Transformation rules from AO4AADL to AspectJ

1 Introduction

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.

2 Transformation of Keywords

Table1. Transformation of keywords

AO4AADL	**AspectJ**
aspect	aspect
pointcut	pointcut
call inport	call
call outport	
call inoutport	
call subprogram	
execution inport	call
execution outport	
execution inoutport	
execution subprogram	
args	args
advice	advice
before	before
after	after
around	void around

AO4AADL	**AspectJ**
proceed	proceed
if	if
else	else
for	for
while	while
variables	-
initially	-
applies	-
to	-
precedence	Declare precedence :
in	-
count	-
thread	-
process	-
subprogram	-

3 Transformation of Data

Since the data used in the AO4AADL description are already declared in the basic AADL model, we simply use generated classes from this model using the

RTSJ code generator. In other words, there will be no new transformation rules of data generation for aspect-oriented model.

4 Transformation of identifiers

In AO4AADL, identifiers still unchanged in AspectJ. If a name conflict can be generated in the AspectJ code, this collision is resolved by prefixing the generated name using an underscore (_).

Identifiers of ports and subprograms they are transformed using the following rules and taking into consideration the rules of the RTSJ generator :

Table2. Transformation rule of a joinpoint intercepting an input port

AO4AADL specification	<Port_Identifier>
Generated AspectJ code	Activity.<PORT_IDENTIFIER>

For example, considering a port called « `RestoreCode_out_V` », the identifier of this port will be in AspectJ « `Activity.RESTORECODE_OUT_V` ».

Table3. Transformation rule of a joinpoint intercepting an input port

AO4AADL specification	<Subprogram_Identifier>
Generated AspectJ code	SubPrograms.<Subprogram_Identifier>Impl

For example, considering a subprogram called « `CheckValidity` », the identifier of this subprogram will be in AspectJ « `SubPrograms.CheckValidityImpl` ».

5 Transformation of parameters

In AO4AADL, the declaration of the list of parameters used in the *pointcuts* and the *advices* have always the following structure :

```
<Parameter_Identifier> : <Parameter_Type>
```

In AspectJ, the order of declaration will be reversed by not reporting the two points separating the identifier of its type. For the type of parameter, it obeys the transformation rules of the RTSJ generator. Thus, the declaration of variables in AspectJ has the following structure :

```
GeneratedTypes.<Parameter_Type> <PARAMETER_IDENTIFIER>
```

6 Transformation of *precedence* subclause

In AO4AADL, the declaration of the priority of aspects is performed in the **precedence** subclause. This subclause has always the following structure :

```
precedence <Aspect_Identifier> { , <Aspect_Identifier>}
```

The **precedence** subclause is mapped to AspectJ while keeping the same structure. We have just to replace the keyword **precedence** by **Declare precedence :**. Thus the generated AspectJ code has the structure as follows :

```
Declare precedence : <Aspect_Identifier> { , <Aspect_Identifier>}
```

7 Transformation of pointcuts

A pointcut consists of one or a combination of a set of joinpoints using logical operators (**&&**) and (||).

The transformation of a pointcut returns then to the combination of the transformations of each joinpoint using the same logical operators.

8 Transformation of joinpoints

The main concepts that we catch in the specification of a joinpoint are ports and subprograms. In the following, we present how all the joinpoint types (**inport outport** and **subprogram**) with a **call** or **execution** primitive are translated into AspectJ to obtain a valid code.

The **inoutport** joinpoints are not yet treated as the **inout** port type is rarely used in AADL models and because it can easily switch ports by using **in** and **out** ports.

For the **args** primitive, it is kept unchanged in AspectJ when it is combined with a **subprogram** joinpoint. Otherwise, it follows the transformation rules that we present in this section.

– *Joinpoint intercepting a subprogram*
  Table 4 presents the rule which translate a joinpoint intercepting a subprogram. As shown in Table 4, 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 the 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. Listing 1.2 presents an example of the generated AspectJ code from a pointcut described in AO4AADL which intercepts the execution of the

Table4. 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>))

`Ping_Spg` subprogram taking a `Simple_Type` parameter. The AO4AADL specification is presented in Listing 1.1.

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

Listing 1.1. Example of an AO4AADL pointcut intercepting a subprogram

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

Listing 1.2. Generated AspectJ code from Listing 1.1

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***'. When the designer wants to intercept the values of the parameters defined in the considered subprogram he aught to use the following transformation rule :

Table5. Transformation rule of a joinpoint intercepting the type of the data to send through an output port

AO4AADL specification	**call/execution subprogram** (<Subprogram_Identifier> (<Paramerter_Types>)) **&& args** (<Parameter_Identifiers>)
Generated AspectJ code	**call/execution (*** SubPrograms.¡Subprogram_Identifier¿Impl (<Generated_Paramerter_Types>)) **&& args** (<PARAMETER_IDENTIFIERS>)

Listing 1.3 presents a joinpoint described in AO4AADL intercepting the call of a subprogram called `spg` tanking two parameters. Listing 1.4 the corresponding generated AspectJ code.

```
1 pointcut P (i:Integer_Type, j:Integer_Type) :
2                                call subprogram spg (..) && args (i,j);
```

Listing 1.3. Transformation d'un *joinpoint* interceptant un sous programme

```
1 pointcut P (GeneratedTypes.IntegerType I,GeneratedTypes.IntegerType J):
2                                 call Subprogram.spgImpl (..) && args (I,J);
```

Listing 1.4. Generated AspectJ code from Listing 1.4

– *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 (`call/execution outport (<Output_Port_Identifier> (..))`) at architectural level, 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 6.

Table6. 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 1.6 the AspectJ code generated from the pointcut of the aspect defined in Listing 1.5.

```
1
2     aspect CheckCode {
3         pointcut Verification(): call outport (Restore_Code_out_V (..));
4         ...
5     }
```

Listing 1.5. Example of AO4AADL pointcut intercepting an output port

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

Listing 1.6. Generated AspectJ pointcut from listing 1.5

– *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 thread, 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 7. While, the interception of the value of this data is translated as shown in Table 8.

Table7. 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>))

The first and second types of the arguments of the method **putValue()** are not specified because we are not interested in these arguments. In addition, these two types of arguments are always fixed.

Table8. 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 generated AspectJ code, the first and the second arguments of the **args** primitive, corresponding to the first and the second parameters of the method **putValue()**, are not specified because they are constant and can be deducted easily from the generated RTSJ code.
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 7 and Table 8.
Listing 1.7 presents a joinpoint intercepting the value of the data to send through an output port described in AO4AADL. The corresponding AspectJ code is presented in Listing 1.8.

```
1 pointcut catchOutportData(value: Integer_Type):
2                         call outport (Port_out(..))&& args (value);
```

Listing 1.7. Transformation d'un *joinpoint* interceptant un port de sortie

```
1 pointcut catchOutportData (GeneratedTypes.IntegerType VALUE):
2                         call(* Activity.putValue(..)) && args (..,..,VALUE);
```

Listing 1.8. Transformation d'un *joinpoint* interceptant un port de sortie

As shown in the transformation rules presented in Tables 6, 7 and 8, 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` thread instance of the `ValidationTh` thread 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();
}
```

Listing 1.9 shows an example of an AO4AADL aspect intercepting the value of information to send via an output port. An **around** advice is applied to the pointcut. Listing 1.10 shows the generated AspectJ code from Listing 1.9.

```
1  aspect Example{
2    pointcut catchOutportData(argument: Integer_Type):
3                          call outport(Port_out(..))&& args (argument);
4    advice around (argument: Integer_Type): catchOutportData(argument){
5         if (condition_1){
6              proceed(argument);
7              argument:=argument+1;
8          }
9          else{ //instructions_else
10         } }}
```

Listing 1.9. Aspect interceptant un port de sortie avec un *advice* ***around***

```
1  aspect Example{
2    pointcut catchOutportData (GeneratedTypes.IntegerType ARGUMENT):
3                        call(* Activity.putValue(..)) && args (..,..,ARGUMENT);
4
5    pointcut catchOutportData_sendOutput ():call(* Activity.sendOutput(..));
6
7  public static boolean put_OK = false;
```

```
 8
 9 void around (GeneratedTypes.IntegerType ARGUMENT): catchOutportData(ARGUMENT){
10
11  if ((OutPort)(thisJoinPoint.getArgs([1])).getPortNumber()==
12                              Deployment.NODE_Customer_VALIDATION_PORT_OUT_K){
13         if (condition_1){
14             proceed(ARGUMENT);
15             put_OK = true;
16             ARGUMENT.value=ARGUMENT.value+1;
17         }
18         else{
19             //instructions_else
20         }}
21       else{ proceed(ARGUMENT);}}
22
23 void around ():catchOutportData_sendOutput (){
24
25 if ((OutPort)(thisJoinPoint.getArgs([1])).getPortNumber()==
26                              Deployment.NODE_CUSTOMER_VALIDATION_PORT_OUT_K){
27          if (put_OK){
28              proceed();
29          }}
30       else{ proceed();}}
```

Listing 1.10. Code AspectJ gnr partir de l'exemple 1.9

– *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 9 presents the rule applied in the the case of intercepting the event of receiving a message.

Table9. Transformation rule of a joinpoint intercepting an input port

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

Listing 1.11 shows an AO4AADL joinpoint intercepting the event of receiving a data on an input port called `Port_in`. The corresponding AspectJ code is presented in Listing 1.12.

```
1 pointcut catchInport():call inport (Port_in(..));
```

Listing 1.11. Transformation d'un *joinpoint* interceptant un port d'entre

```
1 pointcut catchInport():call (* Activity.storeReceivedMessage(..));
```

Listing 1.12. Transformation d'un *joinpoint* interceptant un port d'entre

– *Joinpoint intercepting the type and/or the value of the received data*
In this case, two transformation rules are defined. Table 10 specifies the rule used to intercept the type of the incoming data while the rule presented in Table 11 is used when intercepting the value of the incoming data.

Table10. 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>))

The types of the first and the second arguments of the method **putValue()** are not specified for the same reasons as in the case of output ports.

Table11. 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 first two arguments of the **args** primitive are not specified for the same reasons as in the case of output ports.
The combination of the two last rules leads to the interception of both the type and the value of the incoming data.

Listing 1.13 presents an AO4AADL joinpoint intercepting the type and the value of the received data on an input port. The corresponding AspectJ code is presented in Listing 1.14.

```
1
2 pointcut catchInportTypeData(argument: Integer_Type):
```

```
3                   call inport (Port_in(Integer_Type)) && args (argument);
```

Listing 1.13. Transformation d'un *joinpoint* interceptant un port d'entre

```
1
2 pointcut catchInportTypeData (GeneratedTypes.IntegerType ARGUMENT):
3                   call(* Activity.getValue(..,..,GeneratedTypes.IntegerType))
4                   && args(..,..,ARGUMENT);
```

Listing 1.14. Transformation d'un *joinpoint* interceptant un port d'entre

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. Therefore, the generated advice code has the following structure :

- For a **before** or **after** advice
  - when intercepting the event of receiving a data

```
if ((InPort)(thisJoinPoint.getArgs([0]))).getPortNumber()==
 Deployment.<In_PORT_ID>){
 //Generated advice action code
}
```

  - when intercepting the type and/or the value of the received data

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

  The difference here between these two cases consists in the position of the ***InPort*** parameter in the intercepted method. For the ***public static void storeReceivedMessage (InPort inPort , Message msg , long timeStamp)***, this parameter takes the position 0 ; while for the ***public static void getValue (int entity, InPort inPort, GeneratedType destination)*** the ***InPort*** parameter takes the position 1.
- For an **around** advice
  - when intercepting the event of receiving a data

```
 if ((InPort) (thisJoinPoint.getArgs([0]))).getPortNumber()==
 Deployment.<In_PORT_ID>) {
 //Generated advice action code
}
else{
    proceed(<PARAMETER_PROFILE>);
}
```

  - when intercepting the type and/or the value of the received data

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

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 input port.

9 Transformation rules of the advice

We present in this part the transformation rules used to translate the instructions used in the advice subclause from AO4AADL to AspectJ.

9.1 Transformation rules of the advice declaration and the pointcut reference

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 12 is applied.

Table12. 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 1.15 presents the generated AspectJ code for a declared and initialized integer variable. Listing 1.16 shows the generated code for a declared and not 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 1.15. Generated AspectJ code for a declared and initialized integer variable

```
//AO4AADL specification
variables { counter : Integer_Type; }

//Generated AspectJ code
public static final GeneratedTypes.IntegerType COUNTER =
new GeneratedTypes.IntegerType (0);
```

Listing 1.16. Generated AspectJ code for a declared and not initialized integer variable

9.2 Transformation rules of the action subclause

What about the action subclause, we present the main defined transformation rules to obtain an AspectJ code from an AO4AADL specification.

Transformation of identifiers In this subclause, the identifiers of the variables and parameters are translated according to the transformation rule presented in Table 13.

Table13. 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**

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>`.

Transformation des constants Constants used in AO4AADL are the numerical values, the number of messages in the queue for an entry port and the boolean values.

For the numerical and the boolean values, they are kept without modification. For the second type of constant, the transformation rule is presented in table 14 :

Table14. Transformation rule of the number of messages in the queue for an entry port

AO4AADL specification	<Input_Port_Identifier>**'count**
Generated AspectJ code	Activity.getCount(Deployment.<THREAD_ID>, Activity.<INPUT_PORT_IDENTIFIER>)

Transformation of the assignment operation In AO4AADL, the structure of the assignment operation is :

```
<Variable_or_Parameter_Identifier> := <Expression> ;
```

In AspectJ, this structure is kept as it is replacing the assignment sign (**:=**) by the equal sign (=) while respecting the rules of transformation is presented in table 15 :

Table15. Transformation rule of the number of messages in the queue for an entry port

AO4AADL specification	<Variable_or_Parameter_Identifier> **:=** <Expression> **;**
Generated AspectJ code	<VARIABLE_OR_PARAMETER_IDENTIFIER>**.value =** <Generated_Expression> **;**

Transformation of conditions Conditions used in AO4AADL are projected in AspectJ keeping the same structure and respecting the transformation rules presented in table 16 :

Table16. Transformation rules of the operators

AO4AADL	**AspectJ**
or	||
and	&&
not	!
=	==
>	>
≥	≥
<	<
≤	≤
!=	!=

Transformation of the arithmetic operators Arithmetic operations used in AO4AADL are mapped into AspectJ without modification. Arithmetic operation has always the following structure :

```
<Left_Expression> <operator> <Right_Expression>
```

where `<operator>` belongs to the following set of operators : {*, /, +, -}

Transformation of the communication operations For the communication actions presented in lines 28–30 of Listing 1.19, we followed the transformation rules used by the RTSJ generator.

In AO4AADL, communication operations can be summarized in three main operations namely (1) the execution of a subprogram (2) reading data from an input port and (3) writing data on an output port. The structures of these various operations are respectively :

1. `<Required_Subprogram_Identifier> ! (<Parameter_Profile>)`
2. `<Input_Port_Identifier> ? (<Data_Identifier>)`
3. `<Output_Port_Identifier> ! (<Data_Identifier>)`

According to this generator, for a given node, the generated ***Subprograms*** class includes only the subprograms called by a thread forming this node. Unfortunately, a subprogram can be used inside the advice action without being called by any thread. 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.

Transformation rules applied to the three types of the communication operations are presented in tables 17, 18 and 19 respectively :

Table17. Transformation rule of the number of messages in the queue for an entry port

AO4AADL specification	<Required_Subprogram_Identifier> **!** (<Parameter_Profile>)
Generated AspectJ code	SubPrograms.<Required_Subprogram_Identifier>Impl (PARAMETER_PROFILE)

According to the RTSJ generator, the operation of receiving a message by an input port 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.

Table 19 presents the transformation rule of the operation of sending a message through an output port.

As shown in Table 19, 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.

Transformation of the **proceed** *operations* Concerning the ***proceed()*** action, it keeps the same structure presented in AO4AADL respecting the transformation rules applied to the identifiers.

Transformations of conditions and loops

Table18. Transformation rule of the operation of receiving a message by an input port

AO4AADL specification	<Input_Port_Identifier> **?** (<Data_Identifier>)
Generated AspectJ code	**try {** Activity.getValue (Deploymet.<THREAD_ID>, Activity.<INPUT_PORT_IDENTIFIER>, <DATA_IDENTIFIER>) ; **}** **catch (ProgramException** e) **{** Debug.debugMessage (”error while performing get value in port” + Deployment.<INPUT_PORT_IDENTIFIER>.getPortNumber () + ”from thread” + Deployment.<THREAD_ID>) ; **throw e** ; **}** **try {** Activity.nextValue (Deploymet.<THREAD_ID>, Activity.<INPUT_PORT_IDENTIFIER>) ; **}** **catch (ProgramException** e) **{** Debug.debugMessage (”Error while performing next value in port” + Deployment.<INPUT_PORT_IDENTIFIER>.getPortNumber ()+ ”from thread” + Deployment.<THREAD_ID>) ; **throw e** ; **}**

Table19. 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 (Deploymet.<THREAD_ID>, Activity.<OUTPUT_PORT_IDENTIFIER>, <DATA_IDENTIFIER>) ; **}** **catch (ProgramException** e) **{** Debug.debugMessage (”error while putting value in port” + Deployment.<OUTPUT_PORT_IDENTIFIER>.getPortNumber () + ”from entity” + Deployment.<THREAD_ID>) ; **throw e** ; **}** **try {** Activity.sendOutput (Deploymet.<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>) ; **throw e** ; **}**

Transformation of **if** *conditions* In AspcetJ, the ***if*** condition keeps the same structure used in AO4AADL while respecting the transformation rules presented above.

Transformation of the **while** *loop* The **while** loop keeps the same structure used in AO4AADL while respecting the transformation rules presented above.

Transformation of the **for** *loop* The transformation of the **for** loop from AO4AADL to AspectJ is performed according to the rule presented in table 20 :

Table20. Transformation rule of the ***for*** loop

AO4AADL specification	**for** (<Loop_Variable_Identifier> **in** <Integer_Range>)
Generated AspectJ code	**for** (<LOOP_VARIABLE_IDENTIFIER>**.value** = <Min_Value>**;** <LOOP_VARIABLE_IDENTIFIER>**.value** < <Max_Value>**;** <LOOP_VARIABLE_IDENTIFIER>**.value++**)

where `<Integer_Range>` is the interval in which varies the loop counter (`<loop_variable_identifier>`). It has always the following form :

```
Integer_Range ::= Min_Value.. Max_Value
```

For the **for** loop in AO4AADL, we specify the identifier of the counter `<Loop_Variable_Identifier>` and the interval in which varies the counter `<Integer_Range>`. When transforming the **for** loop to the AspectJ language, the part between brackets will be divided into three parts :

1. **initialization** : The loop counter is initialized to the minimum value of the specified interval.
2. **condition** : Boolean expression that controls the passages in the loop that are as long as it is true. This expression is generally of the form : counter ¡ maximum value of the specified interval.
3. **transition** : instruction executed before each new pass through the loop. It increments the counter by 1.

Listing 1.17 presents a complete example of AO4AADL aspect. Listing 1.18 presents the generated AspectJ code.

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

```
13      else{
14      proceed();
15      counter := counter;
16   } }}
```

Listing 1.17. Complete example of AO4AADL aspect

```
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 1.18. Generated AspectJ code from listing 1.17

In Listing 1.18, 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.

Notes : In order to have a valid code, the following imports must be added to each generated aspect :

```
import fr.enst.ocarina.polyORB_HI_runtime.OutPort;
import fr.enst.ocarina.polyORB_HI_runtime.InPort;
import fr.enst.ocarina.polyORB_HI_runtime.ProgramException;
import fr.enst.ocarina.polyORB_HI_runtime.Debug;
import fr.enst.ocarina.polyORB_HI_runtime.Message;
```

The transformation rules presented here are applicable to aspects that are reported as an annex in a thread or a subprogram. In the case where the aspect affects several architectural components, it is declared in an external package visible to all the system. The transformation rules presented here remain valid ; we have just to generate this aspect for each node of the application while applying the following rules :

- If the aspect uses a subprogram, it will be generated in all ***Subprograms*** classes of all nodes of the application. If the ***Subprograms*** class already contains the corresponding method to this subprogram then the generation will be ignored for this class.
- For the code of the *advice* and in the case of the interception of ports, we have simply to combine the condition of checking the identifier of the port generated in the case of an aspect applied to a single thread with the conditions of verification of the identifiers of the treads intercepted in a node with a logical operator (**&&**). These conditions are separate by a logical operator (**&&**). Identifiers are deduced easily from the list of the specified components in section **applies to** of the aspect and using the ***Deployment*** class. The conditions of verifying identifiers of the intercepted threads are generated using the following rules. :
  - If we intercept the event of receiving a message on an input port, the generated verification condition has the following form :

    ```
    (Message)(thisJoinPoint.getArgs([1])).getmydestination()==
                                                Deployment.<THREAD_ID>
    ```

  - For all other cases, the generated verification condition has the following form :

    ```
    (int)(thisJoinPoint.getArgs([0]))== Deployment.<THREAD_ID>
    ```

Example : We suppose an application composed of two processes :

- A process called `Node_A` composed of three threads `TA1`, `TA2` and `TA3`.
- A process called `Node_B` composed of two threads `TB1` et `TB2`.

The snippet code 1.19 presents a complete example of AO4AADL aspect declared in an external package.

```
1 aspect Logging {
2   applies to thread TA1, thread TA2, process Node_B;
3   pointcut count() : execution outport (* (..));
4   advice after (): count(){
5     variables{counter:Integer_Type;}
6     initially{counter:=1;}
7     counter:=counter+1;}}
```

Listing 1.19. Example of AO4AADL aspect affecting several components

Since the aspect intercepts the threads `TA1` and `TA2` of the process `Node_A` and all threads of the process `Node_B`, then there will be generation of two aspects for each process.

Listing 1.20 shows the AspectJ code generated for the process `Node_A` and listing 1.21 corresponds to the AspectJ code generated for the process `Node_B`.

```
1 import fr.enst.ocarina.polyORB_HI_runtime.OutPort;
2 import fr.enst.ocarina.polyORB_HI_runtime.InPort;
3 import fr.enst.ocarina.polyORB_HI_runtime.ProgramException;
4 import fr.enst.ocarina.polyORB_HI_runtime.Debug;
5 import fr.enst.ocarina.polyORB_HI_runtime.Message;
6
7 aspect Logging {
8
9   pointcut count () : execution (* Activity.sendOutput (..));
10 public static final GeneratedTypes.IntegerType COUNTER =
11     new GeneratedTypes.IntegerType(1);
12   after ():count (){
13     if(((int)(thisJoinPoint.getArgs([0]))== Deployment.NODE_A_TA1_K)
14                 || (((int)(thisJoinPoint.getArgs([0]))==
15                                       Deployment.NODE_A_TA2_K)){
16 COUNTER.value=COUNTER.value+1;}}}
```

Listing 1.20. Generated AspectJ code for Node_A of the application

```
1 import fr.enst.ocarina.polyORB_HI_runtime.OutPort;
2 import fr.enst.ocarina.polyORB_HI_runtime.InPort;
3 import fr.enst.ocarina.polyORB_HI_runtime.ProgramException;
4 import fr.enst.ocarina.polyORB_HI_runtime.Debug;
5 import fr.enst.ocarina.polyORB_HI_runtime.Message;
6
7 aspect Logging {
8
9   pointcut count () : execution (* Activity.sendOutput (..));
10 public static final GeneratedTypes.IntegerType COUNTER =
11     new GeneratedTypes.IntegerType(1);
12   after ():count (){
13     if(((int)(thisJoinPoint.getArgs([0]))== Deployment.NODE_B_TB1_K)
14          || (((int)(thisJoinPoint.getArgs([0]))== Deployment.NODE_B_TB2_K)){
15 COUNTER.value=COUNTER.value+1;}}}
```

Listing 1.21. Generated AspectJ code for Node_B of the application

10 Conclusion

we presented here one of our main contributions.It consists in the definition of a set of transformation rules that enables the designer to translate the AO4AADL specification into AspectJ code.

These transformation rules are based on the RTSJ generator ones.

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.