Hierarchical Behavior Annex: Towards an AADL Functional Specification Extension

Jinmiao Xu
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China
JinMiao_Xu@163.com

Zhibin Yang
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China
yangzhibin168@163.com

Zhiqiu Huang
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China
zqhuang@nuaa.edu.cn

Yong Zhou
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China
zynuaa@nuaa.edu.cn

Chengwei Liu
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China

Lei Xue
Shanghai Academy of Spaceflight Technology
Shanghai, China
jzgunking@163.com

Jean-Paul Bodeveix
IRIT
Université de Toulouse
Toulouse, France
bodeveix@irit.fr

Mamoun Filali
IRIT
Université de Toulouse
Toulouse, France
filali@irit.fr

***Abstract*—AADL is a modeling language to design and analyze embedded real-time systems and is widely used to model safety-critical systems. AADL describes the system models hierarchically through components such as systems, processes, and threads, etc. The Behavioral Annex is a supplement of AADL in terms of functional behavior. It enables modeling component and component interaction behavior in a state-machine-based annex sublanguage. At present, there is no mechanism to represent hierarchical automata in the behavioral annex. However, this is a very important feature because industrial complex systems are always described with concurrent and composite states. Although we can model a system with AADL's own hierarchical description capabilities, it will result in a large amount of threads. In actual development, a refinement process is always needed before system synthesis, in which several threads may be combined into one thread that has concurrent and composite states. This paper proposes a hierarchical extension of the AADL behavioral annex which is named HBA (Hierarchical Behavior Annex). First, the formal syntax of HBA is given, and then we formally define the semantics of HBA. We propose a meta-model of HBA and implement its textual and graphical editor in the OSATE environment. Finally, an industrial case study is given to validate the approach.**

Keywords—safety-critical systems, AADL (architecture analysis and design language), hierarchical behavior annex, functional specification

I. Introduction

Safety-Critical Systems are widely present in fields such as aerospace, communications, nuclear industry, and automotive electronics. As functional and non-functional requirements continue to be extended, it dramatically increases the complexity of the systems. How to design and implement high-quality, safety-critical real-time systems and effectively control development time and costs is a common challenge for both academia and industry. Recently, Model-Driven Development (MDD) has become an important method for the design and development of safety-critical systems [1].

AADL (Architecture Analysis and Design Language) [2] [3] is both a textual and graphical language with component-based modeling concepts specifically designed to represent embedded software systems. AADL provides data and subprogram components organized into packages to abstractly represent application source code that is implemented in any programming language (such as Java, C, or Ada) or in an application design language (such as Simulink for control system components). AADL provides thread, thread groups, and process to represent concurrent tasks executing in protected address spaces (time and space partitioning) and interacting through ports, shared data components, and service call to represent the software runtime architecture. The dynamic of the runtime architecture are captured through mode state machines at different levels of the component hierarchy to present operational modes, dynamic changes to fault-tolerant configurations, and component behavior. In addition, the AADL standard provides well-defined execution semantics for task execution, communication timing, and mode changes using hybrid automata specification to address predictable response times [4].

The AADL behavior annex [5] proposed in 2006, is an extension of AADL to offer a way to specify the behaviors of components without expressing them with the target language, therefore it can support more precise behavioral and timing analysis [6]. The Behavior Annex enhances AADL's ability to describe the functional behavior of components such as thread and subprogram, in the form of a transition system [7]. The execution model defines when the Behavior Annex is executed and which data is exchanged, while the Behavior Annex is located within the component and gives a more accurate description of the execution of components.

However, industrial complex systems are always described with concurrent and composite states which make easier the modeling of complex behaviors. If the nested state machines are unfolded, a huge and unmanageable state machine diagram will be formed. Although we can model a system with AADL's own hierarchical description capabilities, it will result in a large amount of threads. Therefore, it is a very important feature for the AADL behavior annex to express the functional behavior hierarchically.

We know, the flattened model can hardly manage a large number of states or actions and lose structural information. In order to solve this problem, the UML State Charts was proposed [8]. The UML state charts consist of a set of finite state machines containing locations and edges. The state machine can be embedded in a given location. Locations can be "AND-locations" or "XOR-locations". The automata embedded in "AND-locations" are independent of each other, and they are executed in parallel. The automata embedded in the "XOR-locations" can form a connected graph. In addition, David A. et al. proposed HTA [9] (Hierarchical Timed Automata) which is used to facilitate the hierarchical modeling in UPPAAL. In addition, Ricardo Bedin et al. evaluated the AADL Behavior Annex with a reengineering experiment of a flight-control software and introduced the hierarchical concept in AADL Behavior Annex. However, they haven't given formal syntax and semantics definitions, and the implementation.

In this paper, we propose a hierarchical extension of AADL behavior annex named HBA (Hierarchical Behavior Annex). Then, we give the formal syntax and semantics definition of the HBA. In order to implement HBA, we define the meta-model extension of AADL behavior annex. Furthermore, the plugin of HBA is integrated into the AADL open source environment OSATE (Open Source AADL Tool Environment). Finally, we present an actual industrial case using the HBA.

Moreover, as shown in Figure 1, we give a global view of our AADL-based development approach adapted to China industry. Firstly, a Restricted Natural Language (RNL) requirements modeling method and its automatic transformation into AADL models were proposed [10]. The internal behaviors in the generated initial AADL model (M0) may be not so precise, thus there will be a refinement process which may include several steps. For instance, we can use behavior annex or hierarchical behavior annex to refine the internal behaviors of a thread, or several threads will be combined into one thread with a hierarchical behavior annex. Therefore, a relatively complete platform-independent model M1 (PIM) will be obtained. In the second refinement, designers add platform-specific details to the model such as operational system API, interruption, pipeline communication, watchdog, hardware protocol, etc., and obtain the platform-specific model M2 (PSM). Moreover, the AADL models should be formally verified. We use Timed Abstract State Machine (TASM) and UPPAAL to verify the individual properties of each component [11], the compositional verification tool AGREE [12] to verify the system properties of the hierarchical components, the Cheddar [13] tool to verify the schedulability of the system. Finally, the executable C and Ada code will be generated. We also consider how to relate different stages together seamlessly by requirement traceability [14], and the proof of semantics preservation [11].

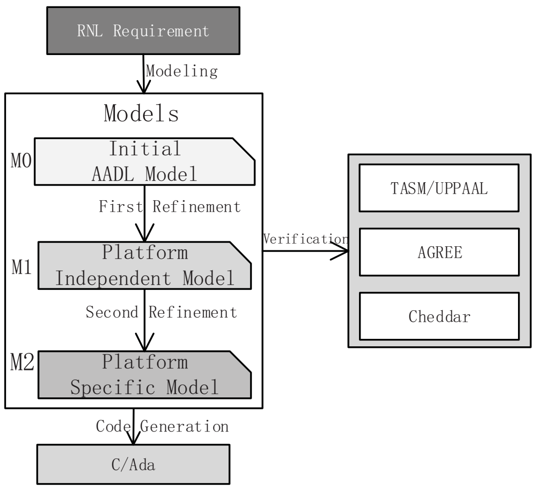

Fig. 1. Global view of an AADL-based development

The rest of the paper is structured as follows. Section II gives an introduction to AADL and Behavior Annex. Section III gives the syntax of the HBA. Section IV defines the formal semantics of the HBA. Section V presents the implementation of the hierarchical extension of AADL Behavior Annex. Section VI presents an industrial case study. Section VII discusses the related work. Section VIII draws conclusions and future work.

II. AADL AND ITS BEHAVIOR ANNEX

A. *The AADL Language*

AADL (Architecture Analysis and Design Language) is designed for the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, schedulability, fault tolerance, security, etc.) distributed systems. It enables the development of highly evolvable systems, early and quantitative analyses of a system's architecture, and evolution of an architecture model for continued analysis throughout the lifecycle.

AADL employs formal modeling concepts for the description of software/hardware architecture and nonfunctional properties of embedded real-time systems in terms of distinct components and their interactions. AADL offers a set of predefined component categories [2] [3].

- Thread, thread group, subprogram, data and process.
- Processor, memory, bus and device.
- System represents composite sets of software and execution platform components.

For instance, a thread represents a sequential flow of execution and it is the only AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another subprogram.

Communication between components can be realized through dataflow, call to server subprogram or access to shared variable. These various connection points are declared in the interface of the communicating components and are called features. They will be Ports, Server Subprograms or Data Access depending on the chosen communication paradigm [6].

System behaviors do not only rely on the architecture defined by such above components and their connections but also rely on the runtime environment [11]. AADL standard has specified execution model as a virtual runtime environment, which contains synchronous as well as asynchronous patterns, to support the execution and management of components. Timing aspects such as deadline, dispatch time, are also defined in the execution model, declared through AADL properties.

Moreover, AADL supports two ways of extension: property set and annex. The property set allows users to introduce new property sets. For example, Cheddar, the scheduling analysis tool, enhances AADL to support more complex scheduling algorithms by defining new property sets. Existing annexes include: AADL Error-Model Annex [15], AADL Behavior Annex [5], ARINC653 Annex, Data-Model Annex, etc.

B. AADL Behavior Annex

Behavior annex lies in the computing state, is an extension of the dispatch mechanism of execution model, to describe more precisely the behaviors such as port communication, subprogram call, timing, asynchronous, etc. The AADL execution model specifies when the behavior annex is executed and on which data it is executed. A full AADL model should contain well-defined structure, execution model and behavior annex. Now, a behavior annex can be attached to any component of AADL. It is described using an extension of AADL mode automata [5]: initial to specify a start state, return to specify the end of a subprogram or complete to specify completion of a thread, transitions may be guarded by conditions and actions, conditions and actions include sending or receiving events, calling or executing subprograms, assigning or testing data variables as well as execution abstractions such as use of CPU time or delay.

The behavior annex mainly includes three parts: Variables, States, and Transitions. The *variable* part declares all the local variables used in the current behavior annex. The local variables can be used to save intermediate results within the scope of the current behavior annex. The *state* part enumerates the states of the machine with their properties (Initial, Complete, Final, or a combination of them). By default, a state is an execution state. The Behavior annex starts in the initial state and terminates in the complete state, waiting for the next dispatch of a thread, or in the final state. *Transitions* define the transitions from a source state to a destination state. A transition has also a guard, and an action.

In the case of subprograms, the automaton consists of one initial state representing the starting point of a call, zero or more intermediate execution states, and one final state. A final state represents the completion of a call. The complete state is not used in behavior specifications of subprograms.

In the case of threads and devices, the automaton consists of one initial state representing the state before initialization actions, one or more complete states, zero or more intermediate execution states, and one or several final states. A complete state acts as a suspend/resume state out of which threads and devices are dispatched. The final state represents the state when a thread or device completes finalization.

Here, we give an example of the behavior annex of a thread as follows.

```
thread implementation example.impl
         annex behavior_specification{**
variables
         a : Base_Types :: Integer;
         has : Base_Types :: Boolean;
states
         s1 : initial complete state ;
         s2 :  state ;
         s3 : state ;
         s4 : complete state;
transitions
  T_0 : s1 -[on dispatch p]->s2{has:=true};
  T_1 : s2 -[has = true]-> s3;
  T_2 : s2 -[ has = false]-> s4{a:= a+1};
**};
end example.impl;
```

III. HIERARCHICAL BEHAVIOR ANNEX（HBA）SYNTAX

The HBA extends the AADL behavior annex to enhance the hierarchical description capabilities of the behavior annex. The HBA retains the variables, states and transitions of the AADL behavior annex, and adds hierarchical mapping functions and hierarchical states.

Definition1 (Hierarchical Behavior Annex (HBA)) A hierarchical behavior annex is a tuple $<Var, S, S_0, \eta, T, type>$, where:

- Var is a finite set of variables.
- S is a finite set of states.
- $S_0 \subseteq S$ is a set of initial states.
- $\eta : S \to 2^S$ is a hierarchical function, it maps S to the sub-states of S .The mapping η is required to give rise to a tree structure where a special super-state $root \in S$ is the root of the tree. We use η to record the hierarchical relationship between a state S and its sub-states.
- $\text{type} : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, it enumerates all the types of states. Composite states can be AND or XOR type and the type of non-composite states is one of BASIC, ENTRY, EXIT, or HISTORY.

The AND state indicates that all the sub-states of the composite state are executed concurrently, which means, when the parent state is executed its internal sub-states are simultaneously executed from their respective initial states. In the subsequent HBA implementation, we represent the AND state as concurrent state. XOR indicates that all the sub-states in the composite state are mutually exclusive, which means, one and only one sub-state is executed at a time. In the subsequent HBA implementation, we represent the XOR state as composite state.

The BASIC state includes the initial state, the complete state, and the final state, which are present in the AADL behavior annex. The AND, XOR, ENTRY, EXIT, and HISTORY states are new state types defined for the HBA. The ENTRY state indicates that the non-composite state is the entrance state of its parent state, which means, when entering its parent state, it enters the sub-state by default. The EXIT state is the out state for the corresponding non-composite state, which means that when the parent state transits to next state, the transition from this exit state to the next state is actually performed. The HISTORY state is a pseudo-state whose purpose is to remember the sub-state in which it exited from the combined state. When entering the composite state again, this sub-state can be entered directly instead of starting from the ENTRY state of the combined state again.

- $T \subseteq s \times (Guard \times Action) \times s$ is the set of transitions. A transition connects two states s and s', and it has a guard and an action. s is called the *source state* and s' is called the *target state*. We use the notion $s \xrightarrow{g,a} s'$ for this and omit g, a, when they are absent. When the transition is completed, actions will be performed.

Here we give an example as shown in Figure 2. In this figure, Fig 2(a) depicts a state chart graphically and Fig 2(b) shows its tree representation. Figure 3 shows the state diagram in the corresponding AADL behavior annex code. We note that state A is an AND state, and that ENTRY and EXIT states do not appear in the tree representation.

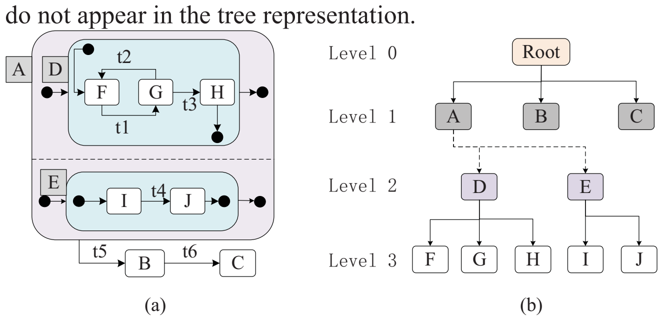

Fig. 2. The syntax of HBA

```
annex behavior_specification{**
          states
           A : initial concurrent state ;
           B : state ;
           C : final state ;
  transitions
           t5 : A-[]->B;
           t6 : B-[]->C;
  concurrent state A
          composite state D
          states
                    F : state;
                    G : state ;
                    H : state ;
          transitions
                    t1 : F-[]->G;
                    t2 : G-[]->F;
                    t3 : G-[]->H;
          end D;
          composite state E
          variables
                    x : Base_Types::Integer;
          states
                    I : state ;
                    J : state ;
          transitions
                    t4 : I-[x > 0]->J{x := x+1};
          end E;
end A;
**};
```

Fig. 3. Textual representation of the HBA example

In the following paragraphs, we will give the state constraints, and we first present the syntactic conventions.

Syntactic Conventions:

For $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}, s \in S$, we use the predicate notation $TYPE(s)$. For example, $AND(s)$ is true when $TYPE(s) = AND$.

The type $HISTORY$ is a special case of an entry, we use $HEntry(s)$ to capture simple entry or history entry. When $HISTORY$ exist $HE\text{ntry}(s)=Histroy(s)$,otherwise $HEntry(s)$ is the default ENTRY state of s.

We define the parent function η^{-1}:

$$\eta^{-1}(S)=\begin{cases} b, where\ \text{s} \in \eta(\text{b}) \text{ if s} \neq \text{root} \\ \perp \qquad\qquad\qquad otherwise \end{cases}$$

So, we define the grandparent function η^{-2} :

$$\eta^{-2}(S)=b, where\ \exists \text{a}, b=\eta^{-1}(\text{a}) \text{ and a} = \eta^{-1}(\text{s})$$

Then, we give a set of constraints to ensure consistency of an HBA:

State Constraints:

- Only composite state contains other states:
  $AND(s) \vee XOR(s) \Leftrightarrow \eta(s) \neq \varnothing$
- Sub-states of AND composite states are not of a basic type:
  $AND(s) \wedge b \in \eta(s) \Rightarrow \neg Basic(b)$
- There is one initial location per XOR composite state:
  $XOR(s) \Rightarrow |\eta(s) \cap S_0| = 1$
- There is only a single history state for each submachine.
- All the sub-states of an AND states must be an XOR type.
- The syntax does not support directly edges to a composite state such as AND or XOR state. A transition having a composite state as target corresponds to its *default entry*. The default entry is connected to the initial location of the given composite state. For transitions having a composite state as source, this is the same as having the source being a *default exit* connected to all internal states. The legal transitions are given in Fig 4 and Table I, where the black solid point represents the ENTRY or EXIT states and the white hollow point represents a basic state. The transitions cannot go directly from an ENTRY state to an EXIT state.

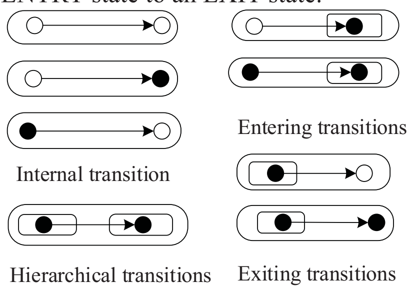

Fig. 4. Legal transitions

TABLE I. Legal transition Example

Transition types	s	s'	Constraint
Internal	BASIC	BASIC	$\eta^{-1}(s)=\eta^{-1}(s')$
	BASIC	EXIT	
	HEntry	BASIC	
Entering	BASIC	HEntry	$\eta^{-1}(s)=\eta^{-2}(s')$
	HEntry	HEntry	
Exiting	EXIT	BASIC	$\eta^{-2}(s)=\eta^{-1}(s')$
	EXIT	EXIT	
Changing	EXIT	HEntry	$\eta^{-2}(s)=\eta^{-2}(s')$

IV. Hierarchical Behavior ANNEX(HBA) Semantics

We define the operational semantics of the HBA formalism. Legal steps between states of an HBA define a set of traces. A state captures a snapshot of the system, i.e., the active states, the integer variable values, and the history of some super-states.

Definition 2 (States) States are of the form (ρ,μ,θ), where:

- $\rho: S \to 2^S$ is a mapping function that gives the set of active states. ρ can be understood as a dynamic version of η that maps every super-state S. If a super-state S is not active, then $\rho(S)=\varnothing$. We define $Active(\mathrm{S}) = \mathrm{S} \in \rho^*(root)$, where $\rho^*(\mathrm{S})$ is the set of all active sub-states of S including S, We note that for $S \neq root$: $Active(\mathrm{S}) \Leftrightarrow \mathrm{S} \in \rho(\eta^{-1}(\mathrm{S}))$, where $\eta^{-1}(S)$ is the parent state of S.
- $\mu: Var \to \mathbb{R}$ maps variables to their values. If $\neg Active(\mathrm{S})$, then for $v \in Var(\mathrm{S})$, $\mu(v)$ is undefined: we note $\mu(\mathrm{v}) = \perp$.
- $\theta: S \to S$ returns the last visited sub-state of S or an entry of the sub-state in the case where the sub-state is not basic. If the exiting state is a BASIC state of the composite state, $\theta(\mathrm{S})$ returns the BASIC state; otherwise, it returns the ENTRY state of the composite.

History state: We capture the existence of a history entry with the predicate $HasHistory(\mathrm{S}) = \exists b \in \eta(\mathrm{S}).\,\mathrm{HISTORY(b)}$. If $HasH\mathrm{istory(S)}$ holds, the term $HE\mathrm{ntry(S)}$ denotes the unique history entry of S. If $HE\mathrm{ntry(S)}$ does not holds, the term $HE\mathrm{ntry(S)}$ denotes the default entry of S. If S is a BASIC state, then $HE\mathrm{ntry(S)=S}$. If none of the above is the case, then $HE\mathrm{ntry(S)}$ is undefined.

Definition3 (Internal Transition) We define the *internal transformation* for transformations that the source state and destination state are within the same composite state. The value of variables will be update to V_2 from V_1 by the action a.

$$\frac{\eta^{-1}(\mathrm{S}) = \eta^{-1}(\mathrm{S}'), g = true}{(\mathrm{S},\mathrm{V}_1) \xrightarrow{g,a} (\mathrm{S}',\mathrm{V}_2)}\, simple\_transition$$

In order to describe the transitions that across the composite state boundary, we define the *Entering transition* and *Exiting transition*.

For each composite state, there is a default entry state and a default exit state. When a transition takes the external state as the source and takes the internal state as the destination, it needs to use the entering transition.

$$\frac{\eta^{-1}(\mathrm{S}) = \eta^{-2}(\mathrm{S}_1), \mathrm{ENTRY}(\mathrm{S}_1) = \mathrm{true}, \mathrm{g} = \mathrm{true}}{(S,V) \xrightarrow{g,a} (S_1,V_1)}\, ent1$$

$$\frac{\eta^{-1}(\mathrm{S}_1) = \eta^{-1}(\mathrm{S}'), \mathrm{ENTRY}(\mathrm{S}_1) = \mathrm{true}}{(S_1,V_1) \xrightarrow{\varepsilon} (S',V_1)}\, ent2$$

$$\frac{ent1 \circ ent2}{(S,\mathrm{V}) \xrightarrow{g,a} (S',V_1)}\, Entering\_transition$$

As the Entering transitions that we can see in Figure 4, the parent state of the source state (i.e. $\eta^{-1}(\mathrm{S})$) and the grandparent state of the destination state (i.e. $\eta^{-2}(\mathrm{S}_1)$) is the same state. Since the source state type is not fixed, we do not constrain the source state. The entry transition is divided into two steps: 1) the transition from the external state to the Entry (i.e. S_1) of the composite state, and 2) the transition from the entry state to the destination.

When a transition takes the internal state as the source and takes the external state as the destination, it needs to use the exiting transition.

$$\frac{\eta^{-1}(\mathrm{S}) = \eta^{-1}(\mathrm{S}_1), EXIT(\mathrm{S}_1) = \mathrm{true}}{(S,V) \xrightarrow{\varepsilon} (S_1,V)}\, ext1$$

$$\frac{\eta^{-2}(\mathrm{S}_1) = \eta^{-1}(\mathrm{S}'), \mathrm{EXIT}(\mathrm{S}_1) = \mathrm{true}, \mathrm{g} = \mathrm{true}}{(S_1,V) \xrightarrow{g,a} (S',V_1)}\, ext2$$

$$\frac{ext1 \circ ext2}{(S,\mathrm{V}) \xrightarrow{g,a} (S',V_1)}\, Exiting\_transition$$

The Exiting transition is similar to the Entering transition, but the grandparent state of the source state (i.e. $\eta^{-2}(\mathrm{S}_1)$) and the parent state of the destination state (i.e. $\eta^{-1}(\mathrm{S}')$) is the same state. The exit transition is divided into two steps: 1) the transition from the source state to the Exit (i.e. S_1) of the composite state, and 2) the transition from the exit state to the destination.

Definition4 (Hierarchical Transition) Hierarchical transition $\mathrm{HT}: S \xrightarrow{g,a} S'$ is a transition exits from a composite state and enters another composite one, as show in Fig 4, including three steps: (1) executing an exiting transition to exit the super-state of S, (2) taking the transition HT itself, and (3) executing an entering transition to enter the super-state of S'. Together, 1-3 define a *hierarchical transition*.

$$\frac{\eta^{-1}(\mathrm{S}) = \eta^{-1}(\mathrm{S}_1), EXIT(\mathrm{S}_1) = \mathrm{true}}{(S,V) \xrightarrow{\varepsilon} (S_1,V)}\, ht1$$

$$\frac{\eta^{-2}(\mathrm{S}_1) = \eta^{-2}(\mathrm{S}_2), \mathrm{EXIT}(\mathrm{S}_1) = \mathrm{true}, \mathrm{ENTRY}(\mathrm{S}_2) = \mathrm{true}, \mathrm{g} = \mathrm{true}}{(S_1,V) \xrightarrow{g,a} (S_2,V_1)}\, ht2$$

$$\frac{\eta^{-1}(\mathrm{S}_2) = \eta^{-1}(\mathrm{S}'), \mathrm{ENTRY}(\mathrm{S}_2) = \mathrm{true}}{(S_2,V_1) \xrightarrow{\varepsilon} (S',V_1)}\, ht3$$

$$\frac{ht1 \circ ht2 \circ ht3}{(S,\mathrm{V}) \xrightarrow{g,a} (S',V_1)}\, Hierarchical\_transition$$

We use the HT to present the hierarchical transition. So, after the hierarchical transition, we can get $(\rho^{1}, \mu^{1}, \theta^{1})$:

$$\rho^{1} = \{x \mid \forall b \in \rho, \mathrm{HT}(b) = x\}$$

$$\mu^1=\{\theta \mid \forall v\in \mu, \mathrm{HT}(v)=\theta\}$$

$$\theta^1 = HISTORY(\theta)$$

Definition5 (State transformation) We define the state transformation Tst :

$$Tst(\rho,\mu,\theta)=(\rho^1,\mu^1,\theta^1)$$

Each system snapshot consists of three parts: ρ , μ , and θ . After a state transition, the snapshot will be updated to ρ^1 , μ^1 , and θ^1 .

Remark 1: Synchronization usually involves a sender and a receiver and data change. Moreover action sending and receiving can be guarded. Since in AADL, we have global variables, one can define many semantics for such a synchronization. Since, we have adopted UPPAAL as a verification engine, we use the UPPAAL semantics for the basic synchronization and exchange data through global variables.

V. THE IMPLEMENTATION OF HIERARCHICAL BEHAVIOR ANNEX

In the implementation, in order to identify the extended syntax, we use ANTLR (ANother Tool for Language Recognition) technology to generate an Abstract Syntax Tree (AST). The input language of ANTLR is very similar to the BNF form, so we give the BNF description of the HBA. In addition, we use the EMF and the Xtext technologies to extend hierarchical elements in the meta-model of behavioral annex. The meta-model is one of the inputs of the EMF, so we give the meta-model extension of the HBA.

A. Extension of the AADL Behavior Annex BNF

It shows the original AADL behavior annex concrete syntax (part) and the extended AADL behavior annex concrete syntax (part) in Figure 5 and Figure 6 respectively. We have added the concept of *composite* states and the expression of inline sub-states in *composite* states.

```
behavior_annex ::=
    [ variables { behavior_variable }+ ]
    [ states { behavior_state }+ ]
    [ transitions { behavior_transition }+ ]
behavior_state ::=
    behavior_state_identifier { , behavior_state_identifier }* : behavior_state_kind
state;
    behavior_state_kind ::= [ initial ][ complete ][ final ]
behavior_transition ::=
  [transition_identifier [ [behavior_transition_priority] ] : ] source_state_dientifier
{,source_state_identifier}*
      -[behavior_condition]->        destination_state_identifier[{behavior_actions}
[timeout behavior_time]]
Behavior_condition ::= dispatch_condition | execute_condition
```

Fig. 5. Original AADL behavior annex syntax (part)

```
behavior_annex ::=
    [ variables { behavior_variable }+ ]
    [ states { behavior_state }+ ]
    [ transitions { behavior_transition }+ ]
    {concurrent concurrent_state}*
    {composite composite_state}*
behavior_state ::=
    behavior_state_identifier { , behavior_state_identifier }* : behavior_state_kind
state;
    behavior_state_kind ::= [ initial ][ complete ][ final ] [composite] [concurrent]
[history] [entry] [exit]
behavior_transition ::=
  [transition_identifier [ [behavior_transition_priority] ] : ] source_state_dientifier
{,source_state_identifier}*
      -[behavior_condition]->        destination_state_identifier[{behavior_actions}
[timeout behavior_time]]
Behavior_condition ::= dispatch_condition | execute_condition
concurrent_state ::=
    state behavior_state_identifier
    {composite composite_state}+
end behavior_state_identifier;
composite_state ::=
    state behavior_state_identifier
    [ variables { behavior_variable }+ ]
    [ states { behavior_state }+ ]
    [ transitions { behavior_transition }+ ]
    {concurrent concurrent_state}*
    { composite composite_state}*
end behavior_state_identifier;
```

Fig. 6. The extended AADL behavior annex syntax (part)

B. Extension of the AADL Behavior Annex meta-model

The AADL behavior annex is closely related to the AADL core, so the AADL behavior annex meta-model reuses the AADL meta-model's EMF framework and is designed on this basis. On the one hand, this aspect facilitates the integration of AADL-BA in OSATE2. On the other hand, using the same form to formulate two meta-models can simplify the expression of object dependencies and simplify the model connection between them.

The behavior annex requires to attach a behavior specification to an AADL component and to link AADL-BA objects with AADL objects. It is reinforced by the implementation of the OSATE2 sub-language extension which requires linking a BehaviorAnnex object to the AnnexSubclause object of the AADL component.

We show the dependency between two meta-models expressed through EMF extensions (e.g. Java's inheritance mechanism) in Figure 7. As we can see, a *BehaviorAnnex* extends an *AnnexSubclause*, and a *BAElement* extends an *Element*. The latter simplifies the link from the AADL-BA model to the AADL model by referencing the AADL objects in the behavior specification and easily retrieving the corresponding AADL objects during the analysis.

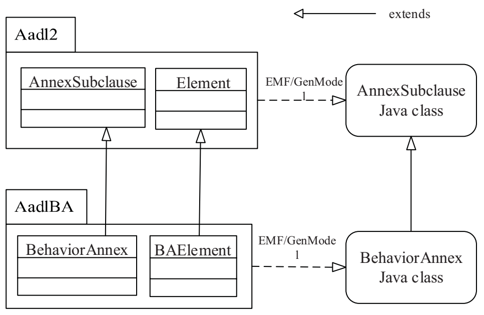

Fig. 7. AADL-BA Meta-model dependency

In order to extend the AADL behavior annex, we first extend the AADL meta-model to increase the expression of concurrent states and compound states (See bold part in Figure 8).

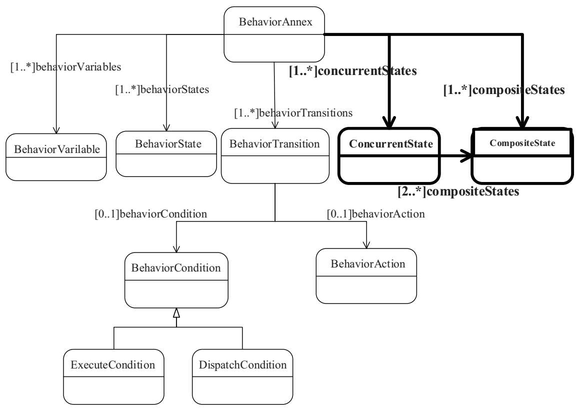

Fig. 8. The extended AADL behavior annex meta-model (part)

As we can see in Fig.8, the main difference between the extended hierarchical behavior annex and the AADL behavior annex is the addition of Composite State and Concurrent Sate. In addition, in order to comply with the aforementioned constraints, we have correspondingly added state types such as entry state and exit state.

C. Java code generation with ANTLR and Xtext

ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

Xtext is a framework for development of programming languages and domain-specific languages. With Xtext we can define our language using a powerful grammar language. As a result we get a full infrastructure, including parser, linker, type checker, and compiler as well as editing support for Eclipse.

The syntax rules of ANTLR are similar to BNF, so we can quickly modify the ANTLR model with the previously proposed BNF syntax. The original ANTLR framework for behavioral annex can be found on GitHub. We use the methods of the AADL-HBA builder factory in the ANTLR grammar to specify how to build the abstract syntax tree (AST) with AADL-HBA objects. It ensures that the AST is compliant with the AADL-HBA meta-model. Finally we use the ANTLR framework to generate the Java classes of the parser and the lexer from the AADL-HBA BNF defined.

Moreover, OSATE is an open source environment that provides supports for the development of architecture models for embedded real-time systems based on AADL, including modeling, compilation and analysis for AADL. OSATE not only provides a set of plug-ins (for validating models or interfacing them with other tools), but also provides an extensible framework for users to develop plug-ins for AADL models. OSATE is based on Eclipse and is constructed using Xtext, which supports the textual editor of AADL. Besides, AADL Behavior Annex plugin-in is also based on Xtext technology and the meta-model (a kind of input of Xtext) of AADL Behavior Annex can be download on the internet. Thus, we extend the meta-model of BA with the EBNF (section IV).

AADL supports textual modeling and graphical modeling, while behavior annex does not support graphical display and modeling. In order to facilitate using HBA, we implement a graphic editor for the AADL behavior annex as a plugin, including graphical display and graphical modeling.

VI. CASE STUDY

A. LCS system

The Launch Control System (LCS) is an important element of the Rocket Ground Test Launch Control System. It protects the equipment during rocket testing, and sets parameters and launch controls before firing. The mission of the LCS system is to receive commands from the command system, complete the test of the rocket on the launch vehicle at the launch site, conduct launch control and simulate rocket launches at the training. A simplified block diagram of the rocket LCS system is given Figure 9.

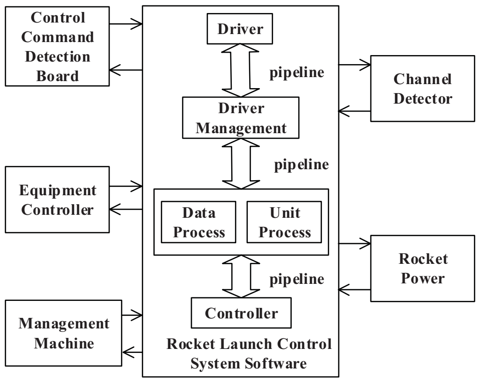

Fig. 9. Rocket launch control system

In the requirement document, the LCS system is divided into five layers, such as driver, driver management, data processing, unit processing, and controller. There are 53 functional modules in total. Each functional module contains a set of complex processing.

Layer	Functional Modules
Driver	17
Driver Management	10
Data Processing	2
Unit Processing	13
Controller	11

We have developed a tool [10] that can automatically generate AADL initial models from restricted natural language (RNL) requirements. We have already specified the requirements of LCS system in RNL, thus, an initial AADL model of the LCS system can be automatically generated. Here, we mainly focus on the refinement of the generated initial AADL models. Due to space limitations, we present the refinement of the Power Control Function in the Unit Processing Layer of the LCS system as a demonstration in this paper.

B. *Power Control Function*

1) *Requirement*

There are 10 modules in the Power Control Function. In order to simplify the description, we only introduce several functional modules.

The first module is *Rocket Power-On 1(RPOn1)*. The purpose of this module is to check the rocket type identification code and launch platform self-check results. If the check result is fault, then the thread calls *Rocket Power-Off 1(RPOff1)* module. If the check result is correct, the thread will call *Rocket Power-On 2(RPOn2)* module. The purpose of this module is to check the result of the rocket self-check. If the self-check result is fault, then the thread calls *Rocket Power-Off 2(RPOff2)* module, otherwise, the thread calls *Rocket Power-On 3(RPOn3)* module. The purpose of this module is to check the results of rocket power supply. If the check result is fault, then the thread calls *Rocket Power-Off 3(RPOff3)* module, otherwise, the thread calls *Open the Cover (OC)* module. The functional behavior of *Power Control Function* is shown in Figure 10.

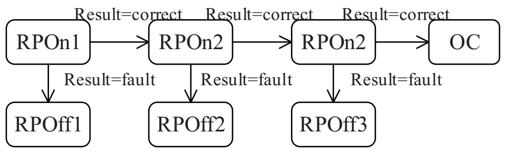

Fig. 10. The functional behavior of Power Control Function

The above requirements are the architecture designed by the architects. They may not consider the internal processing details of each module. The internal details will be supplemented by the designer during the refinement phase.

2) *Modeling with BA*

Here behavior annex is used to present the first refined behaviors. The AADL code is given as follows:

```
annex behavior_specification{**
     states
          RPOn1 : initial state;
          RPOn2 : state;
          RPOn3 : state;
          OC : state;
          RPOff1 : state;
          RPOff2 : state;
          RPOff3 : state;
          Omit : complete state;
     transitions
          T1 : RPOn1 -[result1=correct]-> RPOn2;
          T2 : RPOn1 -[result1=faulty]-> RPOff1 {errMsg!(err1)};
          T3 : RPOn2 -[result2=correct]-> RPOn3;
          T4 : RPOn2 -[result2=faulty]-> RPOff2 {errMsg!(err2)};
          T5 : RPOn3 -[result3=correct]-> OC;
          T6 : RPOn3 -[result3=faulty]-> RPOff3 {errMsg!(err3)};
          T7 : OC -[]->Omit;
**};
```

The corresponding graphical effects in the above case are shown in Figure 11.

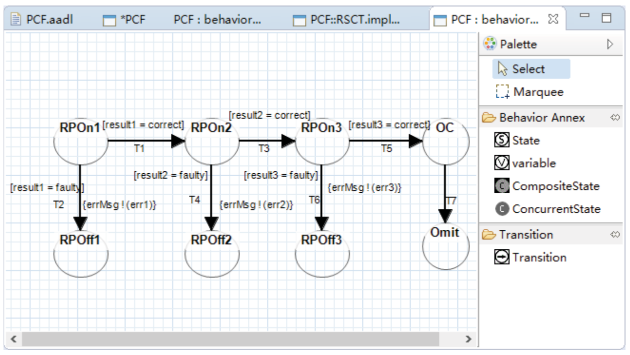

Fig. 11. Power Control Function graphical display

C. *Refinement*

1) *Requirement*

Here we only present the refinement of the *Rocket Power-On 1(RPOn1)* module. In our AADL specification, there are three sub-modules in *RPOn1*: the *Launch Platform Self-Check Module (LPSCM)*, the *Rocket Type Check Module (RTCM)*, and *Rocket Power-On Module (RPOM)*. The *Launch Platform Self-Check Module* checks the launch platform status and sends the check result to the *Rocket Type Check Module*. The *Rocket Type Check Module* confirms whether the rocket type is consistent with the required type and sends the confirmation result to the *Rocket Power-On Module*. The *Rocket Power-On thread* determines whether to power on the rocket.

The *Launch Platform Self-Check Module (LPSCM)* is specified as an automaton which contains four main states: *Platform Information Decoding State*, *Platform Self-Checking State*, *Normal State*, and *Error Report State*. After the platform information is decoded, the automaton sends the platform description information, and then the automaton enters the *Platform Self-Checking State*. The *Launch Platform Self-Checking State* gives the self-check result. If the self-check result is correct, the automata enters in the *Normal State*, and the self-check result message is sent to the *Rocket Type Check Module (MTCM)* through the port; otherwise, an *Error Report State* is entered. The *errorReport* state includes four sub-states: *IO Port Power off (P)*, *Set Flag (S)*, *Report Error Message (R)* and *Update Rocket Configuration (U)*. The inner execution of *errorReport* state is sequential. After cutting the power, the automaton enters the *Set Flag State*. Then the automaton set the flag to *fault.* In the end, the automaton updates the configuration information of the rocket.

The functional behavior of the Rocket Type Check Moudle (RTCM) is presented as an automaton which includes four major states: Self-Check Result Decoding State, Rocket Type Identification State, Type Correctness State, and Error Report State. After receiving the platform self-check result message, the automaton enters the Rocket Type Identification State. The Rocket Type Identification State confirms that the type of rocket is consistent with the type required. If the rocket type is correct, the automata enters the Type Correctness State and sends the confirm result message to the Rocket Power-On Module (RPOM) through the port; otherwise, the automaton enters the Error Report State. The state of the Error Report is the same as described as above.

The functional behavior of the Rocket Power-On Module (RPOM) is described by an automaton which includes two main states: Rocket Power-On and Power-On Report. After receiving the recognition result message, the automaton enters the Rocket Power-On State. After the completion of power-on, the automaton enters the Power-On Report State.

The functional behavior of RPOn1 is shown in Figure 12.

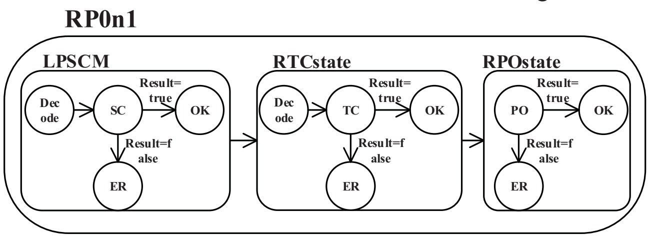

Fig. 12. The functional behavior of RPOn1

2) Modeling with HBA

Here, we take the functional behavior of the *Launch Platform Self-Check Module (LPSCT)* as example, which includes 4 states and 3 transitions. The *errorReport* state is a composite state, it includes 4 sub-states. There is also an automaton in the composite state *errorReport*. The AADL code is given as follows:

```
annex behavior_specification{**
states
    RPOn1 : initial state;
    RPOn2 : state;
    RPOn3 : state;
    OC : state;
    RPOff1 : state;
    RPOff2 : state;
    RPOff3 : state;
    Omit : complete state;
transitions
    T1 : RPOn1 -[result1=correct]-> RPOn2;
    T2 : RPOn1 -[result1=faulty]-> RPOff1 {errMsg!(err1)};
    T3 : RPOn2 -[result2=correct]-> RPOn3;
    T4 : RPOn2 -[result2=faulty]-> RPOff2 {errMsg!(err2)};
    T5 : RPOn3 -[result3=correct]-> OC;
    T6 : RPOn3 -[result3=faulty]-> RPOff3 {errMsg!(err3)};
    T7 : OC -[]->Omit;
composite state RPOn1
states
    LPSCM : initial complete state ;
    RTCstate : state ;
    RPOstate : complete state ;
transitions
     T8 : LPSCM -[on dispatch]->
 RTCstate{selfCheckResult!(PlatformCheckResult)};
     T9 : RTCstate -[]-> RPOstate{confirmPort!(confirmResult)};
--composite state LPSCM
composite state LPSCM
variables
    checkResult : Base_Types::Boolean;
states
    decode : initial state;
    selfCheckState :  state;
    OK : complete state;
    errorReport :  complete state;
transitions
    T10 : decode -[ ]-> selfCheckState{platformInfoPort!(pdMsg)};
    T11 : selfCheckState -[result = true]-> OK{checkResult:=true;
            selfCheckResult!(checkResult)};
    T12 :  selfCheckState -[result = false]-> errorReport{
            checkResult:=false;selfCheckResult!(checkResult)};
composite state errorReport
states
    powerOff : initial state;
    setFlag : state;
    reportBack : state;
    update : complete state;
transitions
    T13 : powerOff -[ ]-> setFlag{power := cutOff};
    T14 : setFlag -[]-> reportBack{flag := fault};
    T15 : reportBack -[]-> update {platformInfoPort!(configInfo)}
end errorReport;
end LPSCM;
--end composite state
end RPOn1;
**};
```

The corresponding graphical modeling is shown in Figure 13, 14 and 15.

When we double-click the *RPOn1* state in Figure 11, we can see the automaton shown in Figure 13. The Fig13 shows the graph of the *Rocket Power-On 1(RPOn1) Module*. Figure 14 shows the graph of the *Launch Platform Self-Check Module*, in which *errorReport* state is a composite state. After double-clicking the *errorReport* state, the internal rendering is shown in Figure 15.

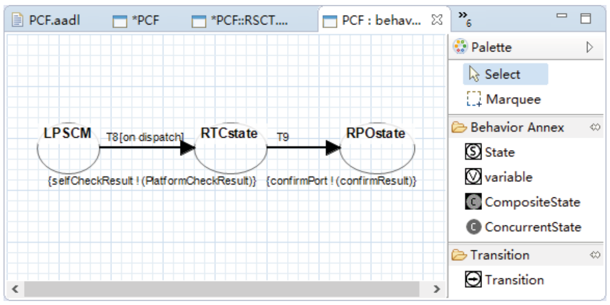

Fig. 13. *RPOn1 state graphical display*

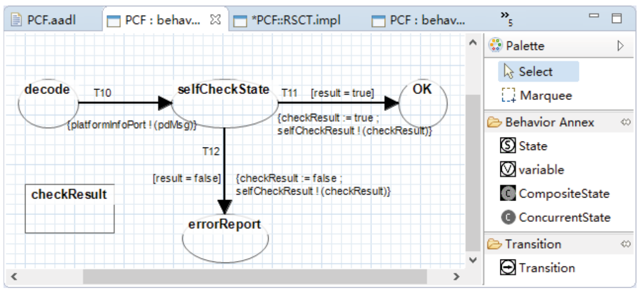

Fig. 14. Graphical representation of *LPSCM state*

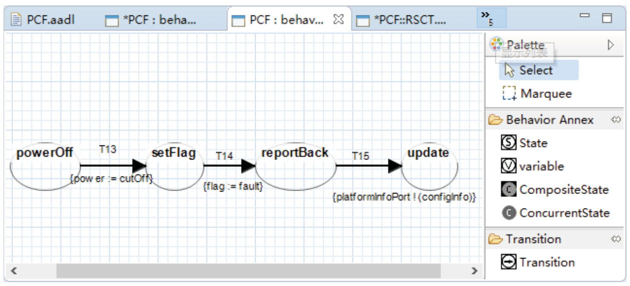

Fig. 15. Internal structure of *errorReport state*

D. Lesson learned

During the collaboration with our industrial partner for devising the methodology and conducting the industrial case study, we learned the following lessons in real industrial contexts.

- The ability to describe hierarchically is very important. Due to the large scale of actual industrial projects and the large amount of background knowledge required, the idea of layer-by-layer refinement goes through the entire development life cycle. Thanks to the idea of layering, designers at each level can focus on the work within their domain knowledge.
- When communicating with engineers, they found it hard to accept such semi-formal modeling languages like AADL. Instead, they are more willing to accept graphical modeling or pseudo-code formative languages. Therefore, a more vivid expression will be more easily accepted by engineers.

VII. RELATED WORK

In terms of hierarchical extension, Harel et.al [16] presented an extension of the conventional formalism of state machines and state diagrams. The statecharts extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication, solved the problems of large-scale state-machines with the deep level nesting of complex system, but the real-time description of embedded safety-critical software is still lacking. In order to solve the problem that the timed automata cannot describe hierarchical models, David et.al [17] proposed the concept of Hierarchical Timed Automata (HTA) and solved the complex hierarchical system modeling and verification issues by extending timed automata. HTA can model hierarchical systems and verify their temporal properties, but it needs to transform the model to the automaton form manually, which increases the complexity of the work.

In terms of formal syntax and semantics, the hierarchical extended HTA formal syntax and operational semantics was given in [17], and a simplified HTA model had been given. Yang et.al. [6] defined the formal semantics of the AADL behavior annex through the timed abstract state machine (TASM) and proposed a real-time behavioral modeling and verification prototype. Zhou et.al. [18] gave the underlying semantics of the UML state machine diagram and the time-dependent modeling elements of MARTE, the configuration files for real-time embedded system modeling and analysis, and proposed the formalization of its operational semantics based on extended hierarchical timed-automata. Ölveczky P.C. et.al. [19] presented a formal real-time rewriting logic semantics for a behavioral subset of AADL which includes the essential aspects of its behavior annex. This semantics is directly executable in Real-Time Maude. In order to support unambiguous reasoning, formal verification, high-fidelity simulation of architecture specifications in a model-based AADL design workflow, Besnard L et.al. [20] defined a formal semantics for the behavior specification of the AADL. Larson et.al. [21] presented the Behavioral Language for Embedded Systems with Software (BLESS), a behavioral interface specification language and proof environment for AADL. BLESS provided a formal semantics for AADL behavioral descriptions and automatic generation of verification conditions that, when proven by the BLESS proof tool, establish that behavioral descriptions conform to AADL contracts. Ehsan Ahmad et.al. [22] presented formal semantics of the synchronous subset of AADL models annotated with Hybrid Annex specifications using HCSP. The semantics was then used to verify the correctness of AADL models (with Hybrid Annex specifications) using an in-house developed theorem prover -- Hybrid Hoare Logic (HHL) prover. In order to support unambiguous reasoning, formal verification, high-fidelity simulation of architecture specifications in a model-based AADL design workflow, Besnard et al. [23] had defined a formal semantics for the behavior specification of the AADL. Johnsen et.al. [24] formalized a subset of AADL-core and transformed it to UPPAAL's input language, the Timed Automata (TA). They gave a formal semantics definition of the AADL subset by mapping the AADL subset to timed automata through a semantically anchored method. Franca R B et.al.[25] respectively evaluated the practicality of AADL-BA through actual engineering projects, analyzed the constraints of AADL-BA, and proposed the idea of hierarchical extension of the behavioral annex, but did not give a formal definition and implementation.

VIII. CONCLUSION AND FUTURE WORK

It is a very important feature to express concurrent and composite states in safety-critical systems. Although we can model a system with AADL’s own hierarchical description capabilities, it will result in a large amount of threads. Moreover, in actual development, a refinement process is always needed before system synthesis, in which several threads will be combined into one thread that has composite states. This paper proposes a hierarchical extension of AADL behavior annex named HBA (Hierarchical Behavior Annex). The formal syntax and the semantics of hierarchical behavioral annex are presented. In addition, through the extension of the AADL behavioral annex meta-model, based on EMF and Xtext technologies, the corresponding plug-ins are implemented on the AADL open source development environment OSATE. Finally, an industrial case study is given and evaluated.

This paper presents the formal definition of the extended hierarchical behavioral automata, which supports the modeling of the behavior of complex embedded real-time systems. However, the work at this stage only stays in the modeling aspect, and there is still a lack of property verification capabilities for the model. We are currently working on the transformation from the hierarchical behavioral annex (HBA) to hierarchical timed automata (HTA) to verify time properties of complex embedded real-time systems. In addition, in order to express both control flows and data flows in the functional behavior, we propose the co-modeling of synchronous language [26] (such as SIGNAL) and behavior annex (or hierarchical behavior annex) to describe the functional behaviors inside the AADL components.

ACKNOWLEDGMENT

This work is partly supported by National Natural Science Foundation of China (61502231); The National Key Research and Development Program of China (2016YFB1000802); National Defense Basic Scientific Research Project under Grant of China (JCKY2016203B011); Natural Science Foundation of Jiangsu Province (BK20150753); the Avionics Science Foundation of China (2015ZC52027), China Postdoctoral Science Foundation.

REFERENCES

[1] Axer P, Ernst R, Falk H, Girault A, Grund D, Guan N, Jonsson B, Marwedel P, Reineke J, Rochange C, Sebastian M, Von hanxleden R, Wilhelm R, Wang Y. Building Timing Predictable Embedded Systems. ACM Trans. Embed. Comput. Syst., 2014, 13(4): 82:1-82:37.

[2] SAE. Architecture Analysis & Design Language (standard SAE AS5506A), 2009, available at http://www.sae.org.

[3] Surhone L M, Tennoe M T, Henssonow S F. Architecture Analysis and Design Language[M]. Betascript Publishing, 2010.

[4] Feiler P H, Lewis B A, Vestal S. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems[C] Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control. IEEE Xplore, 2006:1206-1211.

[5] SAE-AS5506/2, SAE Architecture Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behavior Annex. Int'l Society of Automotive Engineers, 2011.

[6] Yang Z, Hu K, Ma D, et al. Towards a formal semantics for the AADL behavior annex[J]. 2009:1166-1171.

[7] Yang Z B, Kai H U, Zhao Y W, et al. Verification of AADL Models with Timed Abstract State Machines[J]. Journal of Software, 2015.

[8] Elattar M, Luqman H, Karpati P, et al. Extending the UML Statecharts Notation to Model Security Aspects[J]. IEEE Transactions on Software Engineering, 2015, 41(7):661-690.

[9] David A. Hierarchical modeling and analysis of timed systems[J]. Department of Information Technology Uppsala University Sweden, 2006.

[10] Wang F, Yang ZB, Huang ZQ, Zhou Y, Liu CW, Zhang WB, Xue L, Xu JM. Approach for Generating AADL Model Based on Restricted Natural Language Requirement Template [J]. Journal of Software, 2018,29(8) (in Chinese).

[11] Yang Z, Hu K, Ma D, et al. From AADL to Timed Abstract State Machines: A verified model transformation [J]. Journal of Systems & Software, 2014, 93(2):42-68.

[12] Cofer D, Gacek A, Miller S, et al. Compositional Verification of Architectural Models[C] International Conference on NASA Formal Methods. 2012:126-140.

[13] Singhoff F, Plantec A, Rubini S, et al. Teaching Real-Time Scheduling Analysis with Cheddar[C] Ecole d'été Temps Réel. 2015.

[14] Wang Fei, Huang Zhi-Qiu, Yang-Zhi-Bin, KAN Shuang-Long, SHEN Guo-Hua, CHEN Guang-Ying. A Requirements Traceability Approach for Safety-Critical Embedded System [J]. Chinese Journal of computers, 2018(3), (in Chinese).

[15] Delange J, Feiler P. Architecture Fault Modeling with the AADL Error-Model Annex[C] Software Engineering and Advanced Applications. IEEE, 2014:361-368.

[16] Harel D. Statecharts: a visual formalism for complex systems[J]. Science of Computer Programming, 1987, 8(3):231-274.

[17] David A. Hierarchical modeling and analysis of timed systems[J]. Department of Information Technology Uppsala University Sweden, 2006.

[18] Zhou, Luciano, Baresi, et al. Towards a Formal Semantics for UML/MARTE State Machines Based on Hierarchical Timed Automata[J]. Journal of Computer Science and Technology, 2013, 28(1):188-202.

[19] Ölveczky P C, Boronat A, Meseguer J. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude[C] Ifip Wg 6.1 International Conference and, Ifip Wg 6.1 International Conference on Formal Techniques for Distributed Systems. Springer-Verlag, 2010:47-62.

[20] Besnard L, Gautier T, Guy C, et al. Formal semantics of behavior specifications in the architecture analysis and design language standard[C] High Level Design Validation and Test Workshop. IEEE, 2016:30-39.

[21] Larson B R, Chalin P, Hatcliff J. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software[C] NASA Formal Methods Symposium. Springer Berlin Heidelberg, 2013:276-290.

[22] Ahmad E, Dong Y, Wang S, et al. Adding Formal Meanings to AADL Models with Hybrid Annex[C] Facs. 2014.

[23] Besnard L, Gautier T, Guy C, et al. Formal semantics of behavior specifications in the architecture analysis and design language standard[C] High Level Design Validation and Test Workshop. IEEE, 2016:30-39.

[24] Johnsen A, Lundqvist K, Pettersson P, et al. Automated verification of AADL-specifications using UPPAAL[C]. High-Assurance Systems Engineering (HASE), 2012:130-138.

[25] Franca R B, Bodeveix J P, Filali M, et al. The AADL behaviour annex -- experiments and roadmap[C] IEEE International Conference on Engineering Complex Computer Systems. IEEE Computer Society, 2007:377-382.

[26] Gautier T, Guy C, Honorat A, et al. Polychronous Automata and their Use for Formal Validation of AADL Models[J]. Frontiers of Computer Science, 2017.