An Automatic Development Process for Integrated Modular Avionics Software

Ying Wang
State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
wangying@act.buaa.edu.cn

Dianfu Ma
State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
dfma@buaa.edu.cn

Abstract—With the ever-growing avionics functions, the modern avionics architecture is evolving from traditional federated architecture to Integrated Modular Avionics (IMA). ARINC653 is a major industry standard to support partitioning concept introduced in IMA to achieve security isolation between avionics functions with different criticalities. To decrease the complexity and improve the reliability of the design and implementation of IMA-based avionics software, this paper proposes an automatic development process based on Architecture Analysis & Design Language. An automatic model transformation approach from domain-specific models to platform-specific ARINC653 models and safety-critical ARINC653-compliant code generation technology are respectively presented during this process. A simplified multi-task flight application as a case study with preliminary experiment result is given to show the validity of this process.

Index Terms—Integrated Modular Avionics, ARINC653, Architecture Analysis & Design Language, model transformation, code generation

I. INTRODUCTION

With the ever-growing avionics functions, the modern avionics architecture is evolving from traditional federated architecture to Integrated Modular Avionics (IMA) to save hardware resource cost [1]. In IMA, it introduces partitioning concept to achieve security isolation between avionics functions with different criticalities executing on the same processor platform. ARINC653 [2] is a current major industry standard which supports the development of avionics software based on the IMA architecture.

Currently, model-driven engineering has become an important method for the development of safe-critical embedded system [3]. However, there are still some problems when designing and implementing ARINC653-based avionics software during this model-driven engineering: (1) an architecture-level language with precise semantic needs to represent such a partitioned architecture. Although the Architecture Analysis & Design Language (AADL) is extended to AADL ARINC653 annex to describe partition-specific concepts, the annex still has some shortcomings such as lacking description about an ARINC process’s internal dynamic behavior, lacking formal semantic to specify an accurate AADL model which conforms to the ARINC653 requirements; (2) it is not easy to manually build valid ARINC653-compliant models directly because avionics engineers from different application domains mainly focus on the high-level design of domain-specific models (e.g. air data computer) rather than the low-level platform-specific models (e.g. ARINC653 platform). The complexity of the ARINC653 also increases difficulty in building a valid partition model which conforms to the ARINC653 requirements; (3) a partitioned system implementation need to consider its high safety and compatibility with ARINC653 platforms. Whereas existing code generation studies mainly focus on the platform-independent code mapping, the discussion about ARINC653-specific platform is very few. Besides, their less attention to safety-critical constraints in the code generation rules enables the generated system hard to meet safety-critical need.

To address the problems above mentioned, an AADL-based integrated development process is proposed to help the design and implementation of avionics software based on the ARINC653 more productive and reliable as shown in Figure 1. An extended AADL-based formal language for ARINC653 (AADL653) is proposed to accurately describe such a partitioned architecture which is divided into two parts: AADL653 Multi-Task Model and AADL653 Runtime Model (as shown in the Lower-left column).

The whole process takes the platform-independent avionics function model represented by the AADL as input. First, a model transformation is proposed in the AADL Modeling column to automatically transform this input AADL model to a valid AADL653 model to a valid AADL653 Multi-Task Model, and then to integrate it into an AADL653 Runtime Model. Second, the integrated AADL653 model can be used for verification and analysis as shown in the V&V column. Finally, the code generation column shows an ARINC653-compliant and safety-critical system implementation consisting of real-time codes and runtime configurations, automatically generated from this verified AADL653 model.
This paper mainly focuses on the model transformation and code generation parts of this process and the rest of this paper is organized as follows. Section 2 introduces a brief AADL653 language. In Section 3, a model transformation from domain-specific AADL to platform-specific AADL653 model is discussed in detail. An object-oriented safety-critical ARINC653 code generation approach is proposed in Section 4. Section 5 presents a simplified multi-task flight application to illustrate our approach proposed in this paper. Finally, Section 6 ends this paper with a conclusion and future work.

II. AADL653 LANGUAGE

A. AADL Overview

The AADL standard [4] supports a top-down stepwise-refinement component-based design process fit with safety-critical system design. It provides three category components (e.g. system, software, hardware), component features (e.g. data port, requires data access), time properties and modes to model safety-critical system with function and non-function requirements. Some annexes are proposed to further refine or extend the AADL standard for modeling specific requirements, e.g. the AADL behavior annex [5] is used to refine internal function behavior of a thread or a subprogram for function correctness verification and detailed code generation. However, the AADL standard and above annex are high-level abstract and not aiming to any concrete platform implementation, so a new extended language with platform-specific semantic need to be defined to model avionics software constructed on the ARINC653 execution architecture.

B. AADL653 Definition

Existing AADL ARINC653 annex [6], which describes a mapping from ARINC653 elements to AADL elements and extended properties, is aimed to provide a guideline for modeling ARINC653 architecture. But the annex still has some shortcomings, for example, it uses natural language to describe a mapping from ARINC653 elements to AADL models, thus likely to cause ambiguity and inaccuracy of mapping models; it does not give an ARINC process’s internal behavior representation in a partitioned architecture, lacking of complete model descriptions etc. Therefore, the AADL653 language is proposed to extend the AADL standard and combine with the AADL behavior annex [5] to provide an accurate and complete ARINC653-compliant AADL model representation based on classic set theory and first-order logic.

The AADL653 language is divided into two parts: the AADL653 Multi-task Model and the AADL653 Runtime Model. The AADL653 Multi-task Model describes multiple ARINC process communicating in the same or different partitions, as well as internal dynamic behavior of an ARINC process in application software layer. For example, the following Definition 1 gives a formal description for AADL653 Inter-Partition Task Queuing Communication Model. It uses first-order logic to specify the property this model must satisfy, that is the source and destination tasks must communicate with a queuing channel $c_1$ by writing queuing message to the source port $(s qp)$ of $c_1$ and reading queuing message from the destination port $(dqp)$ of $c_2$. The corresponding graphical representation is as shown in Figure 2.

\[ \text{Definition 1 (Inter-Partition Task Queuing Communication).} \]

It consists of a tuple $(t_1, t_2, c_1, c_2)$ where $t_1, t_2 \in \text{ThreadTypeSet}$, $c_1, c_2 \in \text{EvtDataPortConnSet}$  
Satisfy property:

1. $\exists c_3 \in \{s qp, d qp\}$: $s qp \in \text{OutEvtDataPortSet}(p_1) \land d qp \in \text{InEvtDataPortSet}(p_1)$  
2. $(\text{srcPort}(c_1)) = \text{OutEvtDataPortSet}(p_1)$  
3. $(\text{destPort}(c_2)) = \text{InEvtDataPortSet}(p_1)$  

\[ \text{Figure 2. AADL653 Inter-Partition Task Queuing Communication Model} \]

Definition 2 shows a Task Behavior Automation Machine to control the task’s state transition with guards (e.g. receiving message from a queuing port represented by $d q p$?) and actions (e.g. calling intra-partition communication subprograms represented by $\text{intraComSub}$).

\[ \text{Definition 2 (Task Behavior Automation Machine).} \]

It consists of a tuple $(t, S, s_0, G, A, E)$ where

1. $t \in \text{ThreadTypeSet}$  
2. $S$ is a set of state, $s_0 \in S$ is the initial state  
3. $G = \{\text{dqp}\} \cup \{\text{dqp}\} \cup \{\text{dqp}\}$  
4. $A = \{\text{appSub}\} \cup \{\text{appSub}\}$  
5. $E \subseteq S \times G \times A \times S$ is a set of transitions.

The AADL653 Runtime Model describes a partitioned runtime environment in core software layer. It involves inter-partition sampling or queuing communication...
channel, partition scheduling window time requirements, partition memory requirements and health monitor configuration information.

III. FROM AADL TO AADL653 MODELS

The AADL653 language can be used to manually build ARINC653-specific model, but it is still a hard and error-prone work for those avionics engineers focusing on design of avionics applications in high-level. This section proposes a model transformation framework with formal rules to automatically transform a domain-specific AADL model built by the avionics engineers to an AADL653 multi-task model conforming to ARINC653 requirements. Then the model can be integrated into an AADL653 runtime model for validation and analysis.

A. Model Transformation

The transformation rule consists of three parts: pattern, config and action. The pattern specifies an abstract platform-independent model which refers to an AADL model. The config is used to specify concrete ARINC653 task communication style (e.g. inter-partition or intra-partition) applied to current pattern. The action shows a sequence of operation performed on current pattern to transform an AADL model to an AADL653 model.

The transformation rules involves two levels: the first one is inter-task interaction level, aiming to transform abstract inter-task static dependencies to ARINC653-specific inter-task communication styles according to a specified config, e.g. the following Rule 1 shows a rule definition about transforming an AADL Event Data Port Interaction Model (EDPIM) specified in the pattern to a valid AADL653 Task Queuing Communication Model in Definition 1, with an inter-partition config. Likewise, the EDPIM can be transformed to an AADL653 Task Buffer Communication Model which shows each of the source and destination task has a requires data access to a Buffer data if the config is intra-partition.

The second-level transformation is aiming to each ARINC process’s internal dynamic behavior, to transform platform-independent task behaviors to ARINC653-specific task behaviors. For example, if a guard of the transition represents a message arriving at an \textit{in event data port} \textit{(iedp?)} of the task, and it has been transformed to a \textit{requires data access} \textit{(rda)} to a \textit{buffer object} \textit{(bufImpl)} in the first-level transformation, then the second-level transformation rule (e.g. Rule 2) will generate explicit intra-partition communication subprogram call \textit{(RECEIVE_BUFFER!)} which is respectively added to the head of the \textit{action} part of the \textit{transition} and the tail of the subprogram call sequence of the task, instead of reading message from an abstract \textit{in event data port}.

Rule 2: AADL REDPM to AADL653 RBM

\textbf{Pattern:} (Read Event Data Port Model)
\[ a \text{ tuple } (g,e,t) \text{ where } g = \text{GuardSet}(e) ; \text{ iedp} \in \text{InEvtDataPortSet}(t) ; \ t \in \text{ThreadType} ; \ e \in \text{TransitionSet}(t) ; \]
\textbf{Config (Inter-Partition):}
\[ \text{isTransformedTo}(\text{iedp}, \text{rda}, \text{bufImpl}) ; \]
\textbf{Actions:}
\[ \text{addTransAction}(\text{actions}, \text{head}, \text{RECEIVE_BUFFER!}); \]
\[ \text{addSubCall}(\text{psc}(\text{t}), \text{tail}, \text{RECEIVE_BUFFER!}); \]

Rule 3 shows a data exchange behavior conversion example to transform abstract AADL \textit{In Event Data Port Parameter Passing Model} (IEDPPM) to concrete AADL653 Buffer Parameter Passing Out Model (BPPOM). The \textit{pattern} means the value of the \textit{in event data port} \textit{(spar)} of the task will be read and passed to its internal subprogram. The \textit{config} means reading message from abstract port \textit{(spar?)} has been substituted with concrete subprogram call \textit{(RECEIVE_BUFFER!)}. Therefore, the \textit{action} in this rule respectively adds an AADL data access and parameter connection to represent the actual parameter value passing of the subprogram is from the data value read from the concrete buffer resource.

Rule 3: AADL IEDPPM to AADL653 BPPOM

\textbf{Pattern:} (In Event Data Port Parameter Passing Model)
\[ a \text{ tuple } (c,t,s) \text{ where } c \in \{ \text{sp}, \text{dp} \} ; \text{ spar} \in \text{InEvtDataPortSet}(t) ; \ A \text{ dpar} \in \text{InParameterSet}(s) ; \]
\textbf{Config (Intra-Partition):}
\[ \text{isTransformedTo}(\text{spar}, \text{rda}, \text{bufImpl}, \text{RECEIVE_BUFFER!}) ; \]
\textbf{Actions:}
\[ \text{addParameterConn}(\text{c}); \]
\[ \text{addDataAccessConn}(\text{rda}, \text{RECEIVE_BUFFER.rda}); \]
\[ \text{addParameterConn}(\text{RECEIVE_BUFFER.output}, \text{dpar}); \]

B. Transformation Implementation

A rule engine, which implements the above two-level transformation rules, is developed to accomplish automatic model transformation. It takes any valid AADL model instance and associated xml configuration file as input, and output a valid AADL653 Multi-Task Model instance. The configuration file describes current communication style (e.g. inter-partition with a sampling channel or intra-partition with a buffer) and communication resource details (e.g. buffer size) each pair of dependent tasks will use. It is generally specified by the avionics application integrator so as to put the tasks in the appropriate partition.

For each rule, the rule engine first to search for each matched pattern in the input AADL model, e.g. Algorithm 1 is an implementation of Rule 1, line 5-9 shows searching for each EDPIM pattern, that is source and destination threads with an event data port connection between them; second to check the input
configuration file whether the matched model will be configured with the same communication style as it is specified in the config part of this rule (line 10), if yes then perform the action sequence defined in this rule (line 11-14).

Algorithm 1: AADL EDPIM To AADL653 TQCM Rule Implementation

1: input: AADL model and XML configuration file
2: output: AADL653 task queuing communication model
3: begin
4: for all pmpl ∈ ProcessImplSet in AADL model
5:   for all c ∈ ConnectionSet(p) // search for EDPIM pattern
6:       t₁ ← srcComponent(c); sp ← srcPort(c);
7:       t₂ ← destComponent(c); dp ← destPort(c);
8:       if (t₁ ∈ ThreadTypeSet && t₂ ∈ ThreadTypeSet) then
9:         if (XMLconfig(t₁, t₂) == "Inter-Partition") then //do actions
10:            delete(c);
11:            qc = getQueChannel(t₁, t₂);
12:            c₁ = createEventDataPortConn(sqp, scrPort(qc));
13:            c₂ = create EventDataPortConn(destPort(qc), dqp);
14:            add(c₁);
15:            end

C. Model Integration and Analysis

The purpose of the model integration is to integrate the AADL653 Multi-Task Model instance after transformation to an AADL653 Runtime Model instance to enable the integrated model still satisfies system requirements such as soft and hard real-time deadlines etc.

There are two scenarios for the model integration. One is application-driven scenario. That is we have known the multiple tasks and their dependency relation in the AADL653 Multi-Task Model instance, as well as the partition location each task will be put, then to configure the AADL653 Multi-Task Model instance, as well as the known the multiple tasks and their dependency relation in requirements such as soft and hard real-time deadlines etc. to enable the integrated model still satisfies system requirements.

For example, Table 1 gives a RT-Java interface DestQuePort declaration mapped from an in event data port (dap) in Definition 1, which represents a queuing channel’s destination port from which the queuing message can be read.

### Table 1. The DestQuePort Interface.

```java
public interface DestQuePort {
    public Object READ_QUEUING_MESSAGE();
    public int GET_MAX_MSG_NUM();
    public byte GET_MAX_MSG_SIZE();
}
```

A. Code Mapping

The code mapping in this framework aims to build a safety-critical programming model class library for ARINC653. The class library will be used for ARINC653 object instantiation during the code generation process. The mapping involves two levels: first is to map an ARINC653 process interface with inter/intra-partition communication resource defined in the AADL653 Task Communication Model to a RT-Java/C++ task communication programming model. For example, Table 1 gives a RT-Java interface DestQuePort declaration mapped from an in event data port (dap) in Definition 1, which represents a queuing channel’s destination port from which the queuing message can be read.

### Table 2. The PeriodicArincProcessImpl Class.

```java
public class PeriodicArincProcessImpl implements ArincProcess extends NoHeapRealtimeThread {
    public PeriodicArincProcessImpl(int priority, HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler) {
        super(new PriorityParameters(priority),
              new PeriodicParameters(start, period, cost, deadline,
                                      overrunHandler, missHandler), ImmortalMemory.instance());
    }
    // Constructor

    //ScopedMemory with initial size in bytes
    static ScopedMemory smArea = new LTMemory(size);

    public void run() {
        while (true) {
            smArea.enter(new Runnable() {
                public void run() {
                    //execute behavior code in this scoped memory...
                }
            });
            boolean ok = waitForNextPeriod();
        }
        // structure body of run method
    }
```
Second is to map an ARINC653 process instance defined in the AADL653 Task Behavior Model to a RTJava/C++ task behavior programming model. Table 2 shows a PeriodicArincProcessImpl class which represents a periodic ARINC653 process. The run() method consisting of an endless loop body contains business logic followed by a method waitForNextPeriod() [9], which shows a periodic dispatch behavior. The class extends the class NoHeapRealTimeThread [9] which is never allowed to allocate or reference any object allocated in the heap, thus it is always safe to interrupt the garbage collector at any time. Moreover, two types of memory ImmortalMemory and ScopedMemory [9] are used to allocate objects in the non-heap, thus very suitable for safety-critical systems implementation.

Code Generation

A two-phase code generation process is proposed to automatically implement ARINC653-compliant system consisting of real-time code running on the target and runtime configuration code provided to ARINC653-compliant OS.

The first phase is to generate real-time code consisting of partition initialization code and task behavior code. The partition initialization code includes all the ARINC653 processes and communication resource objects required in each partition, generated from the AADL653 Task Communication Model instance of each partition. For example, Algorithm 2 (line 4-6) gives a blackboard communication object instantiation algorithm by parsing the intra-partition task blackboard communication model where there are two threads both having access to the same blackboard data.

Algorithm 2: Intra-Partition Communication Resource Instantiation Code Generation
1: Input: AADL653 Intra-Partition Task Communication Model instance of a Partition: ((tmpl,tmpl1,dmpl,scon,rcon))
2: Output: Intra-Partition Communication Resource Instantiation Code
3: begin
4: if (dest(scon)) &
5: if (dest(rcon)) then
6: instantiate an ArincBlackboard object with
7: getPropertyValue(dataImpl, “MaxMessageSize”);

In order to adapt to multi-task collaboration scenarios, we present detailed internal behavior code generation algorithm for each ARINC653 process so that global interaction sequence among multiple tasks can be determined.

As shown in Algorithm 3, for each transition in the Task Behavior Automation Machine (line 5), first to match each guard in this transition (line 6) to generate inter-partition receiving message code, e.g. line 7-9 shows generating READ_SAMPLING_MESSAGE() call if there is a sampling data arriving the sampling port (dsp?); second to match each action in this transition (line 13) to generate inter-partition sending message and intra-partition communication subprogram call code, e.g. line 14-16 shows generating SEND_BUFFER(rda, message) call of the buffer object referenced by the rda.

Algorithm 3: Task Behavior Code Generation
1: Input: AADL653 Task Behavior Model Instance of Each ARINC653 Process
2: Output: Task Behavior Code of Each ARINC653 Process
3: begin
4: for all tmple ThreadImplSet do
5: for all transition t in BAM(tmple) do
6: for all guard ∈ GuardSet(transition) do
7: if (guard==dsp? && idp ∈ InDataPortSet(tImpl)) then
8: ArincDspImpl ← tmple.destMapPortMap.get(dsp.name)
9: Call ArincDspImpl.READ_SAMPLING_MESSAGE();
......
13: for all action ∈ ActionSet(transition) do
14: if (action==SEND_BUFFER(rda, message) &&
15: rda ∈ ReqDataAccessSet(tImpl)) then
16: Call ArincBufImpl.SEND_BUFFER(rda, message);
......

The second phase of code generation aims to generate runtime XML configuration code which will be used by ARINC653-compliant OS for the core module and partition configuration. Algorithm 4 gives the overall configuration code generation algorithm which will generate inter-partition channel (line 5-6), partition scheduling window (line 8-9) and partition memory requirements (line 11-13) configuration information.

Algorithm 4: Partition XML Configuration Code Generation
1: Input: AADL653 Runtime Model Instance
2: Output: Partition XML Configuration Code
3: begin
4: for all smple SystemImplSet do
5: for all (p,c,p’) ∈ InterParCom(smple) do
6: Call GeChannel(p, c, p’);
7: p←Processset(s); Vp←VirProSubComp(p);
8: for all vP ∈ P do
9: Call GeSchChannel(vP);
10: P←ProcessSubComp(s);
11: for all p ∈ P do
12: if (p ∈ BindMem); p;
13: Call GeMemreq(m);
14: end

A code generation tool which implements all the algorithms in two-phase code generation process is developed to generate ARINC653 system automatically.

V. CASE STUDY

In this section, we take a simplified multi-task flight application [10] as a case study to illustrate our approach proposed in this paper. This example is a scenario concerned with an aircraft system of computing the position and fuel information through multi-task collaboration. It consists of four periodic tasks Pos, Fuel, Para and Fsam each with 60ms period and a sharing data Global_params as shown in Figure 3.
respectively generated in the communication subprogram calls (e.g. updatePos.pos -> updatePos.updpos -> upd_posMes). The task sends a position request (reqPos_refresh) to the task Para for global parameter refreshment. Then, it waits for the notification of the end of refreshment (refresh_end?) from the task Para.

### TABLE 3. AADL BEHAVIOR MODEL OF THE TASK PositionIndicator

<table>
<thead>
<tr>
<th>Subprogram Sequence</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprogram Sequence</td>
<td>calls</td>
</tr>
<tr>
<td>subprogram Sequence</td>
<td>{</td>
</tr>
<tr>
<td>subprogram Sequence</td>
<td>}</td>
</tr>
</tbody>
</table>

First, Figure 4 shows a corresponding AADL653 task communication model instance after the first-level transformation from Figure 3. For example, the task Pos and Para interacting with abstract event data ports is converted to communicating with a buffer resource (buff2) in an intra-partition config. For the second-level transformation, Table 4 shows a textual AADL653 task behavior model of the task Pos, which is transformed from Table 3. We can see explicit intra-partition communication subprogram calls (e.g. send_buff2!) are respectively generated in the subprogram call sequence and transition action part. Besides, data exchange (e.g. a parameter passing ParaCon1 between the subprogram getData and send_buff2!) in ARINC653 platform is also generated in connections part of the thread.

Next, the model after two-level transformation is integrated into a specified AADL653 Runtime Model instance as shown in Figure 5. It shows a core module with On_Flight and Info_Collection partitions communicating with a sampling channel. The two partitions are respectively binding to a separate memory to enforce spatial partitioning and a unique virtual processor to implement temporal partitioning by each allocated 40ms and 20ms scheduling window in a 60ms major frame. The integrated model is then verified schedulable with the Cheddar tool [8]. Finally, the RT-Java partition initialization and task behavior code, as well as XML configuration code can be generated automatically from this verified model.
We choose an ARINC653-compatible OS VxWorks653 [11] and a safety-critical business RT-Java virtual machine Jamaica VM for VxWorks653 [12] as our experiment environment. We first compile the previous generated code to binaries and run it in the VxWorks653 simulator (VxSim). The preliminary output partly is shown in Table 5. We can see the required partition resources (e.g. the buffer resource buff2) and concurrent entities (e.g. the task Pos) are created successfully and global synchronized interaction sequence among the tasks is as expected, e.g. the task Pos with the highest priority is waiting for the event notification until the task Para with the lowest priority calling the SET EVENT method (line 19-20). The results demonstrate the validity of our approach.

TABLE 5. EXPERIMENT RESULT EXCERPT.

1[VxWorks653]: creating the task Pos in Partition OF;
2[VxWorks653]: the task Fuel in Partition OF;
4[VxWorks653]: creating the source sampling port ssp1;
5[VxWorks653]: creating the buffer buff2;

11[VxWorks653]: the Partition OF enters into NORMAL mode;
12[VxWorks653]: the task Pos is started;
13[VxWorks653]: SEND BUFFER of buff2 by Pos
14[VxWorks653]: the task Fuel is started;
15[VxWorks653]: the task Para is started;
16[VxWorks653]: RECEIVE_BUFFER of buff2 by Para;
17[VxWorks653]: WAIT SEMAPHORE of sem1 by Para;
18[VxWorks653]: SIGNAL_SEMAPHORE of sem1 by Para;
19[VxWorks653]: DISPLAY_BLACKBOARD of board1 by Para;
20[VxWorks653]: SET EVENT of evtl by Para;
21[VxWorks653]: WAIT EVENT of evtl by Pos;
......

VI. RELATED WORK

Some studies have been done on safety-critical embedded system development based on model-driven engineering. Esterel Technologies propose correct-by-construction methods and develop SCADE tool to support the automated production of a large part of the software development life-cycle elements [13]. But it mainly focuses on domain-specific applications independent of platforms, whereas we pay attention to those constructed on ARINC653 platform. Hugges, J. et al. presents a rapid prototyping of distributed real-time embedded system using the AADL and Ocarina [14]. It discusses a mapping from the AADL model describing distributed embedded system to high-integrity Ada/C code, while our work targets a safety-critical ARINC653 system automatic generation based on the AADL653 model. J. Delange et al. proposes an AADL-based model-driven method to model, validate and implement ARINC653 systems [15]. It is similar to our work, but there are three differences: first, it models the ARINC653 architecture using the AADL ARINC653 annex which has some shortcomings as we stated previously, while our work is based on the formal AADL653 language we defined to provide an accurate and complete ARINC653-compliant AADL model; second, it suggested to use ARINC653 annex to manual building the AADL model for ARINC653 system directly, while we separate the domain-specific AADL model from the platform-specific AADL653 model and provide a transformation approach to automatically build the AADL653 model; third, it focus on a C code generation strategy for partitioned systems and only discusses simple inter-task communication code mapping strategy, whereas we focus on object-oriented ARINC653 code generation and gives detailed ARINC653 process’s internal dynamic behavior code generation algorithm suitable for multi-task collaboration scenarios.

VII. CONCLUSIONS

In this paper we present an AADL-based integrated development process for ARINC653-based avionics system to ease complexity and improve reliability of this work. Automatic model transformation and safety-critical code generation approach are presented in detail. A simplified multi-task flight application as a case study is given to illustrate and show the validity of our approach. In the future, we will verify the correctness of the model transformation and code generation to facilitate a certification of ARIN653 system as far as possible.

ACKNOWLEDGMENT

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant No.61003017.

REFERENCES


Ying Wang received his B.E. degree at 2005 in National University of Defense Technology. She is a Ph.D candidate at the Department of Computer Science in Beihang University now. Her research interests are software engineering, embedded real-time system and service-oriented computing.

Dianfu Ma is a professor in the State Key Laboratory of Software Development Environment, Beihang University, China. His research interests are severe environment computer, embedded real-time system and service-oriented computing.