World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:6, No:11, 2012

A Mapping Approach of Code Generation for Arinc653-Based Avionics Software

Lu Zou, Dianfu MA, Ying Wang, and Xianqi Zhao

***Abstract*—**Avionic software architecture has transit from a federated avionics architecture to an integrated modular avionics (IMA) .ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in Safety-critical avionics Real-time operating systems. Methods to transform the abstract avionics application logic function to the executable model have been brought up, however with less consideration about the code generating input and output model specific for ARINC 653 platform and inner-task synchronous dynamic interaction order sequence. In this paper, we proposed an AADL-based model-driven design methodology to fulfill the purpose to automatically generating C++ executable model on ARINC 653 platform from the ARINC653 architecture which defined as AADL653 in order to facilitate the development of the avionics software constructed on ARINC653 OS. This paper presents the mapping rules between the AADL653 elements and the elements in C++ language, and define the code generating rules , designs an automatic C ++ code generator .Then, we use a case to illustrate our approach. Finally, we give the related work and future research directions.

***Keywords*—**IMA, ARINC653, AADL653, code generation.

I. Introduction

IN order to reduce Avionic software cost which is rapidly raising nowadays, integrated modular avionics (IMA) architecture has been proposed and broadly accepted. IMA architectures employ a high-integrity, partitioned environment that hosts multiple avionics functions of different criticalities on a shared computing platform.[1]A specificity of Integrated modular avionics in the certification process of avionics systems is that standards such as ARINC 653 , allow each software building block of the overall Integrated modular avionics to be tested, validated, and qualified independently by its supplier.[2]ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in Safety-critical avionics Real-time operating systems. It allows hosting multiple applications of different software levels on the same hardware in the context of a Integrated Modular Avionics architecture [3]. It allows to host multiple applications of different software levels on the same hardware in the context of a Integrated Modular Avionics architecture [4].

On the other hand, an embedded system is a computer system designed for specific control functions often with real-time computing constraints [5]. It is very important for the engineer to analyze the system properties and choose among all the design alternatives in the design cycle. Currently, model-driven (MDA) correct-by-construct methodology has become a major development method in designing safety-critical embedded system [6]. AADL (Architecture Analysis and Design Language) [7] is a standard for architectural modeling of embedded systems widely applied to avionic software modeling and approved in November 2004 as the SAE standard AS5506 [8] including precise semantics to model its components. We have defined an AADL653 model with both the AADL standard and the AADL behavior annex [9] in order to provide a precise model representation for ARINC653-based avionics software.

So far, there have been several tools for code generation of AADL models. There are Ocarina developed by Telecom ParisTech which generates C and Ada codes targeting on PolyORB, PolyORB-HI and the POK [10], STOOD developed by Ellidis which can generate C codes targeting on no-partition platform [11], UCaG [12] developed by University of Electronic Science and Technology of China which generates C codes targeting on Delta OS which is not a portioned platform.

Several platform-independent code generation or mapping methods as listed above have been proposed, the code generator in this paper is different from three points: 1. The input AADL model is different, which is AADL653 model a kind of platform specific model with many additional properties specified by the including project of the project this paper describes; 2. The targeting platform is ARINC653-based platform which has been targeted on by very few code generator 3. Besides, this code generation method considers the inner-task synchronous dynamic interaction order sequence which is very important to describe the situation of multi-task communication. In view of the above, we proposed an AADL-based model-driven design methodology to fulfill the purpose to automatically generating C++ executable model from the AADL653 describing the ARINC653 architecture facilitating the development of the avionics software constructed on ARINC653 OS.

In this paper, we present mapping rules between a subset of AADL653 models which we have define as AADL653

Lu. Zou is with National Lab of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, China (e-mail: zoulu@act.buaa.edu.cn,).

DianFu MA is with National Lab of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, China (e-mail: dfma@ buaa.edu.cn,).

Ying.Wang is with National Lab of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, China (e-mail: wangying@ act.buaa.edu.cn,).

Xianqi Zhao is with National Lab of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, China (e-mail: zhaoxq@act.buaa.edu.cn).

including two parts: AADL653 application model and AADL653 runtime model to elements in C/C++ language based upon the hard-time operating system Vxworks653, and C/C++ application code generation algorithm from the AADL653 multi-task application model; Partition runtime configuration code generation algorithm based on the AADL653 runtime model ,and design an automatic C/C++ code generator named AADL653.

II. An Overview of the AADL653 Model

AADL (Architecture Analysis and Design Language) [6], is a standard for architectural modeling of embedded systems .AADL provides an industry-standard, textual and graphic notation with precise semantics to model three predefined component categories: composite component (system), software components (process, thread group, thread, subprogram, and data) and execution platforms .components (processor, virtual processor, memory, bus, virtual bus and device) [8].The AADL model can just describes the tasks with function interface and inter-task static interaction with no dynamic connection with other task. In order to describe the inter-task dynamic interaction sequence among these tasks which is not included in AADL, we introduce AADL behavior annex to combine with AADL to make it more thorough and complete. We defined AADL 653 which is a more enriched AADL-based model for Arinc653 execution architecture combining with both the AADL standard [8] and the AADL behavior annex [9].

AADL653 model is used to describe the Arinc653 IMA which is mainly divided into two levels-- an AADL653 Multi-Task Application Model in application software layer and an AADL653 Runtime Model in core software layer. Table I presents a brief illustration of AADL653 model.

TABLE I
AADL653 Model Content

AADL 653 Model	Contents	
AADL653 Multi-Task Application Model	AADL653 Task Communication Model	sampling or queuing ports in different partitions
		Inner-Partition communicating with blackboards, buffers, semaphores or events
	Task Behavior Model	Task Behavior Model: Behavior Automation Machine
		Subprogram Call Sequence
AADL653 Runtime Model	AADL653 Inter-Partition Communication Model	
	AADL653 Two-level Scheduling Model	

AADL653 Multi-Task Application Model is an AADL representation of multiple ARINC653 processes executed concurrently in application partitions, and the AADL653 Runtime Model is an AADL representation of an ARINC653 core module and partition related runtime configuration information (such as inter-partition communication, temporal and spatial partitioning etc.) [13].Firstly, AADL653 Multi-Task Application Model consists of two components ,one is AADL653 Task Communication Model which contains sampling or queuing ports in different partitions and Inner-Partition communicating (synchronizing) with blackboards, buffers, semaphores or events in the same partition, the other is AADL653 Task Behavior Model including Behavior Automation Machine and Subprogram Call Sequence .Secondly, AADL653 Runtime Model consists of AADL653 Inter-Partition Communication Model, The AADL653 Two-level Scheduling Model and The Partition Memory Model. For more precise definition and description of each model please refer to [13].

III. The Mapping Rules for AADL 653 Models

We present the mapping rules from AADL653 models which we have defined above including two parts: AADL653 application model and AADL653 runtime model each mapping to the corresponding elements: configuration files used for system and C++ executable based upon the hard-time operating system with the ARINC653 architecture.

Arinc653 platform need both the correct configuration file which specifies the partition, memory, connection and some other configuration information necessary for system and C++ executable codes running on each partition. For AADL653 application model there should be a source file for each partition to start with and the related C++ header files with communication resource Class and task Class defined in; and for AADL653 runtime model there should be a configuration file and a package generated.

A. Mapping Rules for AADL653 Application Model

1) Mapping Rules for Task Communication Model

The AADL653 Task Communication Model mainly describes two kinds of communication situation respectively the Inter-Partition Task Communication situation and the Intra-Partition Task Communication situation .Inter-Partition Task Communication describes the situation that

ARINC653 processes located in different partition communicating with sampling or queuing ports and the Intra-Partition Task Communication describes process located in the same partition communicating (synchronizing) with blackboards, buffers, semaphores or events. So the Communication resource models such as: sampling or queuing ports and blackboards, buffers, semaphores or events should be mapped to the corresponding C++ executable model. For more detailed information about the Communication model in AADL653, please refer to [13].

RULE1 (1.1, 1.2, 1.3, 1.4) describes inner-Partition Task Communication model mapping and RULE 2(2.1, 2.2, 2.3, 2.4) describes the intra-Partition Task Communication model mapping.

RULE1: Each port type in AADL653 is mapping to a specific C++ Class with functions defined in that providing the services mentioned in the Arinc653 standard. Each of properties in AADL port is mapped to a typed parameter of the class constructor

RULE1.1: Source sampling port in AADL653 is mapping to

a specific C++ Class ArincSrcSamPort, each of properties in AADL port is mapped to a typed parameter of the class ArincSrcSamPort constructor

AADL653 Inter-Partition Task Communication resource used by processes in different partitions while communicating consists of these port type: Source Sampling port, Destination Sampling port, Source queuing port, Destination queuing port .

Take the Source Sampling port (Rule1.1) as an example, Source Sampling port corresponds to ArincSrcSamPort class as shown in Listing 1 with the function: WRITE_SAMPLING_MESSAGE function providing the service writing a message in the specified sampling port and GET_SAMPLING_PORT_ID function allowing to obtain a sampling port identifier also with the constructor used to create a sampling port, as what is shown in the List below. The Destination Sampling port similarly corresponds to the ArincDestSamPort, The RefreshTime property (with its AADL Time property type) defined in AADL in data port represents required refresh rate attribute of destination sampling port in ARINC653 ,which is mapped to a C++ parameter refreshTime with SYSTEM_TIME_TYPE type. Source queuing port and Destination queuing port are also mapped to the corresponding Class.

Listing 1 the ArincSrcSamPort class declaration

```
public class ArincSrcSamPort {
private:
        SAMPLING_PORT_NAME_TYPE       samplingName;
      ...  ...
Public:
     ArincSrcSamPort(SAMPLING_PORT_NAME_TYPE
samplingName, MESSAGE_SIZE_TYPE MaxMessageSize,
PORT_DIRECTION_TYPE portDirection,
SYSTEM_TIME_TYPE  refreshTime);
     void WRITE_SAMPLING_MESSAGE
(MESSAGE_ADDR_TYPE MESSAGE_ADDR,
MESSAGE_SIZE_TYPE LENGTH);
      MESSAGE_SIZE_TYPE getMaxMesSize()
}
```

RULE1.2: Destination Sampling port in AADL653 is mapping to a specific C++ Class ArincDestSamPort, each of properties in AADL port is mapped to a typed parameter of the class ArincDestSamPort constructor

RULE1.3: Source queuing port in AADL653 is mapping to a specific C++ Class ArincSrcSamPort, each of properties in AADL port is mapped to a typed parameter of the class ArincSrcSamPort constructor

RULE1.4: Destination queuing port in AADL653 is mapping to a specific C++ Class ArincDestSamPort, each of properties in AADL port is mapped to a typed parameter of the class ArincDestSamPort constructor

The detailed mapping principles for rule1.2, rule 1.3, rule1.4 is similar as those which have be demonstrated above for rule1.1.

RULE2: Each data type and implementation maps to a specific C++ Class with functions defined in that providing the services mentioned in the Arinc653 standard ,Each port type in AADL653 is mapping to a specific C++ Class with functions defined in that providing the services mentioned in the Arinc653 standard. Each of properties in AADL data implementation is mapped to a typed parameter of class constructor. Each of subprogram features in AADL data type is mapped to corresponding public method.

RULE2.1: Each Blackboard data type and implementation maps to a specific C++ Class ArincBlackboardDef with functions defined in that providing the services mentioned in the Arinc653 standard

AADL653 Intra-Partition Task Communication resource used by processes in the same partition while communicating synchronously consists of four data type: Blackboards, Buffers, Semaphores, Events. for example Blackboards with its subprogram features SEND_BUFFER and RECEIVE_BUFFER and etc corresponds to the ArincBlackboardDef Class in Arinc653 architecture as shown in Listing 2 with the function: DISPLAY_BLACKBOARD used to display a message in the specified blackboard; READ_BLACKBOARD used to read a message in the specified blackboard; CLEAR_BLACKBOARD used to clear the message of the specified Blackboard; GET_BLACKBOARD_ID provide the service to get the identifier of a blackboard, also with the constructor to create a blackboard.Other data types such as Buffers, Semaphores, Events correspond to the specific Classes. The MaxMessageSize property (with its AADL aadlinteger property type) defined in AADL653 data Blackboard implementation represents the maximum message number the Blackboard can hold, which is mapped to a RT-Java parameter msgSize with MESSAGE_SIZE_TYPE type.

Listing 2 C++653 ArincBlackboardDef Class declaration

```
public class ArincBlackboardDef{
   private:
       BLACKBOARD_NAME_TYPE name;
     … …
  public:
      ArincBlackboardDef(BLACKBOARD_NAME_TYPE
name, MESSAGE_SIZE_TYPE msgSize);
    Void DISPLAY_BLACKBOARD (MESSAGE_ADDR_TYPE
MESSAGE_ADDR, MESSAGE_SIZE_TYPE LENGTH) ;
     MESSAGE_SIZE_TYPE READ_BLACKBOARD
(SYSTEM_TIME_TYPE INFINITE_TIME_VALUE,
MESSAGE_ADDR_TYPE MESSAGE_ADDR);
  … …
}
```

RULE2.2: Each Buffer data type and implementation maps to a specific C++ Class ArincBufferDef with functions defined in that providing the services mentioned in the Arinc653 standard

RULE2.3: Each Semaphore data type and implementation maps to a specific C++ Class ArincSemaphoreDef with functions defined in that providing the services mentioned in the Arinc653 standard

RULE2.4: Each Events data type and implementation maps to a specific C++ Class ArincEventDef with functions defined in that providing the services mentioned in the Arinc653 standard

The detailed mapping principles for rule2.2, rule 2.3, rule2.4

is similar as those which have be demonstrated above for rule 2.1.

2) Mapping Rules for Task Behavior Model

TABLE II
PERIODIC PROCESS CLASS DECLARATION AND RUN FUNCTION IMPLEMENTATION

```
public class Periodicprocess{
    private:
    PROCESS_ATTRIBUTE_TYPE processTable;
    PROCESS_ID_TYPE  procId;
    RETURN_CODE_TYPE  retCode;
    PROCESS_ATTRIBUTE_TYPE processTable;
    APEX_TYPES.SYSTEM_ADDRESS_TYPE m_pRunnable;
 . . . . . .

public:
   Periodicporcess(PROCESS_NAME_TYPE
name,APEX_TYPES.SYSTEM_ADDRESS_TYPE entry,
STACK_SIZE_TYPE stack,PRIORITY_TYPE prio,
APEX_TYPES.SYSTEM_TIME_TYPE period,
APEX_TYPES.SYSTEM_TIME_TYPE timecap, DEADLINE_TYPE
deadline);
    void create();
    void* runGlobal(void *args);
    void run();
    void startprocess(PROCESS_ID_TYPE procId);
```

```
void Periodicprocess::run(){
 while (true){
  //execute behavior code in this scoped memory...}
  PERIODIC_WAIT (&retCode);
  If (!checkretCode(&retCode){
   // handle overrun or deadline miss here }
)
}
```

RULE3: Each Thread in AADL653 is mapped to a specific (PeriodProcess/ AperiodProcess/ SporadicProcess) Class according to the Dispatch Protocol .Each application subprogram in subprogram calls of the AADL thread implementation is mapped to a method with the same name in the class. The in and out parameter (with AADL property type) of the subprogram is mapped to corresponding C++ typed input and output parameter of the method.

RULE3.1: Period Thread (Dispatch_Protocol is Periodic) in AADL653 is mapped to a Class Periodicprocess according to the Dispatch Protocol, with subprogram calls and parameter mapping to the specific method and corresponding C++ typed parameter respectively.

Task Behavior Model consists of three kinds of Thread: Periodic Thread, Aperiodic Thread and Sporadic Thread according to Dispatch Protocol type (Periodic,Aperiodic,Sporadic)which defined as a property in AADL653 Thread model. These three kinds of threads in AADL653 each corresponds to PeriodProcess Class as shown in Table III, AperiodProcess Class and SporadicProcess Class encapsulated from Process in Arinc653 architecture with the function providing the process management services defined in Arinc653 standard.

Each property defined in AADL thread implementation is mapped to the corresponding parameter of the class constructor. For example, the property Priority which is of integer type to represent thread's execution eligibility, property Period describing the time for one dispatching and property and property Dispatch_Off representing the time at which the first period begins are all mapping to the member of the Arinc653 structure of the type PROCESS_ATTRIBUTE_TYPE which is the input parameter of the Class constructor.

Moreover, need to add run () function body to the Arinc653 thread implementation. Take the periodic Protocol Thread as an example, the corresponding C++ Class is as below in Table III. In the body of the corresponding Period Process Implementation, the run () method contains the sub function logic. In order to represent the periodic dispatch behavior, there is a loop structure with subprograms located in and a method PERIODIC_WAIT () in the end blocking until the start of the next period which is a service that Arinc653 provides for time management. The method will return error return code if the thread is in an overrun or deadline miss condition.

B. Document Modification Mapping Rules for AADL653 Runtime Model

Each AADL653 runtime model component is mapping to the configuration component to compose the whole XML-based Arinc653 configuration file. Table III is the mapping rules between AADL653 model components and ARINC653 configuration element.

TABLE III
MAPPING RULES FOR CONFIGURATION TABLE

AADL653 model components	ARINC653 Configuration element
System	ARINC 653 Module
Process	Partition Table
Processor	Module_Schedule Table
Memory	Partition_Memory Table
Port Connections	Connection_Table Table

According to the ARINC653 standard, this part discuss how to generate XML-based ARINC653 configuration file which will be used by ARINC653-compatible OS. The configuration file generation rules are as below:

RULE4: For each AADL system implementation which represents an ARINC653 core module, first to generate a <ARINC_653_Module> top-level element.

RULE5: For each Process subcomponent in each System implementation, generate a <Partition> element inside with attribute/port configuration in it.

RULE6: For each Actual_Processor_Binding presenting two-level scheduling model of this system implementation properties in System implementation, generate a <Partition_Schedule> element in the <Module_Schedule>element describing scheduling windows configuration.

RULE7: For each Actual_Memory_Binding properties in System implementation generate a <Partition_Memory> Element describing partition memory requirement configuration.

RULE8: For each element of connections in System

implementation, generate <Channel> element in the <Connection_Table> element presenting the inter-partition communication channels configuration.

IV. Code Generation from the AADL653 Multi-Task Application Model

This section represent the code generation with two steps. Firstly, generate the element mapping from AADL653 Task Communication Model initialization code for each partition; Secondly, generate behavior logic function code mapping from AADL653 Task Behavior Model running in each ARINC653 process located in each run() method body.

Algorithm 1 depicts partition initialization code generation from the AADL653 Task Communication Model instance of each partition.

Algorithm 1: Partition Initialization Code Generation

```
1: Input: AADL653 Task Communication Model Instance of Each
Partition
2: Output: Initialization C++ Code of Each Partition
3: begin
4: for all pImpl ∈ ProcessImplSet do
5:   for all tImpl ∈ ThreadSet(pImpl) do
6:    begin // Algorithm 1.1: create ARINC653 processes
7:     if (getPropertyValue(tImpl, "Dispatch_Protocol") =
"Periodic")then
8:        instantiate a PeriodicArincProcessImpl object with
getPropertyValue(tImpl, "Priority"),
getPropertyValue(tImpl, "Dispath_Off"),
getPropertyValue(tImpl, "Period" ),
getPropertyValue(tImpl, "Compute_Execution_Time"),
getPropertyValue(tImpl, "Deadline");
9:     if (getPropertyValue(tImpl,
"Dispatch_Protocol")="Aperiodic")
       .....
10:  for all (tImpl,scon)  or (tImpl,rcon)∈ InterParTCM(pImpl) do
11:   begin // Algorithm 1.2: create InterPar CR
12:   if (src(scon) ∈ OutDataPortSet(tImpl) && dest(scon) ∈
OutDataPortSet(pImpl) )  then
13:          instantiate an ArincSrcSamPort object with
     getPropertyValue(dest(scon), "MaxMessageSize");
14:         tImpl.srcSamPortMap ←ArincSrcSamPort;
15: if (src(rcon) ∈ InDataPortSet(pImpl) && dest(rcon) ∈
InDataPortSet(tImpl))  then
16:         instantiate an ArincDestSamPort object with
getPropertyValue(src(rcon),"MaxMessageSize"),
getPropertyValue(src(rcon), "Refresh_Time");
17:           tImpl.destSamPortMap ←ArincDestSamPort;
.     ......
18:  for all (tImpl,tImpl',data,scon,rcon) ∈ IntraParTCM(pImpl) do
19:    // Algorithm 1.3:create IntraPar CR
20:    Call OS API: SET_PARTITION_MODE("NORMAL")
21:     for  all  tImpl ∈ ThreadSet(pImpl) do
22:          Start the ArincProceeImpl object  representing  tImpl;
23:end
```

Line 4 means that for each AADL process implementation which represents a partition instance: first create and instantiate each ARINC653 process of this partition (line 5-9) according to the value of property Dispatch Protocol in the AADL thread implementation (Algorithm 1.1); second, for each inter-partition task communication model instance in this partition, create and instantiate corresponding inter-partition communication resource instance (line 10-17) according to specific port feature in the AADL thread type (Algorithm 1.2); similarly, line 18-19 means create and instantiate each intra-partition communication resource instance in this partition. Line 20 calls OS API to set partition current mode to "NORMAL". In this mode, the process scheduler is active. All processes have been created and those that are in the ready state are able to be started to run (line21-22).

Algorithm 2 shows generate behavior logic function code mapping from AADL653 Task Behavior Model running in each ARINC653 process located in each run() method body. Line 4 indicates that In each AADL thread implementation which represents an ARINC653 process instance, For each transition(line 5) in the annex behavior specification located in each AADL653 thread implementation, there are two parts—guard and action. Firstly ,generate the behavior code from the guard, line 8 indicates that if there exists sampling data in the input sampling port of this ARINC653 process, then to read it from this port, so generated code is as shown in line 9-10,first to get this ArincDestSamPort object representing this port, and then call its READ_SAMPLING_MESSAGE() method. Secondly, generate the inter-partition communication behavior code (sending message) and intra-partition communication behavior code from the action. Finally, for each parameter connection instance in the Value Passing Model of this thread implementation (line 12), generate assigning expression (line 13) to assign the output parameter value of a method to the input parameter of a method as shown in Algorithm 2.3.

Algorithm 2: Task Behavior Code Generation

```
1: Input: AADL653 Task Behavior Model Instance of Each
ARINC653 Process
2: Output: RT-Java Code of Each ARINC653 Process Behavior
3: begin
4: for all tImpl∈ ThreadImplSet do
5:   for all transition  ∈ BAM(tImpl) do
6:     begin // Algorithm 2.1:match transition guards
7:      for all guard ∈ GuardSet(transition)
8:      if (guard==idp? && idp ∈ InDataPortSet(tImpl)) then
9:     ArincDestSamPort ←tImpl.destSamPortMap.get(idp.name)
10:    Call
ArincDestSamPort.READ_SAMPLING_MESSAGE();
11:     // Algorithm 2.2: match transition actions
       .....
12:   for all paraCon ∈ ValuePM(tImpl) do
13:    begin  // Algorithm 2.3:parameter passing
14:      for all (para,para') ∈  paraConnSet(tImpl) do
15:      if (para ∈OutParameterSet(subi) && para' ∈
InParameterSet(subi+1))  then
16:        Generate Assigning Expression:  para'= para;
17:        Insert it between para = methodi () and method i+1
(para');
       .....
18:end
```

V. Case Study

The multi-task flight application example [14] is a scenario concerned with a flight system of computing the current position and the fuel level of an aircraft during its flight through collaboration among three periodic tasks Position Indicator,

Fuel Indicator, and Parameter Refresher. Detailed explanation of this application example can refer to [14].

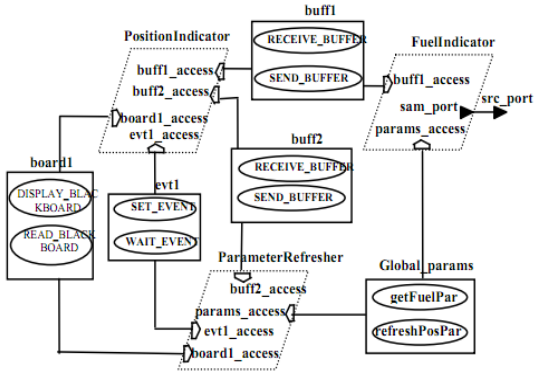

Fig. 1 An AADL653 Task Communication Model of Multi-Task Flight Application

We first build the AADL653 Task Communication Model of this application to describe inter-task static interaction dependency. As shown in Fig. 1, the intra-partition inter-task communication is modeled by the AADL data and the data access connections (e.g. an AADL data implementation named buff2 which is a instance of the data Buffer, and a data access connection from buff2 to the requires data access feature named buff2_access); the inter-partition inter-task communication is modeled by the AADL data/event data port and port connections (e.g. an AADL data port named sam_port which represents a sampling port, as well as a data port connection from sam_port to a sampling channel's source port named src_port).

TABLE IV
AADL653 TASK BEHAVIOR MODEL SLICE OF THE TASK POSITION INDICATOR

```
thread implementation PositionIndicator.Impl
calls
sub_sequence:{
getDate: subprogram getDate;
SEND_BUFFER: subprogram SEND_BUFFER;
WAIT_EVENT: subprogram WAIT_EVENT;
.....
}
connections
ParaCon: parameter getDate.date->SEND_BUFFER.message;
BufferAccessCon: data access buff2_access->
SEND_BUFFER.buffer_access;
.....
annex behavior_specification {**
states
s0:initial return state;
s1,s2,s3...: state;
transitions
so-[]-s1{ getDate!;};
s1-[]-s2{SEND_BUFFER!(this, message);};
s2-[]-s3{WAIT_EVENT!(this);};
.....
**}
end PositionIndicator.Impl;
```

However, the global collaboration interaction sequence is not determined only by building the above model, so we need to build the next refined model that is AADL653 Task Behavior Model for each task respectively so as to determine global synchronous relation and interaction sequence. For example, Table IV shows the AADL653 task behavior model of the task Position Indicator.

Finally, we integrate this AADL653 multi-task flight application model into a specified AADL653 runtime model which is a core module with Flight Management and Flight Control partitions respectively allocated 40ms and 20ms scheduling window in a 60ms major frame.

Taking the partition Flight Management for example, the initialization code excerpt of this partition is shown in Listing 3.

Listing 3 initialization code excerpt of the partition Flight Management

```
void main(){
  Periodicprocess pos = new
PeriodicArincProcessImpl (PROCESS_NAME_TYPE
name,STACK_SIZE_TYPE stack, PRIORITY_TYPE
prio,APEX_TYPES.SYSTEM_TIME_TYPE
period,APEX_TYPES.SYSTEM_TIME_TYPE
timecap,DEADLINE_TYPE deadline); //Arinc Process
Instantiation
  .....
  //Inter-Partition Communication Resource Instantiation
  SrcSamPort sport = new ArincSrcSamPort("ssp1", 30);
//Intra-Partition Communication Resource Instantiation
  Buffer buff1 = new ArincBuffer("buffer1", 20, 30);
  Blackboard Bla1= new ArincBlackboard("board1", 20);
  .....
  pos.start(); // start tasks
 .....
  Set_Partition_Mode("NORMAL"); //call native API
}
```

Taking the behavior model of the task Position Indicator in Table IV for example, the generated behavior code of this task is shown in Listing 4. The global synchronous interaction sequence can be determined by generating each task's behavior code, thus suitable for complex multi-task collaboration interaction situation.

Listing 4 run () method of the task Position Indicator

```
public void run(){
  while ( true ) {
    //Generated task behavior code here
    DateMessage dm = new DateMessage(); // temporary objec
    Date current_date =dm.getDate(); //business logic
    Date message = current_date;  // parameter passing
    ArincBuffer buff2 = getBuffer("buffer2");
    buff2.SEND_BUFFER(message); //buffer comm
    ArincEvent evt1 = getEvent("event1");
    evt1.WAIT_EVENT(); // event sync
    ArincBlackboard board1 = getBlackboard("board1");
    Object pos_mes = board1.READ_BLACKBOARD();
                .....
}
});
    PERIODIC_WAIT (&retCode);
  If (!checkretCode(&retCode){
   // handle overrun or deadline miss here }
}
}
```

The generated XML configuration code excerpt of the multi-task flight application example in Fig. 3 is shown in Listing.

Listing 5 XML configuration code excerpt of the flight example

```
<ARINC_653_Module>
<Partition  PartitionName=" Flight_Management "
Criticality="LEVEL_B">
<Sampling_Port  PortName="ssp1"
MaxMessageSize="30" Direction="SOURCE"/>
</Partition>   //partition attribute and port
.......
<Module_Schedule MajorFrameSeconds="0.060">
        <Partition_Schedule PartitionName="Flight_Management"
PeriodSeconds="0.060"  PeriodDurationSeconds="0.040">
<Window_Schedule WindowIdentifier="101"
WindowStartSeconds="0.0"WindowDurationSeconds="0.040"/>
</Partition_Schedule>
      ……
</Module_Schedule>
<Channel ChannelName="channel1 ">
   <Source>
        <Partition PartitionName=" Flight_Management"
PortName=" ssp1 " />
    </Source>
      ……
     </Channel>
</ARINC_653_Module >
```

Take the multi-task flight application in Fig. 3 for example, running the code of each partition (partly has been shown in previous section) generated automatically by the code generator on the VxWorks 653 Platform. The running results and debugging information of the partition Flight Management are partly presented in Listing 6.

Listing 6 run () method of the task Position Indicator

```
1[test output]:  creating the task PositionIndicator in Partition FM;
2[test output]:  creating the task FuelIndicator in Partition FM;
                 ......
4[test output]:  creating   the source sampling port ssp1;
5[test output]:  creating   the buffer buffer2;
                 ......
11[test output]: the Partition FM enters into NORMAL mode;
12[test output]: the task PositionIndicator is started;
13[test output]: SEND_BUFFER of  buffer2 by PositionIndicator
14[test output]: the task FuelIndicator is started;
15[test output]: the task ParameterRefresher is started;
16[test output]: RECEIVE_BUFFER of  buffer2
byParameterRefresher
17[test output]: WAIT_SEMAPHORE of  sem1 by
ParameterRefresher
18[test output]: SIGNAL_SEMAPHORE of  sem1 by
ParameterRefresher
19[test output]: SET_EVENT of  evt1 by ParameterRefresher
20[test output]: WAIT_EVENT of  evt1 by PositionIndicator
21[test output]: DISPLAY_BLACKBOARD of  board1 by
ParameterRefresher
                 ......
```

VI. Conclusion

In this paper we present an automatic C++ code generation technology from the AADL653 model. A mapping from AADL653 model to high-integrity C++ programming model is proposed and implemented as a basis class library for ARINC653-compatible C++ code generation. Then, we discuss an AADL653-based C++ code generation algorithm in detail. In order to illustrate our approach, a simplified multi-task flight application as a case study is given. In future, we will do real experiment on the VxWorks653 OS to test performance of the generated code.

Acknowledgment

This work is partially supported by Project (No.SKLSDE-2010ZX-05) of the State Key Laboratory of Software Development Environment and National Natural Science Foundation of China (NSFC) under Grant No.61003017.

References

[1] C.B.Watkins and R.Walter, “Transitioning from federated avionics architectures to Integrated Modular Avionics,” In Proceedings of the IEEE/AIAA 26th Digital Avionics Systems Conference (DASC ’07), October 2007.

[2] Airlines electronic engineering committee (AEEC), avionics application software standard interface - ARINC specification 653- part 1 (REQUIRED SERVICES), December 2005, ARINC, Inc.

[3] ARINC 653 - An Avionics Standard for Safe, Partitioned Systems". Wind River Systems / IEEE Seminar. August 2008. http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf. Retrieved 2009-05-30.

[4] "ARINC 653 - An Avionics Standard for Safe, Partitioned Systems" . Wind River Systems / IEEE Seminar. August 2008. Retrieved 2009-05-30.

[5] Heath, Steve (2003). Embedded systems design. EDN series for design engineers (2 ed.). Newnes. p. 2. ISBN 978-0-7506-5546-0.

[6] Sandeep K. Shukla, “Model-Driven Engineering and Safety-Critical Embedded Software,” Computer, vol. 42, no. 9, pp. 93-95,Sept. 2009, doi:10.1109/MC.2009.294

[7] P. Feiler, B. Lewis, and S. Vestal. The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering. In Proceedings of the RTAS 2003, Workshop on Model-Driven Embedded Systems, IEEE CS, 2003.

[8] SAE Aerospace. SAE AS5506: Architecture Analysis and Design Language (AADL), Version 2.0, 2009.

[9] SAE AS5506/2: Behavior Annex, January 17, 2011.

[10] Telecom ParisTech AADL corner, Code generation , Ocarina AADL toolsuite, http://penelope.enst.fr/aadl

[11] Pierre Dissaux, Ellidiss Technologies,STOOD5.2 AADL tutorial,May 2007.

[12] Shenglin Gui, Liang Ma, Lei Luo, Limeng Yin ,Yun Li, UCaG: An Automatic C Code Generator for AADL Based Upon DeltaOS, 978-0-7695-3489-3/08, 2008 IEEE.

[13] Wang, Ying Ma, Dianfu Zhao, Yongwang Zou, Lu Zhao, Xianqi, “An AADL-based modeling method for ARINC653-based avionics software” , 2011 IEEE 35th Annual Computer Software and Applications Conference - COMPSAC 2011 ,pp. 224 - 229, July 2011.

[14] A. Gamatié, T. Gautier, “Synchronous modeling of avionics applications using the Signal language,” In Proceedings of the IEEE 9th Real-Time and Embedded Technology and Applications Symposium (RTAS’03), May 2003.