *reactive*) iff $\tau(s) \mathbin{\widehat{+}} (\Sigma_{(s,t) \in \twoheadrightarrow_A}(trigger_A((s, t)))) = 1_A$. An automaton is *total* (or *reactive*) iff all its states are total (we observe that if C_A is not $\widehat{0}$ then A is not reactive).

Regular expressions

(106) 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

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

(108) *A controlled automaton in the AADL behavioural annex*

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

Notations

(109) Event formulas as regular expressions, extended with counting can be expressed using the property specification language PSL [6]. 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
  - $0 + S = S + 0 = S$, $1; S = S; 1 = S$, $0; S = S; 0 = 0$, $S + S = S$
  - $S^*; S^* = S^{**} = (1 + S)^* = (1 + SS^*) = S^*$, $0^* = 1 + 0; 0^* = 1 + 0 = 1$

(110) Regular expressions with counting, of the form $S[n]$, are inductively defined by $S[0] = 1$ and $S[m + 1] = S; S[m]$. They satisfy

- (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

(111) The alternating automaton of Listing (**??**) 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 $((a\widehat{-}b); (b\widehat{-}a))^*$.

(112) *A controlled automaton in the AADL behavioural annex*

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

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

Synchronous product

(113) 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 $\parallel$ 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 $A \parallel B$ corresponds to the conjunction of the behaviours specified by each of them. $A \parallel B$ 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,
- $(\forall\, st = ((s_1, t_1), (s_2, t_2)) \in \rightarrow_{AB})\ (T_{AB}(st) = T_{AB}((s_1, t_1)) \widehat{*} T_{AB}((s_2, t_2)))$,
- $\mathbf{C}_{AB} = \mathbf{C}_A \mathbin{\widehat{+}} \mathbf{C}_B$.

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.