Front. Comput. Sci., 2019, 13(4): 698–714
https://doi.org/10.1007/s11704-017-6477-y

RESEARCH ARTICLE

Template-based AADL automatic code generation

Kai HU[1], Zhangbo DUAN[1], Jiye WANG[2], Lingchao GAO[3], Lihong SHANG (✉)[1]

1 State Key Laboratory of Software Development Environment, Beihang University, Beijing 100083, China
2 State Grid Corporation of China, Beijing 100031, China
3 Beijing China-Power Information Technology Co., Ltd, State Grid Information & Telecommunication Group, Beijing 100192, China

Abstract Embedded real-time systems employ a variety of operating system platforms. Consequently, for automatic code generation, considerable redevelopment is needed when the platform changes. This results in major challenges with respect to the automatic code generation process of the architecture analysis and design language (AADL). In this paper, we propose a method of template-based automatic code generation to address this issue. Templates are used as carriers of automatic code generation rules from AADL to the object platform. These templates can be easily modified for different platforms. Automatic code generation for different platforms can be accomplished by formulating the corresponding generation rules and transformation templates. We design a set of code generation templates from AADL to the object platform and develop an automatic code generation tool. Finally, we take a typical data processing unit (DPU) system as a case study to test the tool. It is demonstrated that the auto-generated codes can be compiled and executed successfully on the object platform.

Keywords real-time system, template, formal methods, AADL, automatic code generation

1 Introduction

In the field of avionics, aerospace, and automotive control, embedded real-time systems [1] are widely used. These systems are resource-constrained, real-time responsive, fault-tolerant, and have dedicated hardware. They are highly in demand owing to their properties of real-time responsiveness and reliability. Moreover, they are becoming increasingly complex. Thus, how to reduce the development cost and time is an important consideration in the development of these systems. To simplify the development process, the model driven development (MDD) [2] method was proposed. Then, the architecture analysis and design language (AADL) [3–6], which is based on MDD, was invented.

AADL provides a standard and precise way of describing the software/hardware architecture, run-time environment, and functional and non-functional properties of embedded real-time systems. An AADL specification defines various types of software and hardware components (such as system, process, thread, subprogram, data, processor, and bus), their real-time properties (such as period, deadline, and worst-case execution time), 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. For example, AADL standard defines a behavior annex [7,8] to refine the behavior of threads. Currently, the automatic code generation of an AADL model to executable code is a research hotspot in AADL.

The automatic code generation of AADL [9], can contribute to improving the development automation level, shortening development time, and reducing the possibility of error in the coding process. The research of automatic code gener-

Received September 29, 2016; accepted March 8, 2017

E-mail: shanglh@buaa.edu.cn

ation of AADL primarily includes transformation rules from model to code, transformation methods, and tools.

The main contributions of this paper are as follows:

1) We design a set of code generation templates from AADL to object platform VxWorks.
2) We first introduce template-based automatic code generation technology as a carrier of generation rules from AADL to object code, and then develop a related tool.
3) For the AADL model of the DPU, objects can be generated by our tool, which can be compiled and executed correctly.

This paper is structured as follows. Section 2 reviews some related works. Section 3 presents AADL transformation rules of some AADL components. Section 4 shows the architecture of the template-based tool and template-based automatic code generation. Section 5 presents a case study. It takes the example of the data processing unit (DPU) to test the automatic code generation tool. Section 6 concludes this paper.

2 Related work

AADL can model a real-time system as a hierarchy of software components bound to an execution platform. Predefined software component types, such as thread, thread group, process, data, and subprogram are used to model the software architecture of the system. The 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 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 composite sets of software and execution platform components.

Based on the AADL model, there are several languages which can be generated, such as C, SystemC, and Lustre. This section provides a brief introduction to these works.

2.1 Code generation from AADL to C

PolyORB-HI-C is an AADL runtime used by the C code generator. It is developed by Ocarina for distributed high-integrity applications based on the Ada Ravenscar [10] profile. Two object languages are supported, namely, Ada2005 and C. It is used for system primitives and resource management. This piece of software is a link between the Ocarina generated code and the underlying runtime system.

The work [11] presents an experiment: code generation from a sub-part of AADL model to C code (compliant with a specific standard (OSEK/VDX)) using MDA tools. In the experiment, the authors focus on the AADL thread component. Two transformations generate either C language code (compliant with OSEK/VDX) and corresponding OIL configuration code.

The generated code can be executed on both POSIX or RTEMS platforms. With the PolyORB-HI-C tool, the Simulink blocks can be used as AADL subprograms. In that case, the generated code from AADL automatically calls the generated code from Simulink.

Ellidiss Software developed a real-time system analysis tool called STOOD [12]. It carries an AADL model on adjustable analysis by using simulation methods. It supports uniprocessor, multiprocessor, sharing resources, automatic code generation from AADL to C, C++, and Ada, and documentations like HTML, RTF, and MIF. It also supports reverse engineering from Ada or C code to AADL. The University of Electronic Science and Technology of China (UESTC) developed an automatic code generation tool from AADL to C based on the embedded real-time platform DeltaOS: UCaG [13].

2.2 Code generation from AADL to SystemC

SystemC is a set of C++ classes and macros which provide an event-driven simulation kernel in C++. SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis.

For the purpose of simulation, AADS [14] is developed as an AADL SystemC simulation tool. It supports the performance analysis of AADL specifications throughout the refinement process from the initial system architecture till the complete application and execution platform are developed. It parses the AADL model, so the functionality is translated to an equivalent POSIX model and the architecture is represented in XML.

The generation of the SystemC model from the AADL specification is not straight-forward. Nevertheless, the SystemC model generated by AADS is able to capture the fundamental dynamic properties of the initial system specification.

An AADL SystemC simulation tool is developed. However, the behavior specifications have not been taken into consideration.

2.3 Code generation from AADL to Lustre

AADL2SYNC is an AADL to synchronous programs translator, which is extended in the framework of the European project ASSERT, resulting in a system-level tool box that translates AADL to Lustre.

The main object is to perform simulation and validation that take into account both the system architecture and the functional aspects. Jahier et al. [15] study the transformation from AADL to synchronous language Lustre. In the synchronous model, all internal components step forward in a "simultaneous" way. Consequently, the model is deterministic. In contrast, AADL is a modeling language for a realistic platform, so the AADL model is asynchronous. The paper first presents methods to describe the 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 does not support the transformation of mode change or the behavior annex.

2.4 Code generation from AADL to TASM

The timed abstract state machine (TASM) [16] is based on the theory of abstract state machine (ASM) [17]. It extends ASM to enable expression of timing, resource, communication, composition, and parallelism. Technically, a TASM specification is made up of machines. A specification must hold at least one main machine with its set of rules. Beside main machines, it is also possible to define sub machines and function machines.

The work [18] proposes a formal semantics for the AADL behavior annex using TASM. A semantics of AADL execution model is given, and a prototype of behavior modeling and verification is proposed. Only the synchronous execution model is considered in [18]. It gives the relation between the execution model and the behavior annex.

Hu et al. [6] present a methodology for translating AADL to TASM. The authors 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).

The purpose of code generation from AADL to TASM is to perform formal verification. This verification is achieved with UPPAAL by mapping each main machine to a timed automaton. Timing correctness is defined as a reachable state of the system, being reachable within an acceptably bounded amount of time. Code generation from AADL to TASM allows presenting timing semantics of the AADL behavior annex using TASM, which is based on abstract state machines extended with resource consumption annotations. TASM offers more abstract resource consumption mechanisms, but does not offer much support to some scheduling patterns [19]. Resource management allows the specification of a fair preemptive scheduler. However, specifying other scheduling policies has to be evaluated [20].

2.5 Conclusion

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

1) A proper subset of AADL has been chosen as the transformation object including thread components (dispatching, offline scheduling, and execution), port communication, behavior annex, and mode change, which is usually used in safety-critical systems.

2) Template-based automatic code generation is first proposed. For different platforms, we can perform code generation by changing the templates.

3) We have developed an automatic code generation tool based on our method. This tool worked well for our clients.

3 Transformation rules

In this section, we will present transformation rules from AADL to C. We take VxWorks as an object platform and provide a group of transformation rules of AADL elements based on VxWorks. These subsets can basically satisfy the AADL modelling demand. The transformation rules will be described in texts, formal descriptions, and codes.

The AADL standard gives a rich syntactic and semantic,which can enhance expression ability. However, it also increases the complexity of code generation. We will illustrate how to frame the transformation rules using some AADL components as follows.

3.1 The transformation rules of process

An AADL process component can be described as follows:

Definition 1 $Process = \langle Pc, Sc, C, R \rangle$, where

- Pc denotes ports.

- Sc denotes subcomponents, and a subcomponent of a process can be a thread or subprogram.
- C denotes connections of $P \times P$, and a connection is used to check whether the port is used. If the port is not connected, the port may not be processed.
- R denotes the set of process features. It mainly contains scheduling and interrupt features.

The process abstraction represents a protected address space, a space partitioning that prevents other components from accessing anything inside the process. The address space contains:

- executable binary images (executable code and data) directly associated with the process;
- executable binary images associated with subcomponents of the process; and
- server subprograms (executable code) and data that are referred to by external components.

A process does not have an implicit thread. Therefore, to represent an actively executing component, a process must contain a thread.

The transformation rule of a process component is as follows: each process component is transformed into two files, processTypeID.h and processTypeID.c, in the systemTypeID folder. The content of processTypeID.h and processTypeID.c is presented in Table 1. Figure 1 shows the AADL description of process component, and Fig. 2 shows the C code in systemTypeID.c.

Table 1 Content of corresponding file

Filename	Content
processTypeID.h	Statements of functions and data shared by threads of process, references of systemTypeID.h
processTypeID.c	Implementation of data and function declarations

```
process processTypeID
end processTypeID;
process implementation
processTypeID.impl
end processTypeID.impl;
```

Fig. 1 AADL description of process component

3.2 The transformation rule of 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 queuing buffers, but data ports only keep the latest data. To transfer the data and events among components, port connections are provided. The definition is provided as below.

```
#include "systemTypeID.h"
int Fun_systemTypeID()
{
    /*create task*/
return 1;
}
```

Fig. 2 C code in systemTypeID.c

Definition 2 $PortConnection = \langle Sp, Dp, FT \rangle$.

- Sp is the source port of the connection.
- Dp is the object port of the connection.
- FT is the feature type of port connection. It contains the type of port (data port, event port or event data port) and features of port (such as data access and subprogram access).

In the following content, pc denotes the port connection; BA denotes the behavior annex; $t()$ denotes the transformation rule; $\Gamma()$ denotes valid identifier; $c()$ denotes creating; $s()$ denotes sending; and $r()$ denotes receiving. Transformation methods of different port connections are listed in Table 2.

We take the event port as an example, and give the formal description of the transformation rules for the event port as follows.

The transformation rule of event port:

1) Initialization:

- $\Gamma(pc)$:=FALSE.

2) Send:

- $!\exists(BA) \rightarrow s(c(Sig)) \wedge \Gamma(pc)$:=TRUE.
- $\exists BA(\Gamma(BA) = \text{FALSE} \rightarrow \Gamma(pc)$:=FALSE).
- $\exists BA(\Gamma(BA) = \text{TRUE} \rightarrow s(c(Sig)) \wedge \Gamma(pc)$:=TRUE).

3) Receive:

- $r(Sig) \wedge \Gamma(pc)$:= FALSE.

The valid identifier of a signal is declared in a corresponding header file and defaults to FALSE. When the signal is sent, it should be created.

Simultaneously, the valid identifier is set to be TRUE. However, if there is a behavior annex, then the valid identifier of the newly generated signal remains FALSE, and it will not be sent immediately. Only when the state transfer condition of the behavior annex is TRUE, then the newly generated signal will be sent and its valid identifier will become

Table 2 Transformation methods of different port connections

Type of port	Transformation resource	Transformation object	Transformation method
Event port	Event port	Signal	$t(Pc) = Sig$
	Out event port	Create and send a new signal	$t(Pc.out) = s(c(Sig))$
	In event port	Receive the signal	$t(Pc.in) = r(Sig)$
Even data port	Even data port	Message queuing	$t(Pc) = Mesq$
	Out event data port	Create and send a new message queuing	$t(Pc.out) = s(c(Mesq))$
	In event data port	Receive the message queuing	$t(Pc.in) = r(Mesq)$
Data port	Data port	Message	$t(Pc) = Mes$
	Out data port	Create and send a new message	$t(Pc.out) = s(c(Mes))$
	In data port	Receive the message	$t(Pc.in) = r(Mes)$

TRUE. When a signal is received, its valid identifier becomes FALSE.

Figure 3 shows the AADL description of event port, and Fig. 4 shows the C code of the event port after code transformation.

```
thread threadTypeID
  features
    iep: in event port;
    oep: out event port;
  properties
    Dispatch_Protocol => Periodic;
end threadTypeID;
```

Fig. 3 AADL description of event port

```
int TaskEntryPoint_threadTypeID()
{
ReceiveSignal(sigID_iep, ContentID);
hasSigID_iep = FALSE;
SIGNAL sigID_oep = CreateSignal();
SendSignal(sigID_oep, ContentID);
hasSigID_oep = TRUE;
    /*
    TODO task implementation
    */
    return 1;
}
```

Fig. 4 C code of event port

In the C code of event port in Fig. 4, TaskEntryPoint_threadTypeID() is the entry point of tasks. ReceiveSignal() and SendSignal() receive signal and send signal, respectively. sigID_iep and sigID_oep correspond to the signal ID of the input and output event ports, respectively. ContentID is the signal value. hasSigID_iep and hasSigID_oep identify if the input and output signals are valid.

3.3 The transformation rules of thread

The thread component, which is 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 dataor event exchange. The behavior annex is the refinement of the thread execution. The definition is shown as below.

Definition 3 $Thread = \langle IPort, OPort, TProp, BA, Mode, Scheduling\rangle$, where,

- $IPort = \langle IDP, IEP, IEDP\rangle$. IDP is the set of input data ports; IEP is the set of input event ports; and $IEDP$ is the set of input event data ports.
- $OPort = \langle ODP, OEP, OEDP\rangle$. ODP is the set of output data ports; OEP is the set of output event ports; and $OEDP$ is the set of output event data ports.
- $TProp = \langle Dispatch\_type, Period, Compute\_execution\_time, Deadline\rangle$. $Dispatch\_type$ is the dispatch protocol of the thread, including the periodic and aperiodic protocols. The period is the time interval between two dispatches of the periodic threads. $Compute\_execution\_time$ specifies the computation time of a thread. The deadline specifies the longest time interval between the start and end of an execution.
- $BA \in behavior\ annex$, and it is the precise description of the thread's behavior.
- $Mode$ 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 the system and process component of AADL, mode change automata is defined by specifying threads that are capable of execution.
- $Scheduling$ specifies how the thread will be scheduled on the CPU to which it is bound. The Allowed_Processor_Binding_Class property specifies the CPUs to which a thread can be bound when being executed.

Trigger conditions of different types of threads are listed in Table 3. We take the periodic thread as an example, and give the formal description of transformation rules for the periodic thread as follows.

Table 3 Trigger conditions

Thread types	Trigger causes	If there are requirements to arrival time of trigger causes
Periodic thread	Triggered periodically by dispatcher	\
Aperiodic thread	Triggered by the arrival of event or event data, subroutine calls	No
Sporadic thread	Triggered by the arrival of event or event data, subroutine calls	Yes

The transformation rules of periodic thread are as follows. Each thread component of the process component is transformed into two files threadTypeID.h and threadTypeID.c. The content of threadTypeID.h and threadTypeID.c are listed in Table 4.

Table 4 Content of corresponding file of thread component

Filename	Content
threadTypeID.h	Statements of functions and data in threads, references of processTypeID.c
threadTypeID.c	Implement of data and function declarations

We transform a thread component to a $Task = \langle Name, ThreadTask, Status \rangle$, then the formal description of transformation rules for the periodic thread is shown as follows:

- $!\exists(Mode) \rightarrow Task.Status := ready$.
- $\exists Mode(Mode = InitialMode \rightarrow Task.Status := ready)$.
- $\exists Mode(\neg(Mode = InitialMode) \rightarrow Task.Status := blocked)$.

Status is the status of a Task, and its default status is ready. If there is Mode in the process component, then only in initial mode, can task status be set to ready, otherwise it is blocked.

Figure 5 shows the AADL description of the periodic thread. Figure 6 shows the C code in processTypeID.c that corresponds to the periodic thread.

In the C code in processTypeID.c in Fig. 6, Fun_processTypeID()is the entry point of the process task. CreatePeriodicTask() refers to creating a periodic task. TaskName is the name of the task. TaskEntryPoint_threadTypeID() is the entry function of a task. Fun_threadTypeID() is the entry function of a thread task inside the process task. It means when the initialization of a process task is done, the thread task will be executed.

```
thread threadTypeID
end threadTypeID;
thread implementation threadTypeID.impl
properties
    Dispatch_Protocol => Periodic;
    Period=>100ms;
end threadTypeID.impl;
```

Fig. 5 AADL description of periodic thread

```
#include "processTypeID.h"
int Fun_processTypeID()
{
CreatePeriodicTask(TaskName,TaskEntryPoi
nt_threadTypeID,status);
return 1;
}
int TaskEntryPoint_ threadTypeID()
{
Fun_threadTypeID();
return 1;
}
```

Fig. 6 C code in processTypeID.c

3.4 The transformation rule of behavior annex

Behavior annex, taking the form of a state machine, is defined in the thread for the purpose of describing thread behavior such as port communication, computation, and delay. The definition is shown as below.

Definition 4 *Behavior annex* $= \langle S, S0, V, Guard, Action, T \rangle$, where,

- S is the set of states. Types of the state include initial, return, complete, and composite.
- $S0 \in S$ is the initial state.
- V is the set of local variables.
- *Guard* is the set of guards of state transitions, taking the form $\langle BExpr \rangle | [on \langle BExpr \rangle - \rangle] \langle event \rangle [when \langle BExpr \rangle]$, in which $BExpr$ is the predicate on state variables; event is the data or event reading operation $(P?; P?(x))$; when the predicate on the data is received from the event or the event data ports.
- 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.

The execution rule of a transition in the behavior annex is as follows:

$$(Guard = TRUE) \rightarrow \{(S0 \rightarrow St) \wedge (Excute(Action))\}.$$

When the guard of behavior annex is TRUE, the transition takes place. At this time, the original state will jump to the destination state, while the action of the behavior annex will be executed.

Now, we will describe the transformation rules of the behavior annex from three separate aspects: variables in behavior annex, guards of transition, and actions of transition.

1) The transformation rule of variables in behavior annex

Variables in behavior annex are transformed into local variables in a task. For example, the corresponding C code of variable *x* is shown below:

```
int x;
```

Implementation of the transformation rule of variables in the behavior annex: Declarations and definitions of local variables are in the function of a task.The built-in data types (such as Integer) in the behavior annex correspond to the standard data types (e.g., int) in C. Assignments and calculations of variables are specified in the action of transition in the behavior annex.

2) The transformation rule of transition guards

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.

- $(thState = waiting\_executing \wedge thBAstate = ss \wedge tr.Guard) | (thState = execution \wedge thBAstate = ss) \rightarrow G(execution\_tr)$.

If Delay is defined, the thread blocks itself by default. Thus there is no need to waste CPU resources and the condition is

- $(thState = waiting\_execution \wedge thBAstate = ss \wedge tr.Guard)$.

Otherwise, the condition is

- $(thState = execution \wedge thBAstate = ss)$.

Whether the thread can get a CPU and be transitioned into the execution state is judged in the scheduler.Thus the guard condition tr.Guard will be used in the scheduler.

As defined before, *tr.Guard* takes the form:

- $\langle BExpr \rangle | [on \langle BExpr \rangle - \rangle] \langle event \rangle [when \langle BExpr \rangle]$.

It can be a simple boolean expression ($\langle BExpr \rangle$), and the arrival of an event port ($\langle event \rangle$) or a complicated composition of them containing the check on the input data or event ($[when \langle BExpr \rangle]$). BExpr can be translated directly to the logic expression of C. *Event* is used to represent that the transition is enabled by the arrival of an 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 that newly arrived on the port, a translation rule taking the form $when.BExpr(event.p)$ in which $BExpr(event : p)$ means the logic judgment on the data.

3) The transformation rule of transition actions

There are three actions that need to be performed in a state transition: (a) execute actions defined in the 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 C. Port writing operations like $p!$ or $p!(x)$ is translated by the rule defined in Section 3.2.

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 the state waiting_execution and the object state is assigned as the destination state *ds*. Translation rules are presented in Table 5.

Table 5 Transition actions

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

Figure 7 shows the AADL description of the behavior annex. Figure 8 shows the C code of the transition guard after code transformation. Figure 9 shows the C code of the transition action after code transformation.

3.5 The transformation rules of mode change

In AADL, a mode represents an operational state which can be viewed 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 which cause a mode transition and new mode creation. The definition of mode change is shown as below.

```
thread implementation threadTypeID.impl
  subcomponents
    d1: data integer;
    d2: data string;
  calls{
    spClient: subProgram spClient;
    };
  annex behavior_specification{**
      variables
        x: Integer;
      states
        s0: initial state;
        s1: final state;
      transitions
        s0 -[iep01?]-> s1 { d1 := 10; x :=
d1;};
        s1 -[iep10?]-> s0 {spClient!(x)};
     **};
end threadTypeID.impl;
```

Fig. 7 AADL description of behavior annex

```
int TaskEntryPoint_threadTypeID()
{
hasSignal(hasSigID_iep01)
{

hasSignal(hasSigID_iep10){/*action*/}
}
    return 1;
}
```

Fig. 8 C code of transition guard

```
int TaskEntryPoint_threadTypeID()
{
hasSignal(hasSigID_iep01)
{
    d1 = 10;
    x = d1;
    hasSignal(hasSigID_iep10)
    {
        spClient(x); }
}
    return 1;}
```

Fig. 9 C code of transition action

Definition 5 $Mode\ Change = \langle M, m0, Event, Transition\rangle$, 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, then 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.

A mode change state machine is translated into a state machine which is a three tuple $\langle SomMV, SomCV, SomMM\rangle$. $SomMV$ and $SomCV$ are respectively the monitored and controlled variables and $SomMM$ is the set of state transitions. The definition of $SomMV$ and $SomCV$ are shown below.

- $SomMV = \{currentMState, nextHyperperiod, event\}$.
- $SomCV = \{currentMState, nextHyperperiod, event, \cup_{th\in TH} hyperperiod, \cup_{th\in TH} thActive, \cup_{th\in TH} numPeriod, \cup_{th\in TH} syncDis\}$.

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 a mode change. Variable *event* is used to trigger mode change. For every thread $th \in TH$ (TH is the set of threads 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 notify the dispatcher whether the thread can enter into the 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 $mtr = \langle sms, event, dms\rangle$, four state transitions are 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.

- $h_{in}(sms, dms) \rightarrow T(Hyperperiod)$.
- $currentMState = sms \wedge nextHyperperiod = true \rightarrow G(Hyperperiod)$.
- $nextHyperperiod := false \rightarrow A(Hyperperiod)$.

2) State transition *Has_mcr* is executed if the mode change event arrives at the synchronization point of the 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 the old mode and NTH is the set of threads in the new mode.

- $0 \rightarrow T(Has\_mcr)$.
- $currentMState = sms \wedge nextHyperiod = false \wedge event = true \rightarrow G(Has\_mcr)$.
- $(event := false \otimes nextHyperperiod := true \otimes currentMState := smsinprogress \otimes (\otimes_{th \in OTH} thActive(th) := false) \otimes (\otimes_{th \in NTH} thActive(th) := true) \otimes (\otimes_{th \in OTH. \wedge th \in NTH} hyperperiod(th) := h_{out}(sms, dms)/th.period) \rightarrow A(Has\_mcr)$.

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 notify the dispatcher to enter the next cycle of hyper period.

- $0 \rightarrow T(Hasnot\_mcr)$.
- $currentMState = sms \wedge nextHyperperiod = false \wedge event = false \rightarrow G(Hasnot\_mcr)$.
- $nextHyperperiod := true \otimes (\otimes_{th \in OTH} syncDis := true) \rightarrow A(Hasnot\_mcr)$.

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 a new mode. Actions of the transition are the assignment of *curretMState* to new mode *dms* and notifying the dispatcher of threads in new mode as to enter the next cycle of the hyper period.

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

3.6 A discussion about transformation rules

How to prove that the transformation itself preserves the intended semantics of AADL model in the first place or, at least, some of the specific properties or requirements it needs to satisfy is another important research project.

At present, in the AADL research field, the semantic preservation of AADL model transformation is mainly implemented by manual validation, or the semantics is assumed to be preserved. Therefore, semantic preservation is still an open research problem.

IRIT lab [21] in France characterizes the AADL metamodel by using the B-language and Higher-Order Logic (HOL) to support the semantic preservation of AADL model transformation in the future. In [22], Event B is used to describe the semantics of an AADL subset (periodic thread, data port). Moreover, a simple proof of the semantic preservation is given.

In the industrial sector, TOPCASED [23] of Airbus explicitly proposes to demonstrate correctness of the AADL model transformation; ASSERT of EU proposes Proof-Based System Engineering; SPaCIFY also proposes to prove the semantic preservation of AADL model transformation.

In our previous paper [24], we have studied a method of verifying transformation rules. We present a machine checked semantics-preserving transformation of a subset of AADL into timed abstract state machines (TASM). We have considered a formal proof of semantics preservation of the transformation. The theorem prover Coq is used to prove the methodology, i.e., the correctness of the transformation rules.

By using the same methodology, we can enable the proof of semantics preservation of the transformation rules from AADL to C in three steps: (1) the informal execution semantics formalized directly using Timed Transition System (TTS), is considered as a reference semantics, because we cannot directly prove that the translational semantics is equivalent to the informal one which is provided by the AADL standard; (2) combining the translational semantics (expressed by C code) with the semantics of C, we can obtain another way to execute the AADL model, and it is constructed as another TTS; (3) the reference semantics is supposed to be correct, and if there is a simulation equivalence relation between the two TTSs, we say the translation preserves the AADL semantics.

However, the main focus of this paper is on the transformation rules. This method of proving the correctness of trans-

formation rules creates a huge workload. We will not repeat them here.

4 Template-based automatic code generation tool

Our AADL automatic code generation tool uses the template-based automatic code generation technology.

The overall implementation of code generation process is shown in Fig. 10. The metamodel of AADL defines the specification that describes an AADL model, which can be elements that make up the model and the relationships between these elements. First, the model information will be extracted and packaged by analyzing AADL model. Then we will design the template according to the transformation rules of the object platform. Finally, we will integrate the model information and templates by using a template engine to complete code generation. In conclusion, this tool should contain a template design block, a model analysis block, and a template engine block. After the generation of an object, by using the project transform block, the object can be transformed to an object project. It is convenient for compiling and debugging.

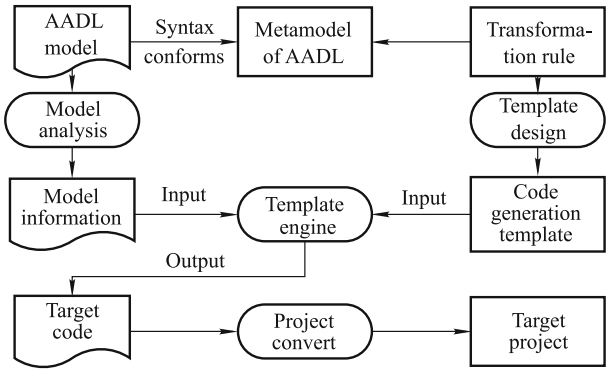

Fig. 10 Code generation framework

4.1 Template design

A template is composed of static text and a placeholder. Static text is the direct output text. Placeholder is the replacement of model information. Template is actually the realization of transformation rules. Automatic code generation template uses Xtend language. Xtend language is a JVM script language, supporting static template expression. Xtend is compiled to Java language and then run on a Java virtual machine.

Xtend templates allow a readable string connection. Template content is surrounded by three single quotes (''' '''). Inside the template we use French quotes (《 》) to handle the insert expression. Template expression supports the judgment statement of 《IF ... 》... 《ENDIF》, the loop statement of 《FOR ... 》... 《ENDFOR》 andswitch statement. Meanwhile, it also supports extension methods, and changes the variable without changing the method of a variable. The switch-case statement sample of the template expression is shown in Fig. 11. The IF and FOR statement sample of the template expression is shown in Fig. 12.

```
def toText(Node n){
 switch n{
    Contents : n.text
    A : '''
        <a href="«n.href»">
        «n.applyContents»
        </a>
    '''
    default : '''
        < «n.tagName» >
        «n.applyContents»
        </ «n.tagName» >
    '''
  }
}
```

Fig. 11 Switch-case template expression

```
def
someHTML(List<Paragraph>paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs»
        «IF p.headLine != null»
          <h1>«p.headLine»</h1>
        «ENDIF»
        <p>«p.text»</p>
      «ENDFOR»
    </body>
  </html>
'''
```

Fig. 12 IF and FOR template expression

Meanwhile, in order to make template operation more flexible, we encapsulate some common file operations and character processing methods which can be used in template expression, whereas character processing methods are provided as extension methods. Figure 13 is an example of a process template.

The Xtend-based AADL automatic code generation tool contains system template, sub-component template, process template, thread template, thread group template, subprogram template, subprogram group template, data template, feature template, connection and flow template, and property

```
def static template(ThreadClassifier thread){
    switch thread{
        ThreadType :'''
            /*Thread Ttoe c file*/
            «Template.head»
            #include "«Template.systemheadfile»"
            void «thread.name.convert»()
            {
                /*features in thread*/
                «IF thread.getAllFeatures!=null»
                    «thread.getAllFeatures.asVariables»
                «ENDIF»
                «IF thread.dispatchProtocol.equalsIgnoreCase("Periodic")»
                    while(1)
                    {

                    }
                «ENDIF»
            }
        '''
        ThreadImplementation : '''
            /*Thread Implementation c file*/
```

Fig. 13 Example of a process template

processing template. Next, we will give a detailed description of the design and implementation of the process template.

Template design of a process component is the core content of design and implementation of a template in the whole project. Process component is the second layer in the AADL model, which means that the process component is the direct subcomponent of the system component. It is a protected address space. According to code transformation rules, as a collection of all tasks in a system or subsystem, a process component corresponds to a function in C language. In order to block the generated code and take into account its reusability, each process component will generate two C files, ComponentName.h and ComponentName.c.

Xtend language supports the downcast of AADL model. Therefore, it is necessary to call the classifier of the process component template for the judgment, and the handling according to the classifier category.

Process classifier will classify the process into process type and process implementation. Process type is used to define some of the features that are visible outside the component, which are defined in the external characteristics of components, such as interface, and data. By using subcomponent, connection, and other aspects, process implementation implements the internal structure of a process, such as the definition of thread components to realize task creation, management, and the correspondence between the process and thread ports through the ports' connection.

When an AADL process component is judged as a process type by the template engine, the generated ProcessName.h file will mainly include the following aspects:

1) The definition of system header files, including the header files associated with operation, such as a clock timer or a task management. System header files definition processes still need to be used in other components (such as thread component and subprogram component). Thus, for convenience of code reuse, we will write a template file called "Template" to add header files.

2) Definition of related initialization function, including ProcessName_init(), which is used to initialize process schedule, and ProcessName_init_task(), which is used to implement the function schedule() and to initialize task management function.

Methods that are invoked in the file generated by a process component include the following aspects:

1) Process initialization function ProcessName_init(). In this function, process schedule parameters are initialized. All thread components of the process component are defined as tasks. Their definitions are added to the system header file SystemName.h. And their schedule parameters are initialized.

2) Task management initialization function ProcessName_init_task(). This function is mainly used to create tasks for thread components inside the process component, and schedule them according to the schedule function.

3) Scheduling function schedule(). This function will generate scheduling code according to the scheduling algorithms

inside the processor. While the scheduling algorithms inside the processor are determined by task scheduling mode provided by the operating platform. The actual operation of task scheduling requires users to call upon the interfaces of an operating system. The main schedule methods: Round-Robin Schedule, makes all ready tasks with same priority equally share the CPU; Priority-based preemptive scheduling, which sorts according to the priority of each task, and it will ensure that the tasks with high priority complete earlier.

4) The behavior annex. Firstly, by traversing we will get the maximum number of thread state. Secondly, state transition can be parallel.

For example,

$s0$ -[guard1]-> $s1${action1;};

$s0$ -[guard2]-> $s1${action2;};

$s1$-[guard3]->$s2${action3;};

$s1$ -[guard4]->$s2${action4;}.

The transition from $s0$ to $s1$ and the transition from $s1$ to $s2$ both have two possibilities. The transition from $s1$ to $s2$ should be generated under the transition from $s0$ to $s1$. When all transitions from $s1$ to $s2$ are done, we will move to the next transition from $s0$ to $s1$. In addition, we also have to consider the case for empty guard or action.

4.2 Model analysis

The AADL standard provides AADL text, XML, and graphical representation. Our automatic code generation tool does the model analysis based on XML representation of AADL (.aaxl file).

The AADL meta-model defines the structure of AADL models. This meta-model is represented as a set of class diagrams with additional EMF-specific properties, that support the automatic generation of methods for manipulation of AADL object models. AADL defines seven EMF Ecore meta-model packages: core, component, feature, connection, flow, instance, and property.

EMF (Eclipse Modeling Framework) is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

We will use Ecore meta-model language to describe AADL model. According to Ecore meta-model of AADL, Java objects of AADL elements will be generated. Meanwhile, the package of API will be provided to get access to AADL elements. At the same time, we use Ecore meta-model language to describe the AADL instance, and generate Java objects corresponding to the AADL instance. aadl.ecore is the basis of instance.ecore. Ecore meta-model description of AADL is shown in Fig. 14.

The construction of Ecore meta-model is the foundation of the AADL development environment. Through the construction of Ecore meta-model, the automatic serialization interface is generated. The Ecore model is essentially a subset of UML, which includes:

- EClass: domain class. It has a name, one or more attributes, and one or more references.
- EAttribute: attribute of domain class. It has a name and a type.
- EReference: relationship between domain classes. It has a name, a Boolean value indicating if it is included, and a reference to object class.
- EDataType: attribute types, such as int or object types.

Through the model analysis, we get a tree storage structure of the AADL model, whose root node is System Implementation. Through achieving the root node, we can traverse all elements of the model.

4.3 Template engine and project transformation

Template engine puts templates and model information as inputs to generate the object. It can be extended and replaced by different template files to generate objects of different languages or different platforms.

We implement the project transformation in order,so that the generated code can be directly compiled and debugged. Thus, we can improve the degree of automation of the automatic code generation tool. The project transformation needs to integrate the compiler environment of object platform. After the generation of an object, an object platform project will be created, and the object will also be set under the object project.

The project transformation must use org.eclipse.ui.menus, commands and handlers to be extended. The use of these extension points must obey the development standard of Eclipse. At the same time, we have to implement project transformation interface. Interface description is presented in Table 6.

5 Case study

We conducted the experiment with a classic AADL model of

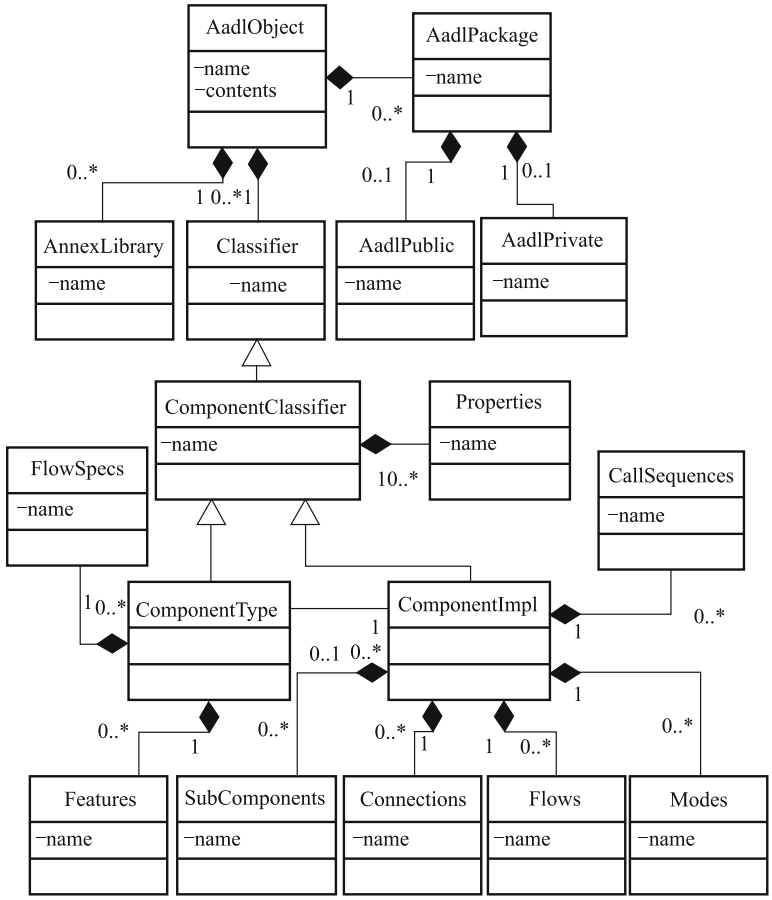

Fig. 14 Ecore meta-model description of AADL

the DPU (data processing unit) to test correctness of our automatic code generation tool. The function of the DPU is to collect the data from two mechanical gyros and three fiber-optic gyros and then send it to the main computer. We have built an AADL model of the DPU. This model of the DPU will be transformed into C code by our tool. Finally, we will run and analyze the code on VxWorks.

Table 6 Project transformation interface

Name	Description
preCprjCreate()	The operation before creating a project
postCprjCreate(IProjectcPrj)	The operation after creating a project, such as income project entity parameters

5.1 AADL model of DPU

The DPU model includes eight sub-tasks: main task, synchronous interrupt handing task, serial port interrupt handing task, two mechanical gyro sampling tasks, and three fiber-optic gyro sampling tasks. Main task is used for initialization of the DPU. There are two kinds of interrupt in DPU: synchronous interrupt and serial port interrupt. They are used for the DPU clock synchronization and the processing of the serial port instruction. Synchronous interrupt handing task will be called when a synchronous interrupt signal arrives. Similarly, serial port interrupt handing task will be called when serial port interrupt signal arrives. There are three input/output interfaces in the DPU: the synchronous interrupt handing task interface, the serial port interrupt handing task interface, and the output data interface.

The AADL model of the DPU is shown in Fig. 15. Task unit of DPU is represented by a process component of AADL: pDpu, which includes five data components (N1, N2, N3, N4, N5), and eight thread components (main, sync, serial, gyro1, gyro2, fog1, fog2, fog3). Sync sends signal to gyro1 and gyro2 when it receives a synchronous signal; then gyro1 and gyro2 start collecting data; the collected data are stored in data components N1 and N2. Serial sends signals to fog1, fog2, and fog3 when it receives serial port signal; then fog1, fog2, and fog3 start collecting data; collected data is stored in data components N3, N4 and N5. After collecting the gyro data, serial will send signal to main, so that main will output the collected data.

Inside pDpu, we also declare a mode change between sync

and serial. Synchronous interrupt handing task interface and serial port interrupt handing task interface are represented by an event port. Output data interface is represented by output data port. We use mode change of the AADL to represent interrupt: the two modes are the synchronous mode and the serial port mode. When a synchronous signal arrives, the synchronous mode is activated as the initial mode. When a serial port signal arrives, mode change occurs. The Synchronous mode transforms to the serial port mode. The whole DPU is represented by a system component. Five device components represent two mechanical gyros and three fiber-optic gyros. One processor component represents the Intel processor. One memory component represents the memory. One system bus component connects the processor component with the memory component. One LAN bus component mounted on a system bus component connects the five gyros.

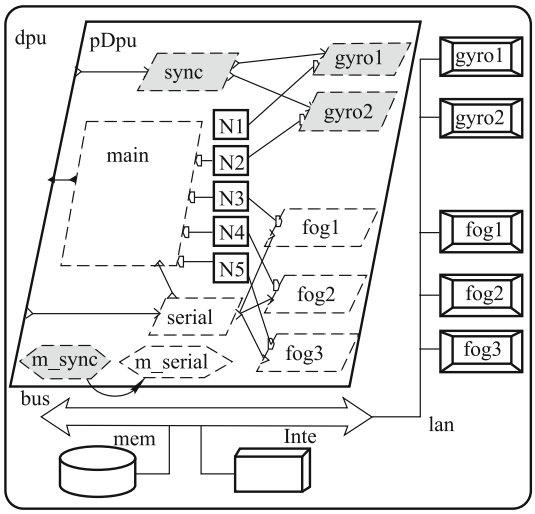

Fig. 15 AADL model of DPU

5.2 Automatic code generation from the AADL model of the DPU to C

First of all, we instantiate the model, package it into a tree structure, then choose the instantiation file of model to generate the object by choosing automatic code generation. Now we can obtain a project. At the same time, we can find the configuration files and object code files in the project directory (shown in Fig. 16).

In the corresponding folder of the system component, each component corresponds to a set of files (.h files and .c files). .h file contains the declaration of variables and functions, .c file includes the initialization of variables and implementation of functions. System component sDpu generates a folder sDpu containing files sDpu.h and sDpu.c. Process components generate two files pDpu.h and pDpu.c. The eight thread components generate corresponding .c and .h files. Meanwhile, the generated code will be transformed to a project in the object platform. 0.S, ldsConfig.xml, and linkscript.lds are project configuration management files. The relation between AADL model of DPU and generated code files is presented in Table 7.

Fig. 16 Generated code files

Table 7 Relationship between AADL model of DPU and generated code files

AADL component of DPU	Generated files
System component sDpu	Folder sDpu, sDpu.h, sDpu.c
Process component pDpu	pDpu.h, pDpu.c
Thread component sync	sync.h, sync.c
Thread component gyro1	gyro1.h, gyro1.c
Thread component gyro2	gyro2.h, gyro2.c
Thread component serial	serial.h, serial.c
Thread component fog1	fog1.h, fog1.c
Thread component fog2	fog2.h, fog2.c
Thread component fog3	fog3.h, fog3.c
Thread component main	main.h, main.c

5.3 Execution of generated code

To ensure the platform-independence of generated code, our automatic code generation tool generates intermediate code. We take VxWorks as test platform. The generated code is implemented in VxWorks (included in targetVxworks.h and targetVxworks.c). targetVxworks.h contains the preprocessing declarations. targetVxworks.c contains the implementation of intermediate code. Some of the code in targetVxworks.c is shown in Fig. 17.

We compile the generated intermediate code with tar-

getVxworks.h and targetVxworks.c in a tornado environment and execute in a tornado emulator.

Result is shown in Fig. 18. Fun_pDpu is entry function of the process. The eight tasks which correspond to the eight thread components of DPU are generated. When we run the i command to see the status of tasks, all threads are in pend state because these threads did not receive any activation signal. sync, gyro1 and gyro1 are activated by p_sync; serial, fog1, fog2, fog3 and main are activated by p_serial.

In Shell, we input a command to simulate the sending of a synchronous signal: semGive(sem_p_sync). The execution result is shown in Fig. 19. Semaphore sem_p_sync correspond to p_sync in the DPU model. semGive(sem_p_sync) means that p_sync is issued. sync is activated when it receives p_sync. Then gyro1 and gyro2 start collecting data. Collected data of gyro1 is stored in N1. After execution of gyro1, value of N1 changes from 0 to 1. Collected data of gyro2 is stored in N2. After execution of gyro2, value of N2 changes from 0 to 2. When we run the command i, we can see that sync, gyro1, and gyro2 are complete. The other five tasks are waiting for p_serial.

```
int CreatePeriodicTask(char* taskName,
FUNCPTR entryPt, char* status){
   int taskId;
   taskId = taskSpawn(taskName, TASK_PRI,
VX_FP_TASK, 4000, (FUNCPTR)entryPt,
0,0,0,0,0,0,0,0,0,0);
   if(taskId == ERROR){
      printf("Spawn task %s failed!\n",
taskName);
   }
   else{
      printf("Spawn task %s(%d)
successed!, task state is %s\n", taskName,
taskId, status);
   }
   return taskId;
}
```

Fig. 17 Part of the code in targetVxworks.c

In Shell, we input a command to simulate sending a serial signal: semGive(sem_p_serial). The execution result is shown in Fig. 20. Semaphore sem_p_serial correspond to p_serial in the DPU model. semGive(sem_p_serial) means that p_serial is issued. serial is activated when it receives

```
->Fun_pDpu
Spawn task      Task_sync(70902976) successed!,tast  state is ready
Spawn task      Task_gyro1(70782816) successed!,task  state is ready
Spawn task      Task_gyro2(70765672) successed!,task  state is ready
Spawn task      Task_seria1(70748528) successde!,task  state is pend
Spawn task      Task_fog1(70731384) successed!,task state  is pend
Spawn task      Task_fog2(70714240) successed!,task  state is pend
Spawn task      Task_fog3(70697096) successed!,task state  is pend
Spawn task      Task_main(70679952) successde!,task state  is ready
value  = 1 = 0x1  = __major_image_version__
->i
  NAME         ENTRY      TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- ------- -------- ------- -----
tExcTask    _excTask     4398de0  0 PEND        408358 4398ce0      0     0
tLogTask    _logTask     43932b0  0 PEND        408358 43931b0      0     0
tWdbTask    _wdbTask     438e668  3 READY       408358 438e518      0     0
Task_sync   _TaskEntryPo 439e4c0150 PEND        408358 439e3d4      0     0
Task_gyro1 _TaskEntryPo 438f60 150 PEND        408358 4380e74      0     0
Task_gyro2 _TaskEntryPo 437cc68150 PEND        408358 437cb7c      0     0
Task_serig1_TaskEntryPo 4378970150 PEND        408358 4378884      0     0
Task_fog1   _TaskEntryPo 4374678150 PEND        408358 437458c      0     0
Task_fog2   _TaskEntryPo 4370380150 PEND        408358 4370294      0     0
Task_fog3   _TaskEntryPo 436c088150 PEND        408358 436bf9c      0     0
Task_main   _TaskEntryPo 4367d90150 PEND        408358 4367ca4      0     0
value  = 0  = 0x0
```

Fig. 18 Execution result of generated code in VxWorks

```
-> N1
_N1 = 0x41d427c:value= 0 = 0x0
-> N2
_N2 = 0x41d42cc:value= 0 = 0x0
-> semGive(sem_p_sync)
value= 0 = 0x0
-> N1
_N1 = 0x41d427c:value= 1 = 0x1 = __major_image_version__
-> N2
_N2 = 0x41d42cc:value= 2 = 0x2 = __subsystem__
-> i
  NAME         ENTRY        TID    PRI   STATUS    PC      SP     ERRNO DELAY
---------- ------------ -------- --- ---------- ------- -------- ------ -----
tExcTask    _excTask      4398de0   0 PEND      408358 4369ce0     0     0
tLogTask    _logTask      43932b0   0 PEND      408358 43671b0     0     0
tWdbTask    _edbTask      438e667   3 READY     408358 438e518     0     0
Task_serial_TaskEntryPo  4348970 150 PEND      408358 4378884     0     0
Task_fog1   _TaskEntryOPo 4374678 150 PEND      408358 437458c     0     0
Task_fog2   _TaskEntryPo  4370380 150 PEND      408358 4370294     0     0
Task_fog3   _TaskEntryPo  436c088 150 PEND      408358 436bf9c     0     0
Task_main   _TaskEntryPo  4367d90 150 PEND      408358 4367ca4     0     0
value= 0 = 0x0
```

Fig. 19 Execution result after sending synchronous signal

```
-> N3
_N3 = 0x3f9465c: value = 0 = 0x0
-> N4
_N4 = 0x3f9469c: value = 0 = 0x0
-> N5
_N5 = 0x3f9461c: value = 0 = 0x0
-> semGive(sem_p_serial)
value = 0 = 0x0
-> N3
_N3 = 0x3f9465c: value = 0 = 0x0 = __subsystem__+ 0x1
-> N4
_N4 = 0x3f9469c: value = 4 = 0x4 = __major_os_version__
-> N5
_N5 = 0x3f9461c: value = 5 = 0x5 = __major_os_version__+ 0x1
-> i
  NAME        ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- -------- -------- ------- -----
tExcTask   _excTask     4158de0   0 PEND       408358  4158ce0       0     0
tLogTask   _logTask     41532b0   0 PEND       408358  41531b0       0     0
tWdbTask   _edbTask     414e668   3 PEADY      408358  414e518       0     0
value = 0 = 0x0
```

Fig. 20 Execution result after sending serial signal

p_serial. Then fog1, fog2 and fog3 start collecting data. Collected data of fog1 is stored in N3. After execution of fog1, value of N3 changes from 0 to 3. Collected data of fog2 is stored in N4. After execution of fog2, value of N4 changes from 0 to 4. Collected data of fog3 is stored in N5. After execution of fog3, value of N5 changes from 0 to 5. When we run the command i, we can see that serial, fog1, fog2, fog3, and main havecompleted.

6 Conclusion

Currently, automatic code generation is a critical application in industry. However, it would be more helpful if cross-platform generation technologies are developed. In this paper, we propose a template-based AADL automatic code generation method. It can make multi-platform automatic code generation easier by modifying templates. Templates are implementations of transformation rules. For different platforms, we formulate different transformation rules and templates. Furthermore, we also develop a tool and test it using DPU. Our tests show that generated codes can be compiled and executed successfully.

However, for different objects, writing new templates will still create a large workload. Therefore, for the next step, we will integrate the template-based automatic code generation with the middle ware-based automatic code generation and perform the classification for the same kind of object platform. By generating middleware code, we can realize code transformation from the middleware code to the object platform in order to better support the multi-platform code generation.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 61672074 and 91538202), Project of the State Key Laboratory of Software Development Environment of China (SKLSDE-2016ZX-16).

References

1. Hu K, Zhang T, Yang Z, Tsai W T. Simulation of real-time systems with clock calculus. Simulation Modelling Practice & Theory, 2015, 51: 69–86
2. Lewis B. Architecture based model driven software and system development for real-time embedded systems. Lecture Notes in Computer Science, 2004, 2941: 249–260
3. Hu K, Lei L, Tsai W T. Multi-tenant Verification-as-a-Service (VaaS) in a cloud. Simulation Modelling Practice & Theory, 2016, 60: 122–143
4. SAE AS5506. Architecture Analysis and Design Language (AADL). SAE International, 2005
5. SAE AS5506A. Architecture Analysis and Design Language (AADL) Standard, Version 2.SAE International, 2008
6. Hu K, Zhang T, Yang Z, Tsai W T. Exploring AADL verification tool through model transformation. Journal of Systems Architecture, 2015, 61(3–4): 141–156
7. SAE AS5506 Annex: Behavior Specification v2.0. 2011
8. Franca R B, Bodeveix J P, Filali M, Rolland J F. The AADL behaviour annex – experiments and roadmap. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems. 2007, 377–382
9. Hu K, Zhang T, Yang Z. Multi-threaded code generation from Signal program to OpenMP. Frontiers of Computer Science, 2013, 7(5): 617–626
10. Lundqvist K, Asplund L, Mitchell S. A formal model of the Ada Ravenscar tasking profile; protected objects. In: Proceedings of the International Conference on Reliable Software Technologies. 1999, 12–25
11. Brun M, Delatour J, Trinquet Y. Code generation from aadl to a real-time operating system: an experimentation feedback on the use of model transformation. In: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems. 2008: 257–262
12. Dissaux P, Singhoff F. Stood and cheddar: AADL as a pivot language for analysing performances of real time architectures. In: Proceedings

of the European Real Time System Conference. 2008
13. Tao Y. Model verification and code generation technology of AADL. Chengdu: University of Electronic Science and Technology of China, 2009
14. Varona-Gomez R, Villar E. AADL simulation and performance analysis in SystemC. In: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems. 2009: 323–328
15. Jahier E, Halbwachs N, Raymond P, Nicollin X, Lesens D. Virtual execution of AADL models via a translation into synchronous programs. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software. 2007, 134–143
16. Ouimet M, Lundqvist K, Nolin M. The timed abstract state machine language: an executable specification language for reactive real-time systems. In: Proceedings of the 15th International Conference on Real-Time and Network Systems. 2007
17. Börger E, Stärk R. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Science & Business Media, 2012
18. Yang Z, Hu K, Ma D, Pi L. Towards a formal semantics for the AADL behavior annex. In: Proceedings of the Conference on Design, Automation and Test in Europe. 2009, 1166–1171
19. Pi L, Yang Z, Bodeveix J P, Filali M, Hu K, Ma D. A comparative study of FIACRE and TASM to define AADL real time concepts. In: Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems. 2009, 347–352
20. Pi L, Bodeveix J P, Filali M. A comparative study of different formalisms to define AADL data communication. Seminaire, 2009
21. Bodeveix J P, Chemouil D, Filali M, Strecker M. Towards formalising AADL in proof assistants. Electronic Notes in Theoretical Computer Science, 2005, 141(3): 153–169
22. Filali-Amine M, Lawall J. Development of a synchronous subset of AADL. In: Proceedings of the International Conference on Abstract State Machines, Alloy, B and Z. 2010, 245–258
23. Farail P, Gaufillet P, Canals A, Camus C L, Sciamma D, Michel P, Crégut X, Pantel M. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. European Congress Embedded Real Time Software, 2006, 781(55–59): 82
24. Yang Z, Hu K, Ma D, Bodeveix J P, Pi L, Talpin J P. From AADL to timed abstract state machines: a verified model transformation. Journal of Systems and Software, 2014, 93: 42–68

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

Zhangbo Duan received his BE degree from Beihang University, China in 2015. In the undergraduate period, he is sent to Institut National des Sciences Appliquées (INSA) Toulouse for an exchange study by China Scholarship Council. He is now a graduate student at Beihang University. His research interests include BlockChain, AADL and formal methods.

Jiye Wang is currently a professor-level senior engineer. His research interests include information management of power systems, smart grid, information security and the next generation energy system.

Lingchao Gao a currently a senior engineer. His research interests include smart grid, big data analytics, cloud computing and blockchain.

Lihong Shang is an associate professor at Beihang University, China. He received his PhD degree from Beihang University in 2001. His research interests include embedded systems and avionics system.