Two Formal Semantics of a Subset of the AADL

[1]Zhibin Yang, [1]Kai Hu, [2]Jean-Paul Bodeveix, [2]Lei Pi, [1]Dianfu Ma, and [3]Jean-Pierre Talpin

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
kenney@cse.buaa.edu.cn, {hukai, dfma}@buaa.edu.cn
[2] IRIT, University of Toulouse, Toulouse, France
{bodeveix, pilei}@irit.fr
[3] INRIA-Rennes, Campus de Beaulieu, Rennes, France
jean-pierre.talpin@irisa.fr

Abstract—The analysis and verification of an AADL model usually requires its transformation into the meta-model of this model-checker or that schedulability analysis tool. However, one challenging problem is to prove that the transformation into the target model of computation (MoC) preserves the semantics of the original AADL model or at least some of its properties. Moreover, the AADL standard lacks a formal semantics to make the validation of this translation possible. Albeit some of the related works give informal explanations on the model transformations they apply to interpret or compile AADL, the formal proof of semantics preservation remains in most cases altogether impossible. Our contribution is to bridge this gap by providing two formal semantics for a synchronous subset of AADL, which includes periodic threads and data port communications. Its operational semantics is formalized as a TTS (Timed Transition System). This formalization is one prerequisite to the formal proof of semantics preservation for our model transformation from AADL sources to our target verification formalism: TASM (Timed Abstract State Machine). 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 family of semantics functions, which associates each AADL construct to a TASM fragment. Then, the proof of simulation equivalence between the TTSs of the AADL and the TASM models is formalized and mechanized using the proof assistant Coq.

Keywords-AADL; TASM; translational semantics; operational semantics; semantics preservation

I. INTRODUCTION

AADL (Architecture Analysis and Design Language)[1] 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.

Safety-critical systems are often required to pass stringent processes of verification to provide sufficiently strong

evidence about their correctness. When described using an AADL model, such a system specification is always transformed to another formal model for verification and analysis, such as, the translations to BIP[2], to ACSR[3], to IF[4], to Fiacre[5], to Real-Time Maude[6], to Lustre[7] or to Polychrony[8]. The goal is to reuse conveniently the existing verification methods and tools of these formal models.

However, a challenge problem is to prove that they preserve the semantics of the model or at least some specific properties. On the one hand, the AADL standard is described informally without a formal semantics, and it is difficult to prove the equivalence between a formal domain and a natural-language domain. On the other hand, the interpretation of AADL in a formal verification framework implicitly defines an intended translational semantics. In most related work on AADL model transformation, the modeling concepts of both AADL and target language are always expressed in natural language, the translation rules are either expressed by natural language or by an algorithm. These make it be also difficult to support the formal proof of semantics preservation.

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[9]. TASM is an extension of the Abstract State Machine formalism, which supports resource consumption and timing as well as behavior and communication.

This work is based on the AADL v2.0. Model checking and simulation are used to verify AADL models through their translation to TASM, while the theorem prover Coq[11] is used to prove the methodology, i.e., the correctness of the translation. The proof of semantics preservation strongly demands for a precise translational semantics and a precise operational semantics of AADL, and it is based on the simulation-equivalence principle of Timed Transition Systems.

The paper is organized as follows. Section II introduces the preliminaries. Section III 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 IV. Section V presents the two formal semantics of the subset of AADL. Section VI shows the mechanized proof of semantics preservation. Section VII discusses the related work, and Section VIII gives some concluding remarks.

II. PRELIMINARIES

A. Timed Transition System

Timed Transition System (TTS) is an abstract and common method for semantics expressions and is used to compare the semantics of real time specification languages.

Given R^+ the set of nonnegative Reals, we formally define our notation of a TTS.

Definition 1: Timed Transition System. A TTS is a tuple $\Gamma = \langle S, S^0, \Sigma, P, \rightarrow, L \rangle$ where:

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

There are two kinds of transition relations: $(s, \sigma, s') \in \rightarrow$ is also written $s \xrightarrow{\sigma} s'$ those from $S \times \Sigma \times S$ are called *discrete* transitions, those from $S \times R^+ \times S$ are the *continuous* transitions.

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

Synchronous product 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 and connections.

Definition 2: Synchronous Product. Consider n TTSs $\Gamma_i = \langle S_i, S_i^0, \Sigma_i, P_i, \rightarrow_i, L_i \rangle, i = 1...n$. The synchronous product is a TTS: $\Gamma = \prod_{i=1..n} \Gamma_i = \langle S, S^0, \Sigma, P, \rightarrow, L \rangle$, such that:

- $S = S_1 \times ... \times S_i \times ... \times S_n$
- $S^0 = S_1^0 \times ... \times S_i^0 \times ... \times S_n^0$
- $\Sigma = \bigcup_{i=1...n} \Sigma_i$
- $P = \bigcup_{i=1...n} P_i$
- $\rightarrow$ satisfies the following rules:

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

- $L = \bigcup_{i=1...n} L_i(S_i)$

Bisimulation and its variants are usually formulated over TTS. In this work, what is proved in Coq is strong simulation equivalence which implies $\forall CTL*$ and $\exists CTL*$ preservation.

Definition 3: Strong Simulation Equivalence. Given an alphabet Σ and two TTSs $\Gamma_1 = \langle S_1, S_1^0, \Sigma, P, \rightarrow_1, L_1 \rangle$ and $\Gamma_2 = \langle S_2, S_2^0, \Sigma, P, \rightarrow_2, L_2 \rangle$, we say that Γ_1 strongly simulates

Γ_2, noted $\Gamma_1 \prec \Gamma_2$, if there is a relation $R \subseteq S_1 \times S_2$ called a strong simulation relation, such that:

- $\forall s_1^0 \in S_1^0, \exists s_2^0 \in S_2^0 : (s_1^0, s_2^0) \in R$
- $\forall (s_1, s_2) \in R$, $\forall \sigma \in (\Sigma \cup R^+)$, $\forall s_1'$ such that $(s_1, \sigma, s_1') \in \rightarrow_1$, it exists $s_2' \in S_2$ such that $(s_2, \sigma, s_2') \in \rightarrow_2$ 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 and invariants when used in the proof.

B. The Coq Theorem Prover

Coq[11] is a theorem prover based on the Calculus of Inductive Constructions which is a variant of type theory, following the "Curry-Howard Isomorphism" paradigm, enriched with support for inductive and co-inductive definitions of data types and predicates.

From the specification perspective, Coq offers a rich specification language to define problems and state theorems. From the proof perspective, proofs are developed interactively using tactics, which can reduce the workload of the users. Moreover, the type-checking performed by Coq is the key point of proof verification.

Here we give the Coq expressions of the definition of TTS and its synchronous product.

```
Variable State: Type.
Record TTS: Type := mkTTS {
 Init: State -> Prop;
 Delay: State -> Time -> State -> Prop;
 Next: State -> State -> Prop;
 Predicate: Type;
 Satisfy: State -> Predicate -> Prop
}.
Definition TTSIndexedProduct Ind (tts: Ind -> TTS): TTS := {
 Init st := forall i, Init (tts i) st;
 Delay st d st' := forall i, Delay (tts i) st d st';
 Next st st' := forall i, Next (tts i) st st';
 Predicate := { i: Ind & Predicate (tts i) };
 Satisfy st p := Satisfy (tts (projT1 p)) st (projT2 p)
}.
```

III. A SYNCHRONOUS SUBSET OF AADL

AADL includes all the standard concepts of any ADL, such as components, ports, and connections to link the ports. AADL provides three kinds of components:

- 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.

We consider periodic threads with data port communication only, and their details will be given in Section V.

A. Periodic Thread

AADL supports the classic thread dispatch protocols: periodic, aperiodic, sporadic or background. Several properties can be assigned to a periodic thread, such as: the *Dispatch_Protocol*, supposed to be *Periodic*, period given by the *Period* property in the form of period=100ms or

frequency=10Hz, *Execution Time*, and *Deadline*. By default, when the deadline is not specified it equals the period.

B. Data Port Connection

Port connections link ports to enable the exchange of data and event among components. AADL defines three types of component ports: data, event and event data ports. Event and event data ports support queueing buffers, but data ports only keep the latest data. Determinism is an important property of safety critical systems. In order to ensure deterministic data communication between the data ports of periodic threads, AADL offers two communication mechanisms: *immediate* and *delayed*. For an immediate connection, the execution of the recipient thread is suspended until the sending thread completes its execution. 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.

Here, we give the abstract syntax of the chosen subset of AADL:

Type Thread := { Period : Duration;
                     WCET: Duration;
                     Deadline : Duration;
                     Iports: set of Port;
                     Oports: set of Port;
                   }
Type Connection := { SourcePort: Port;
                         PortConnectionType: {immediate, delayed};
                         DestinationPort: Port;
                       }
Type Model := { Threads : set of Thread;
                    Connections: set of Connection;
                  }

IV. A BRIEF OVERVIEW OF TASM

A basic TASM[9] specification is a pair: <E, ASM>. The environment E=<EV, TU> contains environment variables and their types that include real numbers, integer, boolean, and user-defined types. The machine ASM=<MV, CV, IV, R> consists of four parts – monitored variables, controlled variables, internal variables and the set of rules with the form "*if condition then action*", where *condition* is an expression depending on the monitored variables, and *action* is a sequence of one or more updates of the controlled variables. The internal variables are not visible to the environment.

A machine can have several rules, but just one rule can be selected for execution. Each rule has a duration and resource usage.

- Time specifies the duration of a rule execution, which can take the form of a single value *t*, 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. Moreover, time can be discrete, like t:=1.

- Resource is defined as a global quantity that has a finite size such as processor usage, memory, power and bandwidth, etc. Each rule specifies how much of a given resource it consumes.

There are three categories of machines: TASM uses a set of *main machines* to support the specification of parallel behavior, and provides *sub-machine* and *function-machine* calls to support the specification of hierarchical behaviors. The communications are only between the main machines and can use channel synchronization and shared variables.

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

$$P ::= x := exp \mid skip \mid Ch! \mid Ch? \mid if \; Bexp \; then \; P \mid time \; (tmin, tmax) \rhd P$$
$$\mid time \; next \; \rhd \; P \mid resource \; r \; (rmin, rmax) \; \rhd P \mid P \oplus P \mid P \otimes P$$
$$TASM ::= <E, P \parallel P \parallel ... \parallel P>$$

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.

The semantics of a main machine is the following: read the shared variables, select a rule of which condition is satisfied, wait for the duration of the execution while consuming the resources, and then apply the updates to the environment. If there are communications with other machines, it also needs to synchronize.

V. FORMAL SEMANTICS OF THE SUBSET OF AADL

In this section, we give the informal interpretation, translational semantics into TASM and operational semantics using a TTS of the subset of AADL.

A. Informal Interpretation

Periodic Thread: A periodic thread is *dispatched* periodically by the clock, and its *inputs* received from other threads are frozen at dispatch time (by default), i.e., at the time zero, or at the Input_Time property of the input ports. 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 the completion time (by default) or as specified by the Output_Time property.

Data Port Connection: We suppose here two threads have the same period and dispatch at the same time.

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 the dispatch time, so the recipient needs to replace the old data using the data got from the sender at the start of execution.

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.

The time line of a periodic thread with data port communication is represented in Fig. 1.
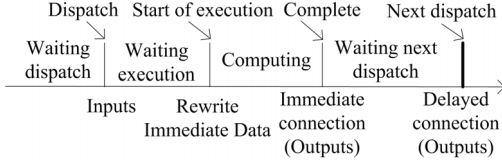

Figure 1. The time line of an AADL periodic thread with data port communication.

B. Translational Semantics

A global semantics function is defined, which has two parts: the mapping of structural aspects into the TASM environment (thread-related variables, connection-related variables), and the mapping of dynamic aspects into the TASM rules (such as the behaviors of threads and connections). The auxiliary functions will be defined later.

$$\text{Translate (m:Model)} =$$
$$< \bigcup_{th \in m.Threads} \text{Trans\_ThreadData(th)} \cup (\bigcup_{cn \in m.Connections} \text{Trans\_ConnectionData (cn)}),$$
$$(\parallel_{th \in m.Threads} \text{Trans\_Thread(th)}) \parallel (\parallel_{cn \in m.Connections} \text{Trans\_Connection(cn)}) >$$

1) The TASM environment is defined as a set of typed variables such as the *state* and *resource consumption* of the thread. A data port only keeps the latest data, so we use an integer variable to express it. 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 1. The TASM environment.

```
Trans_ThreadData (th) =
{ State : { waiting_dispatch, waiting_execution, execution, completed}
        := waiting_dispatch;
  Iport: Integer:= 0;
  Oport: Integer:= 0;
  Resource  Processor, Bus,… /*  used for resource consumption */
  …
}
```

2) In TASM, time can be logical. At each instant, we can use a rule to express the behaviors.

Each thread is expressed by a main machine with five rules: *Dispatch, Waiting Execution, Execution, Write Data,* and *Waiting Next Event*:

Dispatch. A periodic thread is dispatched periodically by the clock. We use a machine to express the clock controller and it sends the dispatch signal to the machine associated to the thread. A family of shared variables *Dispatch:{true, false}* are used to achieve it.

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:{true, false}* are used.

Execution. The duration of execution is the *WCET* and processor consumption is *100%*: we assume execution cannot be preempted.

Write Data. The execution results are copied to the *IportBuffer* of the receiver.

Waiting Next Event. This rule applies when dispatch and execution conditions are not satisfied and allows waiting for the next update of the system.

The behavior of a connection is separated as two parts: *Read* and *Write*. Moreover, the execution is abstracted by a function *Computation (Iports, Oports)* which consumes CPU time, and it can be refined by the AADL behavior annex[12].

Listing 2. The TASM rules.

```
Trans_Thread (th) =
// Rule Dispatch
Time 0  ▷  ( if State(th)=waiting_dispatch and Dispatch(th)=true then
  State(th):= waiting_execution  ⊗ ( ⊗         Iport(ip):=Iport_Buffer(ip))
                                       ip∈Iports(th)
⊕
// 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 WCET(th)  ▷ Resource Processor 100 ▷
   ( if State(th) = execution then
     Computation(Iports(th),Oports(th)) ⊗  State(th):= completed )
⊕
//  Rule Write Data
Time 0 ▷ (if State(th) = completed then
  State(th):= waiting_dispatch ⊗ Trans_Connection_Write(th)
⊕
// Rule Waiting Next Event
Time next  ▷ (else then
          skip)
Trans_Connection_Read(th) =
      ⊗                                                      Iport(ip):=Iport_Buffer(ip))
ip∈Iports (th)∧cn∈Connections∧DestinationPort(cn)=ip∧ConnectionType(cn)=Immediate
Trans_Connection_Write(th) =
      ⊗                          IportBuffer(DestinationPort(cn)):=Oport(op)
op∈Oports (th)∧cn∈Connections∧SourcePort(cn)=op
```

C. Operational Semanitcs

Its operational semantics in terms of TTS is defined, where the state *S* contains the state of the thread (*ThState(th)*), the values of inputs (*Iports(th)*) and outputs (*Oports(th)*), the dispatch condition (*Dispatch(th)*), the execution condition (*Get_CPU(th)*), and the next period of the thread (*NextPeriod(th)*). Its transition is defined by the rules in the following. The rule *WAIT_EVENT* corresponding to the TASM rule *Waiting Next Event*, is a delay rule of which guard is the complement of other guards. The delay we have to wait will be affected by other threads.

$$\frac{\begin{array}{l} S(ThState(th)) = waiting\_dispatch, \\ S(Dispatch(th)) = true, \\ S'(ThState(th)) = waiting\_execution, \\ \forall ip \in Iports(th), S'(ip) = IportBuffer(ip), \\ \forall op \in Oports(th), S'(op) = S(op), \\ S'(NextPeriod(th)) = S(NextPeriod(th)) + Period(th) \end{array}}{S \xrightarrow{dispatch(th),0} S'} \text{ DISPATCH}$$

$$\frac{\begin{array}{l} S(ThState(th)) = waiting\_execution, \\ S(Get\_CPU(th)) = true, S'(ThState(th)) = execution, \\ \forall ip \in Iports(th), S'(ip) = IportBuffer(ip), \\ \forall op \in Oports(th), S'(op) = S(op), \\ S'(NextPeriod(th)) = S(NextPeriod(th)) \end{array}}{S \xrightarrow{waiting(th),immediate,0} S'} \text{ WAITING\_EXE[1]}$$

$S(ThState(th)) = waiting\_execution,$
$S(Get\_CPU(th)) = true, S'(ThState(th)) = execution,$
$\forall ip \in Iports(th), S'(ip) = S(ip),$
$\forall op \in Oports(th), S'(op) = S(op),$
$S'(NextPeriod(th)) = S(NextPeriod(th))$
$$\frac{}{S \xrightarrow{waiting(th),delayed,0} S'}$$ **WAITING_EXE[2]**

$S(ThState(th)) = execution, S'(ThState(th)) = completed,$
$\forall ip \in Iports(th), S'(ip) = S(ip),$
$\forall op \in Oports(th), S'(op) = Computation(ip,op),$
$S'(NextPeriod(th)) = S(NextPeriod(th)), d = WCET(th)$
$$\frac{}{S \xrightarrow{execution(th),d} S'}$$ **EXECUTION**

$S(ThState(th)) = completed, S'(ThState(th)) = waiting\_dispatch,$
$\forall ip \in Iports(th), S'(ip) = S(ip),$
$\forall op \in Oports(th), \forall cn \in Connections,$
$IportBuffer(DestinationPort(cn)) = S'(op),$
$S'(NextPeriod(th)) = S(NextPeriod(th))$
$$\frac{}{S \xrightarrow{write(th),0} S'}$$ **WRITE_DATA**

$(S(ThState(th)) = waiting\_dispatch, S(Dispatch(th)) = false,)$
or
$(S(ThState(th)) = waiting\_execution, S(Get\_CPU(th)) = false,)$
$S'(ThState(th)) = S(ThState(th)),$
$\forall ip \in Iports(th), S'(ip) = S(ip),$
$\forall op \in Oports(th), S'(op) = S(op),$
$S'(NextPeriod(th)) = S(NextPeriod(th))$
$$\frac{}{S \xrightarrow{waiting\_next\_event(th),d} S'}$$ **WAIT_EVENT**

Figure 2. The operational semantics in TTS.

VI. THE PROOF OF SEMANTICS PRESERVATION

In this section, we give the main idea of the proof of semantics preservation from AADL to TASM: compositional proof and strong simulation equivalence.

A. Strong Simulation Equivalence

As defined in Section II, the simulation relation R is general. Here it is specialized using a mapping function and an invariant which restricts considered states to a superset of reachable states. Furthermore, auxiliary variables are needed in order to establish the simulation. They contain information about previous states that are needed to build the TTS of the AADL model from the TTS of TASM model. Moreover, we attach the same set of observers to both models, as for example "*port p of thread t has value v*". Their semantics is defined by a predicate over the state space of each model and be compatible with the chosen mapping function when the invariant is satisfied. Fig.3 shows the principle of the simulation equivalence of this paper.
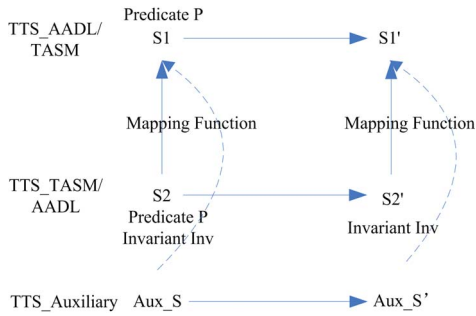

Figure 3. The principle of strong simulation equivalence.

The Coq expressions of TTS and synchronous product have been given in Section II, the Coq expressions of simulation equivalence are shown in the following:

```
Variables StateA StateC: Type.
Record mapping: Type := mkMapping {
  mState: Type; // auxiliary state
  mInit: StateC -> mState;
  mNext: mState -> StateC -> mState;
  mDelay: mState -> StateC -> Time -> mState;
  mabs: mState -> StateC -> StateA
}.
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;
  invDelay: forall ex1 st1 st2 d, Delay _ c st1 d st2 -> inv ex1 st1 -> inv
            (mDelay m ex1 st1 d) st2;
  invNext: forall ex1 st1 st2, Next _ c st1 st2 -> inv ex1 st1 -> inv (mNext m
            ex1 st1) st2;
  simuInit: forall st, Init _ c st -> Init _ a (mabs m (mInit m st)) st;
  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);
  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);
  simuPred: forall ext st, inv ext st -> (forall p, lpSat (Satisfy _ c) st (trc p)
            <-> lpSat (Satisfy _ a) (mabs m ext st) (tra p))
}.
```

B. 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, so we would like to prove the simulation equivalence between the operational semantics of a thread and its translational semantics in a TASM machine, not the full semantics. This method is called as compositional proof and it will reduce the complexity of the proof.

Firstly, we need three Lemmas:

Lemma 1. Let AADL = $< Th_1,…, Th_n>$ be an AADL model, which contains several periodic threads with delayed connections. Let *TTS_AADL* be its operational semantics defined as a TTS, and *TTS_Th(i)* be the operational semantics of the thread *i*, then $TTS\_AADL \prec \prod_{i=1…n} TTS\_Th(i)$, and vice versa.

Lemma 2. Let TASM = $< MM_1,…,MM_n>$ be a TASM model which contains several main machines. Let *TTS_TASM* be its operational semantics defined as a TTS, and *TTS_MM(i)* be the operational semantics of the main machine *i*, then $TTS\_TASM \prec \prod_{i=1…n} TTS\_MM(i)$, and vice versa.

Lemma 3. If $TTS\_i \prec TTS'\_i$, then $\prod_{i=1…n} TTS\_i \prec \prod_{i=1…n} TTS'\_i$.

The abstract syntax of AADL and TASM are presented as inductive data types or record types, and their semantics (*TTS_AADL*, *TTS_Th(i)*, *TTS_TASM*, *TTS_MM(i)*) are presented as inductive predicates. The proof of the three lemmas is done by case analysis.

Then, we give the translation from AADL to TASM.

Finally, we prove the theorems as follows:

Theorem 1. $TTS\_Th(i) \prec TTS\_MM(Trans\_Thread(i))$.

Theorem 2. $TTS\_MM(Trans\_Thread(i)) \prec TTS\_Th(i)$.

The main body of the theorems in Coq is in the following.

```
Theorem AADL2TASM_simu1:
forall th, simu _ _ A2T (AADLPred Value sys) (MM_TTS _ _
(AADL2TASM) th) (Thread_TTS _ sys th) AP2TP (fun x => LPPred _ x).
```

Proof.
  simpl; intros.
    split with (getInvariant th); simpl; intros. **...**
Qed.

Theorem AADL2TASM_simu2:
 forall th, simu _ _ T2A (AADLPred Value sys) (Thread_TTS _ sys th)
 (MM_TTS _ _ AADL2TASM th) (fun x => LPPred _ x) AP2TP.
Proof.
 intros.
  split with (getMMInvariant th); simpl; intros; auto. **...**
Qed.

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 define 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, because some information may be missing when translated AADL state space to TASM. The proof of the two theorems is also done by case analysis. The Coq source can be downloaded at http://www.irit.fr/PERSONNEL/ CADIE/bodeveix/COQ/AADL2TASM/V0/.

VII. RELATED WORK

Model transformation technology is often used to interpret AADL models with a formal specification suitable for analysis and verification. In most of the related works, however, the proof of semantic preservation of such transformations is not addressed. Instead, most of the related work claims manual validation of the proposed technique, such as, the translations to BIP [2], to ACSR[3], to IF[4], to Fiacre[5], to Real-Time Maude[6], to Lustre [7] and to Polychrony [8].

The work presented in [13] gives the formalization for a subset of the AADL Meta model using the B method and the Isabelle proof assistant. This allows proving the correct transformations of AADL models, and an example of AADL flow is given. Similarly, [14] proposes a refinement development of a synchronous subset of AADL using Event B method, and gives simple proof obligations. [15] interprets the AADL mode change protocol using Timed Petri Net, which allows to verify some properties pertaining to this models and validate the proposed translation. In [16], we present a comparative study of FIACRE and TASM to define an AADL subset.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a translation of AADL concepts into TASM and a methodology to prove this translation correct. A synchronous subset of AADL we consider includes periodic threads and data port communications. We first formalize the translational semantics by a family of semantics function which inductively associates a TASM fragment with each AADL construct. Second, the operational semantics is formalized as a Timed Transition System. Finally, we formalize the theorem and proof of simulation equivalence between the two formal semantics and mechanize it using the Coq proof assistant.

In addition to the work presented in this paper, we already formalized other subset of AADL using TASM, such as the behavior annex [18] and mode changes [19]. We are currently working on the proof of these additional elements of AADL.

REFERENCES

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

[2] M.Y.Chkouri, A.Robert, M.Bozga, and J.Sifakis, "Translating AADL into BIP-Application to the Verification of Real-time Systems, " Proc. MoDELS Workshops 08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 5−19.

[3] O. Sokolsky, I. Lee, and D. Clark, "Schedulability Analysis of AADL models, " Proc. IPDPS 06, IEEE Press, Rhodes Island, 2006, 8 pp.

[4] T. Abdoul, J. Champeau, P. Dhaussy, P. Pillain, and J. C. Roger, "AADL execution semantics transformation for formal verification, " Proc. ICECCS 08, IEEE Press, Belfast, 2008, pp. 263-268.

[5] B. Berthomieu, J. P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat, "Formal Verification of AADL Specifications in the Topcased Environment, " Proc. Ada-Europe 09, LNCS, vol. 5570, Springer, 2009, pp.207-221.

[6] P. Csaba Ölveczky, A. Boronat and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. FMOODS/PORTE 2010, pp.47-52.

[7] N. Halbwachs, E. Jahier, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of AADL models via a translation into synchronous programs," Proc. EMSOFT 07, ACM, pp. 134-143.

[8] Y.Ma, J. P.Talpin, S.Shukla, and T. Gautier , "Distributed simulation of AADL specifications in a polychronous model of computation, " Proc. ICESS 09, IEEE Press, 2009, pp. 607-614.

[9] M. Ouimet and K.Lundqvist. The TASM Language Reference Manual Version 1.1. November , 2006.

[10] M. Ouimet and K. Lundqvist, "The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems," Proc. CAV 2007,pp 126-130.

[11] The Coq Proof Assistant Reference Manual Version 8.2. The Coq Development Team. February, 2009.

[12] SAE AS5506 Annex: Behavior_Specification V2.0 September 20, 2007.

[13] J. Bodeveix, D. Chemouil, M. Filali, M. Strecker, "Towards formalising AADL in Proof Assistants," Electronic Notes in Theoretical Computer Science.Volume 141 Issue 3, December, 2005.

[14] M. Filali, J. L. Lawall, "Development of a Synchronous Subset of AADL, " Proc. ASM 2010, pp.245-258.

[15] D. Bertrand, A.M. Déplanche, S. Faucou, O. H. Roux, "A Study of the AADL Mode Change Protocol, " Proc. 13th ICECCS, 2008, pp. 288-293.

[16] L. Pi, Z.Yang, J. Bodeveix and M. Filali, K. Hu and D. Ma, "A Comparative Study of FIACRE and TASM to Define AADL Real Time Concepts., " Proc. 14th ICECCS, 2009, pp.347-352.

[17] M. Ouimet, "A Formal Framework for Specification-Based Embedded Real-Time System Engineering, " Ph.D Thesis. MIT. 2008.

[18] Z. Yang, K. Hu, D. Ma, L.Pi, "Towards a formal semantics for the AADL behavior annex, " Proc. DATE 2009, pp.1166-1171.

[19] Z. Yang, K. Hu, D. Ma, L.Pi, and J.Bodeveix, "Formal Semantics and Verification of AADL Modes in Timed Abstract State Machine, " Proc. PIC 2010, pp.1098-1103.

[20] M. Ouimet and K. Lundqvist, "Verifying Execution Time using the TASM Toolset and UPPAAL," MIT Technical Report 2007.