# From AADL to Timed Abstract State Machines: A Verified Model Transformation

Zhibin Yang, Kai Hu, Dianfu Ma, Jean-Paul Bodeveix, Lei Pi, Jean-Pierre Talpin

# From AADL to Timed Abstract State Machines: A verified model transformation

Zhibin Yang [a,b,*], Kai Hu [a,*], Dianfu Ma [a,*], Jean-Paul Bodeveix [b], Lei Pi [c], Jean-Pierre Talpin [d]

[a] School of Computer Science and Engineering, Beihang University, Beijing, China
[b] IRIT-CNRS, Université de Toulouse, Toulouse, France
[c] INTECS, Toulouse, France
[d] INRIA-Rennes, Campus de Beaulieu, Rennes, France

A B S T R A C T

Architecture Analysis and Design Language (AADL) is an architecture description language standard for embedded real-time systems widely used in the avionics and aerospace industry to model safety-critical applications. To verify and analyze the AADL models, model transformation technologies are often used to automatically extract a formal specification suitable for analysis and verification. In this process, it remains a challenge to prove that the model transformation preserves the semantics of the initial AADL model or, at least, some of the specific properties or requirements it needs to satisfy. This paper presents a machine checked semantics-preserving transformation of a subset of AADL (including periodic threads, data port communications, mode changes, and the AADL behavior annex) into Timed Abstract State Machines (TASM). The AADL standard itself lacks at present a formal semantics to make this translation validation possible. Our contribution is to bridge this gap by providing two formal semantics for the subset of AADL. The execution semantics provided by the AADL standard is formalized as Timed Transition Systems (TTS). This formalization gives a reference expression of AADL semantics which can be compared with the TASM-based translation (for verification purpose). Finally, the verified transformation is mechanized in the theorem prover Coq.

## 1. Introduction

Embedded real-time systems, such as those for avionics and aerospace platforms, represent one of the most safety-critical categories of system. Their system behaviors do not only rely on the software/hardware architecture, but also rely on the runtime environment, such as scheduling, communication and reconfiguration. Moreover, they are more and more complex, so reducing the development cost and time is an important element of design in these systems (van Vliet, 2008).

The Architecture Analysis and Design Language (AADL) (SAE, 2009) is an SAE International (formerly known as the Society of Automotive Engineers) standard (SAE AS5506). AADL employs formal modeling concepts for the description of software/hardware architecture and runtime environment in terms of distinct components and their interactions, and it is especially effective for model-driven design of complex embedded real-time systems (Walker et al., 2013).

A safety-critical system is often required to pass stringent qualification and certification processes (for example DO-178C) before its deployment. When described using an AADL model, such a system specification is often transformed to another formal model for verification and analysis. Examples of such transformations are numerous: translations to Behavior Interaction Priority (BIP) (Chkouri et al., 2008), to TLA+ (Rolland et al., 2008), to real-time process algebra ACSR (Sokolsky et al., 2006), to IF (Abdoul et al., 2008), to Fiacre (Berthomieu et al., 2009), to Real-Time Maude (Ölveczky et al., 2010), to Lustre (Jahier et al., 2007), to Polychrony (Ma et al., 2009), etc. The goal of such a translation is to reuse existing verification and analysis tools and their formal model of computation and communication for the purpose of validating the AADL models.

* Corresponding authors. Tel.: +33 0637053466, +86 13522338282, +86 13701068603.
*E-mail addresses:* Zhibin.Yang@irit.fr (Z. Yang), hukai@buaa.edu.cn (K. Hu), dfma@buaa.edu.cn (D. Ma), bodeveix@irit.fr (J.-P. Bodeveix), lei.pi@intecs.it (L. Pi), Jean-Pierre.Talpin@inria.fr (J.-P. Talpin).

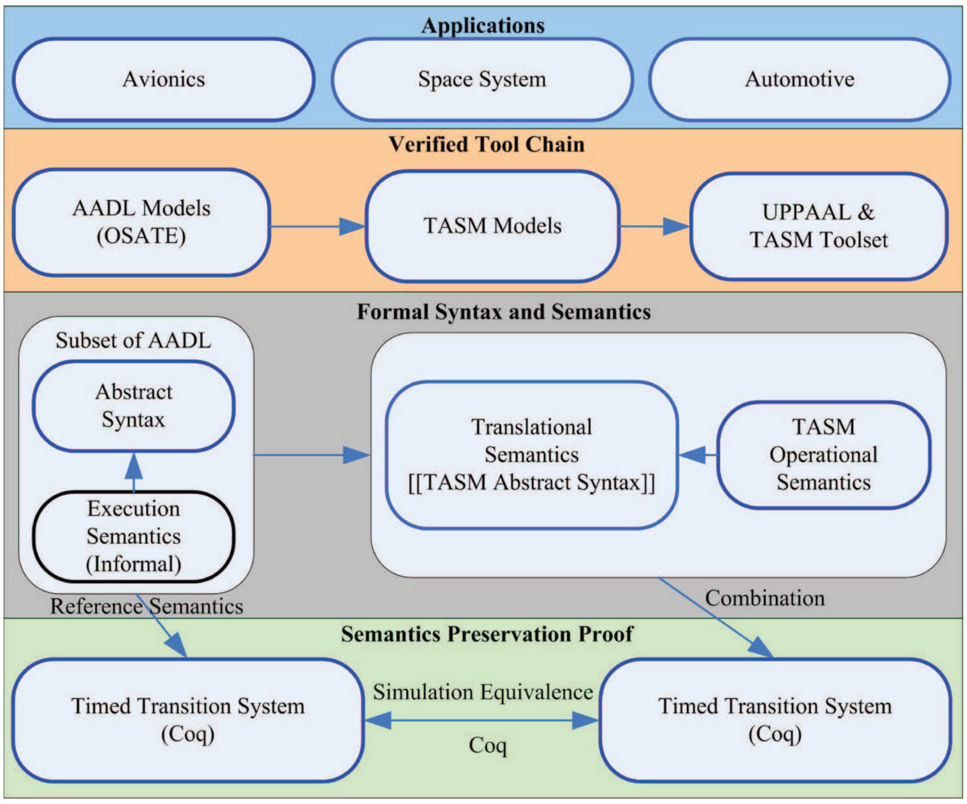

Applications
Avionics
Space System
Automotive
Verified Tool Chain
AADL Models (OSATE)
TASM Models
UPPAAL & TASM Toolset
Formal Syntax and Semantics
Subset of AADL
Abstract Syntax
Execution Semantics (Informal)
Translational Semantics [[TASM Abstract Syntax]]
TASM Operational Semantics
Reference Semantics
Combination
Semantics Preservation Proof
Timed Transition System (Coq)
Simulation Equivalence
Coq
Timed Transition System (Coq)

**Fig. 1.** A global view of the verified AADL model transformation.

One challenge, however, is the problem of proving that the translation itself preserves the intended semantics of the AADL model in the first place or, at least, some of the specific properties or requirements it needs to satisfy (Cabot et al., 2010; Lano and Rahimi, 2013; Giese et al., 2006; Narayanan and Karsai, 2008; Narayanan, 2008; Guerra et al., 2013; Kessentini et al., 2011; Mottu et al., 2008; Xiong et al., 2007).

This paper presents a machine checked semantics-preserving transformation of a subset of AADL into Timed Abstract State Machines (TASM) (Ouimet and Lundqvist, 2006, 2008). TASM is an extension of the Abstract State Machine (ASM) formalism (Börger, 2002), which supports the specification of timing and resource consumption as well as behavior and communication. Compared with the existing approaches such as translations to BIP, TLA+, ACSR, IF, Fiacre, Real-Time Maude, Lustre, etc., we consider some resource information in the transformation. Furthermore, a formal proof of the semantics preservation of the transformation has not been considered by them. In this work, the theorem prover Coq (Bertot and Castéran, 2004) is used to prove the methodology, i.e. the correctness of the translation.

Following the global idea, we would like to provide a verified tool chain for the end users (as shown in Fig. 1). The first version of the model transformation tool prototype AADL2TASM has been implemented in ATL (ATLAS Transformation Language) (Jouault et al., 2006) directly. AADL2TASM is a plug-in of the AADL modeling environment OSATE (CMU, 2006), which supports model checking using UPPAAL (Behrmann et al., 2004) and simulation using the TASM Toolset (Ouimet and Lundqvist, 2007). It is a good way to provide a basis for the Coq mechanization. Finally, we envision the extraction of a complete tool from the mechanization in the future, that is the verified tool chain. Underlying the tool is the formal translational semantics (Combemale et al., 2009; Cleenewerck and Kurtev, 2007; Gargantini et al., 2009) of AADL by a mapping to TASM, namely, the sentences of the target language express the semantics of AADL. To enable the proof of semantics preservation of the model transformation: (1) the informal execution semantics formalized directly using Timed Transition System (TTS), is considered as a reference semantics, because we cannot directly prove that the translational semantics is equivalent with the informal one which is provided by the AADL standard; (2) combining the translational semantics (expressed by the TASM sentences) with the semantics of TASM, we can obtain another way to execute the AADL model, and it is constructed as another TTS; (3) the reference semantics is supposed to be correct, and if there is a simulation equivalence relation between the two TTSs, we say the translation preserves the AADL semantics.

The goal of this paper is to verify the translation from AADL to TASM. The main contributions are as follows: (1) a reference semantics of a subset of AADL is formalized in TTS, including periodic threads, data port communications, mode changes and the behavior annex; (2) a formal translational semantics of the subset of AADL by translating it to TASM; (3) a mechanized proof of the semantics preservation of the transformation from AADL to TASM based on (1) and (2).

The rest of the paper is organized as follows. Section 2 introduces Timed Transition System and its operations. Section 3 presents an overview of the AADL and the abstract syntax of the chosen subset of AADL. The abstract syntax of TASM is expressed in Section 4. Section 5 presents the two formal semantics of the subset of AADL. Section 6 shows the mechanized proof of semantics preservation. We present some discussion about the model transformation tool prototype AADL2TASM in Section 7. Section 8 discusses the related work, and Section 9 gives some concluding remarks.

## 2. Timed Transition Systems

TTS is a well-known model used to express real-time behaviors and is often used to compare semantics equivalence between different real-time specification languages (Bérard et al., 2005, 2008). However, there are different definitions and operations on TTS for different uses. For example, some definitions use clock variables to express time in TTS (Hune and Nielsen, 1998), others express time by associating delays, or time intervals to transitions (Henzinger et al., 1994). In this paper, we use a variant of the TTS defined by Bérard et al. (2005, 2008), where we add a global state to allow communication through shared variables.

Given a global state $S^G$ with its initial set $I^G$ and $\mathbb{R}^+$ the set of nonnegative reals, a TTS is defined as follows.

**Definition 1** *(Timed Transition Systems).* A TTS defined over $S^G$ and $I^G$ is a tuple $\Gamma = \langle S^L, I^L, \Sigma, P, \rightarrow, L\rangle$ such that:

- $S^L$ is a set of *local states*,
- $I^L \subseteq S^L$ is the set of *initial local states*,
- $\Sigma$ is a finite set of *labels*,
- $P$ is a set of *predicates*,
- $\rightarrow \subseteq S \times (\Sigma \cup \mathbb{R}^+) \times S$ is the *transition* relation,
- $L : S \rightarrow 2^P$ is a labeling function, $L$ maps each state to the set of predicates which are true in that state,

where $S = S^G \times S^L$.

We write $s \xrightarrow{\delta} s'$ for $(s, \delta, s') \in \rightarrow$. Here $s, s'$ are values of the type $S$, and $\delta \in (\Sigma \cup \mathbb{R}^+)$. There are two kinds of transition relations: discrete and continuous. Continuous transitions are typically required to obey the following properties ($\forall d, d' \in \mathbb{R}^+$):

- 0-delay: $s \xrightarrow{0} s' \Leftrightarrow s = s'$,
- additivity: $s \xrightarrow{d} s' \wedge s' \xrightarrow{d'} s'' \Rightarrow s \xrightarrow{d+d'} s''$,
- continuity: $s \xrightarrow{d+d'} s' \Rightarrow (\exists s'')(s \xrightarrow{d} s'' \wedge s'' \xrightarrow{d'} s')$,
- time-determinism: $s \xrightarrow{d} s' \wedge s \xrightarrow{d} s'' \Rightarrow s' = s''$.

Then, we consider operations on TTS, including synchronous product and simulation equivalence.

Synchronous product (Arnold, 1994) is used to define the semantics of an AADL model as the composition of the semantics of its constituents. This way, the equivalence proof can be obtained in a compositional way from the correctness of the translation of elementary AADL model elements such as threads, connections and modes.

**Definition 2** *(Synchronous product).* Consider $n$ TTSs over the same alphabet $\Sigma$ and the global state $S^G$, $\Gamma_i = \langle S_i^L, I_i^L, \Sigma, P_i, \rightarrow_i, L_i\rangle$, $i = 1, \ldots, n$, where the set of predicates $P_i$ are supposed to be pair-wise disjoint. The synchronous product is a TTS: $\Gamma = \prod_{i=1,\ldots,n} \Gamma_i = \langle S^L, I^L, \Sigma, P, \rightarrow, L\rangle$, such that:

- $S^L = S_1^L \times \cdots \times S_i^L \times \cdots \times S_n^L$,
- $I^L = I_1^L \times \cdots \times I_i^L \times \cdots \times I_n^L$,
- $P = \bigcup_{i=1,\ldots,n} P_i$,
- $\rightarrow$ satisfies the following rules:

$$\frac{\forall i, (g, l_i) \xrightarrow{e}_i (g', l_i')}{(g, (l_1, \ldots, l_i, \ldots, l_n)) \xrightarrow{e} (g', (l_1', \ldots, l_i', \ldots, l_n'))}, e \in \Sigma$$

$$\frac{\forall i, (g, l_i) \xrightarrow{d}_i (g', l_i')}{(g, (l_1, \ldots, l_i, \ldots, l_n)) \xrightarrow{d} (g', (l_1', \ldots, l_i', \ldots, l_n'))}, d \in \mathbb{R}^+$$

  The rules represent that all TTSs make a step at the same time with compatible updates of the global state, and we consider discrete transitions and continuous transitions separately.
- $L = \{(g, (l_1, \ldots, l_n)) \mapsto \bigcup_{i=1,\ldots,n} L_i(g, l_i)\}$.

Bisimulation and its variants (Milner, 1989) are usually formulated over TTSs. In this work, what is proved in Coq is strong simulation equivalence which implies ACTL (the universal fragment of Computation Tree Logic) and ECTL (the existential fragment of Computation Tree Logic) preservation (Baier and Katoen, 2008) and thus preservation of the UPPAAL query language.

**Definition 3** *(Strong simulation).* Given an alphabet $\Sigma$ and two TTSs over the two global state $S_1^G$ and $S_2^G$, $\Gamma_1 = \langle S_1^L, I_1^L, \Sigma, P, \rightarrow_1, L_1\rangle$ and $\Gamma_2 = \langle S_2^L, I_2^L, \Sigma, P, \rightarrow_2, L_2\rangle$, we say that $\Gamma_2$ strongly simulates $\Gamma_1$, denoted $\Gamma_1 \preceq \Gamma_2$, if there exists a relation $R \subseteq S_1 \times S_2$ called a strong simulation relation where $S_1 = S_1^G \times S_1^L$ and $S_2 = S_2^G \times S_2^L$, such that:

- $\forall s_1^0 \in I_1^G \times I_1^L$, there exists $s_2^0 \in I_2^G \times I_2^L$, such that $(s_1^0, s_2^0) \in R$,
- $\forall (s_1, s_2) \in R, \forall e \in \Sigma, \forall s_1'$ such that $s_1 \xrightarrow{e}_1 s_1'$, there exists $s_2' \in S_2$ such that $s_2 \xrightarrow{e}_2 s_2'$ and $(s_1', s_2') \in R$,
- $\forall (s_1, s_2) \in R, \forall d \in \mathbb{R}^+, \forall s_1'$ such that $s_1 \xrightarrow{d}_1 s_1'$, there exists $s_2' \in S_2$ such that $s_2 \xrightarrow{d}_2 s_2'$ and $(s_1', s_2') \in R$,
- $\forall (s_1, s_2) \in R$, then $L_1(s_1) = L_2(s_2)$.

Here the simulation relation $R$ is general. It will be specialized when used in the proof (see Section 6). Additionally, discrete transitions and continuous transitions are treated separately.

**Definition 4** *(Strong simulation equivalence).* if $\Gamma_1 \preceq \Gamma_2$ and $\Gamma_2 \preceq \Gamma_1$, then we say there is a strong simulation equivalence relation between $\Gamma_1$ and $\Gamma_2$, i.e. two-directions strong simulation, noted $\Gamma_1 \simeq \Gamma_2$.

## 3. A subset of AADL

In this section, we first give an overview of the AADL, and then we elaborate the language subset that we will consider for translation. Finally, the abstract syntax of the considered subset is presented.

### 3.1. Overview of the AADL

AADL describes a system as a hierarchy of software and hardware components. It offers a set of predefined component categories as follows:

- Software components: thread, thread group, subprogram, data and process.
- Hardware components: processor, memory, bus, device, virtual processor and virtual bus.
- System components which represent composite sets of software and hardware components.

A component is given by its *type* and its *implementation*. The type specifies the component's external interface in terms of *features*. Features can be ports, server subprograms or data accesses depending on the chosen communication paradigm. Implementations specify the internal structure of the components in terms of a set of *subcomponents*, their *connections*, *modes* that represent operational states of components, and *properties* that support specialized architecture analysis.

However, system behaviors do not only rely on the *structure* defined by the above components but also rely on the runtime environment (like operating system or virtual machine) (Feiler and Gluch, 2013). AADL offers an *execution model* that covers most of the runtime needs of real-time systems: (1) a set of execution model attributes can be attached to each AADL declaration, such as thread dispatch protocols, communication protocols, scheduling policies, mode change protocols, and partition mechanisms (support of ARINC 653 standard in avionic systems) (Delange et al., 2009), etc.; (2) the semantics of the execution model is also given, namely the execution semantics of AADL.

Moreover, the *behavior annex* (SAE, 2011) describes more precisely the behaviors of threads and subprograms. The behavior annex has an independent syntax and semantics. So, an AADL model is composed of its structure, its execution model and its behavior annex. Correspondingly, the AADL model transformation and translational semantics, should cover these three aspects (Abdoul et al., 2008).

### 3.2. The considered subset of AADL

AADL execution model mixes synchronous and asynchronous aspects (SAE, 2009; França et al., 2007; Filali-Amine and Lawall, 2010). A synchronous execution model is obtained by considering logically synchronized periodic threads communicating through data ports. In the asynchronous execution model, it is possible to raise events, to specify sporadic and aperiodic threads, communication through shared variables, and remote procedure calls, etc. As a main difference with the existing approaches, we focus on the verification of the AADL model transformation. However, describing the formal semantics of the whole AADL is a very hard and time consuming task, as well as formally proving the correctness of its translation to another formalism. Thus we have identified a subset and restricted the correctness proof to this subset. In this paper, we consider the synchronous one, including periodic threads, data port communications, mode changes and the behavior annex. This subset is usually used in safety-critical systems, to guarantee the determinism and predictability of system behaviors. Here multi-partitions and multi-processors mechanisms are excluded, and we just consider simple scheduling features: single-processor and non-preemption.

A quick overview of the considered subset of AADL is given, including the structural elements and the execution model attributes. Its execution semantics will be given and formalized in Section 5.

#### 3.2.1. Periodic threads

Based on the modes of the system, a thread may be *active*, *inactive*, or *halted*. In AADL, only active threads can be dispatched and scheduled for execution.

A thread can have input ports and output ports to receive and send messages. AADL defines three types of ports: *data*, *event* and *event data* ports. Event and event data ports support queuing buffers, but data ports only keep the latest data. We consider data ports. However, out event ports used to trigger mode changes are also considered.

AADL supports the classical thread dispatch protocols: *Periodic*, *Aperiodic*, *Sporadic* and *Background*. Periodic dispatch is the only protocol considered in this paper, and its execution model attribute is expressed as: Dispatch_Protocol $\Rightarrow$ *Periodic*.

Moreover, several properties can be assigned to a periodic thread, such as: period given by the *Period* property in the form of Period $\Rightarrow$ 100 ms (i.e. Frequency = 10 Hz), execution time through the *Compute_Execution_Time* property or the *computation(BCET, WCET)* statement of the behavioral annex, and *Deadline*. By default, when the deadline is not specified it equals the period.

### 3.2.2. *Data port communications*

Port connections link ports to enable the exchange of messages among components. In order to ensure deterministic data communication between the data ports of periodic threads, AADL offers two communication protocols: *Immediate* and *Delayed*. The execution model attribute is attached to the data ports, and expressed as: Timing ⇒{*Immediate*, *Delayed*}.

For an immediate connection, the execution of the recipient thread is suspended until the sending thread completes its execution when the dispatches of sender and receiver threads are simultaneous. For a delayed connection the output of the sending thread is not transferred until the sending thread's deadline, typically the end of the period. Notice that they have not necessarily the same period, which allows over-sampling and under-sampling. A port connection can also be declared with modes specifying whether the connection is part of specific modes or is part of the transition between two specific modes.

### 3.2.3. *Mode change*

A *mode* represents an operational state, which manifests itself as a configuration of contained sub components, connections, and mode-specific property values. When multiple modes are declared for a component, a *mode state machine* identifies which event arrival fires a mode transition, and the new mode. The clause *in modes* indicates which subcomponents and connections are active in a given mode. Connections can also be active during mode transitions.

An AADL model is a tree of components, and each component has one or more operating modes. Thus, AADL uses the concept of *system operation mode* (SOM) to define the hierarchical composition of component modes. A SOM is a vector of modes (Bertrand et al., 2008), where each element is associated to a component. If a component is active, the associated element is valued with the *current mode* of the component. If a component is inactive, the associated element is tagged inactive.

A system has exactly one *initial SOM*, which is the set of initial modes of each component. *SOM transitions* define how a system transits from one SOM to another due to some stimulus. SOM transitions can be declared at each level, and extracting them from a set of mode state machines can be based on a breadth-first walk algorithm among the component tree (Bertrand et al., 2008). When a mode change is requested, a SOM transition is engaged. The new SOM is obtained from the old SOM, by changing the values of the vector elements that are involved in the mode change. In this paper, we mainly consider the relation between SOM transitions and threads/connections.

AADL offers two mode change protocols: *Emergency* and *Planned*. They differ on the instant where mode change actually occurs. The execution model attribute is attached to the mode transitions, and expressed as: Mode_Transition_Response ⇒{*Emergency*, *Planned*}. To guarantee determinism, we consider the planned one.

### 3.2.4. *Behavior annex*

The behavior annex can be used to describe a set of legal behaviors for a thread or a subprogram. It is described using a transition system with annotated *states*: *initial* specifies a start state, *return* specifies the end of a subprogram, *complete* specifies completion of a thread, and zero or more intermediate execution states. *Transitions* define system transitions from a *source* state to a *destination* state. Transitions may be guarded by *conditions* and trigger *actions*. Conditions and actions include sending or receiving messages, assigning variables and execution abstractions such as use of CPU time or delay, etc. Here, the specification of dispatch conditions for sporadic or aperiodic threads, and shared data communication, which imply loss of determinism, are excluded.

*An AADL example*. A simplified example of an Electronic Throttle Controller (ETC) (Ouimet et al., 2006) within our chosen subset is given (Fig. 2). A car may cruise automatically or be controlled by the driver at different speeds. The system is split into a system component, a process component (*Cruise*), and several thread components (*wheel*, *throttle*, *speed*, *command*, *display*). We focus on the modes of the process component, which contains two modes (*automatic*, *manual*). In the *manual* mode, the *command* component reads data from the driver, the *throttle* computes the voltage used to control the car, and *display* the speed parameter. When detecting the *event _ a*, the process switches to *automatic* mode. In the *automatic* mode, the tasks *wheel* and *speed* are released, *command* is deleted, *throttle* and *display* continue to execute. The *speed* component reads the tours of the *wheel* and computes the speed, then sends it to the *throttle* for controlling the car. These threads are periodic with data port connections, and each thread has a behavior annex specification.

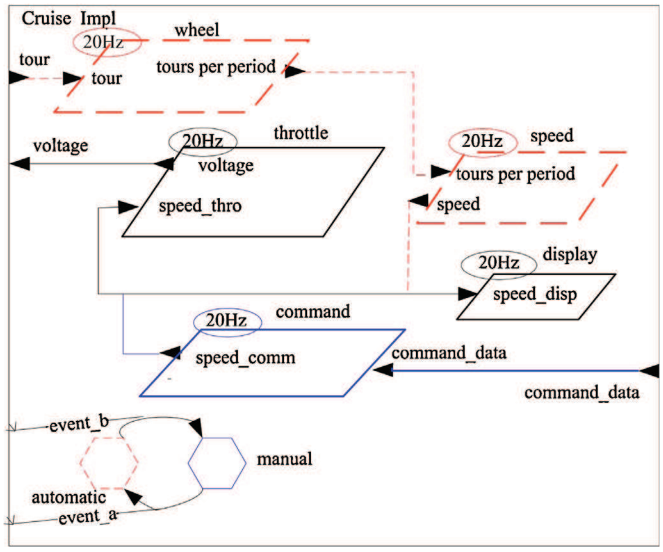


**Fig. 2.** Architecture of the electronic throttle controller system.

We show the *Cruise* process and the *throttle* thread as an example and a part of the textual specification is given in Fig. 3. The behavior of the *throttle* thread is made very simple here: the computed voltage is the difference between the wanted speed and the current speed.

### 3.3. *Abstract syntax of the subset of AADL*

We give the abstract syntax of the considered subset: an AADL model contains several threads, connections, and a SOM automaton. Each thread with its behavior annex belongs to a given SOM, and each connection can belong to a SOM or to a SOM transition. Here, the structural elements and the execution model attributes are expressed in an uniform abstract syntax.

Please notice that: (1) DURATION is a predefined type, EXPRESSION describes the expression language used in the annex and is not given here, VALUE is a type describing all possible values stored in ports, BASTATE and SOM are enumerations extracted from the actual AADL model; (2) the action language of the annex being complex, we abstract it as a function computing outputs from the value of its input ports. The CPU consumption of an action is directly modelled as a *Time* attribute.

**Listing 1** The abstract syntax of the synchronous subset of AADL.

```
Type Thread:=
{ Iports: set of PORT; \* Input  Ports *\
  Oports: set of PORT; \* Output  Ports *\
  Period: DURATION;
  BCET: DURATION;
  WCET: DURATION;
  Deadline: DURATION;
  Behavior: BehaviorAnnex;
  Modes: set of SOM;
}
Type Connection:=
{ SrcThread: Thread; \* Source  Thread *\
  DstThread: Thread; \* Destination  Thread *\
  SrcPort: PORT; \* Source  Port *\
  DstPort: PORT; \* Destination  Port *\
  ConnectionType: {Immediate, Delayed};
  Modes: set of SOM;
}
Type BehaviorAnnex:=
{ States: set of BASTATE;
  Transitions: set of BA_Transition;
}
Type BA_Transition:=
{ SrcState: BASTATE; \* Source  State *\
  DstState: BASTATE; \* Destination  State *\
  Time: DURATION;
  Guard: EXPRESSION;
  Computation:((Iports(th) → VALUE) × Oports(th)) → VALUE;
}
Type SOM_Transition:=
{ SrcMode: SOM; \* Source  Mode *\
  DstMode: SOM; \* Destination  Mode *\
  MCR_Th: Thread;
  MCR_Port: Oports(MCR_Th);
}
Type Model:=
{ Threads: set of Thread;
  Connections: set of Connection;
  Initial_Mode: SOM;
  Mode_Transitions: set of SOM_Transition;
}
```

## 4. Timed Abstract State Machine

In this section, we first introduce the basic concepts of the TASM language, then we give its abstract syntax.

```
process implementation Prc_Cruise.impl
 subcomponents
   wheel: thread Th_wheel.impl in modes (automatic);
   command: thread Th_command.impl in modes (manual);
   ...
 connections
   data port tour -> wheel.tour in modes (automatic);
   ...
 modes
   manual: initial mode;
   automatic: mode;
   manual −[ event_a ]→ automatic;
   automatic −[ event_b ]→ manual;
end Prc_Cruise.impl;
```

```
thread implementation Th_throttle.impl
 properties
   Dispatch_Protocol => Periodic;
   Period => 50ms;
   Compute_Execution_Time => 15ms..15ms;
   Deadline => 50ms;
annex behavior_specification  **
 states
   s0: initial complete state;
 transitions
   s0 −[]→ s0 {computation(15ms); voltage:= speed_thro
                 - current_speed };
 **};
end Th_throttle.impl;
```

**Fig. 3.** A part of textual specification of the AADL model of the ETC system.

### 4.1. *A brief introduction to TASM*

TASM extends the Abstract State Machine (Börger, 2002) to enable the expression of timing and resource consumption. A basic definition of a TASM specification is given as follows:

**Definition 5 (A TASM specification).** A TASM specification is a pair $\langle E, ASM\rangle$ where:

- $E = \langle EV, TU, ER\rangle$ is the environment, including:
  - $EV$ denotes the *Environment Variables*, which are the global variables that affect and are updated by machine execution,
  - $TU$ is the *Types* of environment variables that include integer, boolean, and user-defined types,
  - $ER$ is the set of named *Resources*, $ER = \{(rn, rs) | rn$ is a resource name, and $rs$ is the resource size$\}$. Examples of resources include memory, power, bus bandwidth, etc.
- $ASM = \langle MV, CV, IV, R\rangle$ is a machine, including:
  - $MV$ denotes the *Monitored Variables*, which are the set of environment variables that affect the machine execution,
  - $CV$ denotes the *Controlled Variables*, which are the set of environment variables that the machine updates,
  - $IV$ denotes the *Internal Variables*, which are the set of local variables, and their scope being limited to the machine where they are defined,
  - $R$ is the set of *Rules*, $R = \{ (\mathrm{n, t, RR, r}) | n$ is the rule name; $t$ is the duration of the rule execution, which can take the form of a single value, an interval $[t_{min}, t_{max}]$, or the keyword *next*, the "*next*" construct essentially states that time should elapse until an event of interest occurs, which is especially helpful for a machine which has no enabled rules, but which does not wish to terminate; $RR$ is the resource consumption during the rule execution with the form $rn := rs$; $r$ is a rule of the form "**if** *condition* **then** *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "**else then** *action*".$\}$. A restriction on the set of rules is that they are mutually exclusive, that is, only one rule can be executed at each step.

The concepts of hierarchical composition, parallelism, and communication are also supported by TASM. It uses a set of *main machines* that execute in parallel, to support the specification of parallel behaviors, and provides *sub-machine* and *function-machine* calls to support the specification of hierarchical behaviors. Communications are only between main machines and can use channel synchronization and shared variables.

Simply, the semantics of a main machine can be given as follows: read the shared variables, select a rule of which condition is satisfied, wait for the duration of the execution while consuming the resources, and apply the update set. If there are communications with other machines, it also needs to synchronize.

We give a simplified TASM specification of the *throttle* thread in the example of Section 3 (Fig. 4), and we assume power and memory are consumed during the execution. *Rule1* defines the actual execution of the thread. *Rule2* is triggered at completion and updates the thread state at the next dispatch. Notice that this TASM translation of an AADL thread is correct only if the thread execution time is exactly known and no other threads have access to the processor. The actual translation will need two TASM machines (see Section 5).

```
ENVIRONMENT:
USER-DEFINED TYPES:
  State:= {dispatch,completed};
RESOURCES:
  Power:=[0,100]; Memory:=[0,100];
VARIABLES:
  State CurrentState:= dispatch;
MAIN MACHINE Throttle
MONITORED VARIABLES:
  CurrentState;
CONTROLLED VARIABLES:
  CurrentState;
RULES:
  Rule1: execution
   {   t:= 15;
       Power:= [5,10];
       Memory:= [20,50];
       if CurrentState = dispatch then
       CurrentState:= completed;
   }
   Rule2: next_period
    {  t:=35;
       if CurrentState = completed then
       CurrentState:= dispatch;
    }
```

**Fig. 4.** A TASM example.

## 4.2. *Abstract syntax of TASM*

The abstract syntax of TASM is given as follows:

$$
\begin{array}{ll}
P ::= & x := \text{exp} \\
 & \mid \textit{skip} \\
 & \mid \textit{if Bexp then } P \\
 & \mid \textit{else then } P \\
 & \mid \textit{time}\,(t_{\min}, t_{\max}) \rhd P \\
 & \mid \textit{time next} \rhd P \\
 & \mid \textit{resource } r\,(r_{\min}, r_{\max}) \rhd P \\
 & \mid P \oplus P \\
 & \mid P \otimes P \\
TASM ::= & \langle E, P||P||\cdots||P\rangle
\end{array}
$$

In this paper, we use shared variables to achieve communication. $P$ defines the behaviors of a main machine, $x := exp$ updates the value of the controlled variable $x$, *time* specifies the duration of $P$, *resource* specifies the resource usage during the execution of $P$, $r$ is the name of a resource, $P \oplus P$ is the choice operator that connects several rules within a main machine, $P \otimes P$ is a synchronous parallel operator which connects statements within rule actions, the statements must not update the same variables, and $P||P$ is a parallel operator which connects main machines. Composition is synchronous if updates are simultaneous, else asynchronous.

To get a complete formalization of TASM, an interested reader can refer to (Ouimet and Lundqvist, 2006).

## 5. Two formal semantics for the subset of AADL

In order to enable the proof of semantics preservation of the model transformation, the informal execution semantics of the subset of AADL formalized directly using TTS, is considered as a reference semantics, because we cannot directly prove that the translational semantics is equivalent with the informal one which is provided by the AADL standard. As the behavior is explicitly expressed by atomic transitions, the TTS-based semantics is easy to understand. However, TTSs are mainly a mathematical model which cannot be used to automatically verify properties of a given AADL model. TASM-based translational semantics (for verification purpose) is more complex than the TTS-based one, because we need to know the rather complex TASM semantics to understand the translated TASM sentences. In this paper, the TTS-based semantics is supposed to be correct, and it gives a reference expression of AADL semantics which can be compared with the TASM-based translational semantics.

The TTS-based semantics will be defined as a synchronous product of TTSs communicating through a global state (shared variables). For this purpose, each TTS can leave some variables unspecified, and these variables are assigned by other parallel TTSs; other variables will be modified using the record update notation $s' = s\ \textit{with}\ \{\cdots\}$. So, variables not present in the record update and not declared as unspecified are supposed to be unchanged. Furthermore, in order to avoid deadlocks, we suppose that each deadlock state has a default transition which lets all the variables unchanged. Moreover, as communication between TTSs relies on shared variables, discrete labels are not used and considered to be $\tau$, that is $\Sigma = \{\tau\}$.

As mentioned in Section 3.3, an AADL model ($m$) contains several threads, connections, and a SOM automaton. Each thread ($th$) with its behavior annex belongs to a given SOM, and each connection ($cn$) can belong to a SOM or to a SOM transition ($tr$). Concretely speaking, the operational semantics of a periodic thread with data port communications is defined as the synchronous product of two TTSs: $TTS_{thread\_basic}$ and $TTS_{dispatcher}$, while the computation of the thread will be refined by the behavior annex. For an immediate connection, the execution of the recipient thread is suspended until the sending thread completes its execution when the dispatches of sender and receiver threads are simultaneous. We use an abstract scheduler to manage it and the scheduler only ensures the static alignment of thread execution order implied by the immediate connection. The operational semantics of the scheduler is defined as $TTS_{scheduler}$. The operational semantics of the SOM automaton is defined as $TTS_{mode\_change}$. Thus, the operational semantics of the considered subset of AADL is the composition of a set of TTSs. The definitions of these TTSs will be given later.

Here, we first define the sets $S^G$ and $I^G$.

- The global state is defined by the partial valuation of a set of variables. An *IportBuffer* variable is defined for each input port of each thread. The sender copies values of output ports to the buffer, and the recipient copies values from the buffer to the input ports. Each *IportBuffer* is associated to a boolean variable *NewData* which is true when the input buffer has got the latest data. It is used to guarantee the deadline of the sender occurs before the dispatch of the recipient when they are logically simultaneous and the connection type between them is Delayed. *CurrentTime* represents the global current time.

$$
\begin{aligned}
S^G = \{\ & \text{CurrentMode} : SOM,\\
& \text{Activation\_TH}(th) : \{true, false\},\\
& \text{Activation\_CN}(cn) : \{true, false\},\\
& \text{State}(th) : \{\ waiting\_mode, waiting\_dispatch,\\
& \qquad waiting\_execution, execution,\\
& \qquad computed, completed\},\\
& \text{Dispatch}(th) : \{true, false\},\\
& \text{Get\_CPU}(th) : \{true, false\},\\
& \text{Iport}(th) : \text{Iports}(th) \rightarrow VALUE,\\
& \text{Oport}(th) : \text{Oports}(th) \rightarrow VALUE,\\
& \text{IportBuffer}(th) : \text{Iports}(th) \rightarrow VALUE,\\
& \text{NewData}(th) : \text{Iports}(th) \rightarrow \{true, false\},\\
& \text{DispatchTime}(th) : DURATION,\\
& \text{CurrentTime} : DURATION\}.
\end{aligned}
$$

- $$
\begin{aligned}
I^G = \{\ & \text{CurrentMode} \mapsto Init\_Mode(m),\\
& \text{Activation\_TH} \mapsto \left\{ \begin{array}{l} th \mapsto true \mid Init\_Mode(m) \in Modes(th) \\ th \mapsto false \mid Init\_Mode(m) \notin Modes(th) \end{array} \right\},\\
& \text{Activation\_CN} \mapsto \left\{ \begin{array}{l} cn \mapsto true \mid Init\_Mode(m) \in Modes(cn) \\ cn \mapsto false \mid Init\_Mode(m) \notin Modes(cn) \end{array} \right\},\\
& \text{State}(th) \mapsto waiting\_mode,\\
& \text{Dispatch}(th) \mapsto false,\\
& \text{Get\_CPU}(th) \mapsto false,\\
& \text{Iport}(th) \mapsto \{ip \mapsto 0 \mid ip \in \text{Iports}(th)\},\\
& \text{Oport}(th) \mapsto \{op \mapsto 0 \mid op \in \text{Oports}(th)\},\\
& \text{IportBuffer}(th) \mapsto \{ip \mapsto 0 \mid ip \in \text{Iports}(th)\},\\
& \text{NewData}(th) \mapsto \{ip \mapsto true \mid ip \in \text{Iports}(th)\},\\
& \text{DispatchTime}(th) \mapsto 0,\\
& \text{CurrentTime} \mapsto 0\}.
\end{aligned}
$$

Here, messages exchanged by threads are supposed to be of type Integer.

As the analysis of AADL in Section 3, the translational semantics should take into account the structural aspect, the execution model and the behavior annex. The semantics function has two parts: the mapping of the structural aspect to the TASM environment (such as thread-related variables, connection-related variables, and mode-related variables), and the mapping of the dynamic aspect to the TASM machines (such as the execution of threads, connections, mode change and the behavior annex). The auxiliary functions, for example *Trans_ThreadData*($th$) and *Trans_Thread*($th$), will be defined later.

$$
\begin{aligned}
& Translate(m : Model) = \\
& \langle \bigcup_{th \in Threads(m)} Trans\_ThreadData(th) \\
& \quad \bigcup ( \bigcup_{th \in Threads(m)} Trans\_BehaviorAnnexData(th)) \\
& \quad \bigcup ( \bigcup_{cn \in Connections(m)} Trans\_ConnectionData(cn)) \\
& \quad \bigcup Trans\_SchedulerData, \\
& \quad \bigcup Trans\_ModeData, \\
& \qquad \|_{th \in Threads(m)} Trans\_Thread(th) \\
& \qquad \| \ (\|_{th \in Threads(m)} Trans\_BehaviorAnnex(th)) \\
& \qquad \| \ (\|_{cn \in Connections(m)} Trans\_Connection(cn)) \\
& \qquad \| \ Trans\_Scheduler \\
& \qquad \| \ Trans\_Mode \ \rangle
\end{aligned}
$$

We will give the informal interpretation, the operational semantics using TTSs, and the translational semantics by translating into TASM of the subset of AADL in a modular manner.

## 5.1. Periodic threads with data port communications

### 5.1.1. Informal interpretation

The *current mode* determines the set of threads that are considered *active*. Only the active threads can be dispatched and scheduled for execution and other threads are in the *waiting mode* state or the *halted* state.

*Periodic threads*. A periodic thread is dispatched periodically, and its inputs received from other threads are frozen at dispatch time (by default), i.e. at time zero of the period. As a result the computation performed by a thread is not affected by the arrival of new inputs. Similarly, the outputs are made available to other threads at completion or deadline time, depending on the connection.

*Data port communications between periodic threads*. First, the communication affects the input/output timing of the periodic threads. (1) For an immediate connection, the execution of the recipient is suspended until the sender completes its execution. As mentioned above, the inputs have been copied at dispatch time, so the recipient needs to replace the old data using the data got from the sender at the start of execution. (2) For a delayed connection, the value from the sender is transmitted at its deadline and is available to the recipient at its next dispatch. The recipient just needs the last dispatch data from the sender. Second, the immediate connection implies a static alignment of thread execution order, i.e., it deals with the scheduling of the sending of messages and the processing of the received messages.

In conclusion, the time line of a periodic thread with data port communications is represented in Fig. 5.

### 5.1.2. Operational semantics

Now we give the operational semantics of a periodic thread ($th$) with data port communications, which is defined as the synchronous product of two TTSs: $TTS_{thread\_basic}$ and $TTS_{dispatcher}$.

**Definition 6** *($TTS_{thread\_basic}$).* The basic behavior of a periodic thread ($th$) with data port communications is a timed transition system defined over $S^G$ and $I^G$, that is $TTS_{thread\_basic} = \langle S^L, I^L, \Sigma, \rightarrow \rangle$ where:

- $S^L = \{\text{StartofExeTime}(th) : DURATION\}$.
  As mentioned above, for an immediate connection, the execution of the recipient is suspended until the sender completes its execution. So we save the start time of execution of the thread.
- $I^L = \{\text{StartofExeTime}(th) \mapsto 0\}$.
- $\Sigma = \{\tau\}$.
- $\rightarrow$ is defined by the following rules, including discrete transitions and continuous transitions. $s$ and $s'$ are values of the type $S$ ($S = S^G \times S^L$). The shared variables State($th$), Iport($th$), Oport($th$), IportBuffer($th$), NewData($th$), DispatchTime($th$), and CurrentTime will be modified by this TTS using the schema $s' = s$ *with* $\{\ldots\}$, while CurrentMode, Activation_TH($th$), Activation_CN($cn$), Dispatch($th$) and Get_CPU($th$) are declared as unspecified (namely, they will be modified by other TTSs). We also use the overloading operator <+, which is defined as $s$ <+$\{e \mapsto v\} = \{e' \mapsto v' | (e' \mapsto v') \in s \wedge e' \neq e\} \cup \{e \mapsto v\}$.

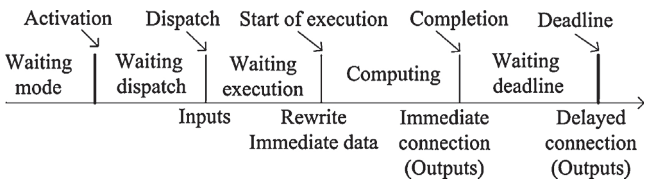


**Fig. 5.** The time line of an AADL periodic thread with data port communications.

**ACTIVATION**. Two rules are used to deal with the relation between the periodic thread *th* and mode change (in Section 5.3). A shared variable Activation _TH(*th*) : {*true*, *false*} is used. ACTIVATION[1] means the thread is in the current mode, while ACTIVATION[2] represents the thread waits to be active. The delay is controlled by the TTS associated to the mode change.

$$\frac{\begin{array}{l} s(State(th)) = waiting\_mode, \\ s(Activation\_TH(th)) = true, \\ s' = s \text{ with } \{State(th) = waiting\_dispatch\} \end{array}}{s \longrightarrow s'} ACTIVATION[1]$$

$$\frac{\begin{array}{l} s(State(th)) = waiting\_mode, \\ s(Activation\_TH(th)) = false, \\ s' = s \text{ with } \{CurrentTime = \\ \qquad\qquad s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} ACTIVATION[2]$$

**DISPATCH**. The rule DISPATCH[1] represents the dispatch of a thread. On the dispatch event, the input ports are read. For this purpose, we must wait for the data transfer made by the sender threads through delayed connections when their deadline occurs at the same logical time as the dispatch of the recipient thread. This condition is defined by the *DelayedSyncCond* predicate. As mentioned in Section 3.2, the thread can trigger mode change requests (MCRs) by raising an event. Here we use the same ports for data and event, and say that event is a value satisfying the *isEvent* predicate. The event buffer can be reset by writing the *None* value. This is done at dispatch time. The rule DISPATCH[2] represents the thread waits to be dispatched. The delay is controlled by the TTS associated to the dispatcher.

$$\frac{\begin{array}{l} s(State(th)) = waiting\_dispatch, \\ s(Dispatch(th)) = true, \\ \text{DelayedSyncCond}(th, s), \\ s' = s \text{ with } \{ \\ \quad State(th) = waiting\_execution, \\ \quad DispatchTime(th) = s(CurrentTime), \\ \quad Iport(th) = \{ip \mapsto s(IportBuffer(th, ip)) \mid ip \in Iports(th)\}, \\ \quad \|_{ip \in Iports(th)} \ NewData(th, ip) = false, \\ \quad \|_{op \in Oports(th) | \text{isEvent}(s(Oport(th, op)))} \ Oport(th, op) = None \\ \} \end{array}}{s \longrightarrow s'} DISPATCH[1]$$

$$\begin{array}{l} \textbf{where } \text{DelayedSyncCond}(th, s) \equiv \\ \quad \forall cn \in Connections(m), \\ \quad\; s(DispatchTime(SrcThread(cn))) + \\ \quad\; Period(SrcThread(cn)) = s(CurrentTime) \\ \quad\; \wedge\ DstThread(cn) = th \\ \quad\; \wedge\ ConnectionType(cn) = Delayed \\ \quad\; \Rightarrow s(NewData(th, DstPort(cn))) = true \end{array}$$

$$\frac{\begin{array}{l} s(State(th)) = waiting\_dispatch, \\ s(Dispatch(th)) = false, \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} DISPATCH[2]$$

**WAITING_EXE**. When the thread is dispatched, its execution is managed by a scheduler. Specially, in presence of immediate connection, the scheduler must ensure that the sender completes before the start of execution of the recipient. Moreover, ports of which incoming connection is immediate must be rewritten. The rule WAITING_EXE[1] means that the thread gets CPU, while the rule WAITING_EXE[2] represents the thread waits to be scheduled. The delay is controlled by a TTS associated to the scheduler (in Section 5.2).

$$\frac{\begin{array}{l} s(State(th)) = waiting\_execution, \\ s(Get\_CPU(th)) = true, \\ s' = s \text{ with } \{ \\ \quad State(th) = execution, \\ \quad Iport(th) = s(Iport(th)) \lhd\!\!\!+ \\ \qquad \{ip \mapsto s(IportBuffer(th, ip)) \mid \text{isImmediate}(th, ip)\}, \\ \quad StartofExeTime(th) = s(CurrentTime) \\ \} \end{array}}{s \rightarrow s'} WAITING\_EXE[1]$$

**where** isImmediate$(th, ip)$ checks if the input port $ip$ of thread $th$ is the destination of an immediate connection. This predicate is defined as follows:

$$\begin{array}{l} \exists cn \in Connections(m), DstPort(cn) = ip \\ \quad \wedge\ ConnectionType(cn) = Immediate \\ \quad \wedge\ ip \in Iports(th) \end{array}$$

$$\frac{\begin{array}{l} s(State(th)) = waiting\_execution, \\ s(Get\_CPU(th)) = false, \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} WAITING\_EXE[2]$$

**EXECUTION**. Here, the execution is abstracted by a function *Computation* (*Inputs* : *Iports*(*th*) → *VALUE*) : *Oports*(*th*) → *VALUE* which consumes CPU time during the interval [BCET, WCET] of the thread, and it will be refined by the AADL behavior annex (in Section 5.4). The rule EXECUTION[1] represents that the thread is in the progress of its execution. The rule EXECUTION[2] is used to write the computation results into the output ports of the thread.

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ s(CurrentTime) - s(StartofExeTime(th)) + d \leq WCET(th), \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} EXECUTION[1]$$

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ BCET(th) \leq (s(CurrentTime) - \\ \qquad\qquad s(StartofExeTime(th))) \leq WCET(th), \\ s' = s \text{ with } \{State(th) = computed, \\ \qquad\quad Oport(th) = \|_{op \in Oports(th)}\ Computation \\ \qquad\qquad\quad (\{ip \mapsto s(Iport(th)) \mid ip \in Iports(th)\}, op) \\ \} \end{array}}{s \rightarrow s'} EXECUTION[2]$$

**WRITE_IMM_DATA**. It is used to copy the data of the output ports of the sender to the *IportBuffer* of the recipient at the completion time of the sender. This copy is only done when the two threads are dispatched at the same time.

$$\frac{\begin{array}{l} s(State(th)) = computed, \\ s' = s \text{ with } \{ \\ \quad IportBuffer = IportBuffer \lhd\!\!\!+ \\ \qquad \{DstThread(cn) \mapsto s(Oport(th)) \mid \\ \qquad\quad cn \in Connections(m) \\ \qquad\quad \wedge\ SrcThread(cn) = th \\ \qquad\quad \wedge\ ConnectionType(cn) = Immediate \\ \qquad\quad \wedge\ s(DispatchTime(th)) = \\ \qquad\qquad s(DispatchTime(DstThread(cn))) \\ \qquad \}, \\ \quad State(th) = completed \\ \} \end{array}}{s \rightarrow s'} WRITE\_IMM\_DATA$$

**WAITING_DEADLINE**. This rule is used to wait for the deadline (i.e. the period here for the sake of simplicity) of the thread.

$$\frac{\begin{array}{l} s(State(th)) = completed, \\ s(Get\_CPU(th)) = false, \\ s(CurrentTime) + d \\ \quad \leq s(DispatchTime(th)) + Period(th), \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} WAITING\_DEADLINE$$

**Definition 7** *(TTS$_{dispatcher}$)*. The behavior of the dispatcher of a periodic thread (*th*) is a timed transition system defined over $S^G$ and $I^G$, that is $TTS_{dispatcher} = \langle S^L, I^L, \Sigma, \rightarrow \rangle$ where:

- $S^L$ = {WaitNextDispatch(*th*) : {*true*, *false*}}.
  Here, an auxiliary variable WaitNextDispatch(*th*) is used.
- $I^L$ = {WaitNextDispatch(*th*) $\mapsto$ *false*}.
- $\Sigma = \{\tau\}$.
- $\rightarrow$ is defined by the following rules. The shared variables State(*th*), IportBuffer(*th*), NewData(*th*), Dispatch(*th*) and CurrentTime will be modified by this TTS, while others are declared as unspecified.

**DISPATCH_THREAD**. This rule is used to dispatch the thread, and synchronize with the thread using the shared variable Dispatch(*th*).

$$\frac{\begin{array}{l} s(Activation\_TH(th)) = true, \\ s(State(th)) = waiting\_dispatch, \\ s(Dispatch(th)) = false, \\ s' = s \text{ with } \{Dispatch(th) = true, \\ \qquad\qquad WaitNextDispatch(th) = true\} \end{array}}{s \rightarrow s'} DISPATCH\_THREAD$$

**WAITING_PERIOD**. This rule does nothing, and it is just used to wait the period of the thread.

$$\frac{\begin{array}{l} s(WaitNextDispatch(th)) = true, \\ s(CurrentTime) + d \\ \quad \leq s(DispatchTime(th)) + Period(th), \\ s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\} \end{array}}{s \xrightarrow{d} s'} WAITING\_PERIOD$$

**REACH_PERIOD**. This rule represents that the thread arrives at its period. The output of the delay connections is also expressed in this rule, namely, copy the data of the output ports of the sender to the *IportBuffer* of the recipient at the deadline of the sender. We also set the *NewData* variable to true in order to indicate the data transfer has been done.

$$\frac{\begin{array}{l} s(WaitNextDispatch(th)) = true, \\ s(CurrentTime) = s(DispatchTime(th)) + Period(th), \\ s' = s \text{ with } \{ \\ \quad State(th) = waiting\_mode, \\ \quad WaitNextDispatch(th) = false, \\ \quad Dispatch(th) = false, \\ \quad IportBuffer = IportBuffer \lessdot\!\!+ \\ \qquad \{DstThread(cn) \mapsto s(Oport(th))\} \mid \text{isDelayOut}(th), \\ \quad NewData(DstThread(cn)) = true \\ \} \end{array}}{s \rightarrow s'} REACH\_PERIOD$$

**where** isDelayOut($th$) checks if the thread $th$ is the source thread of a delayed connection.
This predicate is defined as follows:

$$\begin{array}{ll} \exists cn \in Connections(m), & SrcThread(cn) = th \\ & \wedge\ ConnectionType(cn) = Delayed \end{array}$$

**WAIT**. This rule is used to deal with the waiting time when the conditions of other rules are not satisfied.

$$\frac{s(Activation\_TH(th)) = false, \quad s' = s \text{ with } \{CurrentTime = s(CurrentTime) + d\}}{s \xrightarrow{d} s'} WAIT$$

*5.1.3. Translational semantics*

The TASM environment associated to each thread is defined as a set of typed variables such as the *state* and the *resource consumption* of the thread. Messages exchanged by threads are supposed to be of type Integer. The variables associated to connections are defined as well.

**Listing 2** The TASM environment of an AADL periodic thread with data port communications.

```
Trans_ThreadData(th)=
{ State:{waiting_mode, waiting_dispatch, waiting_execution, execution,
        computed, completed} := waiting_mode;
  Iport:Iports(th) → Integer;
  Oport:Oports(th) → Integer;
  RscUsage:RESOURCE → Integer;
  WaitNextDispatch:Boolean;
   ...
}
```

The dynamic behavior of a periodic thread with data port communications is defined as two TASM machines in parallel: *TASM_Thread(th)* and *Periodic_Dispatcher(th)*, as shown in Listing 3.

- A periodic thread (*th*) is expressed by *TASM_Thread(th)* with six rules: *Activation*, *Dispatch*, *Waiting Execution*, *Execution*, *Write Immediate Data*, and *Waiting Next Event*. In the rule *Execution*, the duration of execution is [BCET,WCET] and processor consumption is 100%. The rule *Waiting Next Event* is used to deal with the waiting time when activation, dispatch and execution conditions are not satisfied. Additionally, the behavior of a connection is separated in two parts: *Read* and *Write*.
- The dispatcher of a periodic thread (*th*) is expressed by *Periodic_Dispatcher(th)* with three rules: *Dispatch Thread*, *Waiting Period*, and *Waiting Next Event*. The rule *Waiting Period* corresponds to the rules WAITING_PERIOD and REACH_PERIOD of $TTS_{dispatcher}$ (Definition 7). According to the TASM semantics (Section 4.1), the system will wait for the duration of the execution of the rule, then apply the update set (i.e. actions). Thus, the TASM-based translational semantics hides implicit behaviors provided by the TASM semantics.

  Translation rules are written using an ML-like language (close to the Coq notations) with bold font keywords as follows. References to other TASM machines are also bold.
- **LET** name = P **AND** . . . **IN** TASM.
- **IF** condition **THEN** P **ELSE** P.

**Listing 3** The TASM machines of an AADL periodic thread with data port communications.

```
Trans_Thread(th)=
LET TASM_Thread(th)=
// Rule Activation
Time 0 ▷ (if State(th)=waiting_mode and Activation_TH(th)=true then
             State(th):=waiting_dispatch)
⊕ //Rule Dispatch
Time 0 ▷ (if State(th)=waiting_dispatch and Dispatch(th)=true then
              State(th):=waiting_execution
              ⊗ Iport(th):=IportBuffer(th)
              ⊗  ⊗_{op∈Oports(th)|isEvent(Oport(th,op))}  Oport(th,op):= None)
⊕ //Rule Waiting Execution
Time 0 ▷ (if State(th)=waiting_execution and Get_CPU(th)=true then
             State(th):=execution ⊗ Trans_Connection_Read(th))
⊕ //Rule Execution
Time (BCET(th),WCET(th)) ▷ Resource Processor 100 ▷
             (if State(th)=execution then
             Oport(th):=Computation(Iport(th)) ⊗ State(th):=computed)
⊕ //Rule Write Immediate Data
Time 0 ▷ (if State(th)=computed then
             State(th):=completed ⊗ Trans_Connection_Write_Imm(th))
⊕ //Rule Waiting Next Event
Time next ▷ (else then
             skip)
```

```
AND Periodic_Dispatcher(th)=
//Rule Dispatch Thread
time 0    (if Activation_TH(th)=true and State(th)=waiting_dispatch
              and Dispatch(th)=false then
              Dispatch(th):=true ⊗ WaitNextDispatch(th):=true)
⊕ //Rule Waiting Period
time Period(th) ▷ (if WaitNextDispatch(th)=true then
              WaitNextDispatch(th):=false
              ⊗ Trans_Connection_Write_Delay(th)
              ⊗ State(th):=waiting_mode
              ⊗ Dispatch(th):=false)
⊕ //Rule Waiting Next Event
time next ▷ (else then
              skip)
IN TASM_Thread(th) || Period_Dispatcher(th)
```

**Trans_Connection_Read(th)**=

$$\bigotimes_{\substack{ip \in Iports(th) \\ \wedge cn \in Connections(m) \\ \wedge DstPort(cn) = ip \\ \wedge ConnectionType(cn) = Immediate}} \text{ip:=IportBuffer(th,ip)}$$

**Trans_Connection_Write_Imm(th)**=

$$\bigotimes_{\substack{op \in Oports(th) \\ \wedge cn \in Connections(m) \\ \wedge SrcPort(cn) = op \\ \wedge ConnectionType(cn) = Immediate}} \text{IportBuffer(DstThread(cn),DstPort(cn)):=op}$$

**Trans_Connection_Write_Delay(th)**=

$$\bigotimes_{\substack{op \in Oports(th) \\ \wedge cn \in Connections(m) \\ \wedge SrcPort(cn) = op \\ \wedge ConnectionType(cn) = Delayed}} \text{IportBuffer(DstThread(cn),DstPort(cn)):=op}$$

### *5.2. Scheduler*

#### *5.2.1. Informal interpretation*

As mentioned above, in presence of immediate connection, the scheduler must ensure that the sender completes before the start of execution of the recipient. Here, we give an abstract view of the scheduler. The given behavior allocates the CPU to a waiting thread *th* of which immediate data emitters that have been synchronously dispatched with *th* have completed their task.

#### *5.2.2. Operational semantics*

**Definition 8 (*$TTS_{scheduler}$*).** The behavior of the scheduler is a timed transition system defined over $S^G$ and $I^G$, that is $TTS_{scheduler} = \langle S^L, I^L, \Sigma, \rightarrow \rangle$ where:

- $S^L = \emptyset$.
- $I^L = \emptyset$.
- $\Sigma = \{\tau\}$.
- $\rightarrow$ is defined by the following rules. The shared variables Get _ CPU(*th*) will be modified by this TTS, while others are declared as unspecified.

**ALLOCATE_CPU**. This rule is used to allocate the CPU to a waiting thread.

$$\frac{\begin{array}{l} s(State(th)) = waiting\_execution, \\ \forall th' \in Threads(m) \\ \quad s(Get\_CPU(th')) = false, \\ \forall cn \in Connections(m) \\ \quad ConnectionType(cn) = Immediate \\ \quad \wedge DstPort(cn) \in Iports(th) \\ \quad \wedge s(DispatchTime(th)) = s(DispatchTime(SrcThread(cn))) \\ \quad \rightarrow s(State(SrcThread(cn))) \in \{completed, waiting\_mode\}, \\ s' = s \text{ with } \{Get\_CPU(th) = true\} \end{array}}{s \rightarrow s'} \; ALLOCATE\_CPU$$

**FREE_CPU**. This rule frees the CPU allocated to a thread which has completed its task.

$$\frac{\begin{array}{l} s(Get\_CPU(th)) = true, \\ s(State(th)) = completed, \\ s' = s \text{ with } \{Get\_CPU(th) = false\} \end{array}}{s \rightarrow s'} FREE\_CPU$$

*5.2.3. Translational semantics*

The scheduler should avoid giving the processor to several threads at the same time. Its behavior is defined as a TASM machine with three rules: *Allocate CPU*, *Free CPU*, and *Waiting Next Event*.

**Listing 4** The TASM machine of the scheduler.

```
Trans_Scheduler=
// Rule Allocate CPU
Time 0 ▷ (if State(th)=waiting_execution
                $\bigwedge_{th' \in Threads(m)}$ (Get_CPU(th')=false)
                $\bigwedge_{cn \in Connections(m)} \left( \begin{array}{l} ConnectionType(cn) = Immediate \\ \wedge\ DstPort(cn) \in Iports(th) \\ \wedge\ DispatchTime(th) \\ \quad = DispatchTime(SrcThread(cn)) \\ \wedge\ State(SrcThread(cn)) \\ \quad \in \{completed, waiting\_mode\} \end{array} \right)$
                then
                Get_CPU(th):=true)

⊕ //Rule Free CPU
Time 0 ▷ (if State(th)=completed and Get_CPU(th)=true then
                Get_CPU(th):=false
⊕ //Rule Waiting Next Event
Time next ▷ (else then
                skip)
```

*5.3. Mode change*

*5.3.1. Informal interpretation*

*The behaviors of a SOM transition*. The AADL mode change protocol comprises two phases: (1) waiting SOM transition; (2) SOM transition. In this paper, we consider the planned one. Fig. 6 shows the time line of an AADL SOM transition.

At the beginning, the system is in the source mode of a SOM transition (i.e. *oldSOM*). After a *mode change request* (MCR) has occurred, execution continues under the *oldSOM* until the dispatches of a set of critical threads that are active align at their hyper-period (called *Hyper(oldSOM)*), then the *mode_transition_in_progress* state is entered. A periodic thread is considered as critical if its *Synchronized_Component* property is true. That means, the duration from the *oldSOM* state to the *mode_transition_in_progress* state is the distance to the next hyper-period of these critical threads.

In the *mode_transition_in_progress* state, some threads enter the new mode (become active), some threads exit the old mode, the critical threads of both new and old modes continue to execute, and the connections belong to *oldSOM* will be deleted. The system is in the *mode_transition_in_progress* state for a limited amount of time, which is the distance to a multiple of the hyper-period of the critical threads that continue to execute (called *Hyper(newSOM)**). Finally, the system enters the destination mode of the SOM transition (i.e. *newSOM*).

When a MCR is responded, all of the other incoming MCRs will be ignored, and we don't consider the higher priority requests here.

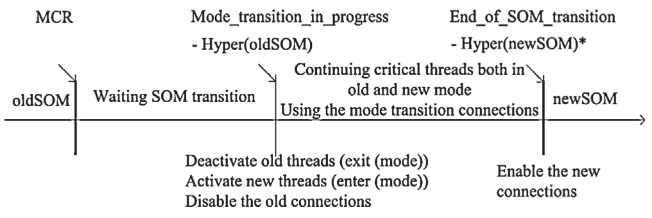

MCR
Mode_transition_in_progress
- Hyper(oldSOM)
End_of_SOM_transition
- Hyper(newSOM)*
oldSOM
Waiting SOM transition
Continuing critical threads both in old and new mode
Using the mode transition connections
newSOM
Deactivate old threads (exit (mode))
Activate new threads (enter (mode))
Disable the old connections
Enable the new connections

**Fig. 6.** The time line of an AADL SOM transition.

*5.3.2. Operational semantics*

Now, we give the operational semantics of the SOM automaton of the AADL model ($m$).

**Definition 9 (*TTS*$_{mode\_change}$).** The behavior of the SOM automaton is a timed transition system defined over $S^G$ and $I^G$, that is *TTS*$_{mode\_change}$ = $\langle S^L, I^L, \Sigma, \rightarrow \rangle$ where:

- $S^L = \left\{ \begin{array}{l} \text{ModeTransitionInProgress} : \{\mathit{true}, \mathit{false}\}, \\ \text{GetMCR}(tr) : \{\mathit{true}, \mathit{false}\} \end{array} \right\}.$

  Here, two auxiliary variables are used.
- $I^L = \left\{ \begin{array}{l} \text{ModeTransitionInProgress} \mapsto \mathit{false}, \\ \text{GetMCR}(tr) \mapsto \mathit{false} \end{array} \right\}.$
- $\Sigma = \{\tau\}$.
- $\rightarrow$ is defined by the following rules. The shared variables CurrentMode, Activation_TH($th$), Activation_CN($cn$), and CurrentTime will be modified by this TTS, while others are declared as unspecified.

**GET_MCR**. This rule gets events sent by threads and selects a corresponding mode transition.

$$\frac{\begin{array}{l} s(CurrentMode) = SrcMode(tr), \\ \text{isEvent}(s(Oport(th, op))), \\ MCR\_Th(tr) = th, \\ MCR\_Port(tr) = op, \\ s' = s \text{ with } \{ \\ \qquad GetMCR(tr) = true\} \end{array}}{s \rightarrow s'} GET\_MCR$$

**ENTER_MC_PL**. This rule makes a random choice of one of the MCRs, other MCRs are disabled. The system goes to the *mode_transition_in_progress* state, old threads and connections are deactivated, while new threads are activated.

$$\frac{\begin{array}{l} s(CurrentMode) = SrcMode(tr), \\ s(GetMCR(tr)) = true, \\ s(CurrentTime) \ mod \ Hyper(CurrentMode) = 0, \\ s(ModeTransitionInProgress) = false, \\ s' = s \text{ with } \{ \\ \quad GetMCR = \{tr' \mapsto (tr = tr')\}, \\ \quad Activation\_TH = \left\{ \begin{array}{l} th \mapsto true \mid DstMode(tr) \in Modes(th) \\ th \mapsto false \mid otherwise \end{array} \right\}, \\ \quad Activation\_CN = \left\{ \begin{array}{l} cn \mapsto true \mid \begin{array}{l} SrcMode(tr) \in Modes(cn) \\ \wedge \ DstMode(tr) \in Modes(cn) \end{array} \\ cn \mapsto false \mid otherwise \end{array} \right\}, \\ \quad ModeTransitionInProgress = true\} \end{array}}{s \rightarrow s'} ENTER\_MC\_PL$$

**MC_PL**. This rule means that the system enters the new SOM.

$$\frac{\begin{array}{l} s(ModeTransitionInProgress) = true, \\ s(GetMCR(tr)) = true, \\ s(CurrentTime) \ mod \ Hyper(DstMode(tr)) = 0, \\ s' = s \text{ with } \{ \\ \qquad CurrentMode = DstMode(tr), \\ \qquad Activation\_CN = \left\{ \begin{array}{l} cn \mapsto true \mid DstMode(tr) \in Modes(cn) \\ cn \mapsto false \mid otherwise \end{array} \right\}, \\ \qquad ModeTransitionInProgress = false, \\ \qquad GetMCR(tr) = false\} \end{array}}{s \rightarrow s'} MC\_PL$$

*5.3.3. Translational semantics*

In the TASM environment, we introduce the *CurrentMode* variable valued in the set of SOM, and two booleans *ArriveHyperPeriod* and *ModeTransitionInProgress*.

**Listing 5** The TASM environment of the SOM automaton.

```
Trans_ModeData=
{ CurrentMode: SOM;
  ArriveHyperPeriod: SOM → Boolean;
  ModeTransitionInProgress: Boolean;
   ...
}
```

The dynamic behavior of the SOM automaton is defined as two TASM machines in parallel: *SOM_Transition* and *Manage_Hyper(mode)*.

- The basic behavior of the SOM automaton is expressed by *SOM_Transition* with three rules: *Get MCR*, *Mode Transition In Progress* and *Mode Transition*. The machine uses the shared variable *Activation _TH*(*th*) to synchronize with the execution of periodic threads. The rule *Get MCR* gets events sent by threads and selects a corresponding mode transition. The rule *Mode Transition In Progress* expresses the behavior when waiting for the actual SOM transition after a MCR. The rule *Mode Transition* expresses the actual SOM transition.
- We use the machine *Manage_Hyper(mode)* to manage the time *Hyper(oldSOM)* and *Hyper(newSOM)**.

**Listing 6** The TASM machines of the SOM automaton.

```
Trans_Mode=
LET SOM_Transition= $\bigoplus_{tr}$ (
LET oldSOM=SrcMode(tr)
AND newSOM=DstMode(tr)
IN
//Rule Get MCR
time 0 $\triangleright$ (if CurrentMode=oldSOM and
            isEvent(Oport(MCR_Th(tr), MCR_Port(tr)))=true then
            GetMCR(tr):=true)
$\oplus$ //Rule Mode Transition In Progress
time 0 $\triangleright$ (if CurrentMode=oldSOM and GetMCR(tr)=true
            and ArriveHyperPeriod(oldSOM)=true
            and ModeTransitionInProgress=false then
            $\bigotimes_{tr'\neq tr}$ GetMCR(tr'):=false
            $\otimes$ ModeTransitionInProgress:=true
            $\otimes$ ArriveHyperPeriod(newSOM):=false
            $\otimes \bigotimes_{th|newSOM\in Modes(th)\setminus oldSOM\in Modes(th)}$ Activation_TH(th):=true
            $\otimes \bigotimes_{th|oldSOM\in Modes(th)\setminus newSOM\in Modes(th)}$ Activation_TH(th):=false
            $\otimes \bigotimes_{cn|oldSOM\in Modes(cn)\setminus newSOM\in Modes(cn)}$ Activation_CN(cn):=false)
$\oplus$ //Rule Mode Transition
time 0 $\triangleright$ (if GetMCR(tr)=true and ModeTransitionInProgress=true
             and ArriveHyperPeriod(newSOM)=true then
             CurrentMode:=newSOM
             $\otimes \bigotimes_{cn|newSOM\in Modes(cn)\setminus oldSOM\in Modes(cn)}$ Activation_CN(cn):=true
             $\otimes$ ModeTransitionInProgress:=false
             $\otimes$ ArriveHyperPeriod(oldSOM):=false
             $\otimes$ GetMCR(tr):=false)
AND Manage_Hyper(mode)=
time Hyper(mode) $\triangleright$ (if true then
             ArriveHyperPeriod(mode):=true)
IN SOM_Transition $\|$ ($\|_{mode\in SOM}$ Manage_Hyper(mode))
```

*5.4. Behavior annex*

AADL supports an input-compute-output model of computation and communication for threads. The input/output timing is defined by the execution model, and the computation can be refined by the behavior annex. So, there is a close relation between the AADL execution model and the behavior annex. The execution model specifies when the behavior annex is executed and on which data it is executed, while the behavior annex acts in a thread (or a subprogram) and describes behaviors more precisely. Thus, the semantics specifications given as above will be enriched by the behavior annex.

### 5.4.1. *Informal interpretation*

The behavior annex is described using a transition system in the following form:

```
annex behavior_specification
{**
<state_variables>
<initialization>
<states>
<transitions>
**};
```

*State Variables* represent the variables with the scope of the behavior annex, and they may hold the inputs, the intermediate results, and the outputs. *States*: *initial* specifies a start state, *return* specifies the end of a subprogram, *complete* specifies completion of a thread, and other states represent intermediate execution steps. *Transitions* define system transitions from a *source* state to a *destination* state. A transition can be guarded with execution conditions. An action part can be attached to a transition and it performs message sending, assignments or time consuming activities. However, the action part is related to the transition and not to the states: if a transition is enabled, the action part is performed and then the current state becomes the transition destination state. When the transition reaches a complete or return state, the thread or the subprogram will be completed.

### 5.4.2. *Operational semantics*

The operational semantics of the behavior annex is defined as the refinement of the *"Execution"* rule of $TTS_{thread\_basic}$, where the state $S^L$ extends the semantics state of $TTS_{thread\_basic}$ with *CurrentBAState* and *StartTransitionTime*. The initial value of *CurrentBAState* is the initial state of the behavior annex. *StartTransitionTime* is used to save the start time of the execution of the transition, and its initial value is the *StartofExeTime* of the thread.

**BA_EXECUTION**. This rule expresses the execution of each transition of the behavior annex, and the *StartTransitionTime* is accumulated for each transition.

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ s(CurrentBAState) = SrcState(BA\_tr), \\ s(CurrentTime) = s(StartTransitionTime) + Time(BA\_tr), \\ s' = s \text{ with } \{ \\ \quad CurrentBAState = DstState(BA\_tr), \\ \quad Oport(th) = Action(BA\_tr)(Iport(th)), \\ \quad StartTransitionTime = s(CurrentTime)\} \end{array}}{s \longrightarrow s'} BA\_EXECUTION$$

if Guard(BA_tr)=true, $\forall BA\_tr \in Transitions(BA) \wedge BA = Behavior(th)$

**BA_COMPLETE**. This rule means the execution of the behavior annex is completed. An auxiliary variable *isFinal* is used to denote the complete state of the behavior annex.

$$\frac{\begin{array}{l} s(State(th)) = execution, \\ isFinal(s(CurrentBAState)) = true, \\ s' = s \text{ with } \{State(th) = computed\} \end{array}}{s \longrightarrow s'} BA\_COMPLETE$$

### 5.4.3. *Translational semantics*

The translational semantics specification of the behavior annex contains three parts: structure, guards and actions.

*Structure*. *State Variables* are mapped to the TASM environment variables with general types such as integer, real numbers and booleans. *States* are translated into TASM internal variables. Each *transition* of the behavior annex is translated into a TASM rule. These rules will take the place of the *"Execution"* rule of the periodic thread.

In each TASM rule, the condition is built from the current state of the thread (*State(th)*), the *source state* of the transition, and the *guards* of the transition, while the actions of the rule include the *actions* of the transition and the *destination state* of the transition.

When all the transitions are translated, a new rule (*Behavior Annex Completion*) is added. Then the TASM machine executes the next rule (i.e. *Write Immediate Data*) of the periodic thread.

*Guards*. The execution conditions which are logical expressions based on the state variables are considered. Here, we do not detail the translation of behavior expressions. The corresponding function is denoted $[\![Guard]\!]$.

*Actions*. We mainly consider the basic actions including message sending, assignments to variables, and timing. (1) Message sending is expressed by the assignment of the port variables. (2) Assignments are expressed by the assignment of the corresponding environment variables. (3) The behavior annex introduces the statements *computation(min,max)* and *delay(min,max)*, which express the use of the CPU and the suspension for a possibly non-deterministic period of time between *min* and *max* respectively. They are related to the transitions and not to the states, which is consistent with the TASM semantics. Thus, the timing actions are translated into the duration of the corresponding TASM rule.

In the TASM environment, we introduce the *CurrentBAState* variable valued in the set of states of the behavior annex and two booleans: *isInitial* and *isFinal*.

**Listing 7** The TASM environment of the behavior annex of a periodic thread.

```
Trans_BehaviorAnnexData(th)=
{ CurrentBAState: BAState;
  isInitial: BAState → Boolean;
  isFinal: BAState → Boolean;
   ...
}
```

We consider all the transitions of the behavior annex of a periodic thread, and the basic semantics specification is presented in Listing 8. It can be detailed when more complex guards and actions are considered.

**Listing 8** The TASM rules of the behavior annex of a periodic thread.

```
Trans_Thread(th)=
...
⊕ //Rule Execution
Trans_BehaviorAnnex(th)
⊕ //Rule Write Immediate Data
...
Trans_BehaviorAnnex(th)=
                 ⊗                          Time Time(BA_tr) ▷
      $BA = Behavior(th)$
  $\wedge BA\_tr \in Transitions(BA)$
       (if State(th)=execution and CurrentBAState=SrcState(BA_tr)
         and ⟦ Guard(BA_tr) ⟧ =true then
         CurrentBAState:=DstState(BA_tr)
         ⊗ Oport(th):=Action(BA_tr)(Iport(th))
⊕ //Rule Behavior Annex Completion
Time 0 ▷ (if isFinal(CurrentBAState)=true then
         State(th):=computed)
```

## 6. The proof of semantics preservation

In this section, we give the main idea of the proof of semantics preservation of the transformation from AADL to TASM, i.e. the exact correspondence between execution steps defined by the AADL operational semantics and those defined by the semantics of the target TASM model. Here, the semantics preservation is defined as a strong-simulation equivalence between the TTSs related to the AADL model and to the TASM model respectively, which implies ACTL (the universal fragment of Computation Tree Logic) and ECTL (the existential fragment of Computation Tree Logic) preservation and thus preservation of the UPPAAL query language. After a brief introduction to Coq, we introduce the property language used to express atomic observers. Then, we introduce a generic definition of simulation equivalence based on a mapping function on states and a mapping function on predicates. At last, simulation equivalence is applied in a compositional manner on AADL and the obtained TASM models.

### *6.1. A brief introduction to Coq*

Coq (Bertot and Castéran, 2004) is a theorem prover based on the Calculus of Inductive Constructions which is a variant of type theory, following the "Curry-Howard Isomorphism" paradigm, enriched with support for inductive and co-inductive definitions of data types and predicates. From the specification perspective, Coq offers a rich specification language to define problems and state theorems. From the proof perspective, proofs are developed interactively using tactics, which can reduce the workload of the users. Moreover, the type-checking performed by Coq is the key point of proof verification.

Here, we try to give an intuitive introduction to the Coq terminologies which are used in this paper. In the spirit of "Curry-Howard Isomorphism" paradigm, types may represent programming data-types or logical propositions. So, the Coq objects used in this paper can be sorted into two categories: the *Type* sort and the *Prop* sort:

- *Type* is the sort for data types and mathematical structures, i.e. well-formed types or structures are of type *Type*. Data types can be basic types such as *nat*, *bool*, *nat* → *nat*, etc., and can be *Inductive* structures or *Record*. We use *Fixpoint* definitions to define functions over inductive data types.
- *Prop* is the sort for propositions, i.e. well-formed propositions are of type *Prop*. We can define new predicates using *Inductive* and *Record* (for conjunctions of properties).

### 6.2. *Properties*

In order to express complex properties on the models, we first introduce the atomic observers (*Predicate*) as for example "*port p of thread th has value v*", "*thread th is in the given state s*", and "*thread th gets the CPU*". From these observers, we build propositional formulas (*LP*), and give their semantics over the TTS states (through the *LPSat* function).

We now give the corresponding Coq definitions:

```
Variable Predicate: Type.
Inductive LP: Type :=
  LPPred: Predicate → LP
| LPAnd: LP → LP → LP
| LPNot: LP → LP.

Variable State: Type.
Variable Sat: State → Predicate → Prop.
Fixpoint LPSat st f: Prop :=
  match f with
    LPPred p ⇒ Sat st p
  | LPAnd f1 f2 ⇒ LPSat st f1 ∧ LPSat st f2
  | LPNot f1 ⇒ ¬LPSat st f1
  end.
```

Complex properties expressed using such as LTL, CTL, ACTL and ECTL logics are built from these atomic observers. In order to automatically verify them, the high-level model (AADL) is transformed into the lower-level model (TASM) which can be checked and so that properties are preserved. We consider the ACTL and ECTL logics, which are preserved through the simulation equivalence.

### 6.3. *Simulation equivalence*

Here, we just give a generic definition of strong simulation between two TTSs named TTS_A (abstract semantics) and TTS_C (concrete semantics) respectively, including a mapping function on states and a mapping function on predicates. It will be concretized as two instances in the proof, one for each direction (Section 6.4).

#### 6.3.1. *Mapping function on states*

We introduce a rather complex definition of a mapping between the set of states (*StateC*) of TTS_C and the set of states (*StateA*) of TTS_A. However, in order to build the image of abstract state, some other information may be needed. For this purpose, we introduce an auxiliary state (*mState*) and define its evolution (*mInit*, *mNext* and *mDelay*).

Consequently, a generic definition of the mapping function on states is defined in Coq as follows:

```
Record mapping: Type := mkMapping {
  mState: Type;
  mInit: StateC → mState;
  mNext: mState → StateC → mState;
  mDelay: mState → StateC → Time → mState;
  mabs: mState → StateC → StateA
}.
```

#### 6.3.2. *Mapping function on predicates*

During the model transformation, atomic observers defined on the StateC must be expressed using a formula defined with atomic observers over StateA (the *p2lp* function). This transformation is extended (*LPTransfo*) to any complex formula (*f*) over StateC.

The corresponding Coq definition is given as follows:

```
Fixpoint LPTransfo Pred1 Pred2 p2lp
          (f: LP Pred1): LP Pred2:=
 match f with
   LPPred p ⇒ p2lp p
 | LPAnd f1 f2 ⇒ LPAnd _ (LPTransfo Pred1 Pred2 p2lp f1)
                          (LPTransfo Pred1 Pred2 p2lp f2)
 | LPNot f1 ⇒ LPNot _ (LPTransfo Pred1 Pred2 p2lp f1)
 end.
```

#### 6.3.3. *Strong simulation*

We now introduce the definition of strong simulation. It is based on the mapping function on states and the mapping function on predicates. The simulation relation takes as parameters the functions *tra* and *trc* which translate common observers (defined by *Pred*)

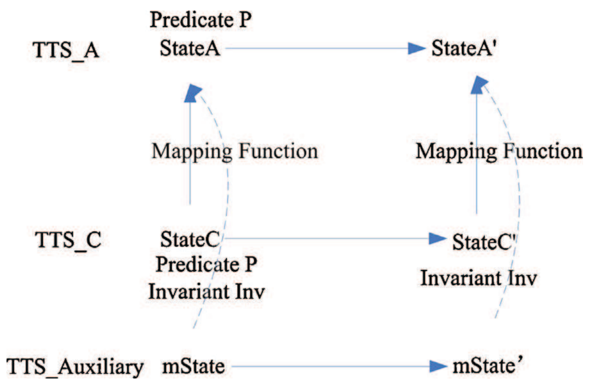


**Fig. 7.** The refined strong simulation.

to propositional formulas on TTS_A and TTS_C respectively. Moreover, in order to restrict the set of considered states, we introduce an invariant over the concrete state (*StateC*) and the auxiliary state (*mState*), which defines a superset of reachable states. It comes to having a partial mapping from concrete states to abstract states. *invInit*, *invNext* and *invDelay* express the invariance property of *inv*. *simuInit*, *simuNext*, *simuDelay* and *simuPred* express the strong simulation property between the two TTSs, including the correspondence of initial state, discrete transitions, continuous transitions, and predicates.

Thus, the definition of strong simulation (Definition 3) is refined, as shown in Fig. 7.

The Coq definition is shown in the following:

```
Variable m: mapping.
Record simu (Pred: Type)(a: TTS StateA)
  (c: TTS StateC)(tra: Pred → LP (Predicate _ a))
  (trc: Pred → LP (Predicate _ c)): Type:= simuPrf {
  inv: (mState m) → StateC → Prop;
  invInit: ∀ st, Init _ c st
    → inv (mInit m st) st;
  invDelay: ∀ ex1 st1 st2 d, Delay _ c st1 d st2
    → inv ex1 st1
    → inv (mDelay m ex1 st1 d) st2;
  invNext: ∀ ex1 st1 st2, Next _ c st1 st2
    → inv ex1 st1
    → inv (mNext m ex1 st1) st2;
  simuInit: ∀ st, Init _ c st
    → Init _ a (mabs m (mInit m st) st);
  simuDelay: ∀ ex1 st1 st2 d, Delay _ c st1 d st2
    → inv ex1 st1
    → Delay _ a (mabs m ex1 st1) d (mabs m (mDelay m ex1 st1 d) st2);
  simuNext: ∀ ex1 st1 st2, Next _ c st1 st2
    → inv ex1 st1
    → Next _ a (mabs m ex1 st1)(mabs m (mNext m ex1 st1) st2);
  simuPred: ∀ ext st, inv ext st
    → (∀ p, LPSat (Satisfy _ c) st (trc p)
    ↔ LPSat (Satisfy _ a)(mabs m ext st)(tra p))
}.
```

We should notice that, in Coq, it is possible to ask the system to infer arbitrary terms, by writing underscores in place of specific values. You may have noticed that we have been calling functions without specifying all of their arguments. For instance, the functions *invInit*, *invDelay* and *invNext* omit the *StateC* argument. Coq's synthesis mechanism automatically inserts underscores for arguments that it will probably be able to infer.

An important property of strong simulation is preservation of ACTL (Baier and Katoen, 2008). More precisely, (1) if TTS_A satisfies an ACTL property, then TTS_C satisfies it also; (2) if TTS_C satisfies an ECTL property, then TTS_A also satisfies it. In the following, we prove strong simulation in both directions, which is called simulation equivalence. Thus it preserves ACTL and ECTL.

## 6.4. Simulation equivalence between the AADL semantics and its TASM transformation

We have given the operational semantics of the subset of AADL in TTS, named AADL_TTS. Moreover, we get a second TTS which is named TASM_TTS. It is the TTS corresponding to the semantics of the TASM model obtained by translating the considered AADL model. The TTS semantics of the subset of AADL, its translation to TASM and the TTS semantics of TASM, are constructed in the proof system (Coq).

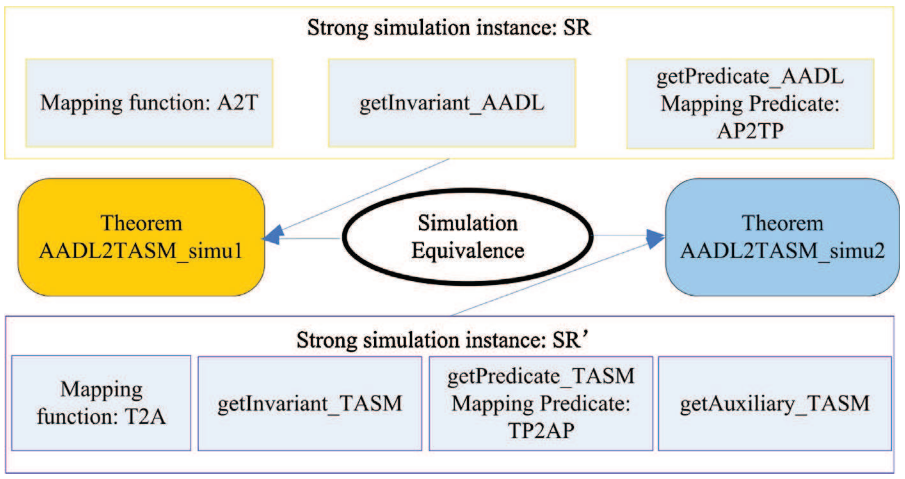


**Fig. 8.** The proof sketch of the strong simulation equivalence.

Here, we use a subset of the proof mechanized by Coq to interpret the idea, which takes into account delayed connections between periodic threads.

### *6.4.1. Compositional proof methodology*

An AADL model can contain several periodic threads. We would like to prove the simulation equivalence between the operational semantics of a thread and its translational semantics in TASM, and then the full semantics will be preserved by using synchronous product in both sides. This method is called as compositional proof and it will reduce the complexity of the proof.

The theorem of simulation of synchronous product is given as follows.

**Theorem 1.** *Let* $\prod_{i=1,\ldots,n} TTS(A_i)$ *be the synchronous product of* $TTS(A_i)$*, and* $\prod_{i=1,\ldots,n} TTS(B_i)$ *be the synchronous product of* $TTS(B_i)$*. If* $\forall i TTS(A_i) \preceq TTS(B_i)$*, then* $\prod_{i=1,\ldots,n} TTS(A_i) \preceq \prod_{i=1,\ldots,n} TTS(B_i)$*.*

The proof of this theorem relies on the definition of simulation relation, including the correspondence of *invInit*, *invNext*, *invDelay*, *simuInit*, *simuNext*, *simuDelay*, and *simuPred*.

### *6.4.2. Applying the compositional principle*

Firstly, the abstract syntax and the operational semantics of the subset of AADL, the abstract syntax and the operational semantics of TASM (named MM_TTS), and the translation from AADL to TASM are specified in Coq.

Secondly, we prove the following theorems:

**Theorem 2.** *For every periodic thread with delayed connections* $Th(i)$*, its operational semantics is strongly simulated by its translational semantics into TASM, noted* $TTS(Th(i)) \preceq MM\_TTS(Trans\_Thread(Th(i)))$*.*

**Theorem 3.** *For every periodic thread with delayed connections* $Th(i)$*, its translational semantics into TASM is strongly simulated by its operational semantics, noted* $MM\_TTS(Trans\_Thread(Th(i))) \preceq TTS(Th(i))$*.*

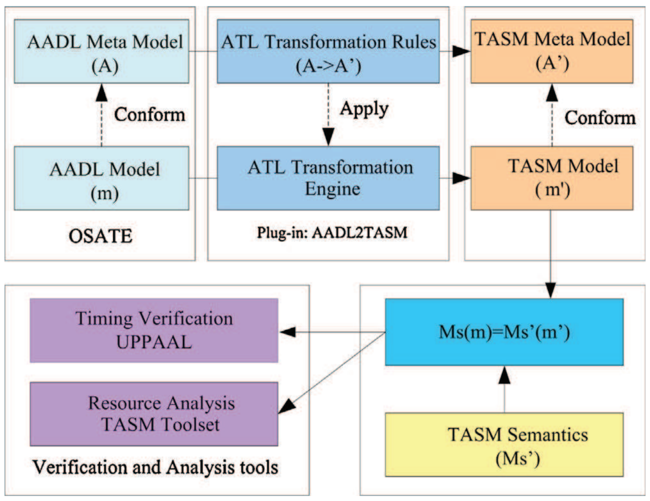
Here, $TTS(Th(i))$ is the synchronous product of $TTS_{thread\_basic}$ and $TTS_{dispatcher}$ defined in Definitions 6 and 7, respectively. $Trans\_Thread(Th(i))$ is the parallel composition of $TASM\_Thread(th)$ and $Periodic\_Dispatcher(th)$ defined in Listing 3.

The sketch of the proof is shown in Fig. 8. Theorem *AADL2TASM_simu1* is declared for the direction from AADL to TASM, and Theorem *AADL2TASM_simu2* is declared for the direction from TASM to AADL. The generic definition of strong simulation is concretized as two instances: *SR* and *SR'*.

- In *SR*, the generic definition of mapping function on states is concretized as *A2T*: $S \to \{env : Env\,;\, upds : Upds\}$, namely, according to the state *S* in the AADL_TTS, we can get the corresponding environment *Env* and the next update set *Upds* in the TASM_TTS; the generic definition of mapping function on predicates is concretized as *AP2TP*; we also construct invariant instances (*getInvariant_AADL*) on the AADL_TTS to restrict to a superset of reachable states.
- In *SR'*, the generic definition of mapping function on states is concretized as *T2A*: $Env \times mState \to S$, namely, according to the environment *Env* in the TASM_TTS, we can get the corresponding *S* in the AADL_TTS, some information are given explicitly in the AADL_TTS, but they are implicit in the TASM rules, so these information need to be complemented into the TASM_TTS (*getAuxiliary_TASM*); the generic definition of mapping function on predicates is concretized as *TP2AP*; we also construct invariant instances (*getInvariant_TASM*) on the TASM_TTS.

The actual definition of *A2T* and *T2A* are complex, because they depend on the management of the TASM update set which introduces delays in assignments.

A Coq formalization of a subset of the paper contents restricted to delayed connections can be downloaded at http://www.irit.fr/ Jean-Paul.Bodeveix/COQ/AADL2TASM/V0/. The proof represents about 1200 lines of Coq. The definitions of TTS, synchronous product, simulation relation and some relative lemmas represent 13% of this code. The abstract syntax and formal semantics of



AADL Meta Model (A)
ATL Transformation Rules (A->A')
TASM Meta Model (A')
Conform
Apply
Conform
AADL Model (m)
ATL Transformation Engine
TASM Model ( m')
OSATE
Plug-in: AADL2TASM
Timing Verification UPPAAL
Ms(m)=Ms'(m')
Resource Analysis TASM Toolset
TASM Semantics (Ms')
Verification and Analysis tools

**Fig. 9.** The model transformation tool prototype AADL2TASM.

both AADL and TASM represent 27% of the code. The transformation from AADL to TASM represents 10% of this code. The rest of the code corresponds to the simulation equivalence proofs (50% of the code).

## 7. Towards a verified tool chain

Actually, the first version of the model transformation tool prototype has been implemented in ATL (ATLAS Transformation Language) directly. It is a good way to provide a basis for the Coq mechanization. Finally, we envision the extraction of a complete tool from the mechanization in the future, that is the verified tool chain. Here we present the prototype.

The prototype is named as AADL2TASM. It has been designed as a plug-in of the OSATE modelling environment, as shown in Fig. 9.

The model transformation language ATL is used as the automatic transformation engine to express the mapping from AADL models to TASM models. The source meta-model is the standard AADL meta-model, and the destination one is defined using the KM3 language.

The first technique of verification is achieved with UPPAAL by mapping each machine to a timed automaton. Here we reuse the translation principle from TASM to UPPAAL given by the authors of TASM (Ouimet, 2008), and implement it using ATL. The verification properties can be expressed by the UPPAAL's query language which is a subset of ACTL ∪ ECTL. Notice that, the behavior annex is an abstraction of the real system behavior where complex data manipulations are replaced by their duration (i.e. $Time(BA\_tr)$ as shown in Listing 8). Moreover, the translation to UPPAAL accepts the data types that are recognized by UPPAAL (records, arrays, bounded integers, enumerated types). Furthermore, as with any model checking approach, the explored state space must not be too large, so the data usage inside the annex should not be complex. In these contexts, we can verify data related safety, and liveness properties as well as schedulability (through deadlock-detection). Other real-time properties (which need observer automata) will be considered in the future.

The second technique of verification is by using TASM Toolset to analyze the timing sequences and resource consumptions. First, it generates graphs showing the time progression of individual main machines, to perform different kinds of analysis, such as average time consumption and minimal and maximal time consumption regarding a specific rule. Second, it generates graphs showing the aggregate resource consumption for each resource versus the time axis, to calculate the minimum, maximum, and average resource consumption for each resource.

In order to extract a complete verified tool chain, a research perspective is to consider how executable code for dedicated model transformation language such as ATL could be extracted from Coq.

## 8. Related work

The AADL description language is a widely used and largely studied standard for the specification of embedded real-time architectures in the avionics and aerospace industry. To extract executable specifications from AADL descriptions in a way suitable for formal verification, one of the easiest ways certainly is to use model transformation technologies, while taking the outmost care to guarantee the preservation of the specification's meaning through the process of its translation in the target modeling framework.

### *8.1. AADL model transformations*

Chkouri et al. (2008) translate most of the AADL concepts into the BIP language, except for the mode change. BIP is a framework for modeling heterogeneous real-time components using three layers: the lower layer describes behaviors, the intermediate layer describes interactions, and the upper layer describes scheduling policies. It provides two categories of verification properties: deadlock detection by using the tool Aldebaran, and some simple timing properties using observers.

Rolland et al. (2008) translate the AADL models into TLA+. TLA+expresses discrete time only. Moreover, the only tool for TLA+is TLC and it is not very efficient.

The translation proposed by Sokolsky et al. (2006) mainly focus on the schedulability analysis of AADL models, a smaller subset (modes and the behavior annex are excluded) is translated into the real-time process algebra ACSR, and use the ACSR-based tool VERSA to explore the state space of the model, looking for violations of timing requirements. ACSR can express the notation of resource explicit in system models.

Abdoul et al. (2008) translate a behavioral subset, minus mode changes, to IF, where they can analyze some safety properties of the system. The behaviors are, however, not expressed using AADL's behavior annex, but their own behavioral language.

In the work of Berthomieu et al. (2009), a synchronous subset of AADL with the behavior annex is translated into Fiacre, and then the Fiacre model is compiled into Timed Petri Nets (TPN) for verification. However, the paper does not explain their semantics, and it just presents the translation principle.

In the work of Ölveczky et al. (2010), a subset of AADL is translated into Real-Time Maude, and it provides an AADL simulator and LTL model checking tool called AADL2Maude. Real-Time Maude is a formal real-time rewriting logic.

The translation proposed by Jahier et al. (2007) covers a subset of AADL except for the behavior annex, and is instead given as a program in the synchronous language Lustre. The translation into Polychrony (Ma et al., 2009) mainly focuses on the multi-partitions structure of an embedded architecture and aims at simulating and verifying a GALS (globally asynchronous locally synchronous) model from the given AADL specifications.

Monteverde et al. (2008) propose Visual Timed Scenarios (VTS) as a graphical language for the specification of AADL behavioral properties. Then a translation from VTS to Timed Petri Nets is presented. Model-checking of properties expressed in VTS is enabled using TPN-based tools.

Rugina et al. (2008) propose a dependability analysis tool (called ADAPT) on the AADL models. Its input is an AADL model annotated with dependability-related information (through the AADL error model annex), and its output is a dependability evaluation model in the form of a Generalized Stochastic Petri Net (GSPN). The GSPN model can be processed by existing dependability evaluation tools, to compute quantitative measures such as reliability, availability, etc.

Bozzano et al. (2011, 2010) present a graphical toolset, called the COMPASS platform, for verifying AADL models by capturing functional, probabilistic and hybrid aspects. Analyses are implemented on top of mature model checking tools and range from requirements validation to functional verification, safety assessment via automatic derivation of FMEA (Failure Mode and Effects Analysis) tables and dynamic fault trees, to performability evaluation, and diagnosability analysis.

The problem of formally analyzing AADL models is an absolutely crucial problem that has been addressed by a number of research groups recently. Most of those efforts have targeted subsets of the large AADL for "functional analysis", "schedulability analysis", or "dependability analysis", etc. In this work, we consider a behavioral subset for functional analysis. Compared with the existing approaches, the main advantage of TASM is its ability to manage resources. Indeed, now we just consider simple resource information such as processor usage in the transformation. However, all the resource declarations contained in an AADL model can be translated to TASM resources. Their usage could be checked by dedicated predicates. Then it would be possible to verify that a given resource is not overloaded. Furthermore, a formal proof of the semantics preservation of the transformation has not been considered by them. Thus, the novel aspects of this work are the verification of the AADL transformation and the ability to consider resource usage.

## *8.2. The verification methods of model transformations*

There are many approaches to gain assurance that the transformation or the translation of a specification or a program is semantic-preserving. This can be done by directly building a theorem-prover-verified compiler (Leroy, 2012, 2009), by using translation validation (Pnueli et al., 1998; Necula, 2000), proof-carrying code (Necula, 1997), semantics-anchoring method (Chen et al., 2005; Narayanan and Karsai, 2006), etc.

A verified compiler is one specified with a theorem prover to formally check that its transformation algorithm defines a perfect match between the semantics of the source and target languages. This is the approach adopted in this paper.

Translation validation consists of using software model-checking techniques to verify the equivalence between a program and its translation.

Proof-carrying consists of returning the transformed program together with a proof of its correctness (the traces of deductions which yield the program) and to check that proof afterwards.

Semantics anchoring is a technique to specify the semantics of DSMLs (Domain Specific Modeling Languages). It relies on the observation that the semantics of DSMLs can be represented by a small set of semantics units such as FSMs (Finite State Machines), Timed Automata, etc., and these semantics units have a common semantics framework, that is ASMs (Abstract State Machines). Based on the semantics-anchoring method, the source and target models can be transformed to the semantics unit respectively, thus verify their equivalence.

Except for semantics preservation, there is another viewpoint for the verification of model transformations, such as the work of Guerra et al. (2013) and Kessentini et al. (2011). They verify some transformation requirements or properties on the execution of actual model transformations written by a dedicated language like ATL, QVT or Kermeta. Lin et al. (2005) and Mottu et al. (2008) research model transformation testing methods. Cariou et al. (2009) propose to specify model transformation using pre and post conditions on the source and target meta-model expressed using OCL (Object Constraint Language) constraints. Cabot et al. (2010) propose verification and validation techniques for model-to-model transformations based on the analysis of a set of OCL invariants automatically derived from the declarative description of the transformations. These invariants state the conditions that must hold between a source and a target model in order to satisfy the transformation definition, i.e. in order to represent a valid mapping.

In the area of AADL research, the related work always use manual validation. The work of Bodeveix et al. (2005) formalizes a subset of the AADL meta-model using the B specification language and the Isabelle proof assistant in order to prove transformations of AADL models correct. Filali-Amine and Lawall (2010) define a refinement of a synchronous subset of AADL using the Event B method, and give simplified proof obligations. Bertrand et al. (2008) translate the AADL mode change protocol into Timed Petri Nets, and verify some properties to validate the proposed translation. Another way is to compare alternative semantics for the same language. In one of our preview work (Pi et al., 2009), we present a comparative study of Fiacre and TASM to define an AADL subset.

## 9. Conclusion and future work

We have presented a translation of AADL into TASM and a methodology to prove the semantics preservation under the assumption that the reference semantics expressed in TTS are correct. However, giving two semantics, the TTS-based one and the translational-based one, which are proved to be equivalent, increases the confidence on the interpretation of the informal semantics. Generally, we hope the idea, i.e., writing a prototype first, doing mechanization based on the prototype and then extracting a verified model transformation tool from the mechanization, can be common to other model to model transformation processes.

The subset of AADL we consider includes periodic threads, data port communications, mode changes and the behavior annex. First, the reference semantics is formalized as a TTS. Second, the translational semantics is formalized by a family of semantics function which inductively associates a TASM fragment with each AADL construct. Third, the theorems and proof of simulation equivalence between the two formal semantics are mechanized by the Coq proof assistant. Our formal semantics and proof of semantics preservation for AADL is an important contribution to the use of AADL for developing safety-critical applications.

Our experience is encouraging, but much more work remains ahead.

Firstly, increasingly larger AADL subsets should be formalized to achieve the goal of giving a formal semantics to the entire AADL standard and having verification and simulation tools for AADL models based on such semantics. For example, a larger subset including such as sporadic and aperiodic thread, event port, event data port, data component, complex scheduling, etc., has been considered in our tool prototype. We will consider this larger subset in the formal proof part in future.

Secondly, in this paper, we just consider simple resource information in the transformation. However, all the resource declaration in the AADL models can be translated to TASM resources. Their usage could be checked by dedicated predicates. Then it would be possible to verify that a given resource is not overloaded. To add such capability, we will extend the definition of TTS with resource information. Intuitionally, the resource properties are also preserved during the transformation.

Thirdly, we will consider how executable code for dedicated model transformation language such as ATL could be extracted from Coq.

## Acknowledgements

We would like to thank the anonymous referees for their comments that helped to substantially improve the quality of the paper. Zhibin Yang would like to thank Mr. Mamoun Filali at IRIT for many help and suggestions.

This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 61073013, 61003017), the Aviation Science Foundation of China under Grant 2012ZC51025, the TOPCASED Project, and the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

## References

Abdoul, T., Champeau, J., Dhaussy, P., Pillain, P.Y., Roger, J.-C., 2008. AADL execution semantics transformation for formal verification. In: ICECCS, IEEE Computer Society, pp. 263–268.

Arnold, A., 1994. Finite Transition Systems – Semantics of Communicating Systems. Prentice Hall International Series in Computer Science. Prentice Hall.

Baier, C., Katoen, J.-P., 2008. Principles of Model Checking. MIT Press.

Behrmann, G., David, A., Larsen, K.G., 2004. A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (Eds.), SFM. Vol. 3185 of Lecture Notes in Computer Science. Springer, pp. 200–236.

Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H., 2005. Comparison of different semantics for time petri nets. In: Peled, D., Tsay, Y.-K. (Eds.), ATVA. Vol. 3707 of Lecture Notes in Computer Science. Springer, pp. 293–307.

Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H., 2008. When are timed automata weakly timed bisimilar to time petri nets? Theor. Comput. Sci. 403 (2–3), 202–220.

Berthomieu, B., Bodeveix, J.-P., Chaudet, C., Dal-Zilio, S., Filali, M., Vernadat, F., 2009. Formal verification of AADL specifications in the topcased environment. In: Kordon, F., Kermarrec, Y. (Eds.), Ada-Europe. Vol. 5570 of Lecture Notes in Computer Science. Springer, pp. 207–221.

Bertot, Y., Castéran, P., 2004. Interactive Theorem Proving and Program Development (CoqArt: The Calculus of Inductive Constructions). Texts in Theoretical Computer Science.

Bertrand, D., Déplanche, A.-M., Faucou, S., Roux, O.H., 2008. A study of the AADL mode change protocol. In: ICECCS, IEEE Computer Society, pp. 288–293.

Bodeveix, J.-P., Chemouil, D., Filali, M., Strecker, M., 2005. Towards formalising AADL in proof assistants. Electr. Notes Theor. Comput. Sci. 141 (3), 153–169.

Börger, E., 2002. The origins and the development of the ASM method for high level system design and analysis. J. Univers. Comput. Sci. 8 (1), 2–74.

Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M., 2011. Safety, dependability and performance analysis of extended AADL models. Comput. J. 54 (5), 754–775.

Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M., Wimmer, R., 2010. A model checker for AADL. In: Touili, T., Cook, B., Jackson, P. (Eds.), CAV. Vol. 6174 of Lecture Notes in Computer Science. Springer, pp. 562–565.

Cabot, J., Clarisó, R., Guerra, E., de Lara, J., 2010. Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. 83 (2), 283–302.

Cariou, E., Belloir, N., Barbier, F., Djemam, N., 2009. OCL contracts for the verification of model transformations. ECEASST, 24.

Chen, K., Sztipanovits, J., Abdelwahed, S., Jackson, E.K., 2005. Semantic anchoring with model transformations. In: ECMDA-FA, pp. 115–129.

Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J., 2008. Translating AADL into BIP – application to the verification of real-time systems. In: Chaudron, M.R.V. (Ed.), MoDELS Workshops. Vol. 5421 of Lecture Notes in Computer Science. Springer, pp. 5–19.

Cleenewerck, T., Kurtev, I., 2007. Separation of concerns in translational semantics for DSLs in model engineering. In: Cho, Y., Wainwright, R.L., Haddad, H., Shin, S.Y., Koo, Y.W. (Eds.), SAC. ACM, pp. 985–992.

CMU, 2006. An Extensible Open Source AADL Tool Environment (OSATE).

Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X., 2009. Essay on semantics definition in MDE – an instrumented approach for model verification. JSW 4 (9), 943–958.

Delange, J., Pautet, L., Feiler, P.H., 2009. Validating safety and security requirements for partitioned architectures. In: Kordon, F., Kermarrec, Y. (Eds.), Ada-Europe. Vol. 5570 of Lecture Notes in Computer Science. Springer, pp. 30–43.

Feiler, P.H., Gluch, D.P., 2013. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley.

Filali-Amine, M., Lawall, J.L., 2010. Development of a synchronous subset of AADL. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (Eds.), ASM. Vol. 5977 of Lecture Notes in Computer Science. Springer. , pp. 245–258.

França, R.B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., Thomas, D., 2007. The AADL behaviour annex – experiments and roadmap. In: ICECCS, IEEE Computer Society, pp. 377–382.

Gargantini, A., Riccobene, E., Scandurra, P., 2009. A Semantic framework for metamodel-based languages. Autom. Softw. Eng. 16 (3-4), 415–454.

Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R., 2006. Towards verified model transformations. In: Proceedings of MODEVA Workshop Associated to MODELS06, pp. 78–93.

Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., 2013. Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. 20 (1), 5–46.

Henzinger, T.A., Manna, Z., Pnueli, A., 1994. Temporal proof methodologies for timed transition systems. Inform. Comput. 112, 273–337.

Hune, T., Nielsen, M., 1998. Timed bisimulation and open maps. In: Brim, L., Gruska, J., Zlatuska, J. (Eds.), MFCS. Vol. 1450 of Lecture Notes in Computer Science. Springer, pp. 378–387.

Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D., 2007. Virtual execution of AADL models via a translation into synchronous programs. In: Kirsch, C.M., Wilhelm, R. (Eds.), EMSOFT. ACM, pp. 134–143.
Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P., 2006. ATL: a QVT-like transformation language. In: Tarr, P.L., Cook, W.R. (Eds.), OOPSLA Companion. ACM, ISBN 1-59593-491-X, pp. 719–720.
Kessentini, M., Sahraoui, H.A., Boukadoum, M., 2011. Example-based model-transformation testing. Autom. Softw. Eng. 18 (2), 199–224.
Lano, K., Rahimi, S.K., 2013. Constraint-based specification of model transformations. J. Syst. Softw. 86 (2), 412–436.
Leroy, X., 2009. Formal verification of a realistic compiler. Commun. ACM 52 (7), 107–115.
Leroy, X., 2012. Mechanized semantics for compiler verification. In: Jhala, R., Igarashi, A. (Eds.), APLAS. Vol. 7705 of Lecture Notes in Computer Science. Springer, pp. 386–388.
Lin, Y., Zhang, J., Gray, J.,2005. A testing framework for model transformations. In: Model-Driven Software Development – Research and Practice in Software Engineering. Springer, pp. 219–236.
Ma, Y., Talpin, J.-P., Shukla, S.K., Gautier, T., 2009. Distributed simulation of AADL specifications in a polychronous model of computation. In: Chen, T., Serpanos, D.N., Taha, W. (Eds.), ICESS. IEEE, pp. 607–614.
Milner, R., 1989. Communication and Concurrency. PHI Series in Computer Science. Prentice Hall.
Monteverde, D., Olivero, A., Yovine, S., Braberman, V., dic 2008. VTS-based Specification and Verification of Behavioral Properties of AADL Models. Toulouse, Francia.
Mottu, J.-M., Baudry, B., Traon, Y.L., 2008. Model transformation testing: oracle issue. In: ICST Workshops. IEEE Computer Society, pp. 105–112.
Narayanan, A., 2008 May. Verification of Model Transformations. Vanderbilt University, USA (PhD thesis).
Narayanan, A., Karsai, G., 2006. Using semantic anchoring to verify behavior preservation in graph transformations. In: ECEASST 4.
Narayanan, A., Karsai, G., 2008. Towards verifying model transformations. Electr. Notes Theor. Comput. Sci. 211, 191–200.
Necula, G.C., 1997. Proof-carrying code. In: Lee, P., Henglein, F., Jones, N.D. (Eds.), POPL. ACM Press, pp. 106–119.
Necula, G.C., 2000. Translation validation for an optimizing compiler. In: Lam, M.S. (Ed.), PLDI. ACM, pp. 83–94.
Ölveczky, P.C., Boronat, A., Meseguer, J., 2010. Formal semantics and analysis of behavioral AADL models in real-time maude. In: Hatcliff, J., Zucca, E. (Eds.), FMOODS/FORTE. Vol. 6117 of Lecture Notes in Computer Science. Springer, pp. 47–62.
Ouimet, M., 2008. A Formal Framework for Specification-Based Embedded Real-Time System Engineering. MIT (PhD thesis).
Ouimet, M., Berteau, G., Lundqvist, K., 2006. Modeling an electronic throttle controller using the timed abstract state machine language and toolset. In: Kühne, T. (Ed.), MoDELS Workshops. Vol. 4364 of Lecture Notes in Computer Science. Springer, pp. 32–41.
Ouimet, M., Lundqvist, K., 2006. The TASM Language Reference Manual Version 1.1.
Ouimet, M., Lundqvist, K., 2007. The TASM toolset: specification, simulation, and formal verification of real-time systems. In: Damm, W., Hermanns, H. (Eds.), CAV. Vol. 4590 of Lecture Notes in Computer Science. Springer, pp. 126–130.
Ouimet, M., Lundqvist, K., 2008. The timed abstract state machine language: abstract state machines for real-time system engineering. J. UCS 14 (12), 2007–2033.
Pi, L., Yang, Z., Bodeveix, J.-P., Filali, M., Hu, K., Ma, D., 2009. A comparative study of FIACRE and TASM to define AADL real time concepts. In: ICECCS. IEEE Computer Society, pp. 347–352.
Pnueli, A., Siegel, M., Singerman, E., 1998. Translation validation. In: Steffen, B. (Ed.), TACAS. Vol. 1384 of Lecture Notes in Computer Science. Springer, pp. 151–166.
Rolland, J.-F., Bodeveix, J.-P., Filali, M., Chemouil, D., Thomas, D., 2008. Modes in asynchronous systems. In: ICECCS. IEEE Computer Society, pp. 282–287.
Rugina, A.-E., Kanoun, K., Kaâniche, M., 2008. The ADAPT tool: from AADL architectural models to stochastic petri nets through model transformation. In: EDCC, IEEE Computer Society, pp. 85–90.
SAE, 2009. AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0.
SAE, 2011. AS5506 Annex: Behavior_Specification V2.0.
Sokolsky, O., Lee, I., Clarke, D.,2006. Schedulability analysis of AADL models. In: IPDPS. IEEE.
van Vliet, J.C., 2008. Software Engineering-Principles and Practice, 3rd ed. Wiley.
Walker, M., Reiser, M.-O., Tucci-Piergiovanni, S., Papadopoulos, Y., Lönn, H., Mraidha, C., Parker, D., Chen, D., Servat, D., 2013. Automatic optimisation of system architectures using EAST-ADL. J. Syst. Softw. 86 (10), 2467–2487.
Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H., 2007. Towards automatic model synchronization from model transformations. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (Eds.), ASE. ACM, pp. 164–173.

**Zhibin Yang** received his PhD degree in Computer Science from Beihang University, Beijing, China, in February 2012. Since April 2012, he has been a postdoc in IRIT, University of Toulouse, France. His research interests include safety-critical real-time systems, formal verification, AADL, synchronous languages, and multi-core systems.

**Kai Hu** is an associate professor at Beihang University, China. He received his PhD degree from Beihang University in 2001. From 2001 to 2004, he did the post-doctoral research at Nanyang Technological University, Singapore. Since 2004, he is the leader of the team of LDMC in the Institute of Computer Architecture (ICA), Beihang University. His research interests concern embedded real time systems and high performance computing. He has good cooperation with IRIT and INRIA at France on study of AADL and synchronous languages.

**Dianfu Ma** is a professor at Beihang University, China. He was the executive director of Chinese Computer Federation, the secretary of the steering committee of Computer Science and Technology Education in Ministry of Education of China. He is the vice director of SOA standards working group under the steering committee of China National Information Technology Standardization. He took charge of the National Basic Research Program (also called 973 Program), National High-tech 863 Program, National Natural Science Foundation of China, Key Technologies Research and Development Program, etc. He has published more than 50 academic papers in international journals or conferences. He received the 3rd prize of Science and Technology Innovation Award from Ministry of Education of China in 2003, and 1st prize of Science and Technology Innovation Award of Beijing in 2011. His research interesting includes services computing, real-time systems and high dependable software.

**Jean-Paul Bodeveix** received a PhD of Computer Science from the University of Paris-Sud 11 in 1989. He has been assistant professor at University of Toulouse III since 1989 and is now professor of computer science since 2003. His main research interests concern formal specifications, automated and assisted verification of protocols as well as of proof environments. He has participated in European and national projects related to these domains. His current activities are linked to the study of real time and synchronous language semantic properties within proof assistants and to real-time modeling and verification.

**Lei Pi** obtained his PhD degree in Computer Science from ISAE (France) with a thesis on model-based methodologies for the design, analysis and implementation of high-integrity real-time systems. He works today as a project manager for INTECS, following projects related to avionics, railroad, space and defense industries, and continues R&D in the area of model-based development, in particular metamodelling, model-based analysis and automated code generation.

**Jean-Pierre Talpin** did his PhD Thesis at Ecole des Mines de Paris under the advisory of Pierre Jouvelot, worked three years at the European Computer-Industry Research Centre in Munich and joined INRIA in 1995, in the EPART project-team of Paul Le Guernic. He led INRIA project-team ESPRESSO from 2000 to 2012 and now leads project-team TEA (on time in embedded architectures). He received the 2004 ACM SIGPLAN Award for the most influential POPL paper, with Mads Tofte; the 2012 ACM-IEEE LICS “Test of Time” Award, with Pierre Jouvelot; and the 2014 ITEA Award of Excellence.