A Synchronous Annex for the AADL

White-paper submitted to the SAE committee on the AADL standard

Loïc Besnard CNRS,
Thierry Gautier INRIA,
Paul Le Guernic INRIA,
Jean-Pierre Talpin INRIA, mailto:talpin@irisa.fr

Abstract

We propose a synchronous timing annex for the AADL standard. Our approach consists of building a synchronous model of computation and communication that best fits the semantic and expressive capability of the AADL and yet requires little to know (syntactic) extension to it, i.e. to identify a synchronous core of the AADL (which prerequisites a formal definition of synchrony at hand) and define a formal design methodology to use the AADL in a way that supports formal analysis, verification and synthesis.

This document first identifies the core AADL concepts from which time events can be described. Then, we naturally consider the behavior annex (BA) as the mean to model synchronous signals and traces through automata. Finally, we consider elements of the constraint annex to reason about abstractions of these signals and traces by clocks and relations among them.

To support the formal presentation of these elements, we define a model of automata that comprises a transition system to express explicit transitions and constraints, in the form of a boolean formula on time, to implicitly constraint its behavior. The implementation of such an automaton amounts to composing its explicit transition system with that of the controller synthesised from its specified constraints.

1 Introduction

The goal of this white-paper is to state some foundational principles towards the definition of a synchronous timing annex for the AADL standard. By putting forward synchrony and timing, we intend to define time starting from software and hardware events incurring synchronisations in an architecture. A 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 computations and communications, applicable to both software or hardware design, which puts emphasis on a domain of time, abstracted through synchronisation points, in order to break down computations into zero-time reactions and regard communications as instantaneous.

While abstracting real-time, the synchronous hypothesis offers an algebraic framework in which both event-driven and time-triggered execution policies can be modelled and bridges to gap symbolic, system-level, static scheduling (using so-called clock calculi) with real-time, time-triggered, distributed, and dynamic scheduling (using periodic or affine clock calculi).

1.1 Outline

Our aim is to equip the AADL standard with a framework allowing for synchronous modelling, verification and synthesis of architecture-focused embedded software. After the presentation of related work in that aim, Section 2, we start a brief outline of the model of computation and communication (MoCC) under consideration, Section 3, whose formal definition is given in Appendix A.

We start from the identification of core AADL events and the behavior annex AS56-02, from which time can be sensed, and on which our model will operate, Section 4. *Compound*

(synchronous) events are defined from core thread and ports events and property fields will be used to annotate core AADL concepts with timing properties. This presentation uses simple examples to illustrate our ideas.

The present timing annex proposal (TA) aims at exploitiong all existing concepts of the core, BA, and upcoming CA of the AADL to express a synchronous model of computations and communications. Its implementation hence reduces mainly to the specification of 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, formally defined in Appendix A. Section 5 concludes our presentation, offering perspectives in that aim.

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 [1] allows to define structural properties on AADL models that are checked inductively visiting the object of a model under verification. [3] presents an extension of this language called LUTE which further uses PSL (Property Specification Language) to check behavioral 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 [2]. It puts the emphasis on capturing multiple aspects of nominal and faulty, timed and hybrid behaviors of models. Formal verification is supported by the nuSMV tool. Similarly, the FIACRE framework [4] uses executable specifications and the TINA model checker to check structural and behavioral properties of AADL models.

RAMSES, on the other hand [5], presents the implementation of the AADL behavior annex. The behavior annex supports the specification of automata and sequences of actions to model the behavior 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 and Ocarina to generate C code for OSs complying the ARINC-653 standard.

Synchronous modeling is central in [6], which presents a formal real-time rewriting logic semantics for a behavioral 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.[7] 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 behaviors 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 frame-

work 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 [8], 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 behavior and constraint annexes in order to implement a synchronous design methodology in the AADL, and specify it within a polychronous MoCC.

2.1 About synchrony and polychrony

The synchronous model of programming 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 behavior 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 polychronous model, which allows to cope with it and provide model refinement techniques to reach the goal of a globally deterministic design.

3 A model of constrained automata in Polychrony

To support the formal presentation of the timing annex, 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 writen $a \widehat{*} b$.

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. Figure 1 depicts a constrained automaton manipulating two events a and b.

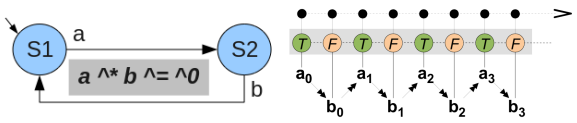

Figure 1 An alternating automaton controlling its input flow.

The automaton specifies the alternation of two input event streams a and b, depicted by the trace. Its reactive behavior, 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 that a and b should never occur simultaneously. With that constraint in place, the automaton behaves as a constrained asynchronous one (event interleaving). Finally, the absence of reflexive transitions specify that b (respectively a) cannot occur alone in state s_1 (respectively s_2). A reactive extension of this automaton allows b (respectively a) to occur in state s_1 (respectively s_2). But the reflexive transition must be fired only when b (respectively a) occurs alone.

The combination of a synchronous automaton and of a temporal constraint yields the hybrid structure of timed automaton depicted Figure 1. It supports an algebraic definition, presented in Appendix A by relying on the model of computation and communication (MoCC) of Polychrony in order to define a framework of constrained automata capable of expressing both the BA and CA 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 behavioral and constraint annex of the AADL much in the flavour of the program depicted in Figure 1 to separately specify explicit reactive behavior using automata (top) and refine these specifications using controller synthesis to enforce satisfaction of implicit timing constraints and temporal requirements.

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

thread implementation alternate
annex behavior_specification {**
   states
      s1: initial complete state;
      s2: complete state;
   transitions
      t1: s1-[on dispatch a]->s2;
      t2: s2-[on dispatch b]->s1;
   constraints
      never a and b;
**};
end alternate;
```

Listing 1 A controlled automaton in the AADL behavioral annex

4 Synchronous specifications in the AADL

All the AADL events that the timing annex needs are present in the behavior annex. Therefore, our proposal for a synchronous timing annex in the AADL will rest on the behavior annex AS56-02 as a foundation. It can be defined as an extension of the behavior annex or, alternatively, as a separate annex or an attached property set that refines or specializes a given behavior annex with synchronous timing constraints.

Figure 2 gives the principle of our approach, which consists in the definition of synchronous specifications inherited from (untouched) behavioral specification and refined with synchronous constraints expressed using regular expression over coumpound events, which we shall explain next.

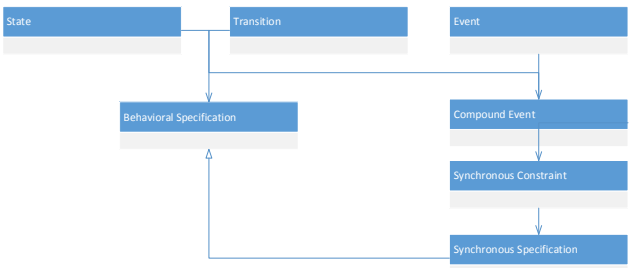

Figure 2 A synchronous annex defined by inheritance of behavioral specifications and constraints

Section 4.2 proposes a synchronous extension of behavior annexes in order to incorporate *compound* events and *actions* although, in principle, a synchronous annex could be detached from behavioral specifications by annexing them into *compound* declarations.

Section 4.3 extends this presentation to that of constraint annexes, in the aim of representing abstractions of automata using regular expressions or, conversely, to express (and possibly enforce by controller synthesis) properties of behavioral annexes using regular expressions representing requirements using observers or invariants.

We will use the behavior annex, Section 4.1, 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 behavior annex, we will use the term *compound* to denote a compound or generalized dispatch condition provided by a core AADL specification or a behavior annex.

4.1 The behavior annex as a foundation

The behavior annex defines a transition system (an extended automaton) described 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 alinea 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 consumption of an execution condition that catches a previously raised timeout (D3 alinea 18)
- the transition from an execution state to an execution state that can send value (to a port, a data, ...) (D3 alinea 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 alinea 4-8)

- an *execution condition* models a behavior on input values from ports, shared data, parameters, and behavior 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 Behavior_Specification (D.2 alinea 7,9).

4.1.1 Towards synchronous data/event flows

Based on the above analysis, we identify some of the extensions to STSs that would be suited toward synchronous extensions, annotations, annexes of AADL behavior specifications.

States

In a synchronous annex, threads should be allowed to run forever. Usually, the system scheduler is such a "thread"; the final state of a thread usually has a final state reached when the whole CPS (cyber-physical system) is halted, but this very much differs from the final state of a subprogram. For instance the "Sender Behavior Specification" (D.4, Figure 2), is better interpreted as the description of a session rather than a sender, thread or process, which would iterate such sessions.

```
thread implementation sender.v2
annex behavior_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.v2;
```

Listing 2 Sender in the behavioral annex, D.4

Transitions

The AADL property that:"dispatch does not depend on the input value", very much corresponds the causal constraint of 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. $\widehat{x} \rightarrow x$ to mean that the clock of x precedes the signal x).

Would the same principle be applied to the AADL (e.g. status of a queue and value, ...), one could then unambiguously put the dispatch and reading of a port a in the same trigger, as depicted in the following example (Listing 3, modifying figure 1 D.4).

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

```
thread implementation sender.v2
annex behavior_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;
      sf-[on dispatch a and a=1]->sf;
**};
end sender.v2;
```

Listing 3 Sender with generalized triggering expressions

An alternative specification would here be to declare transitions and the aggregated events separately, one descrbing the operational behavior, the other the requirements, by making use of the inheritance principle, Figure 4.

```
thread implementation sender.v2
annex timing_specification {**
   annex behavior_specification {**
      states
         st: initial complete state;
         sf: complete final state;

      transitions
         st-[on dispatch timeout]->st {d!(1)};
         st-[on trigger1]->sf;
         st-[on trigger0]->st;
         sf-[on dispatch timeout]->st {d!(0)};
         sf-[on trigger0]->st;
         sf-[on trigger1]->sf;
   **};

   constraints
      trigger0 = on dispatch a and a=0
      trigger1 = on dispatch a and a=1
**};
   end sender.v2;
```

Listing 4 An operational sender and refinement constraints

A modular decomposition would name the behavioral annex from `sender.v2` and inherit its definition in the constrained variant, Figure 5. Semantically, refinement assumes, in both cases, the synchronous composition of the specifications `sender.v2` and `sender.v3`.

This, however, requires `sender.v3` to get not only access to the `sender.v2` interface, but possibly also to its local states (as we will see later).

```
thread implementation sender.v3 refines sender.v2
   annex timing_specification {**
      constraints
         trigger0 = on dispatch a and a=0
         trigger1 = on dispatch a and a=1
**};
```

Listing 5 Inherited operational behavior and refinement constraints

Compounds

A *compound* is an aggregated evaluation condition or trigger, as depicted in Listing 6, that can possibly be declared as a local variable of type event.

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 occur 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 6 outlines a variant of the sender that ensures deterministic behavior of the transition system with priority given to port dispatch.

```
thread implementation sender.v2
annex behavior_specification {**

   states
      st: initial complete state;
      sf: complete final state;

   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]->st {d!(0)};
      sf-[on dispatch a and a=0]->st;
      sf-[on dispatch a and a=1]->sf;
**};
end sender.v2;
```

Listing 6 Sender with andnot expressions

A *compound* constraint subexpression may also be a query on the current state of the STS, written "in s1”, to mean whether the current STS is in state s1, a state identifier.

Figure 7 gives the alternative structure with separate constraints and triggers.

```
thread implementation sender.v2
   annex timing_specification {**
      annex behavior_specification {**
         states
            st: initial complete state;
            sf: complete final state;
         transitions
            st-[on fto]->st {d!(1)};
            st-[on trigger1]->sf;
            st-[on trigger0]->st;
            sf-[on fto]->st {d!(0)};
            sf-[on trigger0]->st;
            sf-[on trigger1]->sf;
      **};
   constraints
      fto      = on dispatch timeout andnot a
      trigger0 = on dispatch a and a=0
      trigger1 = on dispatch a and a=1
   **};
end sender.v2;
```

Listing 7 Sender with constraints

Figure 8 gives the modular structuration with separate annexes.

```
thread implementation sender.v2
   annex behavior_specification {**
      states
         st: initial complete state;
         sf: complete final state;
      transitions
         st-[on fto]->st {d!(1)};
         st-[on trigger1]->sf;
         st-[on trigger0]->st;
         sf-[on fto]->st {d!(0)};
         sf-[on trigger0]->st;
         sf-[on trigger1]->sf;
   **};
end sender.v2;

thread implementation sender.v2 refines sender.v3
   annex timing_specification {**
      constraints
         fto      = on dispatch timeout andnot a
         trigger0 = on dispatch a and a=0
         trigger1 = on dispatch a and a=1
   **};
end sender.v3;
```

Listing 8 Sender with refinement

Actions

A given condition or action may be shared by several transitions of a thread. Labelling a transition by a compound allows one to associate an action with it and execute it.

If the transition is labelled by, say, compound L, then the action associated with L can be written separately (e.g. in a synchronous annex or a timing annex), provided that it is triggered by the *compound* L.

As an example, action selection is introduced in Listing 9. One clear advantage is to factor actions that are common to several conditions. A second advantage is to give the capability of declaring actions or conditions separately through compounds in a timing annex. One last is to provide a uniform way to declare actions in an automaton.

```
thread implementation sender.v2
annex timing_specification {**

   annex behavior_specification {**
      states
         st: initial complete state;
         sf: complete final state;
      transitions
      l0:st-[on fto]->st;
         st-[on trigger1]->sf;
         st-[on trigger0]->st;
      l1:sf-[on fto]->st;
         sf-[on trigger0]->st;
         sf-[on trigger1]->sf;
   **};

   constraints
      fto      = on dispatch timeout andnot a;
      trigger0 = on dispatch a and a=0;
      trigger1 = on dispatch a and a=1;

   actions
      on stto {d!(1)};
      on sfto {d!(0)};
**};
end sender.v2;
```

Listing 9 Final sender automaton

Note that, in this case, actions can in principle be replaced by constraints. Since `d` is an output port, one could instead enforce it to equal 1 on event `fto` in state `st`, Figure 10. Using the principle of controller synthesis, again, one would regard these constraints as the invariants to fulfil and the actions in Figure 9 as the controller enforcing them.

```
   constraints
      on fto and in st = (d=1);
      on fto and in st = (d=0);
```

Listing 10 Final sender automaton

As a result, the modular specification of the sender into the behavioral and timing annex would be the following, Figure 11.

```
thread implementation sender.v2
   annex behavior_specification {**
      states
         st: initial complete state;
         sf: complete final state;
      transitions
      l0:st-[on fto]->st;
         st-[on trigger1]->sf;
         st-[on trigger0]->st;
      l1:sf-[on fto]->st;
         sf-[on trigger0]->st;
         sf-[on trigger1]->sf;
   **};
end sender.v2;

thread implementation sender.v3 refines sender.v2
   annex timing_specification {**
      constraints
         fto      = on dispatch timeout andnot a;
         trigger0 = on dispatch a and a=0;
         trigger1 = on dispatch a and a=1;
         on fto and in st = (d=1);
         on fto and in st = (d=0);
   **};
end sender.v3;
```

Listing 11 Final sender automaton

4.2 A synchronous behavior annex

Starting from the analysis of Section 4.1, 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 coumpound events to express observers, invariants, or guarded actions.

Variables, States, Transitions

See Section 4.1.1.

Actions

All actions are guarded actions executed following a data/event flow model. When the guard is omitted, the implicit guard is the “clock” of the automaton. Each action can be made of sequential code as actions in the transitions of the behavior annex AS5506/2

Static Constraints

A constraint is either an **always** or a **never** constraint. It is defined over *compound* expressions. For instance, the constraint "never a AND b" means that a and b should never occur during the same evaluation step, transition, or instant. Conversally, the constraint "always a AND " means that a occurs if and only if b occurs during the same instant.

The constraint section of an STS may contain a set of constraints that should simultaneously be satisfied (i.e. a conjunction of constraints).

As an example, consider the "alternate" specification introduced in Listing 1. Its automaton is defined in Listing 12, using the constraint "never a and b".

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

thread implementation alternate
annex behavior_specification {**
   states
      s1: initial complete state;
      s2: complete state;
   transitions
      t1: s1-[on dispatch a]->s2;
      t2: s2-[on dispatch b]->s1;
   actions
      on t2 { c! };
   constraints
      never a and b;
      -- i.e. always (a andnot b) or (b andnot a)*
      -- i.e. regexp ((a andnot b) or (b andnot a))*
**};
end alternate;
```

Listing 12 A polychronous automaton with constraints

Again, the specification can be split into an annex describing the operational behavior, Figure 13,

```
thread implementation alternate
   annex behavior_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 13 A behavior annex with constraints

and constraints that control or restrict operations (transitions), Figure 14.

```
thread implementation alternative refines alternate
   annex timing_specification {**
   constraints
      never a and b;
      on t2 = on c;
**};
end alternative;
```

Listing 14 A behavior annex with constraints

Notice that "never a and b" is a partial specification (i.e. a non executable property). It is important, however, to possibly provide for an unexpected behavior, or error, if the constraint cannot be verified or satisfied (at runtime). But here there are many options:

- 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 semantics corresponds to broadcast-synchronous semantics)
- an error is implicitly raised
- an error is explicit raised and handled in the automaton

4.3 Regular expressions in the synchronous behavior annex

Instead of a transition system, one could alternatively just use a regular expression over events extended with counting (Section A.3). A behavior annex using a regular expression (instead of a transition system) would be described in four sections:

- variables declarations;
- regular expression
- guarded actions;
- (static or invariant) constraints;

Variables, actions, constraints remain unchanged. Nevertheless, since there is no state section in a regular expression, a compound like "in s1" cannot be used. We are interested in defining a concrete syntax of regular expressions with counting that could be shared with that of the constraint and requirement annexes of the AADL, should they use regular expression. A initial proposal is given in Section A.3. Using regular expressions, the alternate thread would be defined as follows in Listing 15.

```
thread implementation alternate
   annex timing_specification {**
      constraints
         forever a;b     -- i.e. (a;b)*
         never a and b; -- i.e. ((a andnot b) or (b andnot a))*
      actions
         on b { c! };    -- i.e. on b = on c
   **};
end alternate;
```

Listing 15 A polychronous regular expression with static constraints

In addition to its iterated behavior, one could specify its static constraint as a regular expression as well, Listing 16.

```
thread implementation alternate
annex timing_specification {**
   constraints
      forever (a andnot b);(b andnot a)
   actions
      on b { c! };
**};
end alternate;
```

Listing 16 A polychronous regular expression with implicit constraints

Moreover, the output dispatch could be made implicit and the output dispatch specified in the regular expression as well, listing 18.

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

thread implementation alternate
annex timing_specification {**
   constraints
      forever (a andnot b);((b andnot a) and c)
**};
end alternate;
```

Listing 17 A polychronous automaton with constraints

Counting expression may be used to relate events with time units, as implicit events. For instance, computing (60ms) could be written $ms[60]$ as in the following example extracted from that of the client in the behavior annex D.8:

```
thread a_client
...
annex timing_specification {**
   regexp
      forever dispatch; pre; ms[60]; post(x)
**};
end a_client;
```

Listing 18 A polychronous automaton with constraints

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

5 Conclusion

We observe that the present timing annex (TA), by exploiting all existing concepts of the core AADL, behavioral annex, and possibly some of the forthcoming constraint annex, mostly reduces to the specification of a synchronous, refinement-based, design methodology for the AADL. It relies on a model of computation and communication (MoCC) of Polychrony, presented in Appendix 3, presented as a model of constrained automata, currently under implementation.

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

Acknowledgments

We wish to thank Oleg, Peter, Jérôme, Etienne, Franck and Pierre for valuable discussions at the occasion of our previous meetings and would like to open the present document to their contributions as well as of other interested participants of the committee in order to build a structured and homogeneous proposal that seamlessly accommodates with the core and annexes of the AADL.

References

1 “Expressing and enforcing user-defined constraints of AADL models”. Olivier GILLES, Jerome HUGUES. IEEE ICECCS, 2010.

2 “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.

3 "Compositional Verification of Architectural Models". Darren Cofer,Andrew Gacek, Steven Miller, Michael Whalen. Springer NFM, 2012.

4 "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.

5 “An Implementation of the Behavior Annex in the AADL toolset Osate2”. Gilles Lasnier, Laurent Pautet, Jerome Hugues, Lutz Wrage. IEEE ICECCS 2011.

6 “Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude”. Peter Csaba, Olveczky, Artur Boronat, Jose Meseguer, and Edgar Pek. LNCS FTDS 2010.

7 "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.

8 "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.

9 "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.

10 "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.

11 "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.

12 "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.

13 "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.

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

15 "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.

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

17 Conway, J.H. (1971). Regular algebra and finite machines. London: Chapman and Hall. ISBN 0- 412-10620-5. Zbl 0231.94041. Chap.IV.

18 Arto Salomaa, “Two complete axiom systems for the algebra of regular events,” J. Assoc. Comput. Mach. 13:1 (January, 1966), 158–169.

19 Regular Expressions with Counting: Weak versus Strong Determinism Wouter Gelade, Marc Gyssens, and Wim Martens SIAM J. Comput., 41(1), 160–190. (31 pages)

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

A A framework of constrained automata

A.1 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*: $(\forall x \in (V \cup S))((\widehat{x} = [x] \ \widehat{+} \ [-x]) \wedge ([x] \ \widehat{*} \ [-x] = \widehat{\mathbf{0}}))$
- *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*: $(\forall s_1, s_2 \in S)([s_1] \widehat{*} [s_2] = \widehat{\mathbf{0}}) \vee (s_1 = s_2))$

A.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.

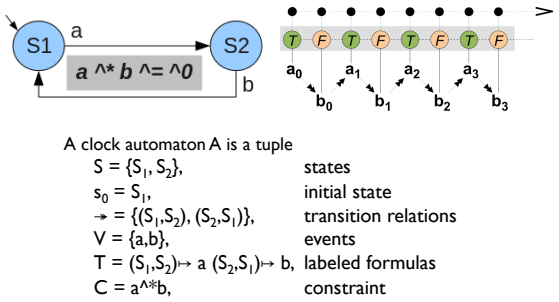

Figure 3 As a result of the above definition, the alternating automaton is decomposed into states $S = \{s_1, s_2\}$, variables $V = \{a, b\}$, transitions labelled by $T = \{(s_1, s_2) \mapsto a, (s_2, s_1) \mapsto b\}$ and constraint $C : (a \widehat{*} b) = \widehat{\mathbf{0}}$. Its control clock 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.

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 $\tau(s) : s \rightarrow_A s$

Usual 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
  $(\forall((s,s_1),(s,s_2)) \in \rightarrow_A \times \rightarrow_A)((s_1 = s_2) \vee (T_A((s,s_1)) \mathbin{\widehat{*}} T_A((s,s_2)) \; is \; null))$
- 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
  $\tau(s) \mathbin{\widehat{+}} (\Sigma_{(s,t) \in \rightarrow_A}(\mathbf{T}_A((s,t)))) = \widehat{\mathbf{1}}_A$
- 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)

Discussion

A constraint in an automaton is a specification constraint. For instance $x := a + b$ specifies an adder behavior such that for each t signals a, b, x are absent, and when one of them is present, all of them are present and $x_t := a_t + b_t$: we have the constraint that a, b and c should be synchronous. In this specification, physical availability, depending on value arrivals, is distinguished from logical behavior. For an asynchronous implementation (or a "microcode" description) then a container can handle asynchronous arrivals of a, b and insure synchronous execution of $x := a + b$.

A.3 Regular expressions

We define the algebra of regular expressions which will be used to abstract constrained automata or represent there null formula [16].

▶ **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

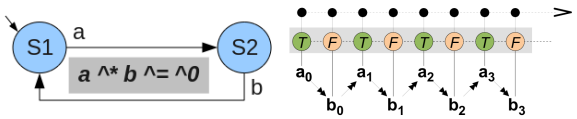

Figure 4 The constraint of the alternating automaton $C = (a\widehat{*}b)$ can equivalently be expressed as the regular event expression $((a\widehat{-}b) + (b\widehat{-}a))*$

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 [20]. 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 "$S_1.S_2$", or "$S_1; S_2$" in PSL; union "$S_1 + S_2$", $S_1|S_2$ in PSL; star "$S*$"; positive $S+ = S; S*$; option $S? = 1 + S$; fusion $S : T$, synchronous product $S|T$, interleaving and subsets.
- 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 [19] 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) $S[m..] = S[m]; S[..] = S[m]; S*$

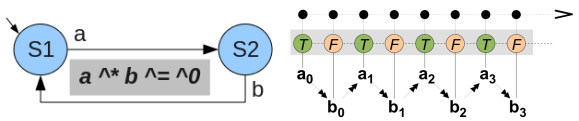

Figure 5 The alternating automaton could itself be alternatively expressed by the composition of two regular event expression consisting of the negation of the constraint $(a\widehat{*}b)*$ and of its transitions $(a.b)*$, which yields $((a\widehat{-}b).(b\widehat{-}a))*$.

A.4 Synchronous product

The global behavior of a component such as a thread can be defined by the composition of features belonging to this component. The synchronous product $\between$ 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 $A \between B$ corresponds to the conjunction of the behaviors specified by each of them. $A \between B$ 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 inital 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: $A \between A = A$.

Normal form outline

Constrained automata have normal forms; a constructive definition is outlined as follow:

- $\mathbf{1} = (\{s\}, s, \emptyset, \emptyset, \emptyset, \widehat{\mathbf{0}})$ is a normal form
- if $A = (S_A, s_0, \rightarrow_A, V_A, T_A, \mathbf{C}_A)$ is a normal form then
  - *Adding a signal variable preserves the normal form:* for every signal variable x that is not in V_A, the automaton $B = (S_A, s_0, \rightarrow_A, V_A \cup \{x\}, T_A, \mathbf{C}_A)$ is a normal form automaton;
  - *Adding a state reachable from an existing state with a non null trigger preserves the normal form:* given a state $s \notin S_A$, for every state $s_i \in S_A$, for every control boolean formula $h \in \mathbf{F}_{V_A,S_A}$ that can be used as a trigger for s_i (i.e. $h\widehat{*}[s_i]$ is not null in A), let
    - h_i be the normal form formula equal to $h\widehat{*}[s_i]$,
    - $S_B = (S_A \cup \{s\})$,
    - $\rightarrow_B = (\rightarrow_A \cup \{(s_i, s)\})$,
    - $T_B = (T_A \cup \{((s_i, s), h_i)\})$,

    the automaton $B = (S_B, s_0, \rightarrow_B, V_A, T_B, \mathbf{C}_A)$ is a normal form automaton;
  - *Adding a local constraint that do not nullify it to the trigger of a transition preserves the normal form:* for every transition $(s_i, s_j) \in \rightarrow_A$, for every control boolean formula $h \in \mathbf{F}_{V_A,S_A}$, let
    - $h_{i,j}$ be the normal form formula equal to $T_A((s_i, s_j))\widehat{*}h$,
    - T_B be the labelling function equal to T_A everywhere unless $T_B((s_i, s_j)) = h_{i,j}$,

    the automaton $B = (S_A, s_0, \rightarrow_A, V_A, T_B, \mathbf{C}_A)$ is a normal form automaton iff $h_{i,j}$ is not null;
  - *Adding a constraint that do not nullify any existing trigger preserves the normal form:* for every control boolean formula $h \in \mathbf{F}_{V_A,S_A}$, let
    - $\mathbf{C}_B$ be the normal form formula equal to $\mathbf{C}_A \widehat{+} h$,
    - T_B be the labelling function that associates to every transition $(s_i, s_j) \in \rightarrow_A$ the normal form formula equal to $T_A((s_i, s_j))\widehat{-}h$,

    the automaton $B = (S_A, s_0, \rightarrow_A, V_A, T_B, \mathbf{C}_B)$ is a normal form automaton iff none of the triggers is null in B (formally, $\forall (s_i, s_j) \in \rightarrow_A, T_B((s_i, s_j))$ is not null in B).
- An automaton is a normal form automaton only when it can be built by iterative application of the above rules.

Note that a normal form automaton is not necessarily deterministic.

The normal form synchronous product is simply defined with the help of a recursive process starting at inital state.