From AADL to Timed Abstract State Machine: A Certified Model Transformation

Zhibin Yang·Kai Hu·Dianfu Ma·Jean-Paul Bodeveix·

Lei Pi·Jean-Pierre Talpin

Abstract 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 preserve the semantics of the initial AADL model in the way it is interpreted in the verification or analysis tool, either entirely or at least some specific properties. Moreover, the AADL standard itself lacks at present a formal semantics to make this translation validation possible. Additionally, related work on interpreting or compiling AADL do, for the most of them, give informal explanations on model transformations. This makes the formal proof of semantics preservation altogether impossible. Our contribution is to bridge this gap by providing two formal semantics for a subset of AADL, including periodic threads, data port communications, mode changes, and the AADL behavior annex (automata). Its operational semantics is formalized as a Timed Transition System (TTS). This formalization is here one prerequisite to the formal proof of semantics preservation from the AADL source to our target verification formalism: Timed Abstract State Machine (TASM).

Z. Yang[1,2](✉)
[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]IRIT-CNRS, University of Toulouse, Toulouse, France
e-mail: kenney@cse.buaa.edu.cn, Zhibin.Yang@irit.fr

K. Hu. D. Ma
School of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: hukai@buaa.edu.cn, dfma@buaa.edu.cn

J. Bodeveix
IRIT-CNRS, University of Toulouse, Toulouse, France
e-mail: bodeveix@irit.fr

L. Pi
e-mail: pilei1207@gmail.com

J. Talpin
INRIA-Rennes, Campus de Beaulieu, Rennes, France
e-mail: jean-pierre.talpin@irisa.fr

In this paper, an abstract syntax of (our subset of) AADL is given, together with the abstract syntax of TASM. The translation is formalized by a semantics function in which translation rules associate each AADL construct to a TASM expression using an ML-like language. Then, the proof of simulation equivalence between the TTSs of the AADL and TASM models is formally mechanized by the theorem prover Coq. Finally, the certified translational semantics is automated in the AADL2TASM tool, which provides model checking and simulation for AADL models.

Keywords Architecture Analysis and Design Language (AADL). Model transformation. Semantics preservation. Timed Abstract State Machine (TASM). Formal verification. Coq . ATL

1 Introduction

Embedded real-time systems deployed for example on 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. They require that operational correctness is not limited to functional correctness and timing correctness but must also include resource utilization correctness. Moreover, they are more and more complex, so reducing the development cost and time is an important element of design in these systems.

Architecture Analysis and Design Language (AADL) (SAE 2009) is an architecture description language standard (SAE AS5506) for embedded real-time systems and has its broad use in the safety-critical area. It 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.

A safety-critical system is often required to pass stringent qualification and certification processes before its deployment and provide sufficiently strong evidence of its correctness. 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) (M.Yassin et al. 2008), to TLA+(Jean-François et al. 2008), to real-time process algebra ACSR(Oleg et al. 2006), to IF(Thomas et al. 2008), to Fiacre(Bernard et al. 2009), to Real-Time Maude(Peter Csaba et al. 2010), to Lustre(Erwan et al. 2007), to Polychrony(Yue 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.

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 (Holger et al. 2006; Anantha and Gabor 2006; Esther et al. 2012; Marouane et al. 2011). This work is concerned with semantics preservation.

On one hand, the AADL standard is informally defined (to allow for some flexibility in its possible implementations) yet lacks a formal semantics, making it altogether difficult to start from a sound basis to prove the equivalence of an AADL model with its interpretation in a formal verification model.

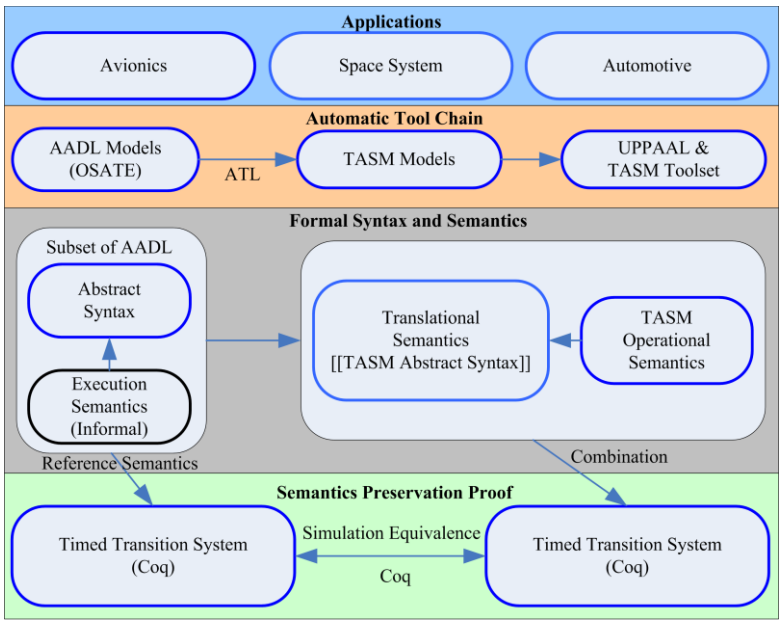
On the other hand, the model transformation also defines a semantics for AADL, called as translational semantics (Benoît et al. 2009; Thomas and Ivan 2007; Angelo et al. 2009), namely, the sentences of the target language express the semantics of AADL. However, in most related work, the modeling concepts of both the AADL and the target language are expressed in natural language and/or with examples and the translation rules are expressed in

natural language or by a pseudo-algorithm. This altogether makes any hope of formal proof of semantics preservation far from possible.

To enable the formal verification of non-functional properties, including timing properties as well as resource consumption properties, we consider transforming AADL models into Timed Abstract State Machine (TASM) (Martin and Kristina 2006; Martin and Kristina 2008). TASM is an extension of the Abstract State Machine formalism (Egon 2002), which supports the specification of resource consumption and timing as well as behavior and communication.

In this work, model checking and simulation are used to verify AADL models through their translation to TASM, while the theorem prover Coq (Coq 2009) is used to prove the methodology, i.e., the correctness of the translation. As shown in Fig.1, we'd like to provide an automatic tool chain for the end users. The model transformation tool AADL2TASM is a plug-in of the AADL modeling environment OSATE(OSATE 2006), which supports model checking using UPPAAL (Behrmann et al. 2004) and simulation using the TASM Toolset (Martin and Kristina 2007). Underlying the tool is the formal translational semantics of AADL by a mapping to TASM. To enable the proof of semantics preservation: (1) the informal execution semantics is formalized directly using Timed Transition System (TTS), considered as a reference operational semantics; (2) combining the translational semantics (expressed by the TASM sentences) with the semantics of TASM, we can obtain a new way to execute an AADL model, and it is constructed as another TTS; (3) if there is a simulation equivalence relation between the two TTSs, we can say the translation preserves the AADL semantics.

Fig. 1 Certified AADL model transformation



The main contributions of this work are: (1) the informal 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 to TASM; (3) a mechanized proof of the semantics preservation of the transformation from AADL to TASM; (4) the AADL2TASM tool, automating the generation of TASM specifications from AADL models based on the certified translational semantics and supporting model checking and simulation.

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 language 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. The AADL2TASM tool and a case study of space Guidance, Navigation, and Control (GNC) subsystem are given in Section 7. Section 8

discusses the related work, and Section 9 gives some concluding remarks.

2 Timed Transition Systems

TTS is often used to compare the semantics of real time specification languages (Bérard et al. 2008; Bérard et al. 2005). However, there are different definitions and operations on TTS for different uses. For example, some definitions use clock variables to express time in TTS (Thomas and Mogens 1998), others express time by associating delays, or time intervals to transitions (Thomas et al. 1994).

Given R^+ the set of nonnegative Reals, we define a TTS as follows.

Definition 1 (Timed Transition Systems). A timed transition system is a tuple $\Gamma = \langle \mathcal{S}, \mathcal{S}^0, \Sigma, P, \rightarrow, L \rangle$ where:

- $\mathcal{S}$ is a set of *states*,
- $\mathcal{S}^0 \in \mathcal{S}$ is the *initial state*,
- $\sum$ is a finite set of *labels*,
- P is a set of *predicates*,
- $\rightarrow \subseteq \mathcal{S} \times (\Sigma \cup R^+) \times \mathcal{S}$ is the *transition* relation,
- $L : \mathcal{S} \rightarrow 2^P$ is a *labeling function*, $L(\mathcal{S})$ maps each state to the set of predicates which are true in that state.

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

- *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 $\Gamma_i = \langle \mathcal{S}_i, \mathcal{S}_i^0, \Sigma_i, P_i, \rightarrow_i, L_i \rangle$, $i = 1 \ldots n$, where the set of predicates P_i are supposed to be pair-wise disjoint. The synchronous product is a TTS: $\Gamma = \prod_{i=1..n} \Gamma_i = \langle \mathcal{S}, \mathcal{S}^0, \Sigma, P, \rightarrow, L \rangle$, such that:

- $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_i \times \ldots \times \mathcal{S}_n$,
- $\mathcal{S}^0 = \mathcal{S}_1^0 \times \ldots \times \mathcal{S}_i^0 \times \ldots \times \mathcal{S}_n^0$,
- $\Sigma = \bigcup_{i=1 \ldots n} \Sigma_i$,
- $P = \bigcup_{i=1 \ldots n} P_i$,
- $\rightarrow$ satisfies the following rules:

$$\frac{\forall i, s_i \xrightarrow{e}_i s_i'}{(s_1, \ldots, s_i, \ldots, s_n) \xrightarrow{e} (s_1', \ldots, s_i', \ldots, s_n')}, e \in \Sigma$$

$$\frac{\forall i, s_i \xrightarrow{d}_i s_i'}{(s_1,...,s_i,...,s_n) \xrightarrow{d} (s_1',...,s_i',...,s_n')}, d \in R^+$$

The rules represent that all TTSs make a step at the same time, and we consider discrete transitions and continuous transitions separately.

- $L = \bigcup_{i=1...n} L_i(\mathbb{S}_i)$.

Bisimulation and its variants (Robin 1989) are usually formulated over TTS. 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 (Christel and Joost-Pieter 2008).

Definition 3 (Strong Simulation). Given an alphabet Σ and two TTSs $\Gamma_1 = \langle \mathbb{S}_1, \mathbb{S}_1^0, \Sigma, P, \rightarrow_1, L_1 \rangle$ and $\Gamma_2 = \langle \mathbb{S}_2, \mathbb{S}_2^0, \Sigma, P, \rightarrow_2, L_2 \rangle$, we say that Γ_2 strongly simulates Γ_1, noted $\Gamma_1 \preceq \Gamma_2$, if there exists a relation $R \subseteq \mathbb{S}_1 \times \mathbb{S}_2$ called a strong simulation relation, such that:

- $\forall s_1^0 \in \mathbb{S}_1^0, \exists s_2^0 \in \mathbb{S}_2^0 : (s_1^0, s_2^0) \in R$,
- $\forall (s_1, s_2) \in R, \forall e \in \Sigma, \forall s_1'$ such that $s_1 \xrightarrow{e}_1 s_1'$, there exists $s_2' \in \mathbb{S}_2$ such that $s_2 \xrightarrow{e}_2 s_2'$ and $(s_1', s_2') \in R$,
- $\forall (s_1, s_2) \in R, \forall d \in R^+, \forall s_1'$ such that $s_1 \xrightarrow{d}_1 s_1'$, there exists $s_2' \in \mathbb{S}_2$ 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 using a mapping function when used in the proof. Additionally, discrete transitions and continuous transitions are also 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 Γ_1 and Γ_2, i.e., two-directions strong simulation, noted $\Gamma_1 \simeq \Gamma_2$.

3 A Subset of AADL

3.1 Overview of the AADL Language

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.

In this paper, we mainly focus on the software parts of AADL. For instance, a thread represents a sequential flow of execution and it is the only AADL component that can be scheduled. A process component defines protected memory that can be accessed by its thread

subcomponents. A subprogram represents a piece of code that can be called by a thread or another subprogram.

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). 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 (Julien et al. 2009a; Julien et al. 2009b) (support of ARINC 653 standard in avionic systems), etc.; (2) the semantics of the execution model is also given, namely, the execution semantics of AADL. However, most of them are defined using natural language.

Moreover, the *behavior annex* (SAE 2011) describes more precisely the behaviors of threads and subprograms. The behavior annex has independent syntax and semantics. However, its semantics is also defined informally.

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 (Thomas et al. 2008)

3.2 The Considered Subset of AADL

AADL execution model mixes synchronous and asynchronous aspects (SAE 2009; Ricardo Bedin et al. 2007; Mamoun and Julia 2010). A synchronous execution model is obtained by considering logically synchronized periodic threads communicating through data ports. In the asynchrony one, it is possible to raise events, to specify sporadic and aperiodic threads, communication through shared variables, and remote procedure calls, etc. In this paper, we mainly 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 Sect. 5.

1) Periodic Thread

In AADL, a thread can be *active* in a specific mode, *inactive* in another mode and *halted*. Only active threads can be dispatched and scheduled for execution.

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

AADL supports the classic 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 => Periodic*.

Several properties can be assigned to a periodic thread, such as: period given by the *Period* property in the form of period=100ms or frequency=10Hz, *Execution Time* such as Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET), and *Deadline*. By default, when the deadline is not specified it equals the period.

2) Data Port Communication

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. Note 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) 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.

However, an AADL model is a tree of components, and each component has one or more operating modes. AADL uses the concept of *system operation mode* (SOM) to define the hierarchical composition of component modes. A SOM is a vector of modes (Dominique 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.

Then *SOM transitions* in which each state is a SOM can be constructed. 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 (Dominique 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*, i.e., the instant where a thread is activated/deactivated or the instant where a connection is added/deleted. The execution model attribute is attached to mode transitions, and expressed as: *Mode_Transition_Response => {emergency, planned}*. To guarantee determinism, we consider the planned one.

4) Behavior Annex

The behavior annex supposed here to act in a thread or a subprogram, and describes the behaviors more precisely. 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) (Martin 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 (*s_etc*), a process component (*cruise*), and several thread components (*command, speed, wheel, throttle, display*). We focus on the modes of the process component, which contains

two modes (*manual*, *automatic*). In *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 *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 connection, and each thread has a behavior annex specification.

We show the *cruise* process and the *throttle* thread as an example and their textual specification is given in Listing 1. 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.

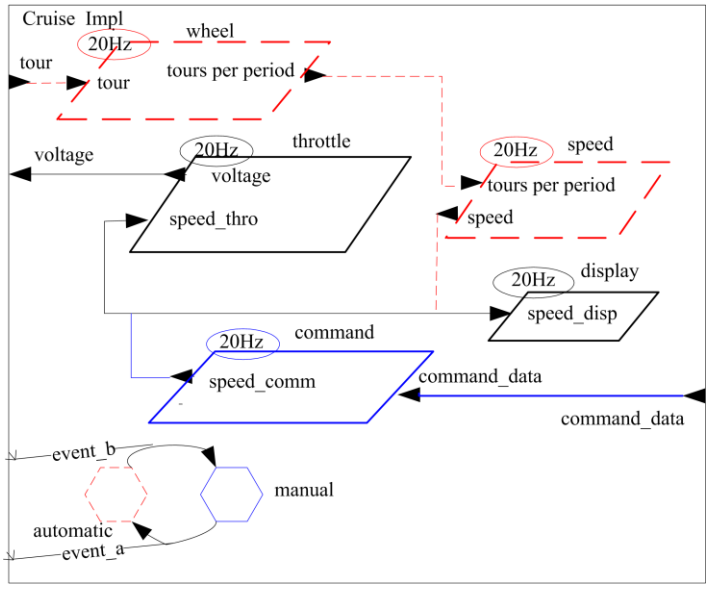


Fig.2 Architecture of the electronic throttle controller system

Listing 1 Textual specification of the AADL model of the ETC system

```
process Cruise
features
  tour: in data port behavior::integer;
  command: in data port behavior::integer;
  ……
end Cruise;
process implementation Cruise.impl
subcomponents
  Thwheel: thread Th_wheel.impl in modes (automatic);
  Thcommand: thread Th_command.impl in modes (manual);
  ……
connections
  data port tour -> Thwheel.tour    in modes (automatic) ;
  ……
modes
  manual : initial mode;
  automatic: mode;
  manual -[ event_a ]-> automatic;
  automatic -[ event_b ]-> manual;
end Cruise.impl;
```

```
thread Th_throttle
features
  speed_thro: in data port behavior::integer;
  voltage: out data port behavior::integer;
end Th_throttle;
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;
```

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 SOM transitions. 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 a uniform abstract syntax.

Please note 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, SOM and EVENT 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.

```
Type Thread :=
 { Period: DURATION;
    BCET: DURATION;
    WCET: DURATION;
    Deadline: DURATION;
    DispatchType: {periodic, aperodic, sporadic};
    Iports: set of PORT;
    Oports: set of PORT;
    Behavior: BehaviorAnnex;
    Modes: set of SOM;
 }
Type Connection :=
 {  SourcePort: PORT;
    ConnectionType: {immediate, delayed};
    DestinationPort: PORT;
    Modes: set of SOM;
 }
Type BehaviorAnnex :=
 { States: set of BASTATE;
   Transitions: set of BA_Transition;
 }
```

```
Type BA_Transition :=
 { SourceState: BASTATE;
   DestinationState: BASTATE;
   Time: DURATION;
   Guard: EXPRESSION;
   Action: (Iports(th)->VALUE) × Oports(th)->VALUE;
 }
Type SOM_Transition :=
{ SourceMode: SOM;
  TransitionType: {emergency, planned};
  DestinationMode: SOM;
  Event : EVENT;
 }
Type Model :=
{ Threads: set of Thread;
  Connections: set of Connection;
  Initial_Mode: SOM;
  Mode_Transitions: set of SOM_Transition;
}
```

4 Timed Abstract State Machine

4.1 A Brief Overview of TASM

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

Definition 5 (TASM Specification). A TASM specification is a pair <E, ASM> where:

- E=<EV, TU, ER> is the environment, including:
- *Environment Variables*, which are the global variables that affect and are updated by machine execution,
- The *Types* of environment variables that include real numbers, integer, boolean, and user-defined types,
- A set of named *Resource*s, ER={(*rn*, *rs*)| *rn* is a resource name, and *rs* is the resource size}. Examples of resources include memory, power, and bus bandwidth.
- ASM=<MV, CV, IV, R> is a machine, including:
- *Monitored Variables*, which are the set of environment variables that affect the machine execution,
- *Controlled Variables*, which are the set of environment variables that the machine updates,
- *Internal Variables*, are the set of local variables, their scope being limited to the machine where they are defined,
- A set of *Rules*, R={<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 [*tmin*, *tmax*], 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 Sect. 3, and we assume power and memory are consumed during execution. *Rule1* defines the actual execution of the thread. *Rule 2* is triggered at completion and updates the thread state at the next dispatch. Note 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 Sect. 5).

Listing 2 A TASM example

```
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:
  Rule 1: 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;
  }
```

4.2 Abstract Syntax of TASM

The abstract syntax of TASM is given in BNF as follows:

P ::= x := exp | skip | if Bexp then P | else then P| time (tmin, tmax) ⊳ P
| time next ⊳ P | resource r (rmin, rmax) ⊳ P | P ⊕ P | P ⊗ P
TASM ::= <E, P || P || ... || P>

In this paper, we just use shared variables to achieve communication. P defines the behaviors of a main machine, x:=exp means update 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.

Due to space limitations, the formalization of the semantics of TASM cannot be given here.

5 Two Formal Semantics for the Subset of AADL

As mentioned in Sect. 1, two formal semantics of AADL are the necessary prerequisites for the formal proof of semantics preservation. The AADL semantics formalized as TTS is an operational style, and its translational semantics is a variant of operational style.

According to the analysis of AADL in Sect. 3, the translation should take into account the structural aspects, the execution model and the behavior annex. The semantics function has two parts: the mapping of structural aspects to the TASM environment (such as thread-related variables, connection-related variables, and mode-related variables), and the mapping of dynamic aspects 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.

$$
\begin{aligned}
&\text{Translate (m:Model)} = \\
&< \bigcup_{th \in m.Threads} \text{Trans\_ThreadData(th)} \cup (\bigcup_{th \in m.Threads} \text{Trans\_BehaviorAnnexData(th)}) \\
&\cup (\bigcup_{cn \in m.Connections} \text{Trans\_ConnectionData (cn)}) \cup (\bigcup_{tr \in m.Mode\_Transitions} \text{Trans\_ModeData(tr)}), \\
&\quad \|_{th \in m.Threads} \text{Trans\_Thread(th)} \ \| \ (\|_{th \in m.Threads} \text{Trans\_BehaviorAnnex(th)}) \\
&\| \ (\|_{cn \in m.Connections} \text{Trans\_Connection(cn)}) \ \| \ (\|_{tr \in m.Mode\_Transitions} \text{Trans\_Mode(tr)}) >
\end{aligned}
$$

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

5.1 Periodic Threads with Data Port Communications

1) Informal Semantics

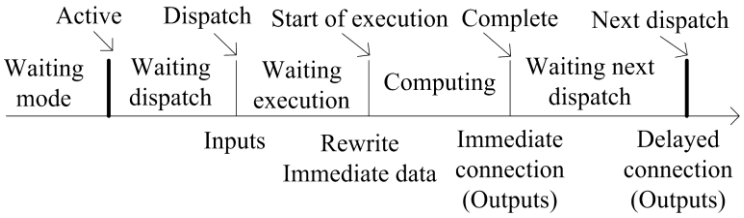
The *current mode* determines the set of threads that are considered *active*. Only the active threads can be dispatched and scheduled for execution and other threads are in the *waiting mode* state or on the *halted* state.

Periodic thread. A periodic thread is *dispatched* periodically by the dispatcher, 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 time (by default).

Data port communication. First, the communication affects the input/output timing of the 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 communication 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.3.

Fig.3 The time line of an AADL periodic thread with data port communications



2) Operational Semantics

The operational semantics is defined as the synchronous product of two TTSs: TTS_{thread_basic} and $TTS_{dispatcher}$.

Definition 6. The operational semantics of a periodic thread with data port communications is a timed transition system $TTS_{thread\_basic} = <\mathbb{S}, \mathbb{S}^0, \Sigma, \rightarrow>$ where:

- $\mathbb{S}$ = {State(th):{waiting_mode,…}, Activation(th):{true, false}, Dispatch(th):{true, false}, Get_CPU(th):{true, false}, Iport(th):Iports(th)->VALUE, Oport(th):Oports(th) ->VALUE, StartofExeTime(th):DURATION, CurrentTime:DURATION}, is the set of states. Here we save the start time of execution of the thread.
- $\mathbb{S}^0$ is considered as {waiting_mode, true, false, false, {ip$\mapsto$0 | ip$\in$Iports(th)}, {op$\mapsto$0 | op$\in$Oports(th)}, 0, 0}.
- Σ is the set of labels.
- $\rightarrow$ is defined by the following rules, including discrete transitions and continuous transitions. ACTIVATION, DISPATCH and WAITING_EXE have two rules respectively, because their behaviors are affected by mode change, dispatcher, and scheduler. S and S' are values of the type $\mathbb{S}$. The schema *S'=S with {}* means we just give the updated fields and $\langle+$ is an overloading operator.

$$\frac{S(State(th)) = waiting\_mode, S(Activation(th)) = true, \quad S' = S\ with\ \{State(th) = waiting\_dispatch\}}{S \longrightarrow S'} \quad \textbf{ACTIVATION[1]}$$

$$\frac{S(State(th)) = waiting\_mode, S(Activation(th)) = false, \quad S' = S\ with\ \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{ACTIVATION[2]}$$

$$\frac{S(State(th)) = waiting\_dispatch, S(Dispatch(th)) = true, \quad S' = S\ with\ \{State(th) = waiting\_execution, Iport(th) = IportBuffer(th)\}}{S \longrightarrow S'} \quad \textbf{DISPATCH[1]}$$

$$\frac{S(State(th)) = waiting\_dispatch, S(Dispatch(th)) = false, \quad S' = S\ with\ \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{DISPATCH[2]}$$

$$\frac{\begin{array}{l} S(State(th)) = waiting\_execution, S(Get\_CPU(th)) = true, \\ S' = S\ with\ \{ \\ \quad State(th) = execution, \\ \quad Iport(th) = S(Iport(th)) \langle + \{ip \mapsto IportBuffer \mid \exists cn \in Connection, \\ \quad\quad DestinationPort(cn) = ip \wedge ConnectionType(cn) = immediate \wedge ip \in Iports(th)\}, \\ \quad StartofExeTime(th) = CurrentTime\} \end{array}}{S \longrightarrow S'} \quad \textbf{WAITING\_EXE[1]}$$

$$\frac{S(State(th)) = waiting\_execution, S(Get\_CPU(th)) = false, \quad S' = S\ with\ \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{WAITING\_EXE[2]}$$

$$\frac{\begin{array}{l} S(State(th)) = execution, \\ S(CurrentTime) = S(StartofExeTime(th)) + t, BCET \le t \le WCET, \\ S' = S\ with\ \{State(th) = completed, \\ \quad Oport(th) = ComputeOutPut(Iport(th))\} \end{array}}{S \longrightarrow S'} \quad \textbf{EXECUTION}$$

$$\frac{\begin{array}{l} S(State(th)) = completed, \\ S' = S\ with\ \{ \\ \quad State(th) = waiting\_next\_dispatch, \\ \quad IportBuffer = S(IportBuffer)\langle + \{th', ip \mapsto S(Oport(th)(op)) \mid \exists cn \in Connection, \\ \quad\quad SourcePort(cn) = op \wedge DestinationPort(cn) = ip \wedge ConnectionType(cn) = immediate \\ \quad\quad \wedge ip \in Iports(th')\} \end{array}}{S \longrightarrow S'} \quad \textbf{WRITE\_DATA}$$

$$\frac{\begin{array}{l} S(State(th)) = waiting\_next\_dispatch, \\ S' = S\ with\ \{CurrentTime = CurrentTime + d\} \end{array}}{S \xrightarrow{d} S'} \quad \textbf{WAIT\_NEXT\_DISPATCH}$$

Definition 7. The operational semantics of the dispatcher of a periodic thread is a timed transition system $TTS_{dispatcher} = <\mathbb{S}, \mathbb{S}^0, \Sigma, \rightarrow>$ where:

- $\mathbb{S}$ = {Activation(th):{true, false}, Dispatch(th):{true, false}, WaitingNextDispatch(th): {true, false}, State(th):{waiting_mode,…}, Iport(th):Iports(th)->VALUE, Oport(th): Oports(th)->VALUE, NextPeriod(th):DURATION, CurrentTime:DURATION}, is the set of states, and we save the time of next period of the thread.
- $\mathbb{S}^0$ is considered as {true, false, false, waiting_mode, {ip$\mapsto$0 | ip$\in$Iports(th)}, {op$\mapsto$0 | op$\in$Oports(th)}, 0, 0}.
- Σ is the set of labels.
- $\rightarrow$ is defined by the following rules. The rule DISPATCH_THREAD is used to dispatch the thread, and synchronize with the thread using the shared variable *Dispatch(th)*. The rule WAITING_PERIOD does nothing, but it is used in the building of product. The rule REACH_PERIOD means that the thread enters the next period, the output of the delay connections is also expressed in this rule, and the final rule is used to deal with the waiting time when the conditions of other rules are not satisfied.

$$\frac{\begin{array}{l} S(Activation(th)) = true, \\ S(State(th)) = waiting\_dispatch, \\ S(Dispatch(th)) = false, \\ S' = S\ with\ \{Dispatch(th) = true, \\ \qquad WaitingNextDispatch(th) = true, \\ \qquad NextPeriod(th) = NextPeriod(th) + Period(th)\} \end{array}}{S \longrightarrow S'} \quad \textbf{DISPATCH\_THREAD}$$

$$\frac{\begin{array}{l} S(WaitingNextDispatch(th)) = true, \\ S(CurrentTime) + d \le S(NextPeriod(th)), \\ S' = S\ with\ \{CurrentTime = CurrentTime + d\} \end{array}}{S \xrightarrow{d} S'} \quad \textbf{WAIT\_PERIOD}$$

$$\frac{\begin{array}{l} S(WaitingNextDispatch(th)) = true, \\ S(CurrentTime) = S(NextPeriod(th)), \\ S' = S\ with\ \{ \\ \quad State(th) = waiting\_mode, \\ \quad WaitingNextDispatch(th) = false, \\ \quad Dispatch(th) = false, \\ \quad IportBuffer = S(IportBuffer)\langle + \{th', ip \mapsto S(Oport(th)(op)) \mid \exists cn \in Connection, \\ \quad\quad SourcePort(cn) = op \wedge DestinationPort(cn) = ip \wedge ConnectionType(cn) = delayed \\ \quad\quad \wedge ip \in Iports(th')\} \end{array}}{S \longrightarrow S'} \quad \textbf{REACH\_PERIOD}$$

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

3) Translational Semantics

The TASM environment associated to each thread is defined as a set of typed variables such as the *state* and *resource consumption* of the thread. Messages exchanged by threads are supposed to be of type Integer. Moreover, an *IportBuffer* variable is defined for each port of each thread. The sender copies values of output ports to the buffer, and the receiver copies values from the buffer to the input ports. The variables associated to connections are defined as well.

Listing 3 The TASM environment of an AADL periodic thread with data port communications

```
Trans_ThreadData (th) =
{ State : { halted, waiting_mode, waiting_dispatch, waiting_execution, execution,
            completed, waiting_next_dispatch} := waiting_mode;
  Iport: Iports(th) -> Integer;
  Oport: Oports(th) -> Integer;
  RscUsage: RESOURCES-> Integer;
  WaitingNextDispatch: Boolean;
  …
}
Trans_ConnectionData(cn) =
{ ConnectionType: {immediate, delayed};
    …
}
```

The basic behavior of a periodic thread is expressed by a main machine with six rules: *Activation*, *Dispatch*, *Waiting Execution*, *Execution*, *Write Data*, and *Waiting Next Event*.

- *Activation*. This rule is used to deal with the relation between periodic threads and mode change (in Sect. 5.2). A family of shared variables *Activation(th):{true, false}* are used.
- *Dispatch*. We use a machine to express the dispatcher and it sends the dispatch signal to the machine associated to the thread. A family of shared variables *Dispatch(th):{true, false}* are used.
- *Waiting Execution*. When the thread is dispatched, its execution is managed by a scheduler. For example, in presence of immediate connection, the scheduler must ensure that the sender completes before the start of execution of the receiver. Similarly, we use a machine to express the scheduler. A family of shared variables *Get_CPU(th):{true, false}* are used. Moreover, ports of which incoming connection is immediate must be rewritten.
- *Execution*. The duration of execution is [BCET,WCET] and processor consumption is 100%.
- *Write Data*. The execution results are copied to the *IportBuffer* of the receiver, and separated as *Trans_Connection_Write_Imm(th)* and *Trans_Connection_Write_Delay(th)*.
- *Waiting Next Event*. This rule 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 execution is abstracted by a function *ComputeOutPut(Iport:Iports(th)->VALUE):Oports(th) ->VALUE* which consumed CPU time for the interval [BCET, WCET] of the thread, and it will be refined by the AADL behavior annex in Sect. 5.3.

Here, the complete TASM expression of a periodic thread with data port communications is defined as two main machines in parallel: *TASM_Thread(th)* and *Periodic_Dispatcher(th)*, as shown in Listing 4.

Translation rules are written using an ML-like language (close to the Coq notations) with bold font keywords:

- **LET** name = P **AND** ... **IN** $\underset{\text{TASM}}{\text{P}}$
- **IF** condition **THEN** $\underset{\text{TASM}}{\text{P}}$ **ELSE** $\underset{\text{TASM}}{\text{P}}$

Listing 4 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)=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) ⊗ Dispatch(th):=false)
⊕ // 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):= ComputeOutPut(Iport(th)) ⊗ State(th):= completed ⊗ Get_CPU(th):=false)
⊕ // Rule Write Data
Time 0 ▷ (if State(th) = completed then
            State(th) := waiting_next_dispatch ⊗ 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)=true and State(th) = waiting_dispatch and Dispatch(th) = false then
            Dispatch(th):= true ⊗ WaitingNextDispatch:=true )
⊕ //Rule Waiting Period
time Period(th) ▷ ( if WaitingNextDispatch= true then
                    WaitingNextDispatch:=false ⊗ Trans_Connection_Write_Delay(th)
                    ⊗ State(th):=waiting_mode)
⊕ //Rule Waiting Next Event
time Next ▷ ( else then
             skip)

IN TASM_Thread(th) || Periodic_Dispatcher(th)

Trans_Connection_Read(th) =
                                  ⊗                          ip:=IportBuffer(th)
                        ip∈Iports(th)∧cn∈Connections
                 ∧DestinationPort(cn)=ip∧ConnectionType(cn)=immediate
Trans_Connection_Write_Imm(th) =
                                  ⊗                          IportBuffer(DestinationPort(cn)):=op
                        op∈Oports(th)∧cn∈Connections
                 ∧ConnectionType(cn)=immediate∧SourcePort(cn)=op
Trans_Connection_Write_Delay(th) =

                                  ⊗                          IportBuffer(DestinationPort(cn)):=op
                        op∈Oports(th)∧cn∈Connections
                 ∧ConnectionType(cn)=delayed∧SourcePort(cn)=op
```

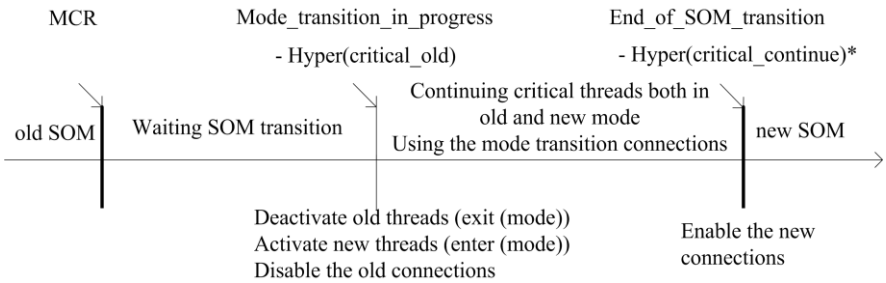
The scheduler should avoid giving the processor to several threads at the same time. Here we don't give the detailed expressions because of the limited space.

5.2 Mode Change

1) Informal Semantics

The behaviors of a SOM transition. The AADL mode change protocol comprises two phases: (1) waiting SOM transition; (2) SOM transition. As mentioned in Sect. 3, we consider the planned one. Fig. 4 shows the time line of an AADL SOM transition.

Fig.4 The time line of an AADL SOM transition



At the beginning, the system is in the source mode of a SOM transition, named *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 as *Hyper(critical_old)*), 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 (i.e., 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 as *Hyper(critical_continue)**). Finally, the system enters the destination mode of the SOM transition, named *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.

2) Operational Semantics

The operational semantics is defined as the synchronous product of two TTSs: TTS_{MCR} and TTS_{mode_change}.

Definition 8. The operational semantics of a MCR is a timed transition system $TTS_{MCR} = <\mathbb{S}, \mathbb{S}^0, \Sigma, \rightarrow>$ where:

- $\mathbb{S}$ = {State(tr):SOM, SOMRequest:{true, false}, CurrentTime:DURATION}, is the set of states.
- $\mathbb{S}^0$ is considered as: State(tr) = oldSOM, SOMRequest = false, CurrentTime = 0.
- Σ is the set of labels.
- $\rightarrow$ is defined by the following rules. The rule PRODUCE_MCR[1] does noting, but it also constrain the advance of time. The rule PRODUCE_MCR[2] is used to produce the MCR, the time of the MCR is assumed as a random MCR_DATE, and the rule WAIT_PRODUCE_MCR waits to produce the next MCR.

$$\frac{S(State(tr)) = oldSOM, S(SOMRequest) = false, \quad S(CurrentTime) < MCR\_DATE, 0 \le MCR\_DATE \le Hyper(critical\_old), \quad S' = S \text{ with } \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{PRODUCE\_MCR[1]}$$

$$\frac{S(State(tr)) = oldSOM, S(SOMRequest) = false, \quad S(CurrentTime) = MCR\_DATE, 0 \le MCR\_DATE \le Hyper(critical\_old), \quad S' = S \text{ with } \{SOMRequest = true\}}{S \longrightarrow S'} \quad \textbf{PRODUCE\_MCR[2]}$$

$$\frac{S(SOMRequest) = true, \quad S' = S \text{ with } \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{WAIT\_PRODUCE\_MCR}$$

Definition 9. The operational semantics of a SOM transition is a timed transition system $TTS_{mode\_change} = <\mathbb{S}, \mathbb{S}^0, \Sigma, \rightarrow>$ where:

- $\mathbb{S}$={State(tr):SOM, SOMRequest:{true, false}, ModeTransitionInProgress:{true, false}, ArriveHyperPeriod:{true, false}, Activation(th):{true, false}, Activation(cn):{true, false}, CurrentTime:DURATION}, is the set of states.
- $\mathbb{S}^0$ is considered as: State(tr) = oldSOM, SOMRequest = false, ModeTransitionInProgress = false, ArriveHyperPeriod = false, $\forall$th, oldSOM $\in$th.Modes, Activation(th)=true, $\forall$th, oldSOM $\notin$ th.Modes, Activation(th) = false, $\forall$cn, oldSOM $\in$cn.Modes, Activation(cn) = true, $\forall$cn, oldSOM $\notin$ cn.Modes, Activation(cn) = false, CurrentTime = 0.
- Σ is the set of labels.
- $\rightarrow$ is defined by the following rules. The rule WAIT_H_PERIOD constrains the advance of time. The rule ENTER_MC_PL is used to activate/deactivate the threads and the connections, and the rule MC_PL means that the system enters the newSOM.

$$\frac{S(State(tr)) = oldSOM, S(SOMRequest) = false, S(CurrentTime) < Hyper(critical\_old), \quad S' = S \text{ with } \{CurrentTime = CurrentTime + d\}}{S \xrightarrow{d} S'} \quad \textbf{WAIT\_H\_PERIOD}$$

$$\frac{S(State(tr)) = oldSOM, S(CurrentTime) = Hyper(critical\_old), \quad S' = S \text{ with } \{ArriveHyperPeriod = true\}}{S \longrightarrow S'} \quad \textbf{REACH\_H\_PERIOD}$$

$$\frac{\begin{array}{l} S(State(tr)) = oldSOM, S(SOMRequest) = true, S(ArriveHyperPeriod) = true, \\ S' = S \text{ with } \{Activation = \{th \mapsto true \text{ if } newSOM \in th.Modes \\ \qquad false \text{ otherwise}\} \\ \qquad \cup \{cn \mapsto true \text{ if } oldSOM \in cn.Modes \wedge newSOM \in cn.Modes \\ \qquad false \text{ otherwise}\}, \\ \qquad ModeTransitionInProgress = true\} \end{array}}{S \longrightarrow S'} \quad \textbf{ENTER\_MC\_PL}$$

$$\frac{\begin{array}{l} S(ModeTransitionInProgress) = true, S(CurrentTime) \bmod Hyper(critical\_continue) = 0, \\ S' = S \text{ with } \{State(tr) = newSOM, \\ \qquad Activation = \{cn \mapsto true \text{ if } newSOM \in cn.Modes \\ \qquad false \text{ otherwise}\}, \\ \qquad SOMRequest = false, \\ \qquad ArriveHyperPeriod = false, \\ \qquad ModeTransitionInProgress = false\} \end{array}}{S \longrightarrow S'} \quad \textbf{MC\_PL}$$

3) Translational Semantics

In the TASM environment, we introduce the *CurrentSOM* variable valued in the set of SOM, and three booleans *ArriveHyperPeriod*, *ModeTransitionInProgress* and *SOMRequest*.

Listing 5 The TASM environment of an AADL SOM transition

```
Trans_ModeData (tr) =
{CurrentSOM : SOM;
 ArriveHyperPeriod: Boolean;
 ModeTransitionInProgress: Boolean;
 SOMRequest : Boolean;
 …
}
```

The behaviors of a SOM transition is expressed by a main machine with three rules: *Waiting Hyper Period*, *Mode Transition In Progress* and *SOM transition*. The machine uses the shared variable *Activation(th)* to synchronize with the execution of periodic threads.

- *Waiting Hyper Period*. This rule is used to wait the next hyper-period of the critical threads under the oldSOM.
- *Mode Transition In Progress*. It expresses the behaviors when waiting for the actual SOM transition after a MCR.
- *SOM transition*. It expresses the actual SOM transition.

However, the duration between the *mode_transition_in_progress* state and the *end_of_SOM_transition* state is not fixed, so we use a machine to manage the time Hyper(critical_continue)*.

Moreover, the mode change request is also expressed by a machine, and we use [0,Hyper(critical_old)] to express the time of the MCR.

Listing 6 The TASM machines of an AADL SOM transition

```
Trans_Mode (tr) =
LET oldSOM = tr.SourceMode
AND newSOM = tr.DestinationMode
AND TASM_SOM_Transition =
// Rule Waiting Hyper Period
time Hyper(critical_old) ▷ ( if CurrentSOM = oldSOM and SOMRequest = false then
                                ArriveHyperPeriod:= true)
⊕ // Rule Mode Transition In Progress
time 0 ▷ ( if CurrentSOM = oldSOM and SOMRequest = true and ArriveHyperPeriod = true then
            ModeTransitionInProgress:=true ⊗ FlagNewSOM := true
            ⊗ ⊗ Activation(th) := true
              th∈threads(newSOM) \ threads(oldSOM)
            ⊗ ⊗ Activation(th) := false
              th∈threads(oldSOM) \ threads(newSOM)
            ⊗ ⊗ Activation(cn) := false)
              cn∈connections(oldSOM) \ threads(newSOM)
⊕ // Rule SOM transition
time 0 ▷ ( if ModeTransitionInProgress = true and FlagNewSOM = false then
            CurrentSOM := newSOM
            ⊗ SOMRequest :=false
            ⊗ ⊗ Activation(cn) := true
              cn∈connections(newSOM) \ threads(oldSOM)
            ⊗ ModeTransitionInProgress :=false
            ⊗ ArriveHyperPeriod :=false)
```

```
AND Manage_Hyper_New =
time Hyper(critical_continue) ▷ (if  true then
                                    FlagNewSOM := false)
AND MCR =
// Produce MCR
time (0,Hyper(critical_old)) ▷ ( if CurrentSOM = oldSOM and SOMRequest = false then
                                    SOMRequest:= true)
⊕ // Rule Waiting Next Event
time next ▷ (else then
            skip)
IN TASM_SOM_Transition || Manage_Hyper_New || MCR
```

5.3 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. The semantics specifications as above will be enriched by the behavior annex.

1) Informal Semantics

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 the transition. 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 complete.

2) Operational Semantics

The operational semantics is defined as the refinement of the *Execution* rule of TTS_{thread_basic}, where the state $\mathbb{S}$ 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.

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

The rule BA_COMPLETE means the execution of the behavior annex is completed.

$$\frac{\begin{array}{l} S(State(th)) = execution, S(CurrentBAState) = SourceState(BA\_tr), \\ S(CurrentTime) = S(StartTransitionTime) + Time(BA\_tr) \\ S' = S \ with \ \{CurrentBAState = DestinationState(BA\_tr), \\ \qquad Oport(th) = Action(BA\_tr)(Iport(th)), \\ \qquad StartTransitionTime = CurrentTime\} \end{array}}{S \longrightarrow S'} \quad \textbf{BA\_EXECUTION}$$

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

$$\frac{\begin{array}{l} S(State(th)) = execution, \\ S' = S \ with \ \{State(th) = completed\} \end{array}}{S \longrightarrow S'} \quad \textbf{BA\_COMPLETE}$$

if $isFinal(CurrentBAState) = true$

3) Translational Semantics

The translational semantics specification 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 the TASM internal variables. Each *transition* of the behavior annex is translated into a TASM rule. These rules will take the place of the rule named "*Execution*" 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 including 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 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 assignments of the port variables. (2) Assignments are expressed by the assignments of the corresponding environment variables. (3) The behavior annex introduces the statements *computation(min,max)* and *delay(min,max)*, which expresses the use of the CPU and 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, so this is consistent with TASM semantics. 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 shown in Listing 8. It can be detailed when the 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 Data
...
Trans_BehaviorAnnex (th) =
                ⊕                    Time Time(BA_tr) ▷
BA=th.Behavior∧BA_tr∈BA.Transitions
                ( if State(th) = execution and CurrentBAState = SourceState(BA_tr)
                    and ⟦Guard(BA_tr)⟧ = true then
                    CurrentBAState:= DestinationState(BA_tr) ⊗
                    Oport(th):=Action(BA_tr)(Iport(th))
⊕// Rule Behavior Annex Completion
Time 0 ▷ (if  isFinal(CurrentBAState) = true then
              State(th) := completed)
```

6 The Proof of Semantics Preservation

In this section, we give the main ideas of the proof of semantics preservation of the transformation from AADL to TASM, i.e., exact correspondence between steps in AADL operational semantics, and in target TASM semantics.

6.1 A Simulation Equivalence Relation

We have given the operational semantics of the subset of AADL in TTS, named as AADL_TTS. Moreover, combining the TASM expressions in the AADL translational semantics with the semantics of TASM, we can obtain another TTS, named as TASM_TTS.

In order to prove that AADL_TTS and TASM_TTS are equivalent we must find a suitable relation R between their states $\mathbb{S}$ and {*env:Env, upds:Upds*}. In this paper, R is a strong simulation-equivalence relation, i.e., two-directions strong simulation. Here it is specialized as mapping functions:

(1) In the direction from AADL to TASM, according to the $\mathbb{S}$ in the AADL_TTS, we can get the corresponding environment *Env* and the next update set *Upds* in the TASM_TTS. We define it as *A2T:* $\mathbb{S}$ *-> {env:Env, upds:Upds}*.

(2) In the direction from TASM to AADL, according to the environment *Env* in the TASM_TTS, we can get the corresponding $\mathbb{S}$ in the AADL_TTS. Moreover, auxiliary states (*ExtState*) are needed in order to establish the simulation. For example, the information "*NextPeriod*" is given explicitly in the AADL_TTS, but it is implicit in the duration of the TASM rules, so this information needs to be complemented into the TASM_TTS. We define it as *T2A: Env×ExtState ->* $\mathbb{S}$.

A common definition of the mapping functions in Coq is given as follows:

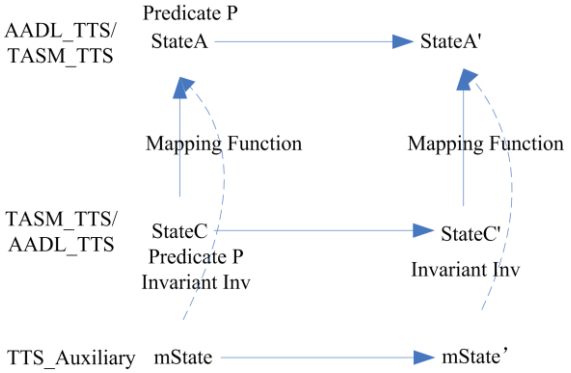
```
Variables StateA StateC: Type.
Record mapping: Type := mkMapping {
   mState: Type;
   mInit: StateC -> mState;
   mNext: mState -> StateC -> mState;    /* discrete transitions*/
   mDelay: mState -> StateC -> Time -> mState;    /*continuous transitions*/
   mabs: mState -> StateC -> StateA
}.
```

StateA and **StateC** represent the set of states of two TTSs respectively, **mState** denotes a set of auxiliary states, **mInit**, **mNext** and **mDelay** are used to construct the state space of the auxiliary states, **mabs** maps the *StateC* and the auxiliary state to the *StateA*.

Moreover, invariants which restrict considered states to a superset of reachable states are demanded. Additionally, we attach the same set of observers to both models, as for example "*port p of thread t has value v*", "*thread t is in the given state s*", and "*thread t gets the CPU*", etc. Their semantics is defined by a predicate over the state space of each model and must be compatible with the chosen mapping function when the invariant is satisfied.

So, here the strong simulation equivalence is refined, as shown in Fig. 5.

Fig.5 The principle of strong simulation equivalence



The Coq expression of the definition of strong simulation equivalence 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: forall st, Init _ c st -> inv (mInit m st) st;
   invNext: forall ex1 st1 st2, Next _ c st1 st2 -> inv ex1 st1 -> inv (mNext m ex1 st1) st2;
   invDelay: forall ex1 st1 st2 d, Delay _ c st1 d st2 -> inv ex1 st1 -> inv (mDelay m ex1 st1 d) st2;
   simuInit: forall st, Init _ c st -> Init _ a (mabs m (mInit m st) st);
   simuNext: forall ex1 st1 st2, Next _ c st1 st2 -> inv ex1 st1 ->
                          Next _ a (mabs m ex1 st1) (mabs m (mNext m ex1 st1) st2);
   simuDelay: forall 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);
   simuPred: forall ext st, inv ext st -> (forall p, lpSat (Satisfy _ c) st (trc p) <-> lpSat (Satisfy _ a)
                                    (mabs m ext st) (tra p))
}.
```

inv is a set of invariants associated with the auxiliary states and *StateC*, **invInit**, **invNext** and **invDelay** are used to get a superset of reachable states of the state space of *StateC* and auxiliary states, **simuInit**, **simuNext**, **simuDelay** and **simuPred** construct the strong simulation relation between the two TTSs, including the correspondence of initial state, discrete transitions, continuous transitions, and predicates.

6.2 Compositional Proof

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

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

Secondly, we prove the two following theorems:

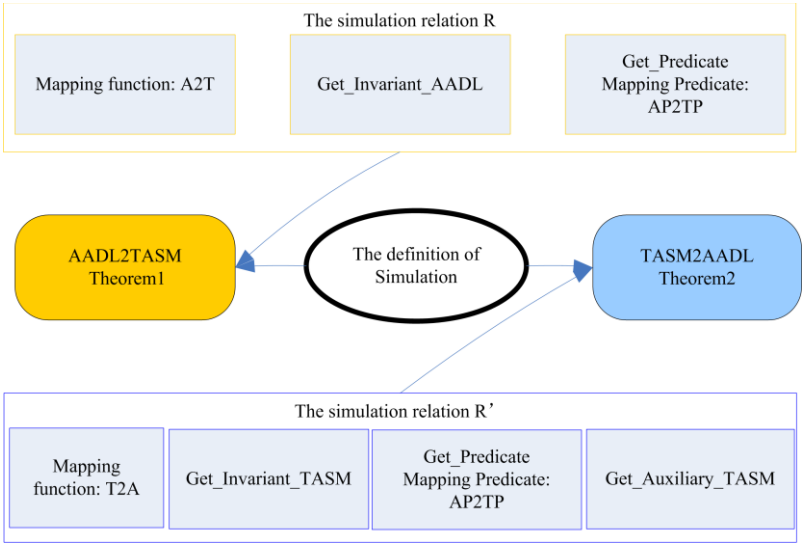
Theorem 1. For every periodic thread with delayed connections Th(i), its operational semantics is strongly simulated by its translational semantics into TASM, noted TTS(Th(i)) $\preceq$ MM_TTS(Trans_Thread(Th(i))).

Theorem 2. 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 Definition 6 and Definition 7. Trans_Thread(Th(i)) is the parallel composition of TASM_Thread(th) and Periodic_Dispatcher(th) defined in Listing 4.

The sketch of the proof is shown in Fig. 6. Theorem 1 is for the direction from AADL to TASM, and theorem 2 is for the direction from TASM to AADL. Before the proof, we need to give the concrete mapping function, get the invariants, get the predicates and map the predicates. Moreover, in the proof of theorem 2, we need auxiliary states. The proof of the two theorems relies on the definition of a simulation relation that is preserved by each transition.

Fig.6 The proof sketch of the strong simulation equivalence



Thirdly, the theorem of simulation of synchronous product is applied.

Theorem 3. Let $\prod_{i=1...n} TTS(A)$ be the synchronous product of TTS(A), and $\prod_{i=1...n} TTS(B)$ be the synchronous product of TTS(B). If TTS(A) $\preceq$ TTS(B), then $\prod_{i=1...n} TTS(A) \preceq \prod_{i=1...n} TTS(B)$.

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

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 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).

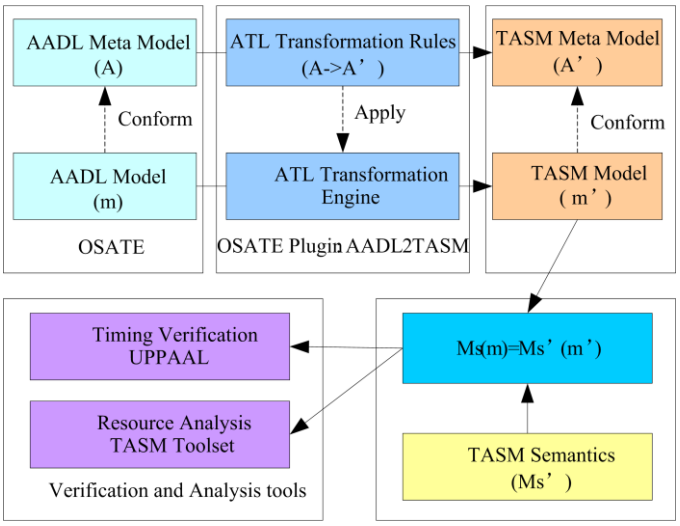
7 Formal Verification and Analysis

This section illustrates how the TASM model corresponding to an AADL model can be formally analyzed. Firstly, the AADL2TASM tool is given. Secondly, a case study of space Guidance, Navigation, and Control(GNC) subsystem is verified in AADL2TASM.

7.1 The AADL2TASM Tool

After the proof of semantics preservation, the translational semantics of the subset of AADL has been automated in the AADL2TASM tool, which is a plug-in of the OSATE modelling environment, as shown in Fig. 7.

Fig.7 Verification and analysis framework of AADL2TASM



The model transformation language ATL (ATLAS Transformation Language) (Jouault and Kurtev 2006) 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, the destination one is defined using the KM3 language, and the ATL transformation rules conform to the certified translational semantics.

Then the model checker UPPAAL is used to verify timing properties and TASM Toolset to analyze resource consumption. Here we reuse the translation principle from TASM to UPPAAL given by the authors of TASM (Martin 2008), and implement it using ATL. We can say TASM is an intermediate model, which can support several verification tools. That's why AADL is not mapped to UPPAAL directly.

Due to space limitations, the details of the meta-models and the ATL rules cannot be given here.

7.2 A GNC System Example

In order to further illustrate how AADL models can benefit from TASM-based formal analysis using the AADL2TASM tool chain, we give an overview of the verification of an example, the GNC system (rather simple) of a satellite.

1) The GNC System

The GNC system involves determining and controlling the attitude and orbit of a satellite as it performs its mission. It always has several working modes, such as velocity damp, solar-panels control, attitude stabilization, maneuver control, three-axis control, etc. In each mode, navigation, guidance, and control will be done periodically. Here, we just consider two modes: attitude stabilization and maneuver control. The architecture of the system is shown in Fig.8.

In the mode of attitude stabilization, *Star Sensor* collects the data of the attitude, *Gyroscope* collects the data of the angular velocity, and then the *Attitude Filter* computes the estimation of attitude. The data of orbit can be got by the processing of the images from the guidance *Camera*, and the estimation of orbit can be got from the *Orbit Filter*. Then the desired attitude is computed using the estimation of orbit by the *Attitude Guidance1*. Finally, the *Attitude Control1* compares the estimation of attitude and the desired attitude, and then controls the *Wheel* to stabilize the satellite.

In the mode of maneuver control, *Star Sensor* and *Camera* are stopped, while *Gyroscope* continues to execute. Firstly, the *Attitude Filter* computes the estimation of attitude according to the angular velocity from the *Gyroscope*, while the *Orbit Filter* computes the estimation of orbit according to the data of orbit from the *Accelerometer*. Secondly, the *Guidance Law* computes the maneuver command using the estimation of orbit and the control commands from the ground (*Operator Command*). Thirdly, the desired attitude can be computed using the maneuver command in the *Attitude Guidance2*. Finally, the *Attitude Control2* compares the estimation of attitude and the desired attitude, controls the *Jets* to change the attitude and then fires *Engines* to move the satellite to the desired trajectory.

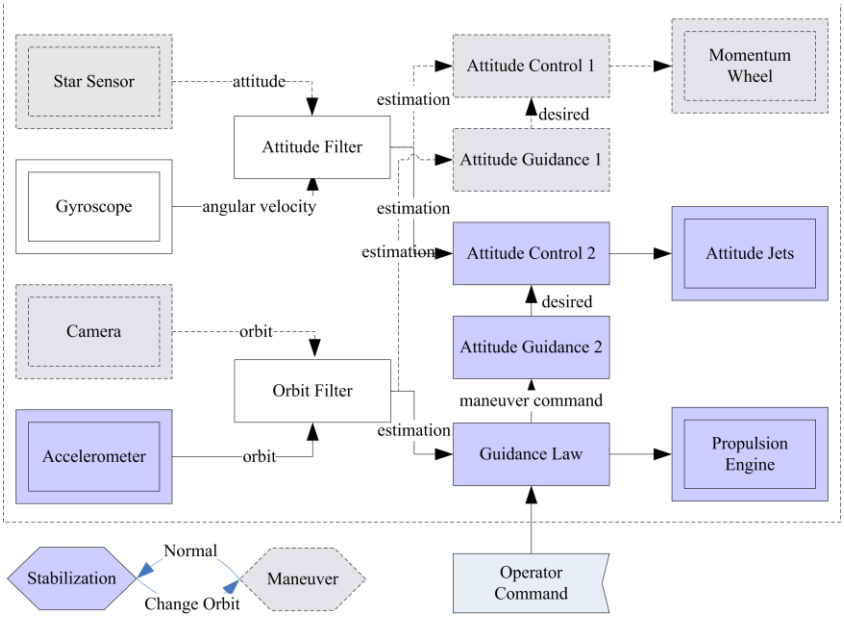


Fig.8 Architecture of the GNC system

2) The AADL Model of the GNC System

Sensors (*Star Sensor*, *Gyroscope*, *Camera* and *Accelerometer*) and actuators (*Wheel*, *Jet* and *Engine*) can be specified by the device components of AADL. Moreover, here we use periodic threads to model the data sampling of the sensors, such as *Star_Sensor_Data_Sampling*, *Gyroscope_Data_Sampling*, etc., and then we can use the behavior annex to express their behaviors. Other tasks such as *Attitude Filter* and *Orbit Filter* are also specified by periodic threads. These threads are linked with data port connection, and each thread has a behavior annex specification.

The threads and connections are active in different modes (*Stabilization*, *Maneuver*), and *Stabilization* is the initial mode. We assume that all the threads need resource consumption, here the power, to estimate the power consumption of the system, and the memory, to ensure that the memory used by the system is adequately bounded. The parameters are given in Table 1.

A single processor is considered, and the tasks execution sequence is static: *Stabilization* mode (*Star Sensor*, *Gyroscope*, *Attitude Filter*, *Camera*, *Orbit Filter*, *Guidance1*, *Control1*), and *Maneuver* mode (*Gyroscope*, *Attitude Filter*, *Accelerometer*, *Orbit Filter*, *Guidance Law*, *Guidance2*, *Control2*). A part of the AADL textual specification is given in Listing 9.

Table 1 Parameters of the threads

Thread	Period (Ms)	Execution Time (Ms)	Power (W)	Memory (KB)	Mode
Star_Sensor _Data_Sampling	360	32	[2,10]	256	Stabilization
Gyroscope_Data _Sampling	360	32	[5,20]	256	Stabilization Maneuver
Camera_Data _Sampling	360	32	[5,10]	1024	Stabilization
Accelerometer _Data_Sampling	360	32	[1,3]	256	Maneuver
Attitude_Filter	360	64	[5,10]	1024	Stabilization Maneuver
Orbit_Filter	360	64	[5,10]	1024	Stabilization Maneuver
Attitude_Guidance1	360	64	[2,10]	512	Stabilization
Attitude_Control1	360	64	[2,10]	512	Stabilization
Attitude_Guidance2	360	64	[2,10]	512	Maneuver
Attitude_Control2	360	64	[2,10]	512	Maneuver
Guidance_Law	360	32	[5,10]	256	Maneuver

Listing 9 Textual specification of the AADL model of the GNC system

```
process implementation GNC_Process.DSP_TI
  subcomponents
    T_Star_Sensor_Data_Sampling: thread Star_Sensor_Data_Sampling.DSP_TI in modes (Stabilization);
    T_Gyroscope_Data_Sampling: thread Gyroscope_Data_Sampling.DSP_TI in modes (Stabilization, Maneuver);
    T_Camera_Data_Sampling: thread Camera_Data_Sampling.DSP_TI in modes (Stabilization);
    ……
  connections
    data port T_Star_Sensor_Data_Sampling.Attitude_Measure -> T_Attitude_Filter.Attitude_Measure in modes (Stabilization)
    data port T_Gyroscope_Data_Sampling.Attitude_Angular_Velocity -> T_Attitude_Filter.Attitude_Angular_Velocity in modes
    (Stabilization, Maneuver)
    ……
  modes
    Stabilization: initial mode ;
    Maneuver: mode ;
    Stabilization -[ Change_Orbit ]-> Maneuver;
    Maneuver -[ Normal ]-> Stabilization;
end GNC_Process.DSP_TI;
```

3) TASM Specifications and Formal Analysis

All the threads are supposed to be critical threads, so the duration of the mode change is the hyper-period, i.e., 360 Ms. The behavior of the mode change is expressed by a main machine, and the MCR is expressed by another machine. The behavior of a periodic thread is expressed by two main machines (basic thread and dispatcher), and the communication is specified as timing synchronization. The scheduler gives a static execution order of the threads. A part of the TASM specification is shown in Listing 10.

Listing 10. The TASM specification

```
Main Machine Mode_Change:
Rule: waiting_hyper_period_stabilization
{ t := 360;
  if CurrentMode = stabilization and
     Change_Orbit = false then
     ArriveHyperPeriod_stabilization := true;
}
Rule: begin_mode_change_stabilization_to_maneuver
{ if CurrentMode = stabilization and
    Change_Orbit = true and
    ArriveHyperPeriod_stabilization = true then
    Activation_SSDS := false;
    Activation_GDS_S := false;
    ...
...
Main Machine Star_Sensor_Data_Sampling_Period:
Rule: dispatch_th
{ t :=0;
  if Activation_SSDS = true and
    ThState_SSDS = waiting_dispatch
    and Dispatch_SSDS = false then
    Dispatch_SSDS := true;
    NextDispatch_SSDS := true;
}
…
Main Machine Scheduler:
Rule: Start_Sensor_Sch
{t :=0;
```

```
    CurrentMode := stabilization_mode_change;
  }
  ...
  Main Machine Star_Sensor_Data_Sampling:
  ...
  Rule: execution
  {t := 32;
    Power := [2,10];
    Memory := 256;
    if ThState_SSDS = execution then
      ThState_SSDS := completed;
      Get_CPU_SSDS := false;
  }
    if ThState_SSDS = waiting_execution and
      Get_CPU_SSDS = false then
      Get_CPU_SSDS := true;
  }
  Rule: Gyroscope_S_Sch
  {t :=0;
    if ThState_GDS_S = waiting_execution and
      ThState_SSDS = completed and
      Get_CPU_GDS_S = false then
      Get_CPU_GDS_S := true;
  }
  …
```

The first technique of verification is achieved with UPPAAL by mapping each machine to a timed automaton. The translation principle is given as follows:

- **Variables and data types.** TASM has more data types than UPPAAL, such as integer, boolean, float and user-defined type. A user-defined type of TASM which contains *n* members, such as the state of the thread, is translated into an integer data type of timed automata with range "[*0*, *n-1*]", where "*0*" corresponds to the first member of the type and "*n-1*" corresponds to the nth member of the type. The boolean type of TASM is expressed by an integer data type of timed automata with range "[*0*,*1*]", where "*0*" represents "*true*" and "*1*" represents "*false*".
- **Time.** In UPPAAL, time elapses in a location, but time is used to denote the duration of a transition in TASM. This can be expressed using timed automata with an extra intermediate location to elapse time, like *begin_mode_change_S2M* in Fig. 9.
- **Resource.** The resource definitions are discarded.
- **Rules.** Each machine is mapped to a timed automaton. First, an urgent location *Pivot* is produced. Then, for each rule of the TASM machine, a branch from the *Pivot* location is added. If the machine contains an "else" rule, an extra branch is added. For rule that contains the "t:=next" annotation, build an urgent edge using an extra automaton and an urgent channel, like Fig. 9 and Fig. 10.

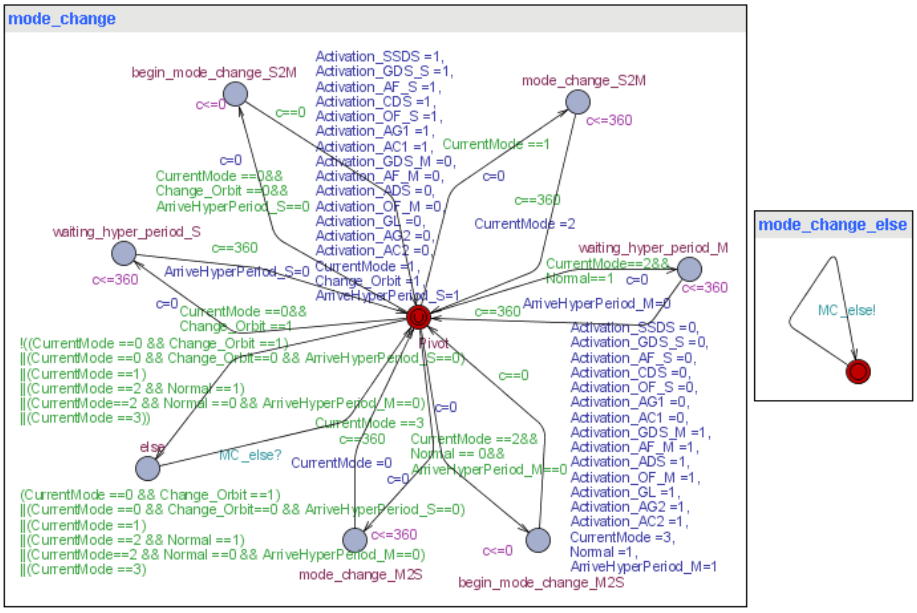


Fig. 9 Timed automaton translation of the mode_change machine

The verification properties can be expressed using the UPPAAL's query language, such as:

- Deadlock-freedom. *A[]!deadlock*.
- The safety properties can fall into two categories – verifying that the correct mode is set depending on environmental conditions and verifying that the correct output is set depending on the mode of operation. For example, when the system in the stabilization

mode, the task of *Star_Sensor_Data_Sampling* must be active, the logic formula is formulated as: *A[](CurrentMode==0) imply (Activation_SSDS == 0)*.

- The model can also be queried to verify certain liveness properties, to ensure that the model behaves correctly. For example, the property "*eventually, the system is in stabilization mode*", and it can be formulated as: *E<>(CurrentMode==0)*.

Moreover, the real-time properties can also be formulated by using a combination of temporal logic and observer automata.

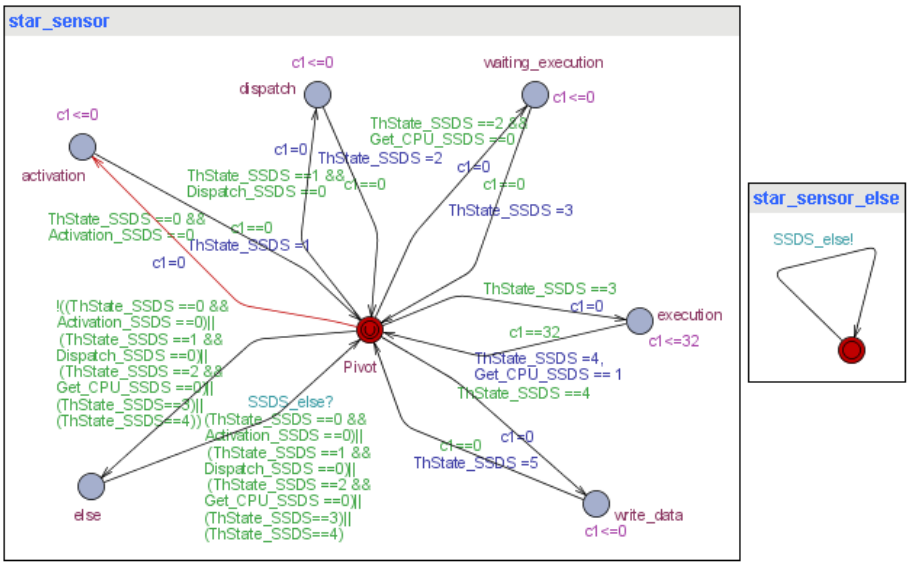


Fig. 10 Timed automaton translation of the star_sensor_ data_sampling machine

The second technique of verification is by using TASM Toolset to analyze the timing sequences and resource consumptions.

As shown in Fig. 11, the graph shows the time progression of individual main machines. The top line, in black, displays the global time progression. A green line means that the machine was active in the last step. A yellow line means that the machine is "blocked" until time progresses to a point where its step becomes complete. A red line means that the machines has terminated.

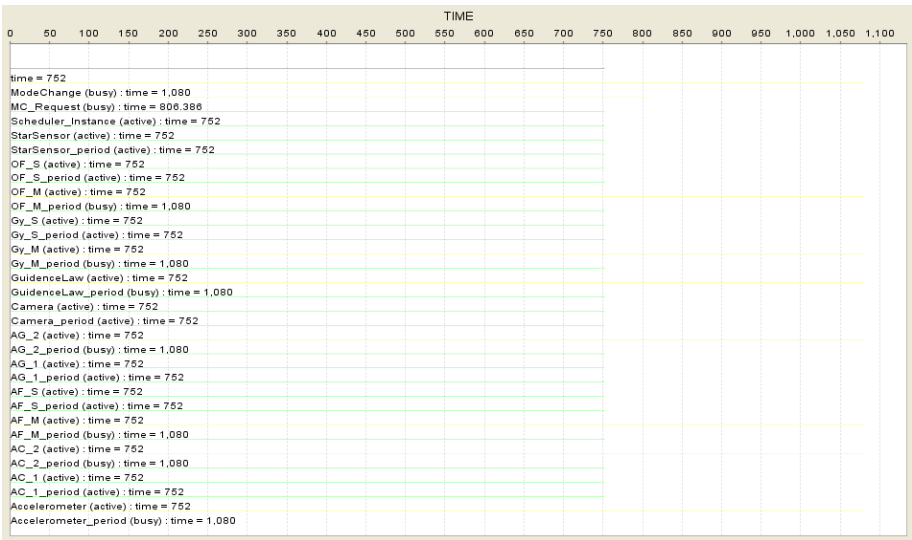


Fig. 11 The simulation time graph of the TASM Toolset

As shown in Fig. 12, the graph shows the aggregate resource consumption for each resource, versus the time axis. There is one graph per resource. The tool can calculate the minimum, maximum, and average resource consumption for each resource.

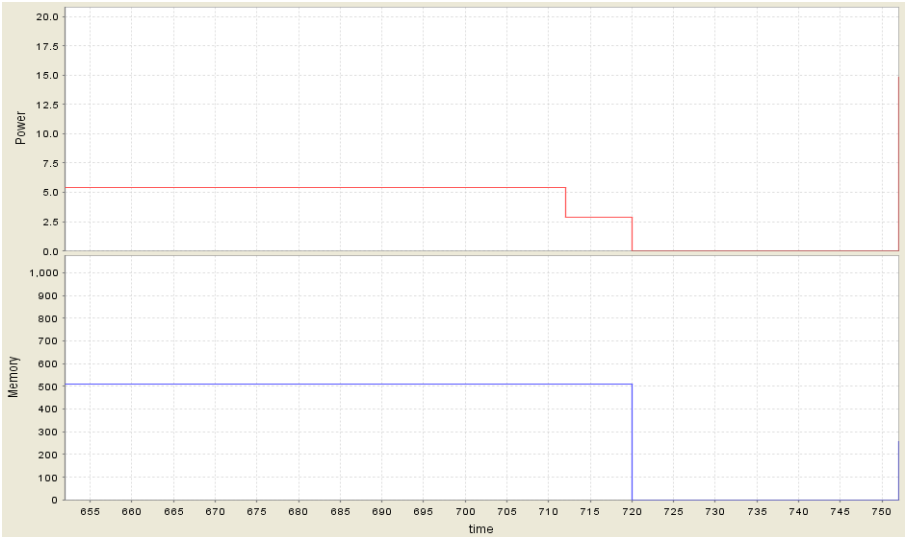


Fig. 12 The simulation resource graph of the TASM Toolset

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.

Related work is henceforth assessed by considering relevant contributions to the objective of building models suitable for the analysis and verification of AADL specifications. We assess these contributions by considering relevant criteria as to the presence and extent of a proof of semantic preservation: the chosen subset of AADL, the way semantics is expressed, the kind of verification properties that are represented and the degree of formality of the translation.

M.Yassin et al. (2008) translate most of the AADL concepts into the BIP language, except for the mode changes, but the semantics of the behavior annex and of connections are not much detailed. BIP is a framework for modeling heterogeneous real-time components using three layers: the lower layer describes the behaviors, the intermediate layer describes the interactions, and the upper layer describes scheduling policies, but BIP cannot express resource consumptions. It provides two categories of verification properties: deadlock detection by using the tool Aldebaran, and some simple timing properties using observers. The modeling concepts of both the AADL and the BIP are expressed with examples. Its translation is expressed using natural language, and the semantics models are simplified as graphs for easier comprehension.

Jean-François et al. (2008) translate the AADL models into TLA+, but the only tool for TLA+ is TLC which is written in Java and not very efficient.

The translation proposed by Oleg 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 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. However, its translation rules are expressed by a pseudo-algorithm.

Thomas 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. The IF language cannot express resource consumption. Its translation rules are also expressed in natural language.

In the work of Bernard 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 Net for verification. However, the paper does not explain their semantics, and it just gives simply the translation principle.

In the work of Peter Csaba 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. A Real-Time Maude specifies a real-time rewrite theory, but it cannot express resource. Moreover, its translation is also expressed using natural language.

The translation proposed by Erwan 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. Now, the translation into Polychrony (Yue 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 (TPN) is presented. Model-checking of properties expressed in VTS is enabled using TPN-based tools.

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-certified compiler (Xavier 2006; Xavier 2009), by using translation validation (Pnueli et al. 1998; George C. 2000) or proof-carrying code (George C. 1997).

A certified 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.

Except for semantics preservation, there is another viewpoint for the verification of model transformations, such as the work of Esther et al. (2012) and Marouane et al. (2011). They verify transformation requirements on the execution of actual model transformations written by a dedicated language like ATL and QVT (OMG 2011).

In the area of AADL research, as mentioned above, most of the related works use manual validation. The work of Jean-Paul 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. Mamoun and Julia (2010) define a refinement of a synchronous subset of AADL using the Event B method, and gives simplified proof obligations. Dominique et al. (2008) translate the AADL mode change protocol into Timed Petri Net, and verifies some properties to validate the proposed translation. Another way is to compare alternative semantics for the same language. In a preview work (Lei 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 operational semantics expressed in TTS are correct. However, giving two semantics, the TTS-based one and the translation-based one, which are proved to be equivalent, increases the confidence on the interpretation of the informal semantics.

A subset of AADL we consider includes periodic threads, data port communications, mode changes and the behavior annex. First, the operational 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. Finally, the certified translational semantics is automated in the AADL2TASM tool, which provides model checking and simulation for AADL models.

Our formal semantics and proof of semantics preservation for AADL contributes important foundations to the use of AADL for developing safety-critical applications and, to that end, our approach has been validated by the case study of a satellite GNC system. Model checking and simulations have been performed to check if an AADL specification meets its functional properties, real-time requirements and resource constraints of the GNC.

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 a semantics. For example, we are currently working on the Integrated Modular Avionics (IMA) modeling and verification using AADL.

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 via observer automata. 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 know, there is a gap between the proof of the semantics preservation and the concrete implementation using ATL rules. However, a research perspective is to consider how executable code for dedicated model transformation language such as ATL could be extracted from Coq.

Acknowledgements

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 under Grant 61073013, Grant 61003017, and Grant 90818024, Project of the State Key Laboratory of Software Development Environment of China under Grant SKLSDE-2010ZX-05, and the TOPCASED Project in France.

References

Arnold A.: Finite Transition Systems. Prentice Hall International, pp. 26-45(1994)

Anantha N., Gabor K.: Towards Verifying Model Transformations. Proc. GT-VMT 06 (2006)

Abdelouahed G., Ferhat K.: From UML/SPT models to schedulability analysis: approach and a prototype implementation using ATL. Automated Software Engineering (2009) 16: 387–414, DOI 10.1007/s10515-008-0045-5. (2009)

Angelo G., Elvinia R., Patrizia S.: A semantic framework for metamodel-based languages. Automated Software Engineering (2009) 16: 415–454, DOI 10.1007/s10515-009-0053-0.(2009)

Bernard B., Jean-Paul B., Christelle C., Silvano-Dal Z., Mamoun F., and François V.: Formal Verification of AADL Specifications in the Topcased Environment. Proc. Ada-Europe 09, LNCS, vol. 5570, Springer, pp.207-221(2009)

Benoît C., Xavier C., Pierre-Loïc G., Xavier T.: Essay on Semantics Definition in MDE- An Instrumented Approach for Model Verification. Journal of Software, Vol. 4, No.9, Nov. pp. 943-958(2009)

Behrmann G., David A., and Larsen Kim G.: A Tutorial on Uppaal. Proc. SFM-RT'04, LNCS vol. 3185, Springer, pp. 200-236(2004)

Bérard B., Cassez F., Haddad S., Lime D., and Roux O.H.: When are timed automata weakly timed bisimilar to time Petri nets? Theoretical Computer Science, 403(2-3), pp.202-220(2008)

Bérard B., Cassez F., Haddad S., Lime D., and Roux O.H.: Comparison of Different Semantics for Time Petri Nets. Proc. ATVA 05, LNCS, Springer, pp. 293-307(2005)

CMU: An Extensible Open Source AADL Tool Environment (OSATE). Software Engineering Institute, CMU, (2006)

Christel B. and Joost-Pieter K.: Principles of Model Checking. The MIT Press (2008)

Dominique B., Anne-Marie D., Sébastien F., Olivier H. R.: A Study of the AADL Mode Change Protocol. Proc. ICECCS 08, IEEE Computer Society Press, Belfast, pp. 288-293(2008)

Egon B.: The Origins and the Development of the ASM Method for High Level System Design and Analysis. Journal of Universal Computer Science, 8(1), pp.2-74(2002)

Esther G., Juan de L., Manuel W., Gerti K., Angelika K., Werner R., Johannes S., Wieland S.: Automated verification of model transformations based on visual contracts. Automated Software Engineering, DOI 10.1007/s10515-012-0102-y. (2012)

Erwan J., Nicolas H., Pascal R., Xavier N., David L.: Virtual execution of AADL models via a translation into synchronous programs. Proc. EMSOFT 07, ACM, pp.134-143(2007)

George C. N.: Translation validation for an optimization compiler. Proc. Prog. Lang. Design and Impl 00. ACM Press, pp 83-95(2000)

George C. N.: Proof-carrying code. Proc. 24th Symp. Principles of Progr. Lang 97. ACM Press, pp 106-119 (1997)

Holger G., Sabine G., Johannes L., Wilhelm S., and Robert W.: Towards Verified Model Transformations. In proceedings of MODEVA workshop associated to MODELS'06. (2006)

Jean-François R., Jean Paul B., Mamoun F., David C., and Dave T.: Modes in Asynchronous Systems. Proc. ICECCS 08, IEEE Computer Society Press, Belfast, pp. 282- 287(2008)

Julien D., Laurent P., Alain P., Mickael K., Frank S., Fabrice K.: Validate, Simulate and Implement ARINC653 Systems using the AADL. ACM SIGAda Ada Letters, Vol. 29, No. 3, Nov. 2009, pp. 31-44 (2009a)

Julien D., Laurent P., Peter F.: Validating safety and security requirements for partitioned architectures. Proc. Ada-Europe 09, Springer-Verlag, Berlin, Heidelberg, pp. 30-43(2009b)

Johan B., Anders H., Bengt J., Paul P.: Specifying and Generating Test Cases Using Observer Automata. Proc.FATES 04, LNCS, Vol 3395, Springer-Verlag, pp. 125-139(2005)

Jouault F., and Kurtev I.: Transforming Models with ATL. Proc. MoDELS 05, LNCS, vol. 3844, Springer-Verlag Berlin Heidelberg, pp.128-138(2006)

Jean-Paul B., Mamoun F., Martin S.: Towards formalising AADL in Proof Assistants. Electronic Notes in Theoretical Computer Science, Vol. 141, Issue 3, pp. 3-25(2005)

Jerome H., Bechir Z., Laurent P., Fabrice K.: From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. ACM Transaction on Embedded Computing Systems, Vol.7, No.4 (2008)

Lei P., Zhibin Y., Jean-Paul B., Mamoun F., Kai H., and Dianfu M.: A Comparative Study of FIACRE and TASM to Define AADL Real Time Concepts. Proc. ICECCS 09, IEEE, pp. 347-352(2009)

M.Yassin C., Anne R., Marius B., and Joseph S.: Translating AADL into BIP-Application to the Verification of Real-time Systems. Proc. MoDELS Workshops 08, Springer-Verlag, Berlin, Heidelberg, pp. 5−19 (2008)

Martin O., Kristina L.: The TASM Language Reference Manual Version 1.1. (2006)

Martin O., Kristina L.: The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. Journal of Universal Computer Science, vol. 14, No. 12, pp. 2007-2033(2008)

Martin O., Kristina L.: The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems. Proc. CAV 07, LNCS, vol. 4590, Springer, pp. 126-130(2007)

Mamoun F., and Julia L.: Development of a Synchronous Subset of AADL. Proc. ASM 10, LNCS vol. 5977, Springer-Verlag, pp. 245-258 (2010)

Martin O., Guillaume B., Kristina L.: Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. Proc. MoDELS 06, Springer-Verlag Berlin, Heidelberg, pp.32-41(2006)

Martin O.: A Formal Framework for Specification-Based Embedded Real-Time System Engineering, Ph.D Thesis, MIT(2008)

Monteverde D., Olivero A., Yovine S., and Braberman V.: VTS-based Specification and Verification of Behavioral Properties of AADL Models. Proc. MoDELS 08 ACES-MB Workshop, Springer-Verlag (2008)

Marco B., Alessandro C., Joost-Pieter K., Viet Yen N., Thomas N. and Marco R.: Safety, Dependability and Performance Analysis of Extended AADL Models. The Computer Journal, Volume 54 Issue 5, May 2011 (2011)

Marouane K., Houari S., Mounir B.: Example-based model-transformation testing. Automated Software Engineering (2011) 18: 199–224, DOI 10.1007/s10515-010-0079-3. (2011)

Oleg S., Insup L., and Duncan C.: Schedulability Analysis of AADL models. Proc. IPDPS 06, IEEE Computer

Society Press, Rhodes Island, 8 pp (2006)

Object Management Group: QVT Specification Version 1.1. http://www.omg.org/spec/QVT/1.1/. (2011)

Peter Csaba O., Artur B., and Jose M.:Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. Proc. FMOODS/PORTE 10, LNCS, vol. 6117, Springer, pp.47-62(2010)

Pnueli A., Siegel M., and Singerman E.: Translation validation. Proc. TACAS 98, LNCS, vol 1384, Springer, pp.151-166(1998)

Robin M.: Communication and Concurrency. Prentics Hall International, Englewood Cliffs. pp. 84-123(1989)

Ricardo Bedin F., Jean-Paul B., Mamoun F., Jean-Francois R.: The AADL behaviour annex – experiments and roadmap. Proc. ICECCS 07, IEEE, pp. 377-382(2007)

SAE Aerospace: SAE AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0 (2009).

SAE: SAE AS5506 Annex: Behavior_Specification V2.0 (2011)

Stefan B., Lars G., and Kristina L.: Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines. Proc. QoSA 09, LNCS, Springer-Verlag Berlin Heidelberg, pp. 101-115(2009)

Stefan B., Cristina S., Kristina L., Paul P.: A Formal Analysis Framework for AADL.The Journal of Science and Technology (2011)

Thomas A., Joël C., Philippe D., Pierre-Yves P., and Jean-Charles R.: AADL execution semantics transformation for formal verification. Proc. ICECCS 08, IEEE Computer Society Press, Belfast, pp. 263-268(2008)

Thomas C., Ivan K.: Separation of Concerns in Translational Semantics for DSLs in Model Engineering. Proc. SAC 07, ACM, pp. 985-992(2007)

The Coq Development Team : The Coq Proof Assistant Reference Manual Version 8.2. (2009)

Thomas H., and Mogens N.: Timed Bisimulation and Open Maps. Proc. MFCS 98, IEEE Computer Society Press, pp. 378-387(1998)

Thomas H., Zohar M., and Amir P.: Temporal Proof Methodologies for Timed Transition Systems. Information and Computation 112, Academic Press, pp. 273-337(1994)

Xavier L.: Formal certification of a complier back-end. Proc. POPL 06, ACM Press, pp. 42-54 (2006)

Xavier L.: Formal verification of a realistic compiler. Communications of the ACM, ACM, 52(7), pp. 107-115 (2009)

Yue M., Jean-Pierre T., Sandeep Kumar S., Thierry G.: Distributed simulation of AADL specifications in a polychronous model of computation. Proc. ICESS 09, IEEE, pp. 607-614(2009)

Zhibin Y., Kai H., Dianfu M., and Lei P.: Towards a formal semantics for the AADL behavior annex. Proc.DATE 09, IEEE, pp. 1166-1171(2009)