AO4AADL Compiler

Sihem Loukil

June 2011

We present in this report the main tools used in our work. Then, we detail one of our main contributions. Finally, we present the several steps to implement our ideas.

1 Ocarina suite tools

Ocarina [4] presents a free tool suite written in Ada to manipulate AADL models. This tool suite includes three code generators which have attempted to generate some programming languages such as Ada [5], C [2] and RTSJ (Real Time Specification for Java) [1] from an AADL specification.

The architecture of Ocarina is composed of three main libraries which are easily extensible :

1. A central library (**libocarina**) which presents a low abstraction level to build and manipulate syntactic trees. It consists in a set of routines that allows the two other libraries to handle and exchange syntactic trees of any formalism.

2. A set of frontends that allows to analyze the syntax and semantics of files written in AADL language using routines of the central library. This part receives as input the AADL source files. It applies a syntactic and semantic analysis on these files based on a file written in a pseudo language that resembles to the IDL syntax to describe the grammar of the used language. The main output of this part is an abstract syntactic tree representing the file structure. The secondary outputs consist of any warnings or error messages.

3. A set of backends whose role is to automatically produce code. They are based on trees resulting from the frontends. This part receives as input the resulting tree from the previous part. It applies a tree expansion to simplify structures and produce an easier and richer tree. After this expansion phase, the obtained AADL tree is transformed into a syntactic tree of the target language. Finally, this tree allows to proceed to the code generation.

The code generators offered by the Ocarina suite tools allow the generation of functional code from basic components of the AADL description. As an example, we present here some of the transformation rules used by the RTSJ generator. It allows to translate each node (process) of the architecture described in AADL into a set of RTSJ classes using a well defined set of transformation rules presented in [1].

In the following, we present some examples of the generated classes :

Subprograms class The RTSJ generator translates each subprogram called by a task into a method of a class called ***Subprograms***. Listing 2 presents an example of this transformation applied on the subprogram `Ping_Spg` given in Listing 1. As shown in the AADL specification, this subprogram takes one `Simple_Type` parameter called `Data_Sink`. So, this subprogram is transformed into a method called `PingSpgImpl` (line 2, Listing 2) which takes as parameter a variable called `dataSink`.

```
1 subprogram Ping_Spg
2 features
3   Data_Sink : in parameter Simple_Type;
4 end Ping_Spg;
```

Listing 1: Example of AADL subprogram specification

```
1 public class SubPrograms {
2    public static void PingSpgImpl(GeneratedTypes.SimpleType dataSink) {
3    ...
4    }
5 }
```

Listing 2: Part of the generated RTSJ code

GeneratedTypes class This class includes all used data types. Listing 3 shows the transformation of the variable `dataSink` defined in line 2 of Listing 2. Each data type is transformed to a class that takes the same name of the data and implements the `GeneratedType` interface (line 6) available at the package ***fr.enst.ocarina.polyORB_HI_runtime*** (line 1).

```
1 import fr.enst.ocarina.polyORB_HI_runtime.GeneratedType;
2 ...
3
4 public class GeneratedTypes {
5
6   public static class SimpleType implements GeneratedType {
7     ....
8   }
9 }
```

Listing 3: Part of the generated RTSJ code

Deployment class This class gathers all information of the current node and the interaction between the current node and the other nodes of the application. This class contains the identification number of the node, the number of nodes in the application, the number of local tasks it possesses and the identification numbers of the ports.

Activity class This class is used to manage the tasks and their job. The tasks which belong to the current node are declared and initialized.

A ***PortsRouter***[1] object has to be created for each task to store or get the messages in the global queue. To perform the proper routing of messages via ports, all functions used by the ***PortsRouter*** object are created in this class. Generally, the same functions as those in the

[1] a class available at the package ***fr.enst.ocarina.polyORB_HI_runtime***

PortsRouter object are created but a new parameter is added for each one, it's the entity which allows in each function to know what port is addressed. Then, in each case statement the right functions with right parameters is called.

For example, we find the method ***public void sendOutput(int entity, OutPort outPort)*** supposed to manage the sending of the data through an output port.

2 Aspect code generation

We present in this section our extension to the Ocarina tool suite. It consists in the generation of aspect programming language from aspectual annexes described in AO4AADL. As mentioned, the generation of functional code from basic components of the AADL description is already ensured by the Ocarina tool suite. Our idea is therefore to extend Ocarina to ensure translation of the aspectual part taking advantage of the available generators.

Aspects described in AO4AADL can be translated in different aspect languages since it is generic.

The study of the various existing aspect-oriented programming languages has proofed that AspectJ [3] is the most popular aspect language due to its widespread use with an emphasis on simplicity and ease of work for end users. AspectJ presents an aspect-oriented extension for Java language. Following this study, we chose to start generating aspects written in AspectJ language from the architectural aspects described in AO4AADL. To apply these aspects, we need a base system described in Java language in order to get a complete Java prototype. For this reason, we adopted in our approach the RTSJ code generator from an AADL specification that is already available in the Ocarina tool suite.

We define a set of transformation rules to map AO4AADL aspects into AspectJ aspects. These transformation rules are based on the RTSJ generator ones in order to ensure the consistency between the RTSJ code and the generated AspectJ code. In this way, a complete Java prototype can be obtained by integrating automatically the generated AspectJ aspects in the RTSJ code.

In the following, we present some examples of the transformation rules from AO4AADL to AspectJ. As we mentioned previously, these rules are based on the RTSJ transformation rules.

The transformation rules from AO4AADL to AspectJ are classified into two categories : transformation rules for the pointcut and those for the advice part.

2.1 Transformation rules of the pointcut

We present in this part the transformation rules[2] used to translate the AO4AADL pointcuts into AspectJ. As the pointcut presents a set of joinpoints, we concentrate on the transformation of each type of joinpoint.

- *Joinpoint intercepting a subprogram*

  Table 1 presents the rule which translates a joinpoint intercepting a subprogram. As shown in Table 1, the interception of a subprogram at architectural level in AO4AADL returns to intercept, in AspectJ, the right method of the ***Subprograms*** class (`SubPrograms.<Subprogram_Identifier>Impl`) already generated by the RTSJ generator. According to the syntax of AspectJ, we have to specify the returning type of the intercepted method. As it was mentioned, all generated methods are ***public static void***. So we are not interested in the generation of the returning type of the intercepted method. That's why, we use here the wildcard "*" in the generated AspectJ code. For the parameter types used in AO4AADL, they obey to the transformation rules of the data types defined by the RTSJ generator.

Table 1: Transformation rule of a joinpoint intercepting a subprogram

AO4AADL specification:
call/execution subprogram (<Subprogram_Identifier> (<Paramerter_types>))
Generated AspectJ code:
call/execution (* SubPrograms.<Subprogram_Identifier>Impl (<Generated_Parameter_Types>))

Listing 5 presents an example of the generated AspectJ code from a pointcut described in AO4AADL which intercepts the execution of the `Ping_Spg` subprogram taking a `Simple_Type` parameter. The AO4AADL specification is presented in Listing 4.

```
1 aspect AspectName {
2    pointcut PointcutName () : execution subprogram (PingSpg (Simple_Type));
3    ...
4    }
```

Listing 4: Example of an AO4AADL pointcut intercepting a subprogram

```
1 aspect AspectName {
2    pointcut PointcutName () : execution (* SubPrograms.PingSpgImpl
3                                          (GeneratedTypes.Simple_Type));
4    ...
5    }
```

[2]The full version of these rules is available at `http://www.redcad.org/projects/AO4AADL`

Listing 5: Generated AspectJ code from Listing 4

We have to note that no changes are applied neither to the aspect and the pointcut names nor to the keywords '***aspect***', '***pointcut***' and '***call/execution***'.

- *Joinpoint intercepting an output port*

  In this case, the designer can intercept either the event of sending a message or the type and/or the value of the data to send.

  - *Joinpoint intercepting the event of sending a message*

    In the case of intercepting the event of sending a message through an output port at architectural level, (`call/execution outport (<Output_Port_Identifier> (..))`) we have to look for the methods (or subprograms) that carry out the exchange of messages through this port at the implementation level. We suppose in this case that we are not interested in the type of the data to send (the wildcard '**..**').

    According to the RTSJ generator, these methods are defined in the ***Activity*** class. The study of these methods shows that the interception of an output port in AO4AADL returns to intercept, in AspectJ, the method ***sendOutput()*** of the ***Activity*** class. This is illustrated in Table 2.

Table 2: Transformation rule of a joinpoint intercepting an output port

AO4AADL specification:
call/execution outport (<Output_Port_Identifier> (..))
Generated AspectJ code:
call/execution (* Activity.sendOutput(..))

Since all the used methods are ***public static void***, we are not then interested in its returning type. We use therefore the wildcard "*****" in the generated code.

As an example of this transformation rule, we present in Listing 7 the AspectJ code generated from the pointcut of the aspect defined in Listing 6.

```
1 thread implementation ValidationTh.Impl
2   ...
3   annex ao4aadl {**
4     aspect CheckCode {
5         pointcut Verification(): call outport (Restore_Code_out_V (..));
6         ...
7     }
```

```
8   **}
9 end ValidationTh.Impl;
```

Listing 6: Example of AO4AADL annex

```
1 aspect CheckCode{
2    pointcut Verification(): call (* Activity.sendOutput(..));
3    ...
4 }
```

Listing 7: Generated AspectJ pointcut from listing 6

– *Joinpoint intercepting the type and/or the value of the data to send*

If the designer is interested in intercepting the type or the value of the data to send through an output port, then the corresponding method at implementation level is the ***public static void putValue*** of the ***Activity*** class. This method has the following structure : ***public static void putValue (int entity, OutPort outPort, GeneratedType value)*** where entity presents the identifier of the intercepted task, the ***Outport*** object refers to the intercepted port while ***value*** is the value of the data to send.

When intercepting only the type of the data to send (`call/execution outport (<Output_Port_Identifier>(<Data_Type>))`), the used rule in this case is presented in Table 3. While, the interception of the value of this data is translated as shown in Table 4.

Table 3: Transformation rule of a joinpoint intercepting the type of the data to send through an output port

AO4AADL specification:
call/execution outport (<Output_Port_Identifier> (<Data_Type>))
Generated AspectJ code:
call/execution (* Activity.putValue(.., .., GeneratedTypes.<Data_Type>))

Table 4: Transformation rule of a joinpoint intercepting the value of the data to send through an output port

AO4AADL specification:
call/execution outport(<Output_Port_Identifier>(..)) && args (<Data_Identifier>)
Generated AspectJ code
call/execution (* Activity.putValue .. && args(..,.., <DATA_IDENTIFIER>)

In the case of intercepting both the type and the value of the data to send, we have simply to combine the rules presented in Table 3 and Table 4.

As shown in the transformation rules presented in Tables 2, 3 and 4, the identifier of the intercepted port is not specified in the generated AspectJ code. To deal with this limitaion and to specify that the code of the advice is executed only when the port specified at the architectural joinpoint is really catched, the generated advice code should be inserted under a condition that verifies the port's identifier. This condition is used to check if the ***OutPort*** parameter of the intercepted method in the generated AspectJ code corresponds to the identifier of the intercepted port specified in the AO4AADL code. Therefore, the generated advice code (more specifically the advice action subclause) has the following structure :

- For a **before** or **after** advice

```
if ((OutPort)(thisJoinPoint.getArgs([1]))).getPortNumber()==
 Deployment.<Out_PORT_ID>){
 //Generated advice action code
}
```

  `thisJoinPoint` is an AspectJ object which refers to the intercepted joinpoint. The method `getArgs([1])` applied to this object refers to the second parameter of the intercepted joinpoint. The returned object is casted to the type ***OutPort*** to apply the method `getPortNumber()` which returns the number of the intercepted port. The `<Out_PORT_ID>` is defined in the ***Deployment*** class. Each port has an identification number which has the format :
  `<NODE_IDENTIFIER>_<THREAD_IDENTIFIER>_<PORT_IDENTIFIER>_K`.
  For example the identification number of the output port `RestoreCode_out_V` of the `Validation` task instance of the `ValidationTh` task defined in the process `Node_Customer` is :
  `Node_Customer_Validation_RestoreCode_out_V_K`.
  If the identifier of the port is not specified in the joinpoint specified in the AO4AADL code (we use the wildcard '**'), this condition will not be generated since this aspect is applied to every output port.

- For an **around** advice
  - when intercepting the event of sending a data through an output port

```
if ((OutPort) (thisJoinPoint.getArgs([1]))).getPortNumber()==
 Deployment .<Out_PORT_ID>) {
 //Generated advice action code
}
else{
    proceed(<PARAMETER_PROFILE>);
}
```

The else condition is used here to define the executed code when the intercepted output port in the AspectJ aspect is different from the one intercepted in the AO4AADL aspect. We call then the ***proceed()*** action to carry on with the execution of the basic RTSJ code.

* when intercepting the type and/or the value of the data to send, an additional pointcut will be generated in the aspect. Actually, the intercepted method ***putValue()*** is associated with the method ***sendOutput()***. If the first method is not executed by calling a ***proceed()*** action then the second method should not be executed. So the second pointcut is supposed to intercept the second method (***sendOutput***). The name of this pointcut is the combination of the first pointcut name and the suffix ***_sendOutput***. Moreover, two advices are generated in this case. The first advice, which is associated to the first pointcut, includes the advice code generated from the AO4AADL advice code. We add to this code a static boolean variable ***put_OK*** initialized to ***false***. This variable takes the value ***true*** whenever the method ***putValue()*** is executed by calling the ***proceed()*** action, otherwise it takes the value ***false***. Concerning the second advice, it is associated to the second pointcut and has the following structure :

```
if ((OutPort) (thisJoinPoint.getArgs([1]))).getPortNumber()==
 Deployment .<Out_PORT_ID>) {
 if (put_OK){
    proceed();
    }
 }
else{
    proceed();
}
```

- *Joinpoint intercepting an input port*

  For the joinpoints which intercept an input port, we follow the same reasoning made in the case of intercepting an output port. Actually, the study of the methods of the `Activity` class has proofed that the interception of an input port at architectural level returns to intercept, at implementation level, two main methods which are : (1) the method ***public static void storeReceivedMessage*** which enables the designer to know if there is a new incoming message to the input port, and (2) ***public static void getValue*** allowing to intercept the type and/or the value of the received data.

  As a result, three transformation rules are used to translate a joinpoint intercepting an input port from AO4AADL to AspectJ.

– *Joinpoint intercepting the event of receiving a message*

Table 5 presents the rule applied in the the case of intercepting the event of receiving a message.

Table 5: Transformation rule of a joinpoint intercepting an input port

AO4AADL specification:
call/execution inport (<Input_Port_Identifier> (..))
Generated AspectJ code:
call/execution (* Activity.storeReceivedMessage(..))

– *Joinpoint intercepting the type and/or the value of the received data*

In this case, two transformation rules are defined. Table 6 specifies the rule used to intercept the type of the incoming data while the rule presented in Table 7 is used when intercepting the value of the incoming data.

Table 6: Transformation rule of a joinpoint intercepting the type of the incoming data to an input port

AO4AADL specification:
call/execution inport (<Input_Port_Identifier> (<Data_Type>))
Generated AspectJ code:
call/execution (* Activity.getValue(.., .., GeneratedTypes.<Data_Type>))

Table 7: Transformation rule of a joinpoint intercepting the value of the incoming data to an input port

AO4AADL specification:
call/execution inport(<Input_Port_Identifier>(..)) && args (<Data_Identifier>)
Generated AspectJ code:
call/execution (* Activity.getValue(..)) && args (.., .., <DATA_IDENTIFIER>)

The combination of the two last rules leads to the interception of both the type and the value of the incoming data.

As we mentioned in the case of intercepting an output port, the generated advice should be inserted under a condition that verifies the port's identifier.

2.2 Transformation rules of the advice

We present in this part some examples of the transformation rules[3] used to translate the instructions used in the advice subclause from AO4AADL to AspectJ.

For the generated advice declaration and the pointcut reference parts, they keep the same structure used in AO4AADL. We have only omitted the keyword '***advice***' and applied the transformation rules defined by the RTSJ generator to translate the declared parameters.

Concerning the ***variables*** and the ***initially*** parts, they will be combined in the generated AspectJ code. For each variable declared in the ***variables*** subclause, the transformation rule presented in Table 8 is applied.

Table 8: Transformation rule of a variable declared in the ***variables*** subclause

AO4AADL specification:
<Variable_Identifier> : <Variable_Type>;
Generated AspectJ code:
public static final GeneratedTypes.<Variable_Type> <VARIABLE_IDENTIFIER> = new GeneratedTypes.<Variable_Type>(<Variable_Value>);

where `<Variable_Value>` corresponds to the initial value of the variable defined in the ***initially*** subclause. If this variable is not initialized then a default value is attributed depending on the type of the variable. For example, the default value of an integer type is 0. Listing 8 presents the generated AspectJ code for a declared and initialized integer variable.

```
//AO4AADL specification
variables { counter : Integer_Type; }
initially { counter := 1; }

//Generated AspectJ code
public static final GeneratedTypes.IntegerType COUNTER =
new GeneratedTypes.IntegerType (1);
```

Listing 8: Generated AspectJ code for a declared and initialized integer variable

What about the action subclause, we present the main defined transformation rules to obtain an AspectJ code from an AO4AADL specification. In this subclause, the identifiers of the variables and parameters are translated according to the transformation rule presented in Table 9.

This rule is applied for all the identifiers except those used in the communication and the ***proceed()*** actions. In this case, the generated identifiers have the following structure : `<VARIABLE_OR_PARAMETER_IDENTIFIER>`.

[3]The full version of these rules is available at `http://www.redcad.org/projects/AO4AADL`

Table 9: Transformation rule of the identifiers of variables and parameters used in the action subclause

AO4AADL specification:
<Variable_or_Parameter_Identifier>
Generated AspectJ code:
<VARIABLE_OR_PARAMETER_IDENTIFIER>.value

For the communication actions, we followed the transformation rules used by the RTSJ generator. According to this generator, for a given node, the generated ***Subprograms*** class includes only the subprograms called by a task forming this node. Unfortunately, a subprogram can be used inside the advice action without being called by any task. Our idea is then to extend this class by generating, for each node, the methods that correspond to the called subprograms in the AO4AADL aspects. Therefore, calling a subprogram inside the advice action in the AO4AADL specification returns to call the corresponding method in the ***Subprograms*** class for the generated AspectJ code. Table 10 presents the transformation rule of the operation of sending a message through an output port.

Table 10: Transformation rule of the operation of sending a message through an output port

AO4AADL specification:
<Output_Port_Identifier> !(<Data_Identifier>)
Generated AspectJ code
try {
Activity.putValue (Deployment.<THREAD_ID>, Activity.<OUTPUT_PORT_IDENTIFIER>,
<DATA_IDENTIFIER>);
}
catch (ProgramException e) {
Debug.debugMessage("error while putting vale in port" +
Deployment.<OUTPUT_PORT_IDENTIFIER>.getPortNumber() +
"from entity" + Deployment.<THREAD_ID>);
}
try {
Activity.sendOutput(Deployment.<THREAD_ID>, Activity.<OUTPUT_PORT_IDENTIFIER>);
}
catch (ProgramException e) {
Debug.debugMessage("Error while sending output in port +
Deployment.<OUTPUT_PORT_IDENTIFIER>.getPortNumber() +
"from entity "+ Deployment.<THREAD_ID>);
}

As shown in Table 10, this operation is translated into two methods : (1) the ***putValue()*** method which enables to put the data on the port and (2) the ***sendOutput()*** method to send the data to the corresponding destination.

For the operation of receiving a message by an input port, it is translated into two other methods from the ***Activity*** class which are : (1) the ***getValue()*** method which enables to get the data on the port and (2) the ***nextValue()*** method to dequeue one entry from the input port's queue.

Concerning the ***proceed()*** action, it keeps the same structure presented in AO4AADL respecting the transformation rules applied to the identifiers. What about the other types of instructions used in the action subclause, they generally keep the same structure with some modification to suite the syntax and semantics of AspectJ. Actually, the transformation from AO4AADL to AspectJ is not so hard since the syntax and semantics of our language is very close to the AspectJ ones.

Listing 10 presents the AspectJ code generated from the AO4AADL aspect presented in Listing 9.

```
1  aspect CheckCode{
2     pointcut Verification(): call outport (Restore_Code_out_V (..));
3     advice around():Verification(){
4
5       variables { counter : Integer_Type; message : String_Type }
6       initially { counter:=1; message:= "Card Rejected !"; }
7
8       if(counter=3){
9         Rejected_Card_out_V!(message);
10      }
11      else{
12        proceed();
13        counter := counter+1;
14      } }}
```

Listing 9: Example of an aspect described in AO4AADL

```
1  //List of import instructions generated in every AspectJ file
2  import fr.enst.ocarina.polyORB_HI_runtime.OutPort;
3  import fr.enst.ocarina.polyORB_HI_runtime.InPort;
4  import fr.enst.ocarina.polyORB_HI_runtime.ProgramException;
5  import fr.enst.ocarina.polyORB_HI_runtime.Debug;
6  import fr.enst.ocarina.polyORB_HI_runtime.Message;
7
8  aspect CheckCode{
9     //The generated AspectJ pointcut
10    //In our case, we intercept the event of sending a data through an output port
11    pointcut Verification(): call (* Activity.sendOutput (..));
12
13    //The variables generated from the variables and initially sections
14    public static final GeneratedTypes.IntegerType COUNTER =
15    new GeneratedTypes.IntegerType(1);
16
17    public static final GeneratedTypes.StringType MESSAGE =
18    new GeneratedTypes.StringType("Card Rejected !");
19    //The generated Advice code
```

```
20    void around():Verification(){ //Advice declaration
21
22      //The condition that checks the intercepted port
23      //In our case the identification number of the output port is
24      //NODE_CUSTOMER_VALIDATION_RESTORECODE_OUT_K
25      if (((OutPort)(thisJoinPoint.getArgs()[1])).getPortNumber()==
26                      Deployment.NODE_CUSTOMER_VALIDATION_RESTORECODE_OUT_K){
27
28        //The generated advice action part
29      if(COUNTER.value==3){
30      // Set the call sequence OUT port values
31           try {
32           Activity.putValue(Deployment.NODE_CUSTOMER_VALIDATION_K,
33                                    Activity.REJECTEDCARD_OUT_V, MESSAGE);
34        }
35        catch (ProgramException e) {
36             Debug.debugMessage("Error while putting value in port "
37              +Deployment.NODE_CUSTOMER_VALIDATION_REJECTEDCARD_OUT_K
38              + " from entity " + Deployment.NODE_CUstomer_VALIDATION_K);
39        }
40        // Send the call sequence OUT port values
41        try {
42                  Activity.sendOutput(Deployment.NODE_CUSTOMER_VALIDATION_K,
43                                           Activity.REJECTEDCARD_OUT_V);}
44        catch (ProgramException e) {
45             Debug.debugMessage("Error while sending output in port "
46                    + Deployment.NODE_CUSTOMER_VALIDATION_REJECTEDCARD_OUT_K
47              + " from entity " + Deployment.NODE_CUSTOMER_VALIDATION_K);}
48            COUNTER.value=1;
49       }
50        else{
51        //The proceed action
52        proceed();
53
54       //The generated code from the assignment action
55       //In this case, the operator ":=" used in AO4AADL is transformed
56       //to the operator "="
57       //For the arithmetic operations, no changes are applied to
58       //the operators {*, /, +, -}
59        COUNTER.value=COUNTER.value+1;
60       }}
61      //This code will be executed if the intercepted port is not RestoreCode_Out_V.
62      else{proceed();}}}
```

Listing 10: AspectJ code generated from listing 9

In Listing 10, lines 2–6 shows the list of the imported files that should be present in every generated AspectJ file. Line 11 corresponds to the generated pointcut. Lines 14–18 presents the list of the generated local variables. The generated advice code is presented in lines 20–62.

References

[1] Thomas Autret. Génération de code Real-Time Java pour systèmes temps-réel. Master's thesis, Universit Pierre & Marie Curie, Paris VI, sep 2009.

[2] J. Delange, J. Hugues, L. Pautet, and B. Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *4th European Congress ERTS*, Toulouse, France, jan 2008.

[3] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[4] T. Vergnaud, B. Zalila, and J. Hugues. Ocarina: a Compiler for the AADL. Technical report, École Nationale Supérieure des Télécommunications, jun 2006.

[5] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, nov 2008.