Exploring AADL verification tool through model transformation

Kai Hu, Teng Zhang, Zhibin Yang, Wei-Tek Tsai

Exploring AADL verification tool through model transformation

Kai Hu [a], Teng Zhang [b,*], Zhibin Yang [b,c,d,*], Wei-Tek Tsai [e]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, PR China
[b] School of Computer Science and Engineering, Beihang University, Beijing, PR China
[c] IRIT-CNRS, Université de Toulouse, Toulouse, France
[d] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, PR China
[e] School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, USA

A B S T R A C T

Architecture Analysis and Design Language (AADL) is often used to model safety-critical real-time systems. Model transformation is widely used to extract a formal specification so that AADL models can be verified and analyzed by existing tools. Timed Abstract State Machine (TASM) is a formalism not only able to specify behavior and communication but also timing and resource aspects of the system. To verify functional and nonfunctional properties of AADL models, this paper presents a methodology for translating AADL to TASM. Our main contribution is to formally define the translation rules from an adequate subset of AADL (including thread component, port communication, behavior annex and mode change) into TASM. Based on these rules, a tool called AADL2TASM is implemented using Atlas Transformation Language (ATL). Finally, a case study from an actual data processing unit of a satellite is provided to validate the transformation and illustrate the practicality of the approach.

1. Introduction

Architecture Analysis and Design Language (AADL) [1] provides a standard and precise way to describe the software/hardware architecture, run-time environment, functional and non-functional properties of embedded real-time systems. An AADL specification defines various kinds of software and hardware components (such as *system*, *process*, *thread*, *subprogram*, *data*, *processor*, and *bus*), their real-time properties (such as *period*, *deadline*, *WCET*) and how they interact with each other using ports, subprogram calls and other interaction mechanisms. Furthermore, mode change, partition, scheduling strategy and other features of real-time systems are also provided. Besides, AADL is an extensible language: AADL standard defines *Behavior Annex* [2,3] to refine the behavior of threads.

To ensure the safety and dependability of embedded real-time systems, rigorous certification process is needed before putting the system into actual use. Although IDE such as OSATE [4] and TOPCASED [5] provide some tools to verify properties of AADL models like flow latency, academics and industries tend to utilize model transformation methodology to verify and analyze AADL model by using existing verification and analysis tools. A lot of studies have been proposed: translation to Behavior Interaction Priority (BIP) [6], to ACSR [7], to IF [8], to Fiacre [9], to TLA+ [10], to RT-Maude [11], to Petri nets [12–14], to UPPAAL timed automata [15], to Lustre [16], to SIGNAL [17,18], to EDA (Event-Data Automata) [19], to UML Marte [20], etc.

This paper presents a transformation from AADL into Timed Abstract State Machine (TASM) [21,22]. TASM is a formal specification language used to specify and simulate the behavior of real-time systems. By extending the abstract state machine (ASM) formalism, the TASM language is able to express three key aspects of all embedded real-time systems: function, timing, and resource usage. Moreover, TASM toolset [21] is provided to analyze timing and resource consumption of the model which also supports consistency and completeness of the model. Compared with studies mentioned above, contributions of this paper are:

- A proper subset of AADL is chosen as the transformation target including thread (dispatching, offline scheduling and execution), port connections, behavior annex and mode change. A safety-critical system with certain scale can be modeled using this subset.
- TASM is chosen as the transformation target such that timing and resource information of AADL can be described. Not only functional properties such as deadlock freeness and state reachability, but also timing correctness and resource usage can be verified and analyzed using TASM toolset.

* Corresponding authors at: School of Computer Science and Engineering, Beihang University, Beijing, PR China. Tel.: +86 13520357769 (T. Zhang). Tel.: +86 18515358736 (Z. Yang).

E-mail addresses: rahxephon89@163.com (T. Zhang), zhibinyang168@gmail.com (Z. Yang).

- The translational rules are defined in a formal way so that the verification of the transformation can be fulfilled.

Some work have been done in translational semantics from AADL to TASM and the formal proof of semantics preservation for the model transformation [23–26]. In this paper, we focus on the definition of the formal and concrete translation rules. A transformation tool AADL2TASM has been implemented based on the translational rules.

After translated into TASM model, several properties can be verified. In this paper, we concentrate on properties of functional correctness such as deadlock freeness [27] and state reachability, timing correctness and resource consumption correctness. Note that we only consider offline scheduling in this paper such that the analysis on preemptive scheduling protocols are not supported.

The rest of the paper is organized as follows. Section 2 gives the introduction to AADL and TASM. Section 3 translates AADL (port communication, thread, mode change and scheduling) into TASM. Section 4 describes the implementation of AADL2TASM transformation tool and gives a case study. Section 5 discusses the related work. Conclusion and future work are presented in Section 6.

2. Introduction to AADL and TASM

2.1. AADL

AADL is able to model a real-time system as a hierarchy of software components bound to an execution platform, as shown in Fig. 1. Predefined software component types such as thread, thread group, process, data and subprogram are used to model the software architecture of the system. Behavior annex is defined for the refinement of thread behaviors. Processor, memory, device and bus components are the execution platform components for modeling the hardware part of the system. Ports and port connections are provided to model the exchange of data and event among components. Functional and non-functional properties like scheduling protocol and execution time of the thread can be specified in components and their interactions. AADL also provides a way to describe multi-mode systems, in which a mode is an explicitly defined configuration of contained components, connections, and property values. System components are used to represent composites sets of software and execution platform components.

2.2. TASM

The Timed Abstract State Machine (TASM) [21,22] is based on the theory of Abstract State Machines (ASM) [28]. It extends ASM with time and resource consumption declarations such as task durations, CPU or memory consumption. The TASM language has a formal semantics, which makes its semantics precise and enables executable specifications.

A TASM specification is a three-tuple $< E, AASM, MASM >$. E is the *environment* defined as a pair: $E = < EV, TU >$ in which EV is the set of *Environment Variables* and TU is the set of types for environment variables, consisting of real numbers, integer, Boolean, and user-defined types. There are three kinds of machines: *main machine*, *sub machine* and *function machine*. *AASM* is the set of auxiliary machines including *sub machines* and *function machines*. *MASM* is the set of main machines *ASM* executed in parallel. An *ASM* is a four-tuple: $ASM = < MV, CV, IV, R >$. The communications are only between the main machines using channel synchronization and shared variables. *MV* denotes the *Monitored Variables*; *CV* denotes the *Controlled Variables*; *IV* denotes the *Internal Variables* and *R* denotes a set of *rules*. Rule takes 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 called "**else then**" *rule*. It will be executed when all conditions of other rules cannot be met. A restriction on the set of rules is that they are mutually exclusive, that is only one rule can be executed at each step. Informally, 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 then apply the updates to the environment. Note that all auxiliary machines in the TASM model can be represented by main machines so that this paper only considers the transformation for main machines.

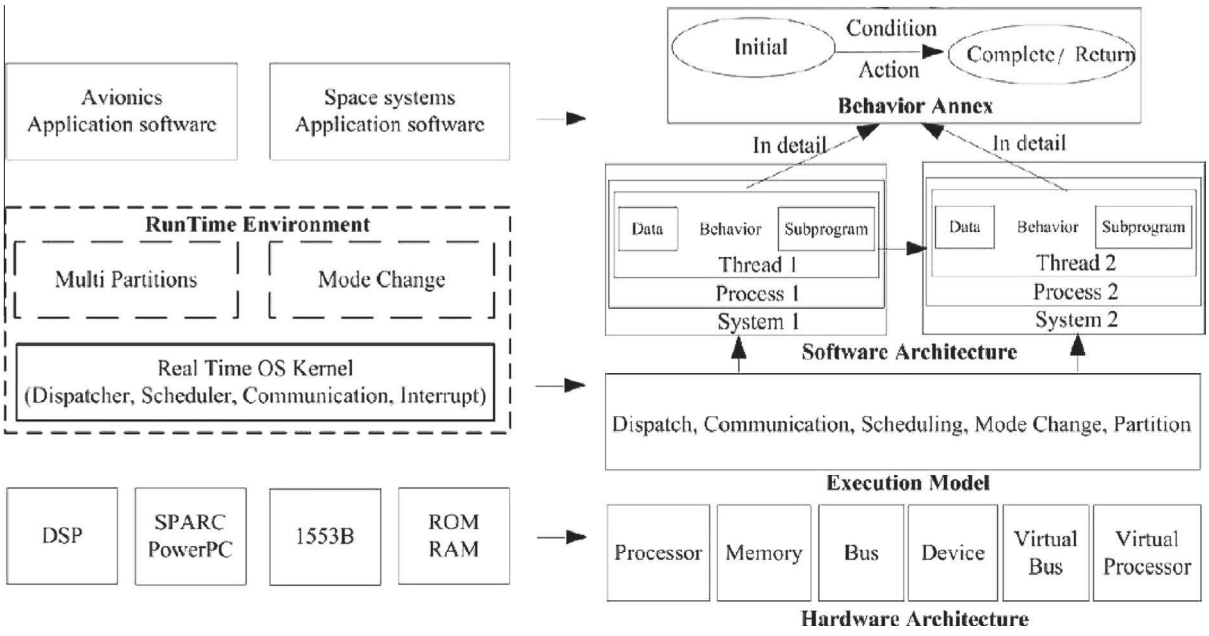

Fig. 1. Correspondence between requirements of real-time system modeling and conception of AADL modeling.

An example of TASM machine is shown in Fig. 2. *ThState* and *BA* are user-defined types (line 3–5). *thstate_th1* and *BAState_th1* are corresponding variables (line 9). *Cpu* is the resource to be consumed during the execution of rules (line 15 and 20), which has been specified in rule *s0_2_s1* (line 14–18) *s1_2_s0* (line 19–22). The TASM specification can be simulated and analyzed by TASM toolset.

2.3. The subset of AADL

For translation, this paper proposes formal definitions of the subset of AADL including port connection, thread component and mode change. As behavior annex is an extension for the specification of AADL components, the definition is also given. For detailed semantics of AADL, the reader is referred to [26].

(1) Port connection
Ports are the logical connection points between 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. To transfer the data and event among components, port connections are provided. The definition is shown as below.

Definition 1. *PortConnection* = $<$ *sourceport*, *targetport*, *EM* $>$, where,

- *sourceport* is the source port of the connection.
- *targetport* is the target port of the connection.
- *EM* = {*immediate*, *delayed*} is the execution property of data port connection. AADL provides two communication paradigms, *immediate* communication and *delayed* communication. In immediate communication paradigm, the writing operation will be executed after the execution of sender thread. The receiver will not begin to execute until having received the data from the port. In delayed communication paradigm, the writing operation is delayed to the deadline of the sender. As a result, the receiver will read the data from the previous execution of the sender when it is dispatched.

(2) Thread
Thread, the abstract of a concurrent task or an active object, is the main execution and scheduling unit in AADL. Input and output ports can be specified in the thread for the data or event exchange. Dispatch, scheduling protocols and mode are the execution model properties. Behavior Annex is the refinement of the thread execution. The definition is shown as below.

Definition 2. *Thread* = $<$*IPort*, *OPort*, *TProp*, *BA*, *Mode*, *Scheduling*$>$, where,

- *IPort* = {*IDP*, *IEP*, *IEDP*}. *IDP* is the set of input data ports; *IEP* is the set of input event ports; *IEDP* is the set of input event data ports.
- *OPort* = {*ODP*, *OEP*, *OEDP*}. *ODP* is the set of output data ports; *OEP* is the set of output event ports; *OEDP* is the set of output event data ports.
- *TProp* = {*Dispatch_type*, *Period*, *Compute_execution_time*, *Deadline*}. *Dispatch_type* is the dispatch protocol of thread, including *Periodic* and *Aperiodic* protocol. *Period* is the time interval between two dispatches of the periodic thread. *Compute_execution_time* specifies computation time of thread. *Deadline* specifies the longest time interval between the start and end of the execution.
- *BA* $\in$ *Behavior Annex*, is the precise description of the thread behavior.

```
1:ENVIRONMENT:
2:USER-DEFINED TYPES:
3: ThState:={waiting_dispatch,waiting_execution,execution
4: ,writing,waiting_mode};
5: BA:={s0,s1};
6:RESOURCES:
7: Cpu:=[0,100];
8:VARIABLES:
9: ThState thstate_th1; BA BAState_th1; ....
10:MAIN MACHINE: th1_thread
11:MONITORED VARIABLES: thstate_th1,BAState_th1;
12:CONTROLLED VARIABLES: thstate_th1,BAState_th1;
13:RULES:
14: s0_2_s1 {
15:  t := [30,30];Cpu:=[50,100];
16:  if thstate_th1 = execution and BAState_th1 = s0 then
17:    BAState_th1 := s1;
18:  }
19: s1_2_s0 {
20:   t := [10,10];Cpu:=[20,50];
21:   if thstate_th1 = execution and BAState_th1 = s1 then
22:     thstate_th1 := writing; BAState_th1 = s0;
23: }
24: ...
```

Fig. 2. An example of TASM model.

- *Mode* specifies the mode property of the thread. A thread can only be dispatched, scheduled and put into execution if it belongs to the current mode. In system and process component of AADL, mode change automata is defined specifying threads that is able to be executed.
- *Scheduling* specifies how the thread will be scheduled on CPU to which it is bound. Property *Allowed_Processor_Binding_Class* specifies the CPUs to which thread can be bound when being executed.

(3) Behavior annex
Behavior Annex, taking the form of state machine, is defined in the thread for the description of the behavior of the thread such as port communication, computation and delay. The definition is shown as below.

Definition 3. Behavior Annex = $<S, S_0, V, Guard, Action, T>$, where,

- S is the set of states. Types of the state include initial, return, complete and composite.
- $S_0 \in S$ is the initial state.
- V is the set of local variables.
- *Guard* is the set of guards of state transitions, taking the form $< BExpr > \mid [\text{on} < BExpr > ---- >] < event > [when < BExpr >]$, in which *BExpr* is the predicate on state variables; *event* is the data or event reading operation $(P?, P?(x))$; *when* is the predicate on the data received from event port or event data port.
- *Action* is the set of actions executed during the transition. Types of action allowed include assignment (P:=x), data or event sending (P!(x),P!), computation (Computation (min,max)) and delay (Delay(min,max)).
- $T = S \times \{G, A\} \times S$ is the set state transitions.

(4) Mode change
In AADL, a mode represents an operational state which can be viewed 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 events, data, and event data arrivals cause a mode transition and the new mode. The definition of mode change (denoted as ModeTransition) is shown as below.

Definition 4. Mode Transition = $<M, m_0, Event, Transition>$, where,

- M is the set of System Operation Mode (SOM). SOM is a vector of modes, where each element is associated to a component (for example a *thread* 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.
- $m_0 \in M$ is the initial mode.
- *Event* is the set of events which trigger the mode change.
- $Transition = M \times Event \times M$ is the set of mode changes.

3. Transformation from AADL to TASM

Translating AADL to TASM is to describe the execution of AADL by the execution of state machines in TASM. Fig. 3 shows the sketch map of translation from AADL to TASM. Each thread in AADL model is to be translated into two state machines in TASM, one for the execution and another for the dispatch. Moreover, port communication, mode change automata and processor component with scheduling protocol defined are also to be respectively translated into machines of port communication, SOM and scheduler in TASM. These generated state machines will synchronize with each other using environment variables.

This paper presents the translation rules from three aspects, *monitored variables*, *controlled variables* and the construction of the *main machine* which is denoted as a set of *rules*. In the following parts of the paper, term *rules*of main machine in TASM will be replaced by state transition, making a distinction with term translation rules of the AADL transformation. Several symbols are defined for the convenience of formal description of translation rules. T stands for time consumed during state transition. Grepresents the guard condition of the state transition. A represents actions of the state transition. Actions are in fact the update of controlled variables. From the perspective of semantics, actions under the same guard condition can be executed in parallel. $\otimes$ is used to represent the composite of multiple actions.

The following section will present the translation of port communication. Then translation methods of thread execution, thread dispatch, Behavior Annex, mode change and scheduling will be given.

3.1. Translation of port communication

In AADL, port is defined as the logic connection point between two components for the transmission of data (by data port), control information (by event port) and both of them (by event data

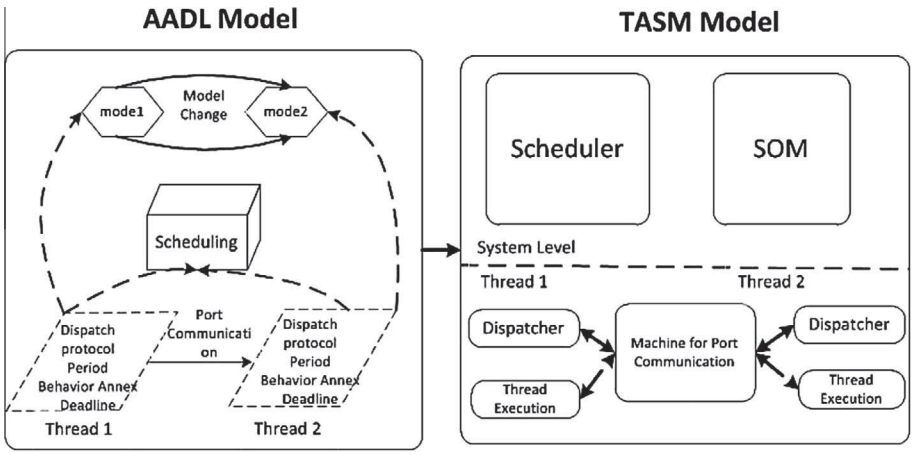

Fig. 3. Map from AADL to TASM.

port). There are two directions of port, input and output. Output port is connected to input port to constitute the port connection.

This paper uses shared environment variables in TASM to describe the port communication in AADL. There are three kinds of ports in AADL, *data port* (denoted as *dp*), *event port* (denoted as *ep*) and *event data port* (denoted as *edp*). The data port and event data port are translated into two environment variables, denoted as Γ (*dp*|*edp*) = {*data*,*event*}. Variable *data* is used to store data in the port while variable *event*, a Boolean variable, has different meanings for these two kinds of ports respectively. For data port, it is used to represent whether new data has arrived. For event data port, it will be true when new event has arrived. Similarly, translation rule of event port is Γ (*ep*) = {*event*}.

For the convenience of simulation and model checking, this paper abstracts the port communication into one state machine, which indicates that there is no concurrent port communication in the model. The state machine is a three-tuple $< PCMV, PCCV, PCMM >$. Note that the set of *Interval Variables* are not used in the transformation.

*PCMV*and *PCCV* are respectively monitored variables and controlled variables needed for the state machine, shown as below.

$$PCMV = \{\bigcup_{spc \in SPC} \Gamma(spc)\}$$
$$PCCV = \{\bigcup_{spc \in SPC} \Gamma(spc), \bigcup_{dpc \in DPC} \Gamma(dpc)\}$$

SPC is the set of source ports of all port connections and *DPC* is the set of corresponding target ports. Therefore, monitored variables needed are all environment variables translated from the source ports and controlled variables are the environment variables translated from both the source and target ports.

PCMM is a set of rules generated, shown as below. Note that rules are also called state transitions in the remainder of the paper.

$$PCMM = \{waiting\_execution, \bigcup_{pc \in PC} Rule(pc)\}$$

State transition *waiting_execution* is used to handle the situation when guard conditions of all other transitions cannot be met. *Rule*(*pc*) represents the translation from a port connection *pc* to the corresponding state transition in *PCMM*. If *pc* is type of data port connection or event data port connection, state transition takes form as below.

$$0 \rightsquigarrow T(\Gamma(pc))$$
$$\Gamma(pc.sourceport).event = true \rightsquigarrow G(\Gamma(pc))$$
$$\Gamma(pc.sourceport).event := false \otimes \Gamma(pc.targetport).event$$
$$:= true \otimes \Gamma(pc.targetport).data := \Gamma(pc.sourceport).data \rightsquigarrow A(\Gamma(pc))$$

It means when new data or event have arrived, event variables for the source ports are assigned to false, followed by the actions of assigning event variables of target ports to true. Then the actions of copy data from source ports to ports will be executed.

Fig. 4 is an example of port connection in AADL and Fig. 5 is the corresponding transitions in the machine. We can see that for event data port *fib_gyro1_in* and *main_sta_order*, corresponding environment variables *event_fib_gyro1_in*, *data_fib_gyro1_in*, *event_main_sta_order* and *data_main_sta_order* are generated. The transition describes the data transfer from *fib_gyro1_in* to *main_sta_order*.

```
thread main_stabilization
  features --interface of the thread
    datacollect : in event data port --ports
    ...
  properties --properties related to the thread
    Dispatch_Protocol => Periodic;
    Period => 250 Ms;
    Deadline => 250 Ms;
    ...
end main_stabilization;
```

Fig. 6. An example of the thread component.

3.2. Translation of thread components

In AADL standard, thread is used to model a concurrent task or an active object, i.e., a schedule unit that can execute concurrently with other threads [1]. Fig. 6 shows an example of the declaration of a thread component. *Features* specifies the interface of the thread; properties specifies important properties related to the execution of the thread such as the dispatch protocol, period of the dispatch for periodic thread and execution deadline. Detailed behaviors of the thread is specified in the implementation part using Behavior Annex (BA). The thread component will be translated into two machines, *execution machine* and *dispatcher*. The execution machine will be used to describe the procedure of the execution and dispatcher is used to control the dispatch according to the properties specified in the declaration. The specification of the BA will be translated as the state transition in TASM and then be integrated into the execution machine. The scheduling problems will be introduced in Section 3.3.

3.2.1. Translation of thread execution

SAE [1] defines hybrid automaton to illustrate the behavior of thread execution, shown in Fig. 7. The six ellipses are the six states of the machine: *halted*, *waiting_mode*, *waiting_dispatch*, *waiting_execution*, *execution* and *writing*. The initial state is *halted*. After the binary file is loaded to the process and it belongs to the current mode, it can transition to the *waiting_dispatch* state. If the thread is not in the current mode, it will stay in the *waiting_mode* state until the new mode contains it. The thread in the *waiting_dispatch* state will be dispatched according to the period

```
portConnection: event data port fib_gyro1_in -> main_sta_order;
```

Fig. 4. Port connection in AADL.

```
fib_gyro1_in_2_main_sta_order
{
t := 0;
  if event_fib_gyro1_in = true then
    event_fib_gyro1_in := false; event_main_sta_order := true;
    data_main_sta_order:=data_fib_gyro1_in;
  }
```

Fig. 5. TASM rule corresponding to port connection in Fig. 4.

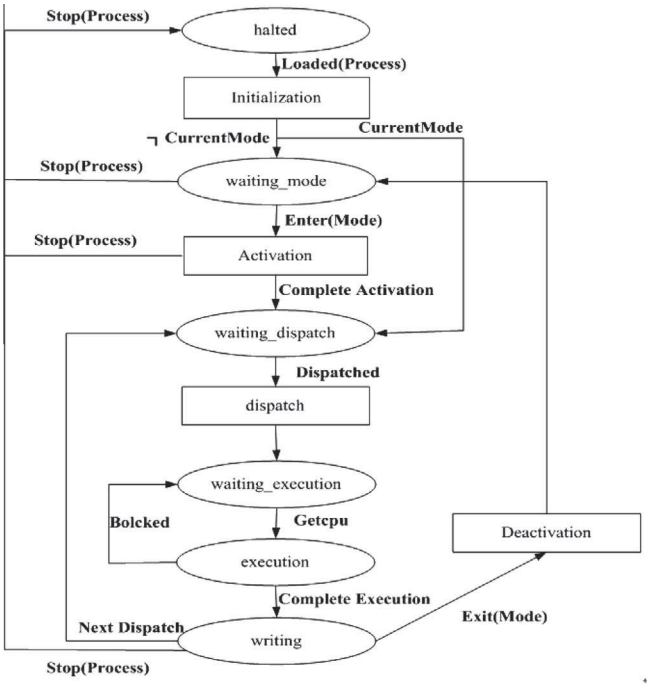

Fig. 7. Hybrid automaton of the thread execution [1].

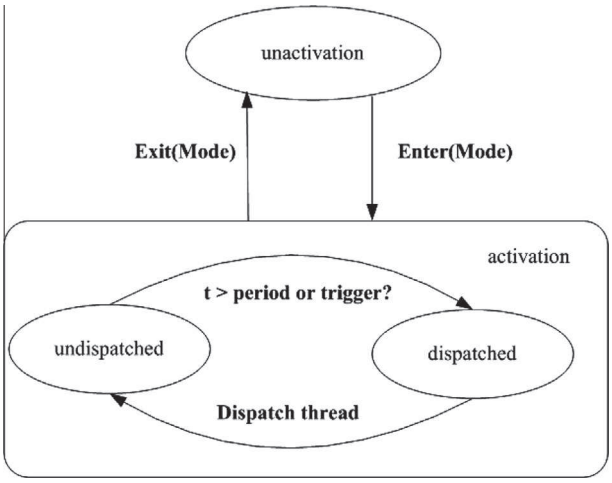

Fig. 8. State machine of the thread dispatcher.

if it is periodic or an event if it is aperiodic. Then the thread is in the *waiting_execution* state. If the thread gets CPU resource, it will transition into *execution* state. Note that if the thread can be preempted, it may be blocked and transitioned back to *waiting_execution* state. After the execution, the thread will transition into *writing* state. The data computed will be written into the output ports. If the thread is still in the current mode, it will transition into *waiting_dispatch* state waiting for the next dispatch. Otherwise, it will be deactivated and transitioned into *waiting_mode* state. If the process containing the thread is stopped, the thread will transition into *waiting_mode* state.

The execution machine is a three-tuple $< ThMV, ThCV, ThMM >$ where $ThMV = \{thState,\ thActive,\ thDispatch,\ thGetcpu\}$ is the set of monitored variables and $ThCV = \{thState\}$ is the set of controlled variables. *ThState* is the variable representing the current state of the machine. *ThActive* is a Boolean variable representing whether the thread belongs to the current mode, controlled by the mode change state machine. *ThDispatch*, controlled by dispatcher state machine, is used to check if the thread has been dispatched. *ThGetcpu* is a Boolean variable representing whether the thread can be put into execution, controlled by the scheduler .

ThMM is the set of state transitions of the execution machine. In this paper, we assume that the process will not be stopped and the load procedure is emitted so *ThMM* has 6 state transitions including *Activation*, *Dispatch*, *Schedule*, *Writing*, *Deactivation* and *Waiting*. The translation rules are shown in Table 1. State transition is performed by the value change of the control variable *thState*. Note that since offline scheduling is used in the processor, no preemption is allowed so that time consumed in the state transition *execution* is the Worst-Case Execution Time (WCET) of the thread. State transition *writing* is used to represent the action to be done when execution completes. For the immediate data port connection, actions of port writing are to be done in this state transition. *Pw(odp)* is the port writing operation where $odp \in IODP$ and *IODP* is the set of immediate output data ports of the thread. State transition *Deactivation* is used to set the thread into state *waiting_mode*. If the thread is *synchronized*, the execution will not be interrupted by the mode change so Deactivation will not be performed until the thread is in *waiting_dispatch* state (thActivate = false $\bigwedge$ thState = waiting_dispatch). Otherwise, execution can be interrupted at any time.

State transition *Waiting* is used to handle the situation when all other transitions in the state machine cannot be executed. All main machines defined in the following part will contain a *waiting* transition, the same as this one.

3.2.2. Translation of dispatcher

As mentioned above, property *Dispatch_Protocol* can be declared in the thread component to describe how the thread will be dispatched by the CPU to which it is bound. Several dispatch protocols are supported in AADL, in which two of them, periodic and aperiodic dispatch, will be introduced in this paper. In the periodic thread, the time interval between two dispatches is specified while in the aperiodic thread, dispatch is triggered by receiving event from defined input event ports. Fig. 8 shows the state machine of the thread dispatcher with three states *unactivation*, *dispatched* and *undispatched*. If the thread does not belong to the current mode, the dispatcher is in the *unactivation* state. If the dispatcher is in *undispatched* state, it will wait for the trigger condition to dispatch the thread. For periodic thread, the condition is that the time that the machine has spent on the state is longer than the period of the thread. For the aperiodic thread, the condition is the arrivals of events received from the input events ports defined in the thread.

Table 1
Translation rules of execution machine.

State transition	Time (*T*)	Guard (G)	Action (A)
Activation	0	thActivate = true $\bigwedge$ thState = waiting_mode	thState := waiting_dispatch
Dispatch	0	thDispatch = true $\bigwedge$ thState = waiting_dispatch	thState := waiting_execution
Schedule	0	thDispatch = true $\bigwedge$ thState = waiting_dispatch	thState := execution
Execution	WCET	thState = execution	thState := writing
Writing	0	thState=writing	thState := waiting_dispatch $\otimes$ $(\otimes_{odp \in IODP} pw(odp))$
Deactivation	0	(thActive = false)|(thActivate = false$\bigwedge$ thState = waiting_dispatch)	thState := waiting_mode

After the state transition from *undispatched* to *dispatched*, the dispatcher will dispatch the thread and the transition back to *undispatched* state.

The dispatch of the thread is translated into a TASM state machine in this paper. It is a three-tuple $< DisMV, DisCV, DisMM >$ where $disMV = \{thDispatch, nextDispatch, thActive\}$ is the set of monitored variables and $disCV = \{nextDispatch, thActive\}$ is the set of controlled variables. *thDispatch* and *thActive* have been introduced in the definition of execution machine while *nextDispatch* is a Boolean variable representing the state of the dispatcher: if it is true, the machine is in *undispatched* state; otherwise the machine is in the *dispatched* state.

DisMM is the set of transitions of dispatcher: *Nextdispatch*, *Dispatch* and *Waiting*. State transition *Nextdispatch* represents the state transition from *undispatched* to *dispatched* and *Dispatch* is the reverse transition. Since the condition of thread is different between periodic and aperiodic thread, the execution time and guard condition of the transition need to be defined respectively, as shown in Table 2 and Table 3. For periodic thread, the dispatch is triggered by the period of the time so the execution time of the state transition is the period of the thread denoted as *th.period*. For aperiodic thread, however, the execution time of the state transition is 0 and the check on the input event ports is added on the condition of the transition, where *TIEP* is the set of the trigger ports and *pr(iep)* is the reading operation of the event port *iep* to judge whether the event has arrived.

Actions of the rules of *period_nextdispatch* and *aperiodic_nextdispatch* are the same. *Pw(odp)* is the writing operation of the data port *odp* and *th.DODP* is the set of delay output data ports.

When *Dispatch* is enabled, *thDispatch* is set to true, informing the execution machine that the thread has been dispatched. Then the state machine transition back to *undispatched* state (nextDispatch:=true), waiting for the next dispatch.

3.2.3. Translation of behavior annex

Behavior Annex (BA) can be defined as a five-tuple $<S, S_0, V, Guard, Action, T>$ where S is the set of states; S_0 is the initial state; V is the set of local variables; *Guard* is the set of guard conditions of the state transition; *Action* is the set of actions to be executed on the transitions; T is the set of state transition $S \times \{G, A\} \times S$. Fig. 9 shows an example of BA specification with five states (*s0-s4*, line 4-6)and 4 state transitions (*t1-t4*, line 8–12).

First, states in BA will be translated into a user-defined typed environment variable called *thBAstate*, as shown in Table 4. We can see that each state in BA is translated as a value of *thBAstate*. Similarly, for each local variable in *V*, a corresponding environment variable is generated in TASM model.

Each state transition $t \in$ T of BA is translated into a corresponding state transition in thread execution state machine of TASM, denoted as *execution_tr*. According to the definitions of state transitions in BA, execution time (T), guard condition (G) and actions (A) of the state transition in TASM will take the different forms.

(1) Execution time (T)
Execution time of the transition is decided by the actions of the transition, shown as below.
$(Computation(min,\ max)|Delay(min,\ max)|0) \rightsquigarrow T(execution\_tr)$
If actions contain operation *Computation* (*min*, *max*), execution time is between value of min and max of *Computation*. Similarly, if action *Delay* (*min*, *max*) is defined, execution time is between value of min and max of *Delay*. Otherwise, execution time is 0.

Table 2
Translation rules of dispatcher for periodic thread.

State transition	Time (T)	Guard(G)	Action(A)
Nextdispatch	th.period	nextDispatch = true ⋀ thActive = true	nextDispatch:=false ⊗($\otimes_{odp \in th.DODP} pw(odp)$)
Dispatch	0	nextDispatch = false	thDispatch:=true⊗nextDispatch:=true

Table 3
Translation rules of dispatcher for aperiodic thread.

State transition	Time (T)	Guard(G)	Action(A)
Nextdispatch	0	(nextDispatch=true ⋀ theActive=true ⋀($\bigvee_{iep \in TIEP}(pr(iep))$))	nextDispatch:=false ⊗($\otimes_{odp \in th.DODP} pw(odp)$)
Dispatch	0	nextDispatch = false	thDispatch:=true⊗nextDispatch:=true

```
1:thread implementation main_stabilization.impl
2:    annex behavior_specification{**
3:    states:
4:      s0:initial state;
5:      s1,s2,s3:state;
6:      s4:complete state;
7:    transitions:
8:      t1: s0-[]->s1{t_fib1_sync!;t_fib2_sync!;t_fib3_sync!;
9:      t_mech_sync!;}
10:      t2: s1-[]->s2{delay(50ms);}
11:      t3: s2-[from_handler?]->s3{t_mech_serial!;}
12:      t4: s3-[]->s4{}
13:    **}
14:end main_stabilization.impl;
```

Fig. 9. An example of BA.

Table 4
BA states and corresponding TASM environment variable.

BA states	TASM environment variable
S = {s0,s1,s2,. . .sn}	thBAstate={s0,s1,. . .,sn}

(2) Guard condition (G)

According to whether *Delay* is defined in the action of state transition, there are two translation rules, shown as below. Note that *tr.Guard* is the guard condition of the BA state transition and *ss* is the source state of the transition.

$(thState = waiting\_execution \bigwedge thBAstate = ss \bigwedge tr.Guard) | (thState = execution \bigwedge thBAstate = ss) \rightsquigarrow G(execution\_tr)$

If *Delay* is defined, the thread Initiatively blocks itself so there is no need to get CPU resource and the condition is $(thState = waiting\_execution \bigwedge thBAstate = ss \bigwedge tr.Guard)$. Otherwise, the condition is $(thState = execution \bigwedge thBAstate = ss)$. Whether the thread can get CPU and be transitioned into the state execution is judged in the scheduler so the guard condition *tr.Guard* will be used in the scheduler which will be introduced in Section 3.3.

As defined in Section 2, *tr.Guard* takes the form:

$< BExpr > \mid [\text{on } < BExpr > -->] < event > [when < BExpr >]$

It can be a simple boolean expression ($< BExpr >$), the arrival of an event port ($< event >$) or a complicated composition of them containing the check on the input data or event ($[when < BExpr >]$).

BExpr can be translated directly to the logic expression of TASM. *Event* is used to represent that the transition is enabled by the arrival of event. Therefore, the translation rule is defined as $pr(\Gamma(event.p))$ in which *pr* is the port reading operation. *When* is used to represent the judgment on the data newly arrived on the port, translation rule taking the form $when.BExpr(event.p)$ in which $BExpr(event.p)$ means the logic judgment on the data.

(3) Action (A)

There are three actions need to be performed in the state transition: (a) execute actions defined in AADL state transition; (b) perform the state transition; (c) release the CPU resource. As introduced above, for each execution machine, an environment variable *thGetcpu* is defined. To release the CPU resource, *thGetcpu* is set to false. Here are the translation rules for (a) and (b).

(a) Actions in BA include assignments and port writing. Assignment operations defined in BA are directly translated into corresponding assignment statements in TASM. Port writing operations like $p!$ or $p!(x)$ is translated by the rule defined in Section 3.1.

(b) If destination state *ds* is typed *complete* in AADL, it means the thread execution is completed so that the state of the thread is transitioned to state *writing* and the state of BA needs to be transitioned to the initial state. Otherwise, the thread needs to be transitioned to state *waiting_execution* and the target state is assigned as the destination state *ds*. Translation rules are shown in Table 5.

Table 5
Transition actions.

State type	Transition actions
Complete	thBAstate := $S_0 \otimes$ thState = writing
Otherwise	thBAstate:=ds$\otimes$ thState = waiting_execution

Translation of BA is the decomposition and refinement of the state transition *execution* of the execution machine. The environment variables and state transitions generated from BA will be integrated into the corresponding execution machine.

3.3. Translation of scheduler

To schedule the execution of threads bound to the same CPU, we generate a TASM machine called scheduler in which the idea of token ring is used. The process of token passing is shown in Fig. 10. When a thread gets the token, it will check whether CPU is free. As soon as CPU becomes free, CPU resource is distributed to the thread with the highest priority. If the thread cannot be executed and there is a thread with lower priority can be executed, the token is passed to it. Otherwise, the passing of the token is stopped till one of the threads becomes ready to execute.

Each processor component in AADL model is translated into a scheduler state machine which is a three tuple $< SchMV, SchCV, SchMM >$ respectively represents the monitored variables, controlled variables and the main machine. *SchMV* and *SchCV* are defined as below.

$SchMV = \{cpustate, current\_token, \bigcup_{th \in STH} thState(th), \bigcup_{th \in STH} thGetcpu(th), \Gamma(port), \bigcup_{th \in STH} th.ba.V\}$

$SchCV = \{cpustate, current\_token, \bigcup_{th \in STH} thGetcpu(th), \Gamma(port), \bigcup_{th \in STH} th.ba.V\}$

STH is the set of threads to be scheduled. User-defined typed variable *cpustate={free,busy}* is used to represent the current state of CPU. Variable *current_token* is used to represent the thread which has the token currently. $\Gamma(port)$ is the set variables of data ports used to judge if the data dependencies are met. $\bigcup_{th \in STH}(th.ba.V)$ is the set of local variables of specification of BA used in the guard condition of BA.

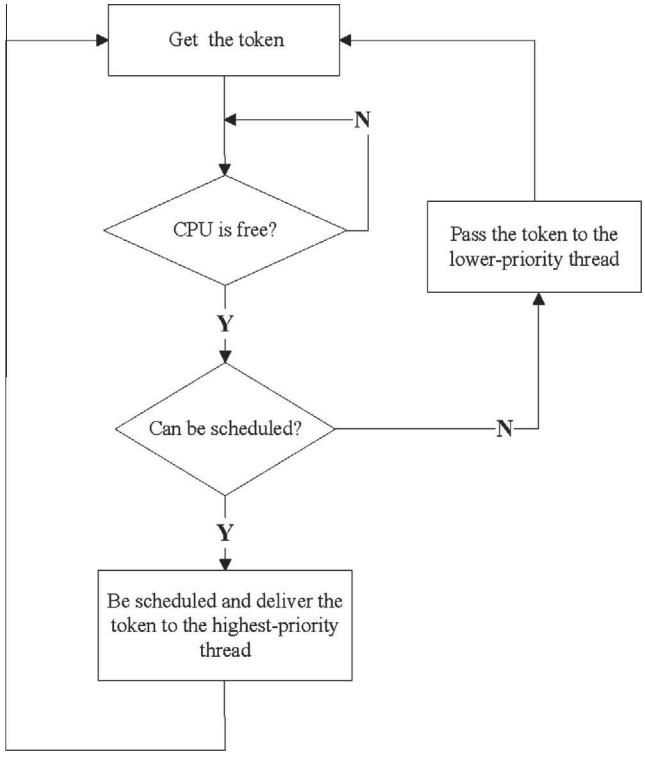

Fig. 10. Process of token passing.

Table 6
Translation rules of scheduler.

State transition	Time (T)	Guard (G)	Action (A)
Schedule(th)	0	cpustate=free$\wedge$ current_token=th $\wedge$ sa(th)	cpustate:=busy$\otimes$ current_token:=next(th)$\otimes$ thGetcpu(th):=true
Pass(th)	0	cpustate=free$\wedge$ current_schedule=th$\wedge$ ($\bigvee_{th\in LTH} sa(th)$)	current_token:=next(th)

SchMM is the set of state transitions. For each thread *th*, there are two corresponding state transitions *Schedule(th)* and *Pass(th)*. The translation rules are shown in Table 6.

State transition *Schedule(th)* is used to describe the behavior when *th* is able to execute. Apart from the condition that CPU resource is free(cpustate = free) and the token belongs to *th*(current_token = th), there are two conditions need to be met: (a) *th* is in *waiting_execution* state; (b) data dependencies are met. Moreover, if there are BA defined in *th*, at least one of the guard conditions of the state transitions in BA needs to be met. The composition of these three conditions, $Sa(th)$, is shown below.

$$(thState(th) = waiting\_execution \wedge (\bigwedge_{idp\in IDP} pr(idp)) \wedge (\bigvee_{th.ba.tr\in th.ba.T} th.ba.tr.G)) \rightsquigarrow sa(th)$$

IDP is the set of input data port. $\bigwedge_{idp\in IDP} pr(idp)$ means all data dependencies need to be met. $th.ba.tr.G$ is the guard condition of specification of BA. $\bigvee_{th.ba.tr\in th.ba.T}$ means at least one guard condition needs to be met.

Actions of the transition is to set current state of CPU to busy, set *thGetcpu(th)* to true and pass the token to the thread with the highest priority next to *th*(next(th)).

If *th* has the token but it can not be executed, state transition *pass* will be performed to pass the token to the thread with the highest priority next to *th*. Note that *LTH* is the set of threads which has the lower priority than the current thread and $\bigvee_{th\in LTH} sa(th)$ means at least one of the threads in the set is able to be scheduled.

3.4. Translation of mode change

Safety-critical systems need to perform different functions in different time or running environments so there are distinguished configurations in such systems. In AADL, *mode* is used to describe the process of the dynamic configurations. It is an explicitly defined configuration of contained components, connections, and property values. Fig. 11 shows an example defined in an AADL *process* component called *dpu_process*. There are two modes, *stabilization* and *maneuver* in the specification. Two threads in *dpu_process*, *sta_main* and *man_main*, respectively belong to mode *stabilization* and *maneuver*. Since multiple modes can be declared in a component, a mode change state machine is needed for the available mode transitions triggered by the arrival of event. In this example, we can see that the mode change state machine has two state transitions: the one from stabilization to maneuver and the reverse one. The trigger event for the former transition is from event port *t_mech_return* of thread *sta_main* and the one for the later transition is from event port *t_mech_return* of thread *man_main*.

Mode change is triggered by the event port in AADL, translating the system from an old mode into a new mode. Tasks in the old modes need to be deleted and deactivated and ones in the new mode need to be activated and dispatched. Mode change protocols are used to deal with the condition *i* when a mode change comes with old tasks still in execution. In synchronous protocols, tasks in the new mode will not be dispatched until all tasks in the old mode have completed, which is not efficient. In asynchronous protocols, there will be a time interval with tasks in both old mode and new mode in execution. This paper will discuss the condition of asynchronous protocols.

There are some tasks called critical tasks in the system. Mode change should not influence on the execution of these tasks. In AADL, threads corresponding to critical tasks are marked with key words *synchronized*. When mode change request comes during the execution of critical tasks in the old mode, tasks need to be completed before the mode change begins. The process for the completion of critical tasks is called *preparation process*. After the preparation period, common tasks(tasks except critical ones) in the old mode will be deleted and deactivated and tasks in the new mode will be activated and dispatched. The process is called *execution process*. When all tasks in the old mode have been deactivated and all tasks in the new mode has been dispatched and executed for the first time, the system is in the new *steady* state. The system is taken over by the new mode and waits for the next mode change request.

Fig. 12 shows the state machine of mode change. Three ellipses are the three processes or state given above. When mode change request comes (Current MCR = Event), the system transitions into the *preparation process*. The interval on the preparation process is denoted as *syncout (C,old,new)*. It is the hyper period of all critical tasks (hyper period is the LCM of periods of these tasks). After the preparation, system clock is set to zero (C:=0) and the system transitions into the *execution process*. The interval on the execution process is denoted as *syncin (C,old,new)*. It is the hyper period of tasks in the old mode. After this process, the system is in the new mode (currentmode = new) and transitions into the *steady* state.

A mode change state machine is translated into a state machine which is a three tuple $\{SomMV, SomCV, SomMM\}$. *SomMV* and

```
process implementation dpu_process.impl
    subcomponents:
      ...
      sta_main: thread main_stabilization.impl in modes
      (stabilization);
      man_main: thread main_maneuver in modes (maneuver);
      ...
    modes:
      stabilization: initial mode;
      maneuver: mode;
      stabilization -[sta_main.t_mech_return]-> maneuver;
      maneuver -[man_man.t_mech_return]-> stabilization;
end dpu_process;
```

Fig. 11. An example of mode in the process definition.

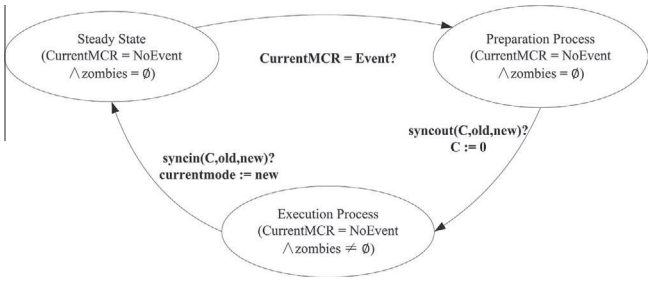

Fig. 12. Mode change state machine.

SomCV are respectively the monitored and controlled variables and *SomMM* is the set of state transitions. The definition of *SomMV* and *SomCV* are shown below.

$somMV = \{currentMState,\ nextHyperperiod,\ event\}$
$somCV = \{currentMState,\ nextHyperperiod,\ event\ , \bigcup_{th\in TH} hyperperiod \bigcup_{th\in TH} th\ Active, \bigcup_{th\in TH} numPeriod, \bigcup_{th\in TH} syncDis\}$

Variable *currentMState*, typed as user-defined type, is used to represent the current state of mode. Variable *nextHyperperiod*, typed Boolean, is used to control if the mode change automata can accept the request of mode change. Variable *event* is used to trigger mode change. For every thread $th \in TH$ (TH is the set of thread defined in the process component), there are corresponding variables *hyperPeriod*, *numPeriod*, *thActive* and *syncDis* to be generated. Variable *hyperPeriod* is the value of hyper period of current mode (product of period value of all threads in this mode) divided by the period of thread *th*. Variable *numPeriod* is the value representing the number that thread *th* has been dispatched in the current mode. Every time the thread enters a new hyper period, *numPeriod* is assigned to 0 and every time the thread is dispatched, *numPeriod* increases by one. Variable *syncDis* is used to synchronize between the main machine of mode change and the main machine of dispatcher of *th*. When a hyper period completes, main machine of mode change needs to notice the dispatcher whether the thread can enter into next hyper period. Variable *thActive* is used to represent whether *th* is activated in the current mode.

SomMV is the set of state transitions generated. For every mode change transition $mtr =< sms, event, dms >$, four state transitions in TASM is generated, shown as below.

(1) State transition *Hyperperiod* is used to represent the hyper period of source mode *sms*, execution time is $h_{in}(sms, dms)$, the hyper period of the synchronized threads.
$h_{in}(sms, dms) \rightsquigarrow T(Hyperperiod)$
$(currentMState = sms \bigwedge$
$nextHyperperiod = true \rightsquigarrow G(Hyperperiod)$
$nextHyperperiod := false \rightsquigarrow A(Hyperperiod)$

(2) State transition *Has_mcr* is executed if the mode change event arrives at the synchronization point of hyper period, shown as below. Variables *thActive* of threads which are not in the new mode are set to false while ones in the new mode are set to true. Variables *HyperPeroid* are also needed to be recalculated and variable *currentMState* is set to *sms_improgress* to represent the progress of mode change. *OTH* is the set of threads in old mode and *NTH* is the set of threads in new mode.
$0 \rightsquigarrow T(Has\_mcr)$
$currentMState =$
$sms \bigwedge nextHyperid = false \bigwedge event = true \rightsquigarrow G(Has\_mcr)$
$(event := false \otimes nextHyperperiod := true \otimes$
$currentMState := sms\_inprogress \otimes (\otimes_{th\in OTH}$
$thActive(th) := false) \otimes$
$(\otimes_{th\in NTH} thActive(th) := true)$
$\otimes(\otimes_{th\in OTH} \bigwedge_{th\in NTH} hyperperiod(th) :=$
$h_{out}(sms, dms)/th.period) \rightsquigarrow A(Has\_mcr)$

(3) Transition *Hasnot_mcr* is executed if no mode change event arrives at the synchronization point of hyper period, shown as below. Variables *nextHyperperiod* are set to true to notice the dispatcher to enter the next cycle of hyper period.
$0 \rightsquigarrow T(Hasnot\_mcr)$
$currentMState = sms \bigwedge$
$nextHyperperiod = false \bigwedge event = false \rightsquigarrow$
$G(Hasnot\_mcr)$
$nextHyperperiod := true \otimes$
$(\otimes_{th\in OTH} syncDis := true) \rightsquigarrow A(Hasnot\_mcr)$

(4) State transition *Inprogress* is used to represent the progress of mode change. Execution time of the transition is the hyper period of threads in new mode. Action of the transition are the assignment of *curretMState* to new mode *dms* and notice the dispatcher of threads in new mode to enter the next cycle of the hyper period.

$h_{out}(sms, dms) \rightsquigarrow T(Inprogress)$
$currentMState = sms\_inprogress \rightsquigarrow G(Inprogress)$
$currentMState := dms \otimes$
$(\otimes_{th\in NTH} syncDis(th) := true)) \rightsquigarrow A(Inprogress)$

3.5. Generation of observer state machine

Observer state machine is usually used to monitor the state of other machines. To verify the timing properties of AADL model, for each thread, an observer state machine is generated to monitor if the thread execution exceeds its deadline, shown in Fig. 13. After

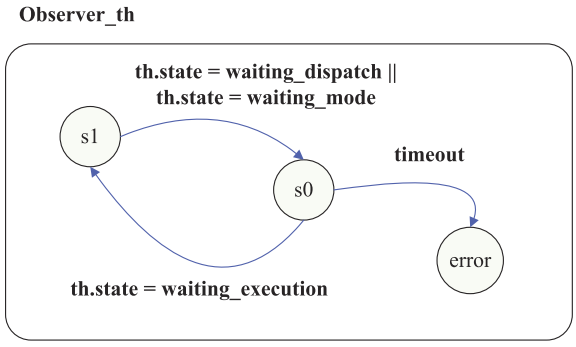

Fig. 13. Structure of AADL2TASM model transformation tool.

Table 7
Translation rules of observer.

State transition	Time (T)	Guard (G)	Action (A)
Initialization	deadline	thState=waiting_execution ∧ obState=s0	obState=s1
End	0	(thState=waiting_dispatch∨ thState=waiting_mode) ∧obState=s1	obState=s0
Timeout	0	(thState=execution∨thState =writing)∧obState=s0	obState=error

the thread has been dispatched and entered into *waiting_execution* state, observer will transit from *s0* to *s1*. If the execution time exceeds the deadline, observer will transit to *error* state. Otherwise, it will transit back to *s0*, beginning the next observation.

The observer state machine is a three tuple $< dlMV, dlCV, dlMM >$ where *dlMV* and *dlCV* are respectively monitored and controlled variables and *dlMM* is the set of state transitions. The definition of *dlMV* and *dlCV* are shown below. *ThState* represents the state of thread and *obState* represents the state of the observer state machine.

$$dlMV = \{thState, obState\}$$
$$dlCV = \{thState, obState\}$$

DlMM contains four state transitions: *Initialization*, *End*, *Timeout* and *Waiting*. The translation rules are shown in Table 7. *Initialization* is the transition from *s0* to *s1*; *End* is the transition from *s1* to *s0* and *Timeout* is the transition from *s0* to *error*. It is obvious that if the execution time exceeds the deadline, transition *Timeout* will be performed to alarm that the thread does not meet the timing properties.

4. Implementation of transformation tool and case study

4.1. Implementation of AADL2TASM transformation tool

Based on the translation rules defined in Section 3, we implemented AADL2TASM transformation tool which can be integrated into OSATE environment. The translational rules are defined using ATL. ATL is a model transformation language used in MDD (Model Driven Development). For more detailed information of ATL, the reader is referred to [29–31].

The tool architecture is shown in Fig. 14. The input of the tool is the standard AADL model and the output is the standard TASM model. After the pre-processing, AADL model will be modified to fit in the ATL transformation. The metamodels of AADL (denoted as *M* in Fig. 14) and TASM (denoted as M' in Fig. 14) and ATL transformation (denoted as $A \rightarrow A'$ in Fig. 14) are needed for the transformation. Metamodel describes the abstract syntax of language while the ATL transformation defines the mapping rules between the source model and the target model. The concrete model "conforms to" the meta model. The metamodels are implemented using KM3 (Kernel MetaMetaModel) language [32], which are then translated into *ecore* format. The ATL transformation rule is used to implement the mapping between two models. ATL Engine will do the transformation and output the corresponding TASM model. After the post-processing, the standard TASM model will be obtained for further verification and analysis.

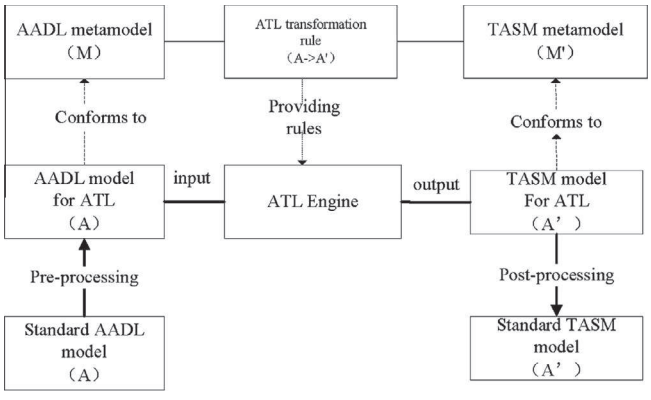

Fig. 14. Structure of AADL2TASM model transformation tool.

4.2. DPU: a case study

This paper uses a subsystem called DPU (Data Processing Unit) from a satellite to illustrate the transformation and verification of AADL model. Functional properties such as state reachability and non-functional properties such as timing correctness and resource correctness will be verified by model transformation. Fig. 15 shows the AADL model of DPU.

The function of DPU is to collect the data from one Gyro and five FOGs and then send it to the main computer. There are two modes, *stabilization* and *maneuver*, in this system. In stabilization mode, DPU will execute the following actions for every 150 ms:

- Send synchronous interruption signals to three FOGs (FOG 1 to FOG 3) to notice them preparing for the data; then send the same signal to the Gyro and the data from Gyro will be collected to DPU, denoted as *N3*.
- 50 ms after, send asynchronous interruption to the Gyro and get the data, denoted as *N4*; then replace *N1* (*N3* collected from the previous cycle) with *N3*, *N2* (*N4* collected from the previous cycle) with *N4*.
- Collect data from three FOGs and Gyro and then send them to the main computer (not included in the model).

In *maneuver* mode, the only difference is that DPU will collect data from FOG 4 and FOG 5 instead of FOG 1 to FOG 3. Data from Gyro will be collected in both modes. Each time execution in one mode has completed, the model will switch to another model. In AADL model, DPU is represented by system component. There are 6 device components representing one Gyro (*mech_gyroscope*) and five FOGs (*fib_gyroscope1* to *fib_gryoscope5*). Component *Intel* and *mem* respectively represent the CPU and memory of the system. Two bus components (*bus* and *lan*) are used to connect devices, CPU and memory. The core part of the model is the *process* component containing 11 threads (represented by dotted quads), three data components *N1*, *N2* and *buf* and mode change between *stabilization* and *maneuver*. Behavior annex specifications are defined in threads to describe detailed actions to be taken during the execution. Table 8 lists properties of threads.

Fig. 16 shows the behavior annex specification defined in the thread *main_stabilization*, of which the detailed specification has been defined in Fig. 9. The *period* and *deadline* of the thread is both 150 ms. 5 output event ports (*t_fib1_sync - t_fib3_sync, t_mech_sync and t_mech_serial*) are used to send the command of the data collection; input event data port *from_handler* receives the packed data and output event data port *write_mc* is used to send the packed data to the main computer.

4.3. Verification by model transformation

4.3.1. Simulation and analysis by TASM Toolset

Using the AADL2TASM tool, the AADL model is translated into the corresponding TASM model. For each thread, there are 2 machines: dispatcher and execution machines. To analyze the timing properties, an observer state machine is also generated for each thread. Furthermore, there are 3 main machines respectively for mode change, scheduling and port communication. The behaviors and resource usage will be simulated in the TASM toolset.

There are five perspectives in TASM toolset: *Environment*, *Machines*, *Time*, *Resource* and *Update Set*. *Environment* perspective shows the value change of environment variables during each step of the execution; *Machines* perspective illustrates the information respect to the machines such as time elapsed in the execution, the rule executed in the last step and how many steps the machine has been executed; *Time* perspective shows the timing among machines; Resource perspective shows the usage of the resource

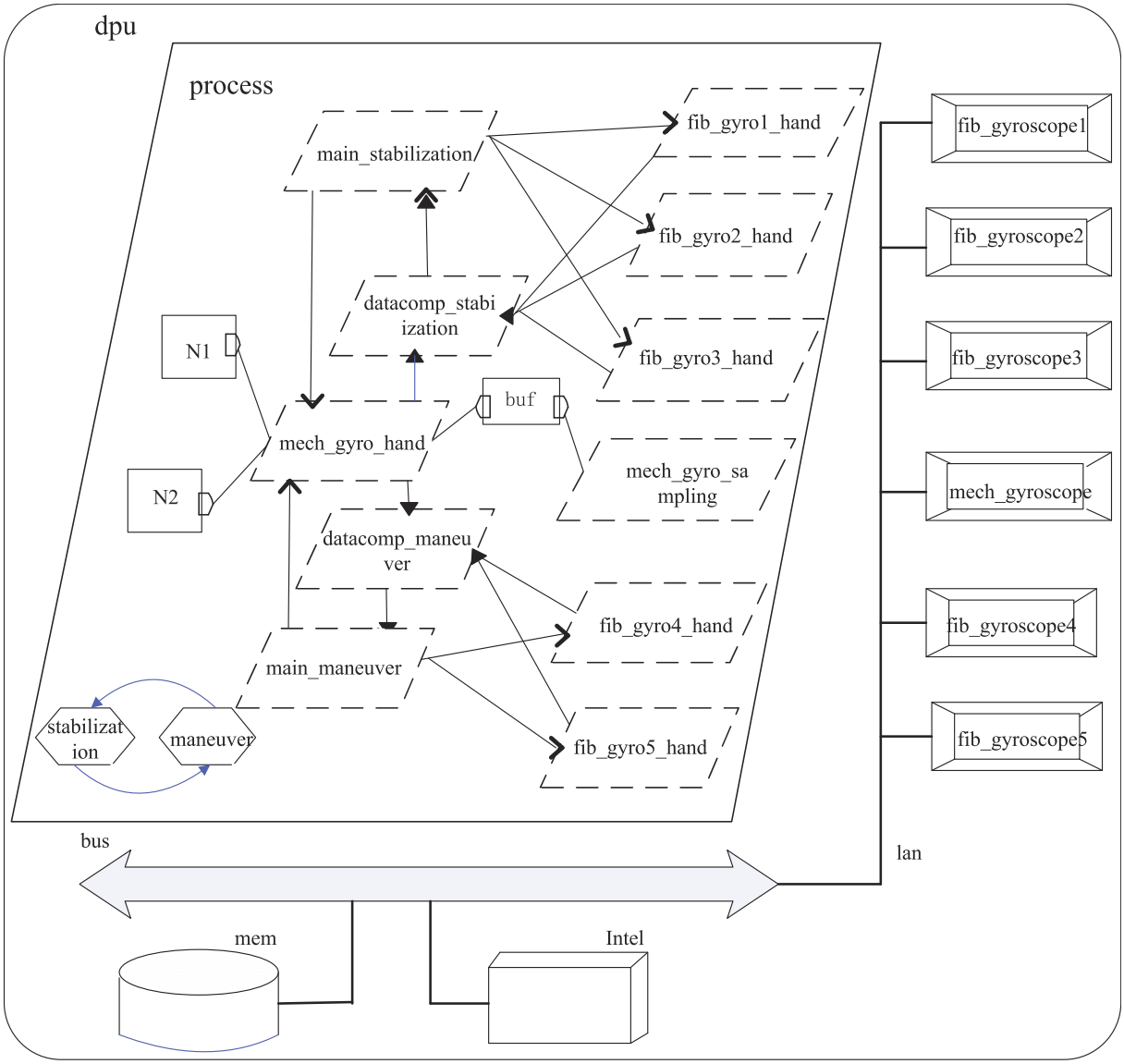

Fig. 15. An example of AADL model.

Table 8
Properties of threads.

Thread	Dispatching	Period	Deadline	Execution time	Mode
Main_stabilization	Periodic	150	150	Behavior annex	Stabilization
Datacomp_stabilization	Periodic	150	150	Behavior annex	Stabilization
Main_maneuver	Periodic	150	150	Behavior annex	Maneuver
Datacomp_maneuver	Periodic	150	150	Behavior annex	Maneuver
Mech_gyro_sampling	Periodic	20	20	5	Both
Mech_gyro_hand	Speriodic	x	80	Behavior annex	Both
Fib_gyro_hand (1–3)	Periodic	150	150	Behavior annex	Stabilization
Fib_gyro_hand (4–5)	Periodic	150	150	Behavior annex	Maneuver

defined in the model; *Update Set* perspective shows the state transitions of the model. In this paper, we define that when the thread is being executed, CPU usage is 100%. By observing the Resource perspective, CPU usage can be counted. Note that although we consider only CPU usage, all resource declarations in AADL can be transformed into TASM resources in a similar way.

4.3.2. Model Checking By UPPAAL

Apart from simulation, model checking is a major method for the verification and validation of the AADL model. Ouimet [22] proposes a methodology translating TASM into UPPAAL timed automata [33], based on which a model transformation tool TASM2UPPAAL has been implemented [34].

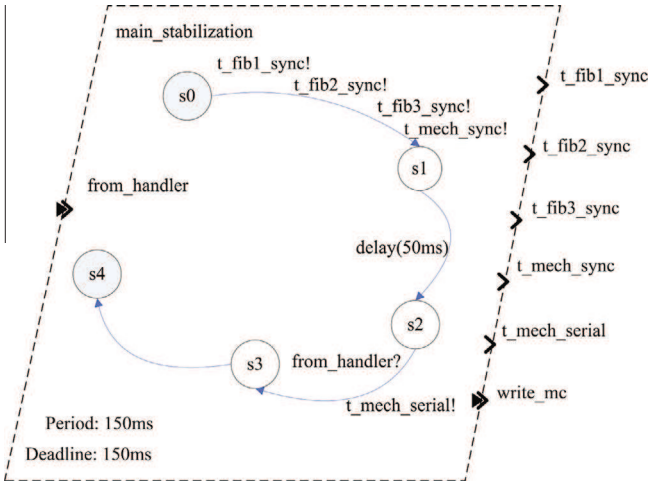

Fig. 16. Behavior annex of thread main_stabilization.

UPPAAL uses a subset of CTL (Computation Tree Logic) to describe properties to be verified. Here we take following properties as examples to show the verification of AADL model by model checking.

4.3.2.1. Deadlock freeness. Deadlock freeness is fundamental property related to the function of a reactive system. If a system is deadlocked, it will not be able to interact with the environment, which can probably bring about serious system failures. In UPPAAL, deadlock freeness is described as the following CTL formula.

$A[]\ !deadlock$

For instance, if we delete the port connection between threads *mech_gyro _hand* and *data_comp_stabilization*, *data_comp_stabilization* can not pack data from three FOGs and Gyro such that thread main_stabilization will not be able to continue the execution. As a result, the system will turn into deadlock status.

4.3.2.2. State reachability. State reachability is a kind of functional property checking if there is dead state in the model. several properties can be described as state reachability. For instance, for the behavior annex shown in Fig. 16, the formula to check if state *s3* is reachable is shown below.

$thState\_main\_sta == 4 -- > thBAState\_main\_sta == 3$

ThState_main_sta represents the execution state whereas *thBAState_main_sta* depicts the state of behavior annex specification. The formula means that if the thread has begun executing, state machine of behavior annex will finally arrive at state *s3*. If *from_handler* cannot received the signal, the transition from *s2* to *s3* cannot be performed and the property will not be satisfied.

Schedulability can also be represented in a similar way. The formula shown below means that once *main_stabilization* has been dispatched ($thState\_main\_sta == 2$), it will finally come into execution ($thState\_main\_sta == 4$).

$thState\_main\_sta == 2 -- > thState\_main\_sta == 4$

Another property able to be represented as state reachability is the mode reachability. For instance, to verify if mode *maneuver* can be reached, the following formula will be checked in UPPAAL.

$A <> currentmode == 1$

4.3.2.3. Timing correctness. This paper uses observer state machine to monitor whether the thread execution time exceeds the deadline. For instance, if the process time of asynchronous interruption of *mech_gyro_hand* increases from 15 ms to 16 ms, since dispatch period and deadline of *mech_gyro_sampling* is 20 ms and the execution time is 5 ms, it will exceed the deadline at certain periods (16 ms + 5 ms > 20 ms) and the property formulated as the following formula will not be satisfied.

$A[]\ !(obState\_mech\_gyro\_sampling == 2)$

4.3.3. Performance Evaluations and Discussion on the practicality

The implemented verification toolchain "AADL-TASM-UPPAAL" has been successfully put into practical use to verify the DPU system. DPU system is a typical real-time system with 11 threads and 2 modes. Behavior annex specifications are also defined to describe the detailed behaviors and port communications are used for the interactions among threads. After translating into TASM models, there are 38 state machines (with observer state machines). The verification was performed on a basic development PC with 2 cores and 4GB RAM memory. Properties mentioned above have been verified. When the model has been injected with errors, it takes less than 1 minute to get the result. If the model is correct, however, much more time is needed. In addition, to verify the model is not dead-locked, state machines such as observers have must be deleted to reduce the state space. It has to be admitted that state explosion [35] is an obstacle for the verification of larger system. According to the verification test, if AADL model has more than 5 threads, the model with observer state machines will be hard to verify by model checking. If observer state machines are deleted from the model, system with 11 threads can be verified. Therefore, it is necessary to model a system in a modular way: components are modeled and verified first and then the whole system can be verified with fewer details.

5. Related work

A considerable number of works have been proposed respect to AADL transformation and formal analysis of AADL models. This section provides a brief introduction to these works.

In order to analyze timing properties of safety-critical systems, [36] chooses AADL as the modeling language and proposes an extension of behavior annex with time annotation to specify the time properties. Then a transformation from extended behavior annex to TASM is presented to facilitate the timing analysis with timed simulation or timed model checking. However, one can observe from the case study that the system is modeled at a high level: each behavior annex specification describes a subsystem and how to deal with the communication among subsystems is not described clearly.

Verimag research lab in France presents model transformation methodology from AADL to BIP (Behavior Interaction Priority) [6,37]. BIP is a modeling language for real-time system that uses automata to describe behavior and supports combination between heterogeneous components. The basic modeling element of BIP is *Atomic Component* and *Compound Component*. In the transformation, each component in AADL corresponds to a BIP component. The input and output ports of AADL components correspond to ports of BIP components and behaviors of AADL component are described as state transitions in the BIP component. Moreover, all transformation rules are depicted in natural language and shown as graphs in the paper. Compared with work presented in this paper, mode change, communication among threads are not supported.

Table 9
Summary of AADL transformations.

Target formalism	Supported subset	Transformation objectives
TASM	Behavior annex, port communications	Timing properties
BIP	Process, thread, processor,	Timing properties
	Behavior annex, sub program, port connections	Deadlock
ACSR	Thread execution, communication	Schedulability
IF	Thread execution,	Deadlock
	Communication	Buffer overflow
Fiacre	Thread, communication	Deadlock, livness, time-bounded verification
TLA+	Port communications	Not specified
	Preemptive scheduling, mode	
RT-Maude	Thread, port communication, mode change, behavior annex	Reachability,time-bounded verification
VTS	Not specified	Behavior properties
GSPN	Error model annex	Dependability-related properties
Petri net	Thread	Deadlock, livelock
UPPAAL	Thread dispatching, scheduling, port communication	Control-flow, data-flow reachability
Lustre	Processor, thread, port communication, resource sharing	Safety
SIGNAL	Thread, scheduling, behavior annex	Safety verification, architecture exploring
EDA (COMPASS)	SLIM (AADL dialect enriched by error model annex)	Requirement validation, functional properties, safety, dependability and performance analysis, FDIR analysis, diagnosability
UML Marte	Dispatch protocols, port communication, end to end flow specification	Timing properties, flow analysis

University of Pennsylvania presents model transformation methodology from AADL to ACSR [7] that supports the transformation of static structure, execution and communication of threads. ACSR can describe competition for resources to support the schedulability analysis. However, the transformation methodology presented does not support the complete transformation of mode change and behavior annex.

Thomas Abdoul et al. present model transformation methodology from AADL to IF [8] to fulfill the formal verification like safety properties. IF is a description language for real-time system presented by Verimag research laboratory. Translation rules from static structure, thread execution, behavior annex (not standard version), communication, scheduling and other elements of AADL to IF are defined but transformation of mode change is not supported.

Project TOPCASED of Airbus presents the transformation from AADL to Fiacre [9], an intermediate language for model verification. The transformation omits the hierarchical information of AADL syntax and concentrates on the thread execution and communication. Every thread in AADL is mapped to a corresponding process in Fiacre and a process called *Glue* is defined in Fiacre to manage the dispatch and scheduling of threads and communications among threads. The translated Fiacre model will be then compiled into a format for *Tina* [38] tool to verify the system using LTL.

IRIT in France chooses TLA+ [10] as the target language of AADL model transformation. Transformation of port communication, component communication based on shared variables, preemptive scheduling strategy and mode change are studied while transformation of behavior annex is not supported.

Ölveczky et al. [11] presents a methodology transforming AADL to RT-Maude. RT-Maude is a modeling language for embedded real-time system based on *Rewriting Logic*. The paper proposes the translation rules for thread component, port connection, mode change, dispatch protocol and behavior annex. Generated model can be simulated or verified using the tool *AADL2Maude*. However, the scheduling and resource sharing are not supported.

Several studies use Petri nets to verify the AADL model. Monteverde et al. [12] uses VTS (Visual Timed Scenarios) to specify the behavioral properties of AADL models. The transformation from VTS to Timed Petri Nets (TPNs) is defined to verify the mode change behaviors and flow specifications of AADL models. However, how to combine VTS with AADL is not clearly stated in the paper. Rugina et al. [13] translates *AADL Error Model Annex* into GSPNs (Generalized Stochastic Petri Nets) to verify dependability-related properties such as reliability and availability by ADAPT tool [39]. To support the qualitative analysis of distributed systems specified by AADL, [14] proposes a methodology translating AADL thread automata to symmetric net of Petri net. Properties such as deadlock-freeness and livelock-freeness can be then verified.

Johnsen et al. [15] proposes methodology translating AADL to UPPAAL timed automata. By using *semantic anchoring*, semantics of a subset of AADL is anchored to semantics of UPPAAL timed automata. Thread dispatching, fixed-priority preemptive scheduling, port communication are supported. By using the verification technique proposed in [40], control-flow reachability, data-flow reachability and other properties can be verified by the UPPAAL tool.

Jahier et al. [16] studies the transformation from AADL to synchronous language Lustre. In synchronous model, all internal components steps forward in a "simultaneous" way. Consequently, the model is deterministic. In contrast, AADL is a modeling language for realistic platform so that the AADL model is asynchronous. The paper first presents methods to describe asynchronous model in Lustre. Then the translation of processor, scheduling, thread dispatch and execution, port connection and resource sharing are given. The generated Lustre model will be verified using Lesar. However, the methodology do not support the transformation of mode change and behavior annex.

Yu et al. [17] and Ma et al. [41] propose a concept of co-design for safety critical system in which AADL is used to model the architecture and Simulink is used to model the behavior of threads. Both models of AADL and Simulink are then translated into a so called SME [42] model based on synchronous language SIGNAL [43]. Thread, port communication, scheduling and binding of AADL are translated into SIGNAL specification. In [18], translation from AADL behavior annex to synchronous equations via an intermediate formalism SSA (Static Single Assignment). In addition, [44] uses SynDEx [45] to explore the architecture of AADL model.

Bozzano et al. [19,46] propose an integrated toolset COMPASS to verify functional correctness, safety, dependability, performance, diagnosability and other properties of AADL specification.

An extension of AADL called SLIM (System Level Integrated Modeling) and the transformation to EDAs (Event-Data Automatons) are presented. Then networks of EDAs will be verified using mature tools by which formal analyses such as model checking, safety and dependability analysis can be performed .

André et al. [20] and Lee et al. [47] proposes transformation from AADL to UML Marte, a profile extension for real-time and embedded system. The transformation concentrates on port communication and end-to-end flow specification.

Table 9 illustrates a summary on current studies of AADL transformations.

Apart from verification based on model transformation, several studies also concentrate on the simulation of AADL models. Varona-Gomez and Villar [48] proposes a tool AADS to simulate and analyze AADL model. Software and hardware components are translated into model of SystemC, a C++ based modeling platform. HW/SW partition and behavior of hardware components will be analyzed by SCoPE [49], a simulation and analysis tool. However, AADS tool do not support the verification of behavior annex and AADL v2.0. LISYC Team from Brest University developed Cheddar [50] for scheduling analysis of AADL model. Number of classical scheduling protocols are supported by Cheddar such as RM, EDF, DM and LLF. In addition, resource consumption [51] and performance [52] can also be analyzed by Cheddar. Scheduling with mode change and behavior annex, however, have not been supported yet. Other simulation tools include Furness[1] and ADeS[2] and so on. Senn et al. [53] proposes a method estimating the power consumption of AADL model.

Compared with current studies, the methodology proposed in this paper have the following features:

(1) A proper subset of AADL has been chosen as the transformation target including thread components (dispatching, offline scheduling and execution), port communication, behavior annex and mode change, which is usually used in safety-critical systems.
(2) TASM is chosen as the transformation target. TASM can describe the multi-capacity resources and the TASM toolset can support the quantitative analysis of resource usage.
(3) Translational rules were defined in a good form and then it is easier to verify the correctness of the transformation (presented in [26]).

6. Conclusions and future work

This paper describes the AADL model transformation based on TASM. We first present formal description of key modeling elements of AADL including thread component, port communication, behavior annex, mode change. Then translation rules from these AADL modeling elements to TASM are formally defined. Finally, we implement model transformation tool, AADL2TASM, using ATL, through which the properties of AADL can be analyzed by the toolset developed for TASM.

In our future work, we will study the translation rules for more elements of AADL, such as data access, subprogram and so on. At the same time, we will improve the implementation of modeling transformation tool to support analysis and formal verification of AADL model. In addition, the use of Multi-core architecture in safety critical systems will become a trend. Based on the study we are conducting in multi-threaded code generation of SIGNAL [54], we will study the extension of AADL with multi-core and how to verify multi-core AADL model with transformation methodology.

[1] http://www.furnesstoolset.com.
[2] http://www.axlog.fr/aadl/ades_en.html.

Acknowledgments

This work was supported by National Natural Science Foundations of China (No. 61073013), State Key Laboratory of Software Development Environment (No. SKLSDE-2014ZX-09) and Aviation Science Foundation of China (No. 2012ZC51025).

References

[1] SAE, Architecture Analysis & Design Language (AADL) v2, AS-5506, SAE International (2009).
[2] SAE, SAE AS5506 Annex: Behavior Specification v2.0 (2011).
[3] J. BODEVEIX, M. FLLALI, et al., The AADL behavior annex-experiments and roadmap, in: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, IEEE Computer Society, Washington, DC, 2007, pp. 377–382.
[4] P. Feiler, Open source AADL tool environment (OSATE), in: AADL Workshop, Paris, 2004.
[5] P. Farail, P. GOUTILLET, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, M. Pantel, The TOPCASED project: a toolkit in open source for critical aeronautic systems design, Ingenieurs de l'Automobile (2006) 54–59.
[6] M.Y. Chkouri, A. Robert, M. Bozga, J. Sifakis, Translating AADL into BIP-application to the verification of real-time systems, in: Models in Software Engineering, Springer, 2009, pp. 5–19.
[7] O. Sokolsky, I. Lee, D. Clarke, Schedulability analysis of AADL models, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 8–pp.
[8] T. Abdoul, J. Champeau, P. Dhaussy, P.-Y. Pillain, J. Roger, AADL execution semantics transformation for formal verification, in: 13th IEEE International Conference on Engineering of Complex Computer Systems, 2008. ICECCS 2008, IEEE, 2008, pp. 263–268.
[9] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, F. Vernadat, Formal verification of AADL specifications in the topcased environment, in: Reliable Software Technologies–Ada-Europe 2009, Springer, 2009, pp. 207–221.
[10] R. Jean-Francois, B. Jean-Paul, F. Mamoun, C. David, T. Dave, Modes in asynchronous systems, in: 13th IEEE International Conference on Engineering of Complex Computer Systems, 2008. ICECCS 2008, IEEE, 2008, pp. 282–287.
[11] P.C. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in real-time Maude, in: Formal Techniques for Distributed Systems, Springer, 2010, pp. 47–62.
[12] D. Monteverde, A. Olivero, S. Yovine, V. Braberman, VTS-based specification and verification of behavioral properties of AADL models, in: International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES08), Springer-Verlag, 2008.
[13] A.-E. Rugina, K. Kanoun, M. Kaâniche, A system dependability modeling framework using AADL and GSPNs, in: Architecting Dependable Systems IV, Springer, 2007, pp. 14–38.
[14] X. Renault, F. Kordon, J. Hugues, From AADL architectural models to petri nets: Checking model viability, in: IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09, IEEE, 2009, pp. 313–320.
[15] A. Johnsen, K. Lundqvist, P. Pettersson, O. Jaradat, Automated verification of AADL-specifications using UPPAAL, in: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), IEEE, 2012, pp. 130–138.
[16] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens, Virtual execution of AADL models via a translation into synchronous programs, in: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, ACM, 2007, pp. 134–143.
[17] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P.L. Guernic, A. Toom, O. Laurent, System-level co-simulation of integrated avionics using polychrony, in: Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, 2011, pp. 354–359.
[18] Y. Ma, J. Talpin, T. Gautier, Interpretation of AADL behavior annex into synchronous formalism using ssa, in: 2010 IEEE 10th International Conference on Computer and Information Technology (CIT), IEEE, 2010, pp. 2361–2366.
[19] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, X. Olive, Formal verification and validation of AADL models, in: Proc. ERTS (2010).
[20] C. André, F. Mallet, R. De Simone, et al., Modeling of immediate vs. delayed data communications: from AADL to UML MARTE, in: ECSI Forum on Specification & Design Languages (FDL), 2007, pp. 249–254.
[21] M. Ouimet, K. Lundqvist, The TASM toolset: specification, simulation, and formal verification of real-time systems, in: Computer Aided Verification, Springer, 2007, pp. 126–130.
[22] M. Ouimet, A Formal Framework for Specification-based Embedded Real-time System Engineering, Ph.D. Thesis, Massachusetts Institute of Technology, 2008.

[23] Z. Yang, K. Hu, D. Ma, L. Pi, Towards a formal semantics for the AADL behavior annex, in: Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09., IEEE, 2009, pp. 1166–1171.
[24] Z. Yang, K. Hu, D. Ma, L. Pi, J.-P. Bodeveix, Formal semantics and verification of AADL modes in Timed Abstract State Machine, in: 2010 IEEE International Conference on Progress in Informatics and Computing (PIC), volume 2, IEEE, 2010, pp. 1098–1103.
[25] Z. Yang, K. Hu, J.-P. Bodeveix, L. Pi, D. Ma, J. Talpin, Two formal semantics of a subset of the AADL, in: 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2011, IEEE, 2011, pp. 344–349.
[26] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, J.-P. Talpin, From AADL to timed abstract state machines: a verified model transformation, J. Syst. Softw. 93 (2014) 42–68.
[27] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Systems and Software Verification: Model-checking Techniques and Tools, Springer Publishing Company, Incorporated, 2010.
[28] E. Börger, The origins and the development of the ASM method for high level system design and analysis, J. Univ. Comput. Sci. 8 (2002) 2–74.
[29] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, Sci. Comput. Program. 72 (2008) 31–39.
[30] A. Group, et al., ATL User Manual Version 0.7, LINA & INRIA (2006).
[31] F. Jouault, I. Kurtev, Transforming models with ATL, in: Satellite Events at the MoDELS 2005 Conference, Springer, 2006, pp. 128–138.
[32] F. Jouault, J. Bézivin, KM3: a DSL for metamodel specification, in: Formal Methods for Open Object-Based Distributed Systems, Springer, 2006, pp. 171–185.
[33] K.G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, Int. J. Softw. Tools Technol. Transfer (STTT) 1 (1997) 134–152.
[34] H. Kai, T. Zhang, Y. Zhi-bin, G. Bin, S. Jiang, P.-C. Jiang, Formal verification of TASM models by translating into UPPAAL, J. Donghua Univ. 29 (2012).
[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Progress on the state explosion problem in model checking, in: Informatics, Springer, 2001, pp. 176–194.
[36] S. Björnander, L. Grunske, K. Lundqvist, Timed simulation of extended AADL-based architecture specifications with timed abstract state machines, in: Architectures for Adaptive Software Systems, Springer, 2009, pp. 101–115.
[37] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006. SEFM 2006, IEEE, 2006, pp. 3–12.
[38] B. Berthomieu, P.-O. Ribet, F. Vernadat, The tool TINA-construction of abstract state spaces for petri nets and time petri nets, Int. J. Product. Res. 42 (2004) 2741–2756.
[39] A.-E. Rugina, K. Kanoun, M. Kaâniche, The ADAPT tool: From AADL architectural models to stochastic petri nets through model transformation, in: Dependable Computing Conference, 2008. EDCC 2008. Seventh European, IEEE, 2008, pp. 85–90.
[40] A. Johnsen, P. Pettersson, K. Lundqvist, An architecture-based verification technique for AADL specifications, in: Software Architecture, Springer, 2011, pp. 105–113.
[41] Y. Ma, H. Yu, T. Gautier, J. Talpin, L. Besnard, P. Le Guernic, System synthesis from AADL using polychrony, in: Electronic System Level Synthesis Conference (ESLsyn), 2011, IEEE, 2011, pp. 1–6.
[42] C. Brunette, J.-P. Talpin, A. Gamatié, T. Gautier, A metamodel for the design of polychronous systems, J. Logic Algebraic Program. 78 (2009) 233–259.
[43] P. LeGuernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming real-time applications with SIGNAL, Proc. IEEE 79 (1991) 1321–1336.
[44] H. Yu, Y. Ma, T. Gautier, L. Besnard, J.-P. Talpin, P. Le Guernic, Y. Sorel, Exploring system architectures in AADL via polychrony and SynDEx, Front. Comput. Sci. 7 (2013) 627–649.
[45] C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine, The SynDEx software environment for real-time distributed systems design and implementation, in: European Control Conference, volume 2, Citeseer, 1991, pp. 1684–1689.
[46] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri, Safety, dependability and performance analysis of extended AADL models, Comput. J. 54 (2011) 754–775.
[47] S.-Y. Lee, F. Mallet, R. De Simone, et al., Dealing with AADL end-to-end flow latency with UML MARTE (2008).
[48] R. Varona-Gomez, E. Villar, AADL simulation and performance analysis in SystemC, in: 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009, IEEE, 2009, pp. 323–328.
[49] H. Posadas, D. Quijano, E. Villar, M. Martínez, SCope: SoC co-simulation and performance estimation in SystemC, in: IEEE/ACM Design, Automation and Test in Europe, 2007.
[50] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Cheddar: a flexible real time scheduling framework, in: ACM SIGAda Ada Letters, 24, ACM, 2004, pp. 1–8.
[51] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Scheduling and memory requirements analysis with AADL, ACM SIGAda Ada Letters, 25, ACM, 2005, pp. 1–10.
[52] F. Singhoff, A. Plantec, P. Dissaux, J. Legrand, Investigating the usability of real-time scheduling theory with the Cheddar project, Real-Time Syst. 43 (2009) 259–295.
[53] E. Senn, J. Laurent, E. Juin, J.-P. Diguet, Refining power consumption estimations in the component based AADL design flow, in: Specification, Verification and Design Languages, 2008. FDL 2008. Forum on, IEEE, 2008, pp. 173–178.
[54] K. Hu, T. Zhang, Z. Yang, Multi-threaded code generation from Signal program to openMP, Front. Comput. Sci. 7 (2013) 617–626.

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

Teng Zhang received his B.E. in computer science and engineering from Beihang University in 2011. He is now the master's degree student at the same university. His research interests include synchronous languages, modeling of embedded system and formal methods.

Zhibin Yang received his Ph.D. degree from Beihang University, China, in February 2012. Since April 2012, he has been a Postdoc in IRIT research laboratory of University of Toulouse, France. His research interests include safety-critical real-time system, formal verification, AADL, and synchronous languages.

Dr. Wei-Tek Tsai received his B.S. from Computer Science and Engineering from Massachusetts Institute of Technology at Cambridge in 1979, M.S. and Ph.D. in Computer Science from University of California at Berkeley in 1982 and 1986. He is now Professor in the School of Computer Science and Engineering at Beihang University, Beijing, China, and Emeritus Professor at Arizona State University. He has authored more than 400 papers in software engineering, service-oriented computing, and cloud computing. He travels widely and has held various professorships in US, Asia and Europe.