Formal semantics of behavior specifications in the architecture analysis and design language standard

Loïc Besnard, Thierry Gautier, Paul Le Guernic, Clément Guy, Jean-Pierre Talpin, Brian Larson, Etienne Borde

Formal Semantics of Behavior Specifications in the Architecture Analysis and Design Language Standard

Loïc Besnard, , Thierry Gautier, Paul le Guernic, Clément Guy, Jean-Pierre Talpin, Brian Larson , Etienne Borde

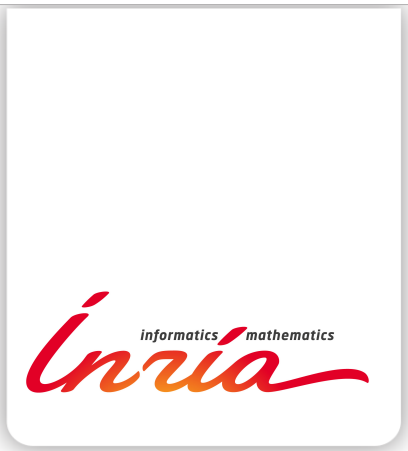
RESEARCH REPORT

N° 8950

August 2016

Project-Teams TEA

ISSN 0249-6399 ISRN INRIA/RR--8950--FR+ENG



Formal Semantics of Behavior Specifications in the Architecture Analysis and Design Language Standard

Loïc Besnard*, , Thierry Gautier†, Paul le Guernic†, Clément Guy†, Jean-Pierre Talpin†, Brian Larson ‡, Etienne Borde §

Project-Teams TEA

Research Report n° 8950 — August 2016 — 2 pages

Abstract: In system design, an architecture specification or model serves, among other purposes, as a repository to share knowledge about the system being designed. Such a repository enables automatic generation of analytical models for different aspects relevant to system design (timing, reliability, security, etc.). The Architecture Analysis and Design Language (AADL) is a standard proposed by SAE to express architecture specifications and share knowledge between the different stakeholders about the system being designed. To support unambiguous reasoning, formal verification, high-fidelity simulation of architecture specifications in a model-based AADL design workflow, we have defined a formal semantics for the behavior specification of the AADL, the presentation of this semantics is the aim of this paper.

Key-words: System design, real-time systems, architecture modeling, standards, formal semantics, automata theory

* Irisa
† Inria
‡ FDA Scholar in residence, KSU
§ Telecom ParisTech

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Sémantique formelle des spécifications fonctionnelles du standard Architecture Analysis Design Language (AADL)

Résumé : En génie système, une spécification ou un modèle d'architecture occupe un rôle central pour partager et renseigner la connaissance du système pendant sa conception. cette base de connaissance permet de générer automatiquement des modèles analytiques pour valider différentes problèmiques de conception (temps, sureté, sécurité, etc). AADL (architecture analysis and design standard) est un standard de la SAE pour spécifier des modèles d'architecture et ainsi partager les spécifications d'un système en construction avec ses différents intervenants. Afin de permettre un raisonnement précis et univoque au moyen de spécifications AADL, nous présentons une sémantique formelle des spécifications fonctionnelles au coeur du standard: les annexes comportementales.

Mots-clés : Génie système, systèmes temps réel, modèles d'architecture, standard, sémantique formelle, théorie des automates

1 Introduction

In system design, an architecture specification serves several important purposes. First, it breaks down a system model into manageable components to establish clear interfaces between them. In this way, complexity becomes manageable by hiding details that are not relevant at a given level of abstraction. Clear, formally defined, component interfaces allow us to avoid integration problems at the implementation phase. Connections between components, which specify how components affect each other, help propagate the effects of a change in one component to the linked components.

Most importantly, an architecture model is a repository to share knowledge about the system being designed. This knowledge can be represented as requirements, design artefacts, component implementations, held together by a structural backbone. Such a repository enables automatic generation of analytical models for different aspects relevant to system design, such as timing, reliability, security, performance, energy, etc. Since all the analyses are generated from the same source, the consistency of assumptions w.r.t. guarantees, of abstractions w.r.t. refinements, used for different analyses, becomes easier, and can be properly ensured in a design methodology based on formal verification and synthesis methods.

Several standards for modeling embedded architectures have emerged in recent years: the SAE AADL[1] [1], SysML[2], and UML MARTE [17]. Each of them represents different design approaches, embodies different concepts, and serves different purposes. We focus on the AADL, and the scope and precision of concepts defined by this standard, to define a formal semantics for a significant subset of its behavioral specification annex language, often called 'BA'. Just as non-functional properties (timing, performance, energy, security properties), such descriptions can be attached to threads, processes, or any object of the standard (bus, sensor, actuator, port) to formally specify its behavior, as specified in the standard (e.g. a bus), or refine it (e.g. as an AFDX bus).

Since it began being discussed in the AADL standard committee, the formal semantics defined in this article evolved from a synchronous model of computation and communication [4] to a semantic framework for time and concurrency in the standard: asynchronous, synchronous or timed, to serve as a reference for model checking, code generation or simulation tools uses with the standard. These semantics are simple, relying on the structure of automata present in the standard already, yet provide tagged, trace semantics framework to establish formal relations between (synchronous, asynchronous, timed) usages or interpretations of behavior.

2 Case study of an Adaptive Cruise Control

To illustrate the definition and use of a formal semantics for the AADL behavior annex, we consider the case study of an Adaptive Cruise Control (ACC) system, Figure 1.

> An Adaptive Cruise Control System is an optional cruise control system for road vehicles that automatically adjusts the vehicle speed to maintain a safe distance from vehicles ahead. [...] Control is imposed based on sensor information from on-board sensors [19].

ACC systems implement two main functions:

1. an ACC can automatically sustain a preset speed (as a conventional Cruise Control system), and
2. an ACC can adapt the vehicle's speed to maintain a safe distance with other vehicles ahead of it and prevent collisions.

To implement these functions, the ACC requires information from different sensors: speedometer, laser or radar to detect vehicles or obstacles ahead, and wheel sensor to adjust the focus point of the laser or radar. It receives commands from the driver through buttons allowing to set the preferred speed and to activate or deactivate the system.

Depending on the situation (presence of an obstacle or not, activation of the cruise control or not), it computes the acceleration and deceleration for the vehicle to reach the needed speed: the preferred speed of the driver if there is no obstacle and the cruise control is on, or the speed of the vehicle ahead if one is detected. Finally, it acts on the vehicle speed through its brakes and throttle.

An ACC is a safety-critical system. Hence, in addition to meeting its functional requirements, its design must satisfy design correctness objectives that concern several aspects specified in its architecture model:

[1] `http://www.aadl.info/aadl/currentsite/`

[2] `http://www.omg.org/spec/SysML/1.4/`

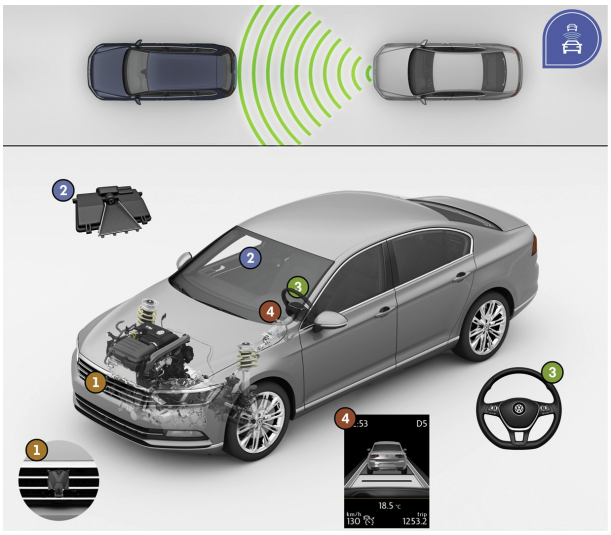


Fig. 1 Adaptative Cruise Control

- from the timing and scheduling perspective, all threads must meet their deadlines and the overall task of reacting to the presence or absence of an obstacle must meet a maximum reaction time;
- from the logical perspective, the system must be free of deadlock and race condition;
- from the security perspective, critical software components (processes or systems) must be protected from less critical components, thus executed on dedicated processors;
- from the consumption perspective, the system must draw minimal power from the car battery, thus processors must run on the minimal possible frequency;
- from the cost perspective, the overall cost of the system should be minimal, which means minimizing hardware component size and complexity.

3 Architecture Analysis and Design Language

AADL [1] is SAE International standard AS5506C, dedicated to modeling embedded real-time system architectures. As an architecture description language, based on a component modeling approach, AADL describes the structure of systems as an assembly of software components allocated on execution platform components together with constraints and properties, including timing ones.

3.1 Architecture

In AADL, three distinct families of components are provided:

- software application components which include process, thread, thread group, subprogram, and data components,
- execution platform components that model the hardware part of a system including (possibly virtual) processor, memory, device, and (possibly virtual) bus components,
- composite components (systems).

Figure 2 presents an overview of an ACC system, consisting of:

- devices, such as sensors (speedometer, radar, wheel sensor), console with buttons and display, throttle and brakes;
- buses allowing subsystems to communicate with each other and with devices;
- controller and console subsystems.

Each subsystem in Figure 2 consists of hardware components, such as processors, memories and buses; and software components: processes containing threads. Figure 3 presents the controller subsystem and its components: one processor, one memory, one bus connecting the processor and the memory and

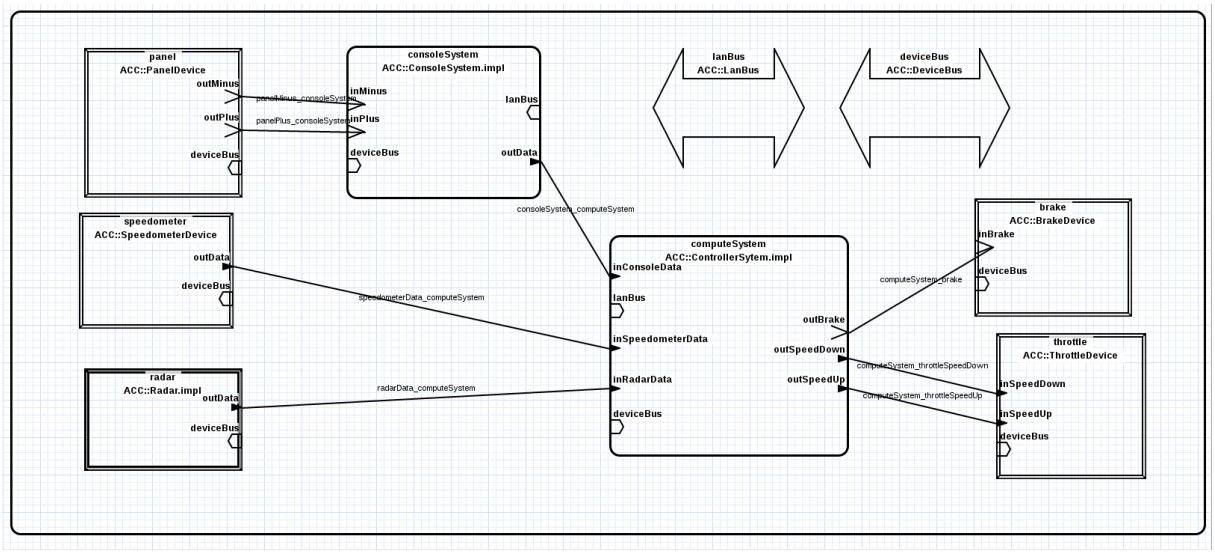


Fig. 2 Overview of the Adaptive Cruise Control system modeled with AADL.
Double-lined rectangles represent devices, double-arrows buses and rectangles with rounded corners systems and subsystems.

one controller process. The controller process itself contains four threads, one for each sensor, and the `ComputeActionThread`, which is responsible for sending speed up, slow down or complete stop signals to the throttle and brakes of the vehicle.

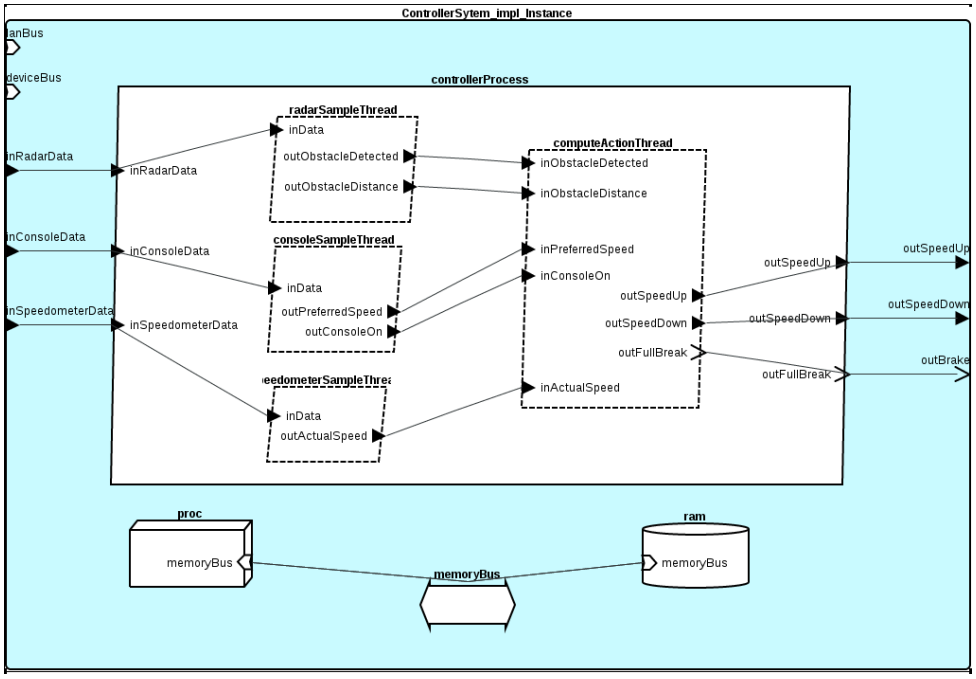


Fig. 3 Controller subsystem of the Adaptive Cruise Control system modeled with AADL.
Rectangles represent processors, double-arrows buses, cylinders memories and rhombuses processes and threads.

The AADL components communicate via data, event, and event data ports. On Figure 2, ports are represented by arrows, and connections between ports by lines. Data ports are represented using filled arrowheads and event ports using empty arrowheads.

Each component has a type, which represents the functional interface of the component and externally observable attributes. Each type may be associated with zero, one or more *implementation(s)* that describe the contents of the component, as well as the *connections* between components.

3.2 Properties

AADL properties provide various information about model elements of an AADL specification. For example, a property `Dispatch_Protocol` is used to provide the dispatch type of a thread. Property associations in component declarations assign a particular property value, e.g., `Periodic`, to a particular property, e.g., `Dispatch_Protocol`, for a particular component.

For example, Listing 1 presents such properties attached to the `ComputeActionThread` thread.

Listing 1 Timing and scheduling properties of the `ComputeActionThread` thread implementation

```
[2]thread [2]implementation ComputeActionThread.impl
  [2]properties
    -- periodic thread
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
    -- thread deadline
    Deadline => 40 ms;
    -- thread WCET
    Compute_Execution_Time => 20 ms;
[2]end ComputeActionThread;
```

3.3 AADL timing execution model

Threads are dispatched periodically, triggered by the arrival of data or events on ports, or from the arrival of a subprogram call (from another thread), depending on the thread type. Three event ports are predeclared: `dispatch`, `complete` and `error` (Figure 4).

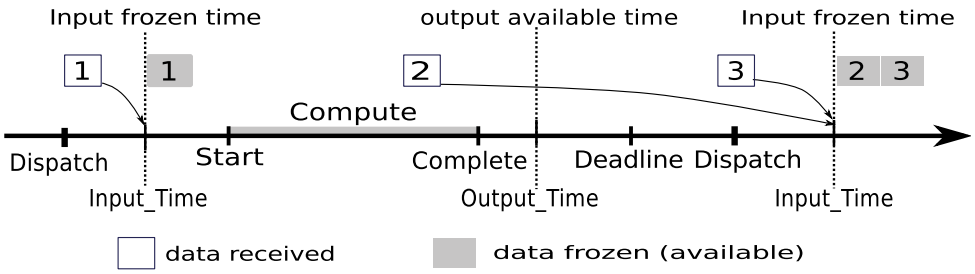


Fig. 4 Execution time model for an AADL thread

A thread is activated to perform a computation at *start* time, and has to be finished before the *deadline*. A *complete* event is sent at the end of the execution. The received inputs are frozen at a specified time (*Input_Time*), by default the *dispatch* time. This implies that the content of a dispatched port does not change during the execution of a thread dispatch, even though the sender may send new values in its input FIFO. For example, values 2 and 3 (Figure 4) arriving after the first *Input_Time* will not be processed until the next *Input_Time*. As a result, the performed computation is not affected by a new input arrival until an explicit request for input (another dispatch). Similarly, the output is made available to other components at a specified point of *Output_Time*, by default at *complete* (resp., *deadline*) time if the associated port connection is immediate (resp., delayed) communication.

4 A formalization using constrained automata

We define the model of computation and communication of a behavior specification by the synchronous, timed or asynchronous traces of automata with variables [18]. These constrained automata are derived

from *polychronous automata* defined within the polychronous model of computation and communication [12]. Automata define a behavior using transitions. A transition is composed of an initial state, a guard, an action, a final state. The guard and action of a transition are defined using logical formulas. The logical formula of the guard must be true for the transition to occur.

4.1 Vocabulary

These multi-sorted logical formulas are defined on the vocabulary W of AADL constants and of the states S, variables V, connections and ports P defined in the lexical scope of the denoted AADL object. An identifier w in W has a type $T = \mathit{typeof}(w)$ and is valuated on the corresponding domain $\mathbb{D}_T$, e.g., Booleans, integers or reals, $\mathbb{D} \supseteq \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R}$.[3]

We write $\mathbb{D}_x$ for the value domain of a typed identifier x. The domain of a port identifier p of type T is defined by $\mathbb{D}_p = \mathbb{D}_T^{\perp} = \mathbb{D}_T \cup \{\perp\}$. The bottom sign $\perp$ denotes the absence of a value at the given step of execution. A port value is said absent if the port is not frozen and its value is neither read or written.

4.2 Formulas

The set of typed formulas F_W on the vocabulary W is an algebraic set of terms that denotes the conditions, actions and constraints of an AADL object of vocabulary W. It is defined by induction from:

- Constants 0 (*false*), to mean "never", and 1 (*true*), to mean "always". Always is discrete, relative to the vocabulary W.
- Atoms w of W, to mean the value of an identifier w.
- Unitary expressions:
  - $\hat{}p$ is the clock of p: a Boolean that denotes the presence of a value on a frozen port p, i.e., $p \neq \perp$;
  - $@p$ is the date of p: a real number that denotes the time of an event present on a port p;
  - v' denotes the next value of a variable v;
  - $\neg f$ denotes the complement of formula f, for all f in F_W.
- Binary expressions f *op* g:
  - for all Boolean formula f, g in F_W and Boolean operators $\vee$, $\wedge$, $\Rightarrow$, etc. (in particular, $f - g = f \wedge \neg g$);
  - for all numerical formula f, g in F_W and numerical operators $+$, $-$, $*$, $/$, $\%$, $=$, $<$, etc.

A formula f is the denotation of a well-typed AADL condition or action. It is hence assumed to be a well-typed, multi-sorted, logical expression. Ill-typed expressions do not define formula.

Example

The formula $\hat{}a \wedge \hat{}b = 0$ stipulates that the ports a and b should never carry a value (sent or dispatched) at the same logical period of time. In the AADL, it refers to the condition "*on dispatch* a" of an object possibly triggered by a or b and allows to explicit the status of a being frozen and b not (or, alternatively, empty, with "*not* b'*fresh*" or "*not* b'*count* $= 0$").

Conversely, $\hat{}a = \hat{}b$ expresses the synchrony of a and b at any step of execution. It can be refined by the real-time constraint $d \leq @a,\ @b < d + p$, where d is the date of the behavior's dispatch and p its period.

[3] Although BA supports other types (strings, enumerations, records, arrays) our formalization focuses on numbers and Booleans without loss of generality.

4.3 Model

A model m is a function $W \to \mathbb{D}_W$ from a vocabulary W to its domain of valuation $\mathbb{D}_W$ that is true for a formula f of F_W, written $m \models f$. A timed model $m^@$ is a function $W \to \mathbb{R} \times \mathbb{D}_W$ associating also each event with a date, that the formula must satisfy as well.

4.4 Automaton

The meaning of a behavior annex is defined by an incomplete automaton with variables $A = (S_A, s_0, V_A, P_A, T_A, C_A)$ defined by:

- S_A, the set of states of the automaton A, s_0 is the initial state.
- V_A, the set of local variables of the automaton A. (${V_A}'$ designates the set of next values v' for all $v \in V_A$)
- P_A, the set of ports of automaton A, both inputs I_A and outputs O_A, $P_A = I_A \cup O_A$.
- F_A is the set of Boolean formulas of A, defined on the vocabulary $W_A = S_A \cup V_A \cup P_A$.
- The transition function $T_A \in S_A \times F_A \to F_A \times S_A$ defines the transition system of A.
  - The source formula of a transition is its guard g, defined on V_A and I_A.
  - The target formula of a transition is its action f, defined from V_A and P_A.
- $C_A \in F_A$ is the constraint of A. It must always equal 0. It is a formula that denotes the invariants (properties, requirements) of the denoted AADL object in a form of a logical formula.

Since variables V are private to the automaton, a transition function T_A is equivalent to one in $X_A = Q_A \times F_P \to F_P \times Q_A$ over extended states $Q_A = S_A \times \mathbb{D}^{V_A}$ for all valuations $\mathbb{D}^{V_A} = \prod_{v \in V_A} \mathbb{D}_v$ of the variables V_A: for all transition $(s, g, f, d) \in T_A$, for all model $m \in (V_A + {V_A}') \to \mathbb{D}^{V_A}$, we have $((s, m(V_A)), m(g), m(f), (d, m({V_A}')) \in X_A$ and, for all $v' \in {V_A}'$ undefined in f, $m(v') = m(v)$.

Example

A behavior alternating between two states of receiving a's and b's can be represented (Figure 5) by a transition system defined on the vocabulary $\{a, b\}$ with two complete states s_0 and s_1 (complete states are observable states—see Section 5) and two transitions:

- $(s_0, \hat{}a \ - \ \hat{}b, \mathit{true}, s_1)$ denotes the transition from $s0$ to $s1$ if port a carries a value and b does not;
- $(s_1, \hat{}b \ - \ \hat{}a, \mathit{true}, s_0)$ denotes the transition from $s1$ to $s0$ if port b carries a value and a does not.

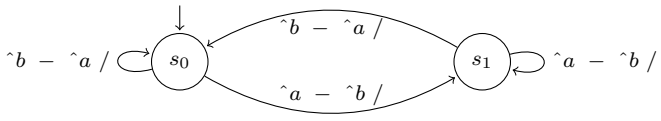


Fig. 5 Alternating behavior

The role of a constraint formula such as $\hat{}a \ \wedge \hat{}b = 0$ is to guarantee a property by all models of the incomplete automaton. For instance:

- if an a is received in state s_1, or a b in state s_0, both the automaton and the constraint allow it: the event is consumed and the automaton remains in the same state;
- if both a and b are received in either s_0 or s_1 then the transition is denied by the constraint.

4.5 Properties

- The control clock 1_A of an automaton A is defined by the sum (union) of its port clocks $1_A = \sum_{p \in P_A} \hat{} p$.
- The trigger $tick_A(s) = \sum_{(s,g,f,d) \in T_A}(g)$ of a state s is defined by the upper bound of guard formulas g from s.
- The stuttering clock of a state s is defined by $\tau_A(s) = 1_A - ((s * C_A) + tick_A(s))$. It means that an automaton A is silent in state s if and only if its model m satisfies the constraint C_A in state s and no guard can be triggered from s with m.

4.6 Product

The synchronous product of two automata $A = (S_A, s_0, V_A, P_A, T_A, C_A)$ and $B = (S_B, t_0, V_B, P_B, T_B, C_B)$ is defined by $A \mid B = (S_{AB}, (s_0, t_0), V_{AB}, P_{AB}, T_{AB}, C_{AB})$ with

$$
\begin{aligned}
S_{AB} &= S_A \times S_B \\
V_{AB} &= V_A \cup V_B \\
P_{AB} &= P_A \cup P_B \\
C_{AB} &= C_A \vee C_B \\
T_{AB} &= \{((s_1, t_1), g_1 \wedge g_2, f_1 \wedge f_2, (s_2, t_2)) \mid (s_1, g_1, f_1, t_1) \in T_A \wedge (s_2, g_2, f_2, t_2) \in T_B\}
\end{aligned}
$$

Product is commutative, associative, has neutral element $(\{s\}, s, \emptyset, \emptyset, \emptyset, 0)$ and is idempotent for deterministic automata.

Example

The synchronous composition of two automata A and B communicating through an immediate connection of port p can be represented by the synchronous product of A and B with the automaton representing a point-to-point one-place first-in-first-out buffer (Figure 6). A queue of size n can be defined by the product of n copies of $FIFO_1$.

$$
\begin{aligned}
FIFO_1 &= (\{s_0, s_1\}, s_0, \{v\}, \{p_A, p_B\}, T_{FIFO_1}, 0) \\
T_{FIFO_1} &= \{(s_0, \hat{} p_A, v' = p_A, s_1), (s_1, true, p_B = v, s_0)\}
\end{aligned}
$$

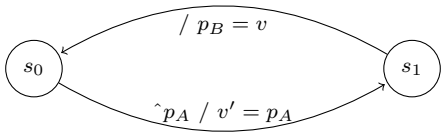


Fig. 6 $FIFO_1$

4.7 Small step

The model m of a transition in an automaton A consists of a pre-condition pre(m) defined on input ports $I \to \mathbb{D}_I^{\perp}$ and state variables $V \to \mathbb{D}_V$ and a post condition post(m) defined on output ports $O \to \mathbb{D}_O^{\perp}$ and next values of variables $V' \to \mathbb{D}_V$.

A *small step* of an automaton A from state s to state t is defined by a model m of A that satisfies its constraint C_A, written $m \models \neg C_A$, and both the guard g and action f of a transition (s, g, f, t) of A, written $m \models g \wedge f$.

Example

A small step of an automaton denotes an atomic and untimed execution step of the denoted behavior. For instance, the model $m = \{(v, 0), (v', 0), (p_A, 0), (p_B, 0)\}$ is a small step of the automaton $FIFO_1$ from s_0 to s_1: it satisfies both guard $m \models \hat{}\, p_A$ and action $m \models v' = p_A$.

4.8 Big step

Let $n > 1$, $q_1 = (s_1, r_1)$ and $q_n = (s_n, r_n)$ two extended states of an automaton A with complete states $s_1, s_n \in S_A$ and variable valuations $r_1, r_n \in \mathbb{D}^{V_A} \simeq V_A \to \mathbb{D}_{V_A}$ (note that it may be the case that $q_n = q_1$). A *big step* of automaton A from s_1 to s_n is defined by a model $m \in P_A \to \mathbb{D}^{\perp}_{P_A}$ that, for all $1 \leq i < n$ satisfies:

- $\mathrm{pre}\, r_{i+1}(v) = \mathrm{post}\, r_i(v')$ for all $v \in V_A$ (the *next* variable values v' at step i, $\mathrm{post}\, r_i(v')$, are the regular variable values v at step $i+1$, $\mathrm{pre}\, r_{i+1}(v)$)
- $m_i = r_i \uplus m$
- $m_i \models \neg C_A$
- $(s_i, g_i, f_i, s_{i+1}) \in T_A$ and $m_i \models g_i \wedge f_i$
- $\mathrm{pre}\, r_i(v) = \mathrm{post}\, r_i(v')$ for all $v' \in V'_A$ not occurring in f_i and g_i
- s_i is an *execution state* if $1 < i < n$ (an *execution state* is a non observable, internal state—see Section 5).

We write $m, s_1 \models A, s_n$ to mean that m is the model of a big step of A from s_1 to s_n.

Example

For instance, the model $m = \{(p_A, 0), (p_B, 0)\}$ is a big step of the automaton $FIFO_1$ from s_0 back to s_0. It abstracts the meaning of A over its port interface for the corresponding valuation of its local variables $\{(v, 0), (v', 0)\}$ that satisfies the guard and action.

4.9 Synchronous and asynchronous trace

A *synchronous trace* $B \in P_A \to (\mathbb{D}^{\perp}_{P_A})^*$ of an automaton A is a finite sequence of valuation over P_A obtained by concatenating the codomains of successive big steps. The length of B is denoted $|B|$. The set of synchronous traces of an automaton A from its initial state s_0 is defined as:

$$T(A, s_0) = \{B \in P_A \to (\mathbb{D}^{\perp}_{P_A})^* \mid 0 \leq i < |B|,\ m_i, s_i \models A, s_{i+1} \wedge \forall x \in dom(B), (B(x))_i = m_i(x)\}$$

An *asynchronous trace* $B^{\#} \in P_A \to (\mathbb{D}_{P_A})^*$ is the abstraction of a synchronous trace $B \in P_A \to (\mathbb{D}^{\perp}_{P_A})^*$ obtained by the removal of all absence marks $\perp$. For a sequence s in $(\mathbb{D}^{\perp})^*$, we denote by $s_{/\perp}$ the projection of s on $\mathbb{D}^*$. The set of asynchronous traces of an automaton A from its initial state s_0 is defined as:

$$T^{\#}(A, s_0) = \{B \in P_A \to (\mathbb{D}_{P_A})^* \mid C \in T(A, s_0) \wedge \forall x \in dom(B), B(x) = C(x)_{/\perp}\}$$

4.10 Timed step and timed trace

A *timed step* of an automaton A from state s to state t is defined by a timed model $m^{@}$ defined on W_A that satisfies its constraint and the guard g and action f of a transition (s, g, f, t) of A. For all w in W_A, $m^{@}(w)$ refers to the value of w in $m^{@}$ and $m^{@}(@w)$ refers to the date of w in m.

A *timed trace* $B^{@} \in P_A \to (\mathbb{R} \times \mathbb{D}^{\perp}_{P_A})^*$ of an automaton A is defined by the concatenation of the codomains of successive timed steps $(m_i^{@})_{i \geq 0}$ of A such that for all $0 \leq i < j$, for all x in $dom(m_i^{@})$, for all y in $dom(m_j^{@})$, $m_i^{@}(@x) < m_j^{@}(@y)$. A timed trace $B^{@}$ is therefore the refinement of a synchronous trace $B \in P_A \to (\mathbb{D}^{\perp}_{P_A})^*$ associating each event in B with a date.

5 Behavior Annex Model

BA provides an extension to AADL to associate functional behavior specifications with AADL components. A behavior is expressed by transition systems with conditions and actions [2]. Actions can be abstract, e.g., describe the consumption of time or resources, describe error scenarios. They can be refined to simulate and define the functional behavior of the AADL component using an imperative action language.

This section first presents how we formally express the meaning of a behavior annex (through an automaton). Then, the different elements of the behavior annex are defined (transition system, action and expression language, interaction protocols, etc.) and their formal semantics given.

5.1 Formalization

Formally, the meaning of a behavior annex is defined by the axiomatic, denotational and operational interpretation of constrained, incomplete, automata with variables $A = (S_A, s_0, V_A, P_A, T_A, C_A)$ such as defined in Section 4. The sets S_A, V_A, P_A represent the states (including the `initial state` s_0), variables and ports of A. The guard, action and constraints of its transitions T_A and constraints C_A are denoted by multi-sorted logical formula F_A.

F_A is defined over the vocabulary W_A available in the scope of a behavior annex: AADL value constants, port, state, and variable names. They are combined using AADL logical operators and numeric operators. Operators that are specific to the model of computation and communication of a given behavior annex are $\hat{}p$, a Boolean value to mean the presence of a value on port p under synchronous interpretation (i.e., $p \neq \perp$); and $@p$, a numeric value to mean the time of an event on p, under timed interpretation.

The transition system of an automaton A is defined by the function $T_A \in S_A \times F_A \rightarrow F_A \times S_A$ whose quadruples (s, g, a, t) define the source state s, guard formula g, action formula a and target state t of a specified transition.

In the reminder of the section, we present each element of the behavior annex, with examples using our motivating case study, and the semantics of the element with respect to our framework.

5.2 Transition system

The AADL behavior annex defines a transition system (an extended automaton) described by three sections: variables declarations, states declarations, and transitions declarations. This transition system of the behavior annex is not to be confused with the transition system of the automaton interpreted to give its meaning to a behavior annex. On the one hand, we have a transition system which is part of the behavior annex, and on the other hand an automaton which is used to express the meaning of the whole behavior annex.

The automaton A of a `behavior_annex` instance is defined on the vocabulary consisting of its private variables `behavior_variable`, of its states `behavior_state`, and ports of its parent component. Its transition system T_A is the union of the transitions specified by a `behavior_transition`.

```
behavior_annex ::=
     [ variables { behavior_variable }+ ]
     [ states { behavior_state }+ ]
     [ transitions { behavior_transition }+ ]
```

We first describe how a thread can be described in the Adaptive Cruise Control model using the behavior annex. Then we present the three different sections of the transition system of the behavior annex, detailing the example. Finally, we give the formal semantics of the different elements of the transition system.

5.2.1 Transition system of a thread of the Adaptive Cruise Control

In the Adaptive Cruise Control (ACC) system, the `ComputeActionThread` thread is responsible for processing the correct behavior the system should adopt (slow down, speed up or keep the speed constant) depending on the situation. Figure 7 pictures the transition system describing the behavior of the `ComputeActionThread` thread.

For readability sake, conditions and actions have been omitted. In the case of this transition system, conditions are tests on input signals (are they present or not) and on variables (value comparison), and actions are of two kinds: either the sending of a signal through one of the output ports of the thread; or the computation of an intermediate value, such as the vehicle speed relative to the obstacle one, or the acceleration/deceleration needed to reach a given speed, and its assignment to a variable.

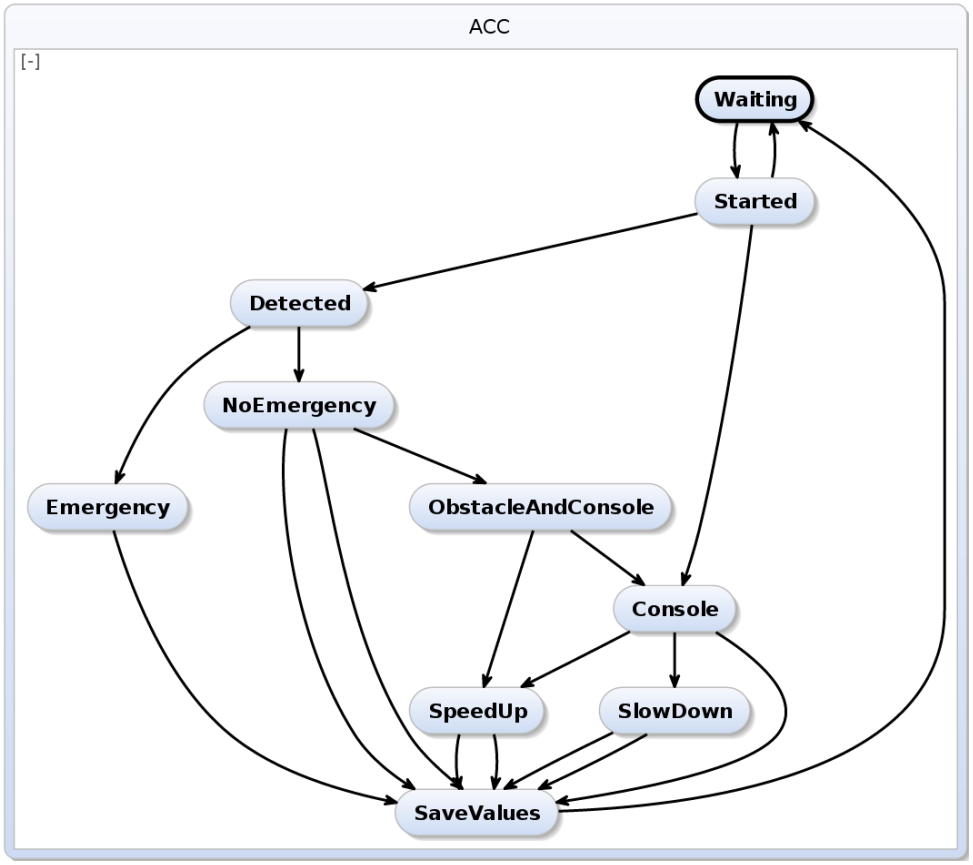


Fig. 7 Transition system for the `ComputeActionThread` thread.

The state transition system starts in the `Waiting` state, waiting for its thread to be periodically dispatched, and to pass in `Started` state. The `Waiting` state is a `complete` one, that is, a state in which a thread pauses its execution when entering in, waiting for a new dispatch.

After entering the `Started` state, depending on the inputs, the state transition system can pass in `Detected` (the system detected an obstacle) or `Console` state (the system did not detect an obstacle and the cruise control is on), or go back to the `Waiting` state (the system did not detect an obstacle and the cruise control is off).

In the `Detected` state, the system must decide the emergency of the situation: if the obstacle is in an unsafe range, the system goes into the `Emergency` state and its next transition will send a signal to brakes in order to stop the vehicle; if the obstacle is outside this range, the system enters the `NoEmergency` state and then determines whether it should slow down to adapt its speed to the obstacle speed, speed up or keep the speed constant (each transition sending the corresponding signal to the throttle after the computation of the needed acceleration/deceleration). The same happens in the `Console` state depending on the current speed of the vehicle and the speed preset by the driver.

After saving useful values (e.g., current speed, current obstacle speed and distance in the `SaveValues` state, the state transition system returns in the `Waiting` state, waiting for the next dispatch of its thread.

5.2.2 Variables section

The variables section of the transition system of a behavior annex declares identifiers that represent variables with the scope of the behavior annex subclause. Local variables can be used to keep track of intermediate results within the scope of the annex subclause. They may hold the values of out parameters on subprogram calls to be made available as parameter values to other calls, as output through enclosing out parameters and ports, or as value to be written to a data component in the AADL specification. They can also be used to hold input from incoming port queues or values read from data components in the AADL specification. They are not persistent across the various invocations of the same behavior annex subclause. Listing 2 presents a sample of the variables section of the behavior annex of thread `ComputeActionThread`.

Listing 2 Sample of the variables section of the behavior annex of the `ComputeActionThread` thread.

```
[2]variables
...
  --vv : vehicle speed (from accelerometer)
  actual_speed: Base_Types::Float;
  --vv' : previous vehicle speed
  previous_actual_speed: Base_Types::Float;
  --vo : obstacle speed (vv-vv'+(d-d'/T))
  obstacle_speed: Base_Types::Float;
  --vo' : previous obstacle speed
  previous_obstacle_speed: Base_Types::Float;
...
```

5.2.3 States section

The states section declares all the states of the automaton. Some states may be qualified as `initial` state (thread halted), `final` state (thread stopped), or `complete` state (thread awaiting for dispatch), or combinations thereof. A state without qualification is referred to as `execution` state. A behavior automaton starts from an initial state and terminates in a final state. A complete state acts as a suspend/resume state out of which threads and devices are dispatched. Complete states thus correspond (with initial and final states) to the observable states of the behavior, in which computations are "paused", inputs read and outputs produced. Listing 3 shows an excerpt of the states section of the `ComputeActionThread` thread of our ACC example.

Listing 3 Sample of the states section of the behavior annex of the `ComputeActionThread` thread.

```
[2]states
  Waiting: [2]initial [2]complete [2]state;
  Started, Detected, ..., ComputeBreak: [2]state;
```

5.2.4 Transitions section

The transitions section defines transitions from a source state to a destination state. Transitions in a behavior automaton represent the execution sequence within a thread. A transition out of a complete state is initiated by a dispatch once the dispatch condition is satisfied. Transitions can be guarded by dispatch conditions, or execute conditions, and can have actions. Listing 4 presents three transitions (`t0`, `t1` and `t3`) from the transitions section of the behavior annex of the `ComputeActionThread` thread.

Dispatch conditions explicitly specify dispatch trigger conditions out of a complete state. A dispatch condition is a Boolean expression that specifies the logical combination of triggering events: arrival of an

Listing 4 Sample if the transitions section of the behavior annex of the `ComputeActionThread` thread.

```
[2]transitions
  --Periodically check input data
  t0 :
  Waiting -[[2]on [2]dispatch]-> Started {
    actual_speed:=inActualSpeed
  };
  --Detecting obstacles
  t1 :
  Started -[inObstacleDetected [2]and (actual_speed != 0)]-> Detected {
    obstacle_distance:=inObstacleDistance
  };
...
  --Stopping car in case of emergency
  t3 :
  Emergency -[]-> SaveValues {
    outFullBreak!
  };
...
```

event or event data on an event port or an event data port, receipt of a call on a provided subprogram access, or timeout event.

Execute conditions specify transition conditions out of an execution state to another state. They effectively select between multiple transitions out of a given state to different states. These conditions are logical expressions based on component inputs, subcomponent outputs, and values of data components, state variable values, and property constants. They can also result in catching a previously raised execution timeout exception.

If transitions have been assigned a priority number, then the priority determines the transition to be taken. The higher the priority number is, the higher the priority of the transition is. If more than one transition out of a state evaluates its condition to true and no priority is specified, then one transition is chosen non-deterministically. For multiple transitions with the same priority value the selection is also non-deterministic. Transitions with no specified priority have the lowest priority.

Each transition can have actions. Actions can be subprogram calls, retrieval of input and sending of output, assignments to variables, read/write to data components, and time consuming activities. An action is related to the transition and not to the states: if a transition is taken, the sequence of actions is performed and then the state specified as the destination of the transition becomes the new current state.

5.2.5 Transition Semantics

States of a behavior annex transition system can be either observable from the outside (`initial`, `final` or `complete` states), that is states in which the execution of the component is paused or stopped and its outputs are available; or non observable, execution states, that is internal states. The semantics of the AADL considers the observable states of the automaton. The set S_A of automaton A thus only contains states corresponding to these observable states and set T_A big-step transitions from an observable state to another (by opposition with small-step transitions from or to an execution state).

A transition `behavior_transition` has source state $s =$ `source_state_identifier`. Its guard formula g is defined by the translation of the expression `behavior_condition` as a logical formula. Its target state $d =$ `destination_state_identifier` is that of the transition system defined by the semantic function $T(s, d)$ (defined Section 5.4) applied to its action block `behavior_action_block`.

A `transition_identifier`, if present, is represented by a label L that names the clock of the transition. It is a (virtual) event considered present and true iff the guard formula of that transition holds and the constraint of the automaton is enforced: the transition $(L : s, g, f, d)$ is equivalent to the transition (s, g, f, d) with the constraint $\hat{}L \Leftrightarrow (\hat{}s \wedge g)$.

A `behavior_transition_priority`, if present, enforces a deterministic logical order of evaluation among transitions. A pair of transitions $(s[m], g1, f1, s1)$ and $(s[n], g2, f2, s2)$ from a state s and such that $m < n$ (to mean that m has higher priority as n) is equivalent to the transitions $(s, g1, f1, s1)$ and

$(s, g2 \wedge \neg g1, f2, s2)$: the guard formula of a prioritized transition is subtracted to all transition in the same state of lower or no priority.

```
behavior_transition ::=
  [ transition_identifier [ [ behavior_transition_priority ] ] : ]
  source_state_identifier { , source_state_identifier }*
    -[ behavior_condition ]-> destination_state_identifier
    [ behavior_action_block ] ;
```

5.3 Behavior Conditions

Behavior conditions that cause transitions may be either execute conditions or dispatch conditions.[4]

```
behavior_condition ::=  execute_condition | dispatch_condition
```

Execute conditions are Boolean-valued expressions, and may only be used in transitions leaving an execution (or initial) state. State machines may never 'stall' in execution states; there must always be an enabled, outgoing transition from an execution state. The `otherwise` condition occurs when no other execute condition of a transition leaving an execution state is true.

```
execute_condition ::= logical_value_expression | otherwise
```

Dispatch conditions can only be associated with transitions from a `complete` state. A thread scheduler evaluates dispatch conditions to determine when threads are dispatched. A dispatch trigger condition can be the arrival of events or event data on ports (expressed as a disjunction of conjunctions)or timeout.

Periodic dispatches are always considered to be implicit unconditional dispatch triggers on complete states and handled by dispatch conditions without dispatch trigger condition. This is the case for transition `t0` presented in Listing 4.

```
dispatch_condition  ::= on dispatch [ dispatch_trigger_condition ]
     [ frozen ( frozen_ports ) ]

dispatch_trigger_condition ::= dispatch_trigger_logical_expression
  | stop | timeout_catch
```

Dispatch can be triggered by arrival of events at an `event port` or event-data at an `event data port`. To provide flexibility while avoiding paradoxes, dispatch conditions may be a disjunction, of conjunctions, of event (data) arrival at event (data) ports. Dispatch can also be triggered by event arrival at the predeclared `Stop` port.

```
dispatch_trigger_logical_expression ::=
  dispatch_conjunction { or dispatch_conjunction }*

dispatch_conjunction ::= port_identifier { and port_identifier }*
```

Timeout catch is a dispatch trigger condition that is raised after the specified amount of time since the last dispatch or the last completion is expired.

```
  timeout_catch ::= timeout
    [ [ ( port_identifier { or port_identifier }*  ) ] behavior_time ]
```

[4] The grammar for `behavior_condition`, here, is slightly simplified from that in the BA standard.

5.3.1 Behavior Condition Semantics

A `dispatch_condition` is represented by a guarding formula g that is formed by referring to the clock $\hat{}p$ of the logical combination of ports specified as its `dispatch_trigger_condition`.

An `execute_condition` is represented by a guarding formula that encodes its `logical_value_expression` using the current state of its persistent variables V. The `otherwise` clause is handled as the guard of least priority. The `otherwise` guard, if present in a transition leaving execution state s, applies if none of the guards from other transitions leaving s are true. It is hence defined by $(\hat{}s - tick_A(s))$, which differs from the stuttering clock of $s, \tau_A(s)$.

In the case of a time-triggered dispatch, when the dispatch trigger condition of an `on dispatch` clause is empty, the Boolean true is assumed, but only in the scope of the denoted object. It means that the dispatch condition is considered to be present as soon as time-triggered and an event is to be handled (otherwise, it can be regarded as silent, i.e., absent).

A `timeout` clause, if present, is denoted by the dispatch of the virtual event port `timeout`, whose trigger is associated with a real time constraint of the parent component behavior action block. It can be associated with a port list to reset timer from before timing out by arrival of an event at listed port. *The parent component is responsible for triggering this event by respecting the real time constraint behavior time, if specified, as well as with the specified frozen ports list, if present.*

5.4 Action Language

The action language of BA defines actions performed during transitions. Actions associated with transitions are action blocks that are built from basic actions and a minimal set of control structures: sequences, sets, conditionals and loops. Action sequences are executed in order, while actions in actions sets can be executed in any order.

Basic actions can be assignment actions, communication actions or time consuming actions. Assignments consist of a value expression and a target reference (local variables, data components acting as persistent state variables, or outgoing features such as ports and parameters) for the value assignment, separated by the assignment symbol `:=`. For example transitions `t0` and `t1` presented in Listing 4 both have associated assignment actions.

Communication actions can be freezing the content of incoming ports, initiating a send on an event, data, or event data port, initiating a subprogram call or catching a previously raised execution timeout exception. Listing 4 presents the transition `t3` with associated action to initiate a send on the event port `outFullBreak`.

Timed actions can be predefined computation actions. Computation actions specify computation time intervals. An execution timeout exception can be raised after any behavior action block. Raising such a timeout event may trigger a transition with a timeout catch execute condition.

5.4.1 Action Semantics

Let us recall that the transition system T representing a behavior transition is defined by $T = (s, g, true, s') \bigcup T'$. It has source state s and a guard formula g. Its target state d is that of the transition system T' defined by the semantic function call encoding the *behavior_action_block* block as $\mathcal{T}(s, d)[behavior\_action\_block] = T'$. T' is constructed by recursively calling function $\mathcal{T}$ on the action block's sub-expressions.

The recursive function $\mathcal{T}(s, d)[behavior\_actions] = T$ associates the action block *behavior_actions* guarded by a behavior condition of formula g, of source and target states s and d, to a transition system T. It is defined by case analysis on *behavior_actions*:

- a behavior action sequence is represented by concatenating the transition systems of its elements. For instance, $\mathcal{T}(s, d)[action_1 \ ; \ action_2]$ is translated by the union $T_1 \bigcup T_2$ of its transition systems $T_1 = \mathcal{T}(s, e)[action_1]$ and $T_2 = \mathcal{T}(e, d)[action_2]$, by introducing a new execution state e;
- a behavior action set is represented by composing the transition systems of its elements. For instance, $\mathcal{T}(s, d)[action_1 \ \& \ action_2]$ is translated by the synchronous composition

$$T = (T_1 | T_2)[(s_1, s_2)/s, (d_1, d_2)/d]$$

```
behavior_action_block ::=  { behavior_actions } [ timeout behavior_time ]
behavior_actions ::=
  behavior_action | behavior_action_sequence | behavior_action_set
behavior_action_sequence ::=   behavior_action { ; behavior_action }+
behavior_action_set ::=   behavior_action { & behavior_action }+
behavior_action ::=
  basic_action | behavior_action_block
  | if ( logical_value_expression ) behavior_actions
    { elsif ( logical_value_expression ) behavior_actions }*
    [ else behavior_actions ]
    end if
  | for ( element_identifier in element_values ) { behavior_actions }
  | forall ( element_identifier in element_values ) { behavior_actions }
  | while ( logical_value_expression ) { behavior_actions }
  | do behavior_actions until ( logical_value_expression )
basic_action ::=   assignment_action | communication_action | timed_action
```

of its transition systems $T1 = \mathcal{T}(s_1, d_1)[action_1]$ and $T2 = \mathcal{T}(s_2, d_2)[action_2]$, substituting the composed states (s_1, s_2) and (d_1, d_2) by s and d.

A behavior action is translated by case analysis of its form:

- `[2]if (b) a1 [2]else a2 [2]end [2]if` is translated by a guard formula g corresponding to *logical_expression* and returning the union

$$T = T_1 \bigcup T_2 \bigcup \{(s, g, true, s_1), (s, \neg g, true, s_2)\}$$

of its transition systems $T_1 = \mathcal{T}(s_1, d)[a_1]$ and $T_2 = \mathcal{T}(s_2, d)[a_2]$ where the guard formula g is the translation of the logical value expression, b;
- `[2]while ( b ) { a }` is translated by the union $T_1 \bigcup T_2$ of its transition systems $T_1 = \mathcal{T}(s_1, s_2)[a]$ and $T_2 = \{(s, h, true, s_1), (s, \neg h, true, d), (s_2, h, true, s_1), (s_2, \neg h, true, d)\}$ where the guard formula h is the translation of the logical value expression, b;
- `[2]do a [2]until (b)` is translated by the union $T_1 \bigcup T_2$ of its transition systems $T_1 = [a]$ and $T_2 = (s_1, h, true, s), (s_1, \neg h, true, d)$ where the guard formula h is the translation of the logical value expression, b;
- `[2]forall (j [2]in e) { a }` can be translated by the action set $a_1 \& \ldots \& a_n$ where a_i results from the substitution of j by the i^{th} element value of e in a.
- `[2]for (j [2]in e) { a }` can be translated by the action sequence $a_1; \ldots; a_n$ where a_i results from the substitution of j by the i^{th} element value of e in a.

A basic action is translated by case analysis of its grammar's sub-clauses:

- an assignment action to a variable v`:=`e is represented by updating v with e as $\mathcal{T}(s, d)[v := e] = \{(s, true, v' = e, d)\}$ where v' represents the next value of v;
- an output port action $port!(value)$ is represented by an action formula that binds $value$ to $port$ by $\mathcal{T}(s, d)[port!value] = \{(s, true, port = value, d)\}$;
- an input port action $port?(target)$ is represented by an action formula that updates $target$ to $port$ by $\mathcal{T}[port?target] = \{(s, true, target' = port, d)\}$;
- a timed action of the form `[2]computation` $(t_1[..t_2])$ is a timing constraint imposed on the execution time of the action block. It can either be represented by a timing property of the parent thread object or simulated by a protocol interacting with the scheduler using two virtual ports ps (start) and pf (finish) to specify a delay of time between exclusive occurrences of ps and pf, and to translate the timing specification by $\mathcal{T}(s, d)(t_1[..t_2])] = (s, true, ps, c), (c, pf, true, d)$ using a complete state c and the timed constraint $@ps + t_1 \leq @pf + t_2$;
- subprogram invocations are specified using the communication protocols HSER, LSER or ASER (cf. Section 5.7). A subprogram invocation is hence translated by the composition of the client (the caller) and server (the callee) with the behavior of the calling protocol. For instance, a subprogram call $subprogram!(parameter)$ using the HSER protocol is encoded by $\mathcal{T}(s, d)[subprogram!(parameter))] = \{(s, true, sps = pv, c), (c, spf, true, d)\}$. The output port sps encodes the call, the variable pv its parameter, and the input port spf signals the return from the callee;

5.5 Communication Actions

The communication actions defined by BA allows threads to interact with each other.

Threads can interact through shared data, connected ports and subprogram calls. The AADL execution model defines the way queued event/data of a port are transferred to the thread in order to be processed and when a component is dispatched.

Messages can be received by the annex subclause through declared features of the current component type. They can be in or in out data ports; in or in out event ports; in or in out event data ports and in or in out parameters of subprogram access.

The AADL standard defines that input on ports is determined by default *freeze* at dispatch time, or at a time specified by the *Input_Time* property and initiated by a *Receive_Input* service call in the source text. From that point in time the input of the port during this execution is not affected by arrival of new data, events, or event data until the next time input is frozen. For example, after transition `t0` (in Listing 4) is fired by the periodic dispatch of the thread, all input ports of the thread are *frozen*, new arrival of data or events will not be taken into account before the next periodic dispatch.

The AADL standard also defines that data from data ports are made available through a port variable with the name of the port. The same transition `t0` in Listing 4 uses the port variable `inActualSpeed` to get the data available on the same name port. If no new value is available since the previous freeze, the previous value remains available and the variable is marked as not fresh. Freshness can be tested in the application source code via service calls.

5.6 Expression Language

The expression language of BA is used to define expressions, the results of which are used either as logical conditions of transitions or conditional statements, or as values for assignment actions. Expressions consist of logical expressions, relational expressions, and arithmetic expressions. Values of expressions can be variables, constants or the result of another expression.

Variable expression values are evaluated from incoming ports and parameters, local variables, referenced data subcomponents, as well as port count, port fresh, and port dequeue. For example, transition `t1` presented in Listing 4 is conditioned by an expression based on one event input (`inObstacleDetected`) and one variable value (`actual_speed`). Constant expression values are Boolean, numeric or string literals, property constants or property values.

5.7 Synchronization Protocols

Thanks to provides subprogram access features, an AADL thread can receive execution requests and execute the corresponding subprogram. With proper statements in a behavior annex subclause, it is possible to specify the states where specific requests can be accepted, which correspond to Ada selective accept statements or to HOOD[5] (Hierarchical Object-Oriented Design) functional activation conditions. This mechanism also allows a clean separation between the functional part of the component defined by a set of subprograms and the synchronization aspects specified by the behavior annex automaton. The internal behavior of a server component together with the specification of the interaction protocols between the server component and its clients define the global synchronization aspects.

The behavior annex introduces precise communication protocols that can be used to better control the blocking duration of a client thread during a remote call to a server thread. These protocols are derived from the main HOOD functional execution requests:

- HSER for Highly Synchronous Execution Request;
- LSER for Loosely Synchronous Execution Request;
- ASER for ASynchronous Execution Request.

[5] `http://www.esa.int/TEC/Software_engineering_and_standardisation/TECKLAUXBQE_0.html`

5.7.1 Synchronization Semantics

Let *cs* and *cd* delimit the source and target state of subprogram call. Let *ss* and *sd* delimit the transition system of the server's subprogram. Let *pc* be the client request port and *ps* be the server reply port.

- the HSER protocol is encoded by the client transitions $\{(cs, true, pc, s), (s, \hat{}ps, true, cd)\}$, using a complete state s, and the server transition $\{(s_0, \hat{}pc, true, ss), (sd, true, ps, s_0)\}$;
- the LSER protocol is encoded by the client transitions $\{(cs, true, pc, s), (s, \hat{}ps, true, cd)\}$ and the server transition $\{(s_0, \hat{}pc, ps, ss)\}$;
- the ASER protocol is encoded by the client transitions $\{(cs, true, pc, s)\}$ and the server transition $\{(s_0, \hat{}pc, true, ss)\}$.

6 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 [9] allows to define structural properties on AADL models that are checked inductively visiting the object of a model under verification. [8] 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 [7]. 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 [3] uses executable specifications and the TINA model checker to check structural and behavioral properties of AADL models.

RAMSES, on the other hand [6], 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 to generate C code for OSs complying the ARINC-653 standard.

Synchronous modeling is central in [16], 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 tool AADL2MAUDE using OSATE.

Similarly, Yang et al. [20] define a formal semantics for an implicitly synchronous subset of the AADL, which includes periodic threads and data port communications. Its operational semantics is formalized 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 endeavor in an effort of structuring and using them together within the framework of a more expressive multi-rate or multi-clocked, synchronous, model of computation and communication: that of 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 [15], 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 annex and specify it within a polychronous MoCC.

Finally, the Behavior Language for Embedded Systems with Software (BLESS) [10, 11] was derived from BA by adding non-executable assertions to behavior to become a proof outline. With human guidance, a proof engine transforms proof oullines into deductive proofs that every execution conforms to a formal behavior specification. Although the formal semantics defined for BLESS are expressed much differently than the semantics for BA defined here, they are not incompatible. We are endeavoring to merge the semantics so that deductively proved BLESS behaviors can also be analyzed with polychronous tools such as Polychrony.

Our previous work demonstrated that the all concepts and artifact of the AADL core could, as specified in its normative documents, be given an interpretation in the polychronous model of computation and communication [14, 23, 13, 21, 22, 5], by mean of its import and simulation in the Eclipse project POP's toolset[6].

7 Conclusion

We propose a formal semantics for a significant subset of the behavioral specification annex of the Architecture Analysis and Design Language (AADL). This annex allows one to attach a behavior specification to any components of a system modeled using the AADL, and can be then analyzed for different purposes which could be, for example, the verification of logical, timing or scheduling requirements.

The addressed subset includes the transition system (state variables, states and transitions), the conditions that can be attached to transitions, the action language allowing to describe actions to be computed when a transition is fired and the expression language, used for logical conditions and assignment actions.

The semantics we presented for this subset relies on constrained automata (automata with variables derived from polychronous automata) and supports unambiguous reasoning, formal verification and simulation of the modeled system.

In future work, we will provide semantics for the remaining subset of the behavior specification annex of the AADL (mainly the synchronization protocols allowing to send and receive execution request in a client-server configuration). We will also implement the semantics of the behavior specification annex through a model transformation from the annex to the Signal language, in which the constrained automata are already implemented.

Acknowledgements

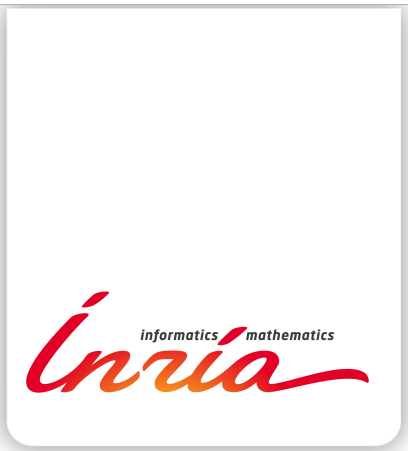
This work was partly funded by Toyota InfoTechnology Center (ITC) and by INRIA D2T's standardisation support program. The authors wish to thank Pierre Dissaux, and all the SAE sub-committee on the AADL for valuable comments on the model and method presented in this work.

References

1. Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL), 2009.
2. Aerospace Standard AS5506/2: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behavior Model Annex, 2011.
3. Bernard Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, Sebastien Heim, and François Vernadat. Formal Verification of AADL models with Fiacre and Tina. In *ERTSS 2010 - Embedded Real-Time Software and Systems*, pages 1–9, TOULOUSE (31000), France, May 2010.
4. Loïc Besnard, Étienne Borde, Pierre Dissaux, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Logically timed specifications in the AADL : a synchronous model of computation and communication (recommendations to the SAE committee on AADL). Technical Report RT-0446, INRIA, April 2014.
5. Loïc Besnard, Adnan Bouakaz, Thierry Gautier, Paul Le Guernic, Yue Ma, Jean-Pierre Talpin, and Huafeng Yu. Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony. *Science of Computer Programming*, pages 54–77, August 2015.

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

6. Etienne Borde, Smail Rahmoun, Fabien Cadoret, Laurent Pautet, Frank Singhoff, and Pierre Dissaux. Architecture models refinement for fine grain timing analysis of embedded systems. In *25nd IEEE International Symposium on Rapid System Prototyping, RSP 2014, New Delhi, India, October 16-17, 2014*, pages 44–50, 2014.
7. Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Xavier Olive. Formal verification and validation of AADL models. In *Proc. of Embedded Real Time Software and Systems Conference*, 2010.
8. Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 126–140, Berlin, Heidelberg, 2012. Springer-Verlag.
9. Olivier Gilles and Jerôme Hugues. Expressing and enforcing user-defined constraints of aadl models. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:337–342, 2010.
10. B.R. Larson, P. Chalin, and J. Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. In *Proceedings of the 2013 NASA Formal Methods Conference*, volume 7871 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2013.
11. B.R. Larson, Y. Zhang, S.Cc Barrett, J. Hatcliff, and P.L. Jones. Enabling safe interoperation by medical device virtual integration. *IEEE Design and Test*, October 2015.
12. Paul Le Guernic, Thierry Gautier, Jean-Pierre Talpin, and Loïc Besnard. Polychronous Automata. In *TASE 2015, 9th International Symposium on Theoretical Aspects of Software Engineering*, pages 95–102, Nanjing, China, September 2015. IEEE Computer Society.
13. Yue Ma, Huafeng Yu, Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin, Loï¿½c Besnard, and Maurice Heitz. Toward polychronous analysis and validation for timed software architectures in aadl. In *The Design, Automation, and Test in Europe (DATE) conference*, pages 1173–1178, Grenoble, France, March 2013.
14. Yue Ma, Huafeng Yu, Thierry Gautier, Jean-Pierre Talpin, Loïc Besnard, and Paul Le Guernic. System Synthesis from AADL using Polychrony. In *Electronic System Level Synthesis Conference*, June 2011.
15. Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1):99–106, 2010.
16. Peter Csaba Ölveczky, Artur Boronat, and José Meseguer. Formal semantics and analysis of behavioral aadl models in real-time maude. In *Proceedings of the 12th IFIP WG 6.1 International Conference and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'10/FORTE'10, pages 47–62, Berlin, Heidelberg, 2010. Springer-Verlag.
17. Bran Selic and Sébastien Gérard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
18. Markus Skoldstam, Knut Akesson, and Martin Fabian. Modeling of discrete event systems using finite automata with variables. In *46th IEEE Conference on Decision and Control, 2007*, pages 3387–3392, 2007.
19. Wikipedia. Autonomous cruise control system — wikipedia, the free encyclopedia, 2015. [Online; accessed 26-November-2015].
20. Zhibin Yang, Kai Hu, Jean-Paul Bodeveix, Lei Pi, Dianfu Ma, and Jean-Pierre Talpin. Two formal semantics of a subset of the AADL. In *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011*, pages 344–349, 2011.
21. Huafeng Yu, Yue Ma, Thierry Gautier, Loïc Besnard, Paul Le Guernic, and Jean-Pierre Talpin. Polychronous modeling, analysis, verification and simulation for timed software architectures. *Journal of Systems Architecture*, 59(10):1157–1170, November 2013.
22. Huafeng Yu, Yue Ma, Thierry Gautier, Loïc Besnard, Jean-Pierre Talpin, Paul Le Guernic, and Yves Sorel. Exploring system architectures in AADL via Polychrony and SynDEx. *Frontiers of Computer Science*, 7(5):627–649, October 2013.
23. Huafeng Yu, Yue Ma, Yann Glouche, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Paul Le Guernic, Andres Toom, and Odile Laurent. System-level co-simulation of integrated avionics using Polychrony. In *ACM Symp. on Applied Computing*, pages 354–359, TaiChung, Taiwan, March 2011.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

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

ISSN 0249-6399