Cent. Eur. J. Comp. Sci. • 3(2) • 2013 • 43-68
DOI: 10.2478/s13537-013-0105-1

VERSITA

Central European Journal of **Computer Science**

AO4AADL: Aspect oriented extension for AADL

Research Article

Sihem Loukil*, Slim Kallel†, Bechir Zalila‡, Mohamed Jmaiel§

ReDCAD Laboratory
National Engineering School of Sfax
B.P. 1173, 3038 Sfax
University of Sfax, Tunisia

Received 13 December 2011; accepted 22 May 2013

Abstract: Managing embedded system complexity and scalability is one of the most important problems in software development. To better address this problem, it is very recommended to have an abstraction level high enough to model such systems. Architectural description languages (ADLs) intend to model complex systems and manage their structure at a high abstraction level. Traditional ADLs do not normally provide appropriate formalisms to separate any kind of crosscutting concerns. This frequently results in poor descriptions of the software architectures and a tedious adaptation to constantly changing user requirements and specifications. AOSD (Aspect Oriented Software Development) deals with these problems by considering crosscutting concerns in software development. The effectiveness of considering an aspect-oriented architectural design appears when aspect concepts are taken into account early in the software's life-cycle.
In this paper, we propose a new aspect language called AO4AADL that adequately manipulates aspect oriented concepts at architecture level in order to master complexity and ensure scalability. The abstract nature of our proposed language allows the generation of aspect code for several programming languages and platforms.

Keywords: ADL • AADL • software architecture • aspect oriented • programming

1. Introduction

Implementing and managing software embedded systems is a tedious task, due to complexity and strict requirements of such systems. A possible solution to deal with these problems is to model such systems at architecture level. Software architecture helps the designer to define the structure of the application in terms of architectural elements that compose it and the definition of assembly. In fact, software embedded systems are defined as a set of components describing the functional level. These components are connected through a set of architectural connectors. Thus the software architecture allows to clearly separate the technical details from functional requirements.

* *E-mail: sihem.loukil@redcad.org*
† *E-mail: slim.kallel@fsegs.rnu.tn*
‡ *E-mail: bechir.zalila@enis.rnu.tn*
§ *E-mail: mohamed.jmaiel@enis.rnu.tn*

Architecture description languages (ADLs) [9, 30] are considered as an important tool for early analysis and feasibility testing. They can also support automatic generation of the functional code and allow easier management of the configuration and the deployment of systems.
Software architecture can be presented as a set of functional and non-functional concerns. A concern is a problem and a set of properties determining acceptable solutions. The functional concerns define what the system is supposed to do, while non-functional concerns state the quality of service and the conditions under which the system correctly operates. Traditional ADLs provide the same formalisms to describe both functional and non-functional concerns. They lack appropriate formalisms to provide modularity, scalability and represent crosscutting concerns behaviors (behavior that cuts across the typical divisions of responsibility).
This lack frequently results in poor descriptions of the architectures and a tedious adaptation to constantly changing user requirements and execution context. The specification of non-functional concerns is not well-modularized, as it is *tangled* with the specification of each component's core functionality or *scattered* across the specification of different components. The mixture of multiple concerns in an architectural model greatly increases the complexity of such a model. Furthermore, when the designer modifies one of the concerns, he should manipulate all parts of the model related to that concern which is challenging since these parts are mixed with the elements of other concerns.
AOSD deals with these problems by considering crosscutting concerns in software development. Several aspect based approaches have been proposed to model crosscutting concerns from requirements to implementation. These approaches describe architectural aspects as components, connectors, or new architectural abstractions and with either a symmetrical [3, 27] or an asymmetrical approach [11, 20].
On the one hand, the symmetrical approaches use the same module type to represent all concerns of a system (each component of the architectural model is considered as a concern). Thus, all components are considered as first class elements with the same structure, without any more important than another. The majority of these models does not use the term aspect (no need to distinguish between entities which are aspects and those which are not), but require other elements to represent this concept. For this reason, symmetrical models are often considered more flexible and more abstract. However, the proposed symmetric aspect-oriented ADLs treat the behavior of a crosscutting concern and the weaving mechanism in separate modules : the aspects are generally defined as components and the weaving mechanism of these aspects with the basic components is defined in a section of composition outside the component. This separation adds complexity to the maintenance and evolution phase. Then, it would be better to model the pointcuts and the behavior of the aspect in the same section inside the components to make easier the evolution of the architecture. In addition, it is not easy to visually distinguish aspects from non-aspectual components. All of this may hinder the comprehensibility of the architecture.
On the other hand, however, an asymmetrical approach introduces a special type of module to represent aspects. The asymmetry is manifested in the fact to explicitly distinguish aspects from non-aspectual components. Thus, components and aspects have different structures. This visual asymmetry in such approaches allows to easily distinguish between crosscutting concerns and non-crosscutting ones. In these approaches, the components define the initial structure of the system and the aspects are complementary. Hence, the evolution of systems is becoming increasingly easy. Asymmetrical approaches are more widely used than symmetrical approaches because they are easier to integrate into current approaches of software development and put them into practice [15, 16]. However, the obtained aspect oriented models in such approaches may not be analyzable and compatible with other tools manipulating the models described using traditional languages. Moreover, the distinction between base components and aspects reduces the reusability of the architectural elements in the sense that aspectual component cannot be reused as base component. Finally, none of the symmetrical and asymmetrical approaches is intended to model real-time embedded systems.
To address these issues, we propose an aspect-oriented extension of AADL [30], a well known ADL for specifying the architecture of real-time embedded systems. Our approach allows modeling the aspect-oriented architecture of such systems using a hybrid approach that brings together the advantages of both symmetrical and asymmetrical approaches. On the one hand, we keep the basic architectural elements without modification or extension to avoid introducing additional complexity to the architectural description and to keep compatibility of the resulting models with existing tools, which provides a symmetrical appearance to our approach. On the other hand, we defined a new formalism to model aspects, which gives asymmetrical aspect to our approach.
The extension of AADL consists in defining a new language christened AO4AADL to integrate aspect-oriented concepts in AADL. This language contains most of the AspectJ constructs. It is an interesting contribution since AADL is a standard unlike the other ADLs. Moreover, AADL allows modeling both hardware and software parts of the system and

the resulting architecture enables simulation and analysis of architectural characteristics using precise execution and communication semantics. Also, AADL is easily extendable thanks to two extension mechanisms : properties and annexes. Hence, it becomes possible to enrich the AADL description with a new formalism different from AADL. Additionally, the AADL language already has tool support and code generation capabilities that can be reused : different code generators are developed to allow the generation of functional code ready to be executed from the AADL description using the Ocarina tool suite [31]. This suite tool can be easily extended to support new concepts.
Our proposed language considers aspects as an extension concept of AADL using "annexes", an intrinsic mechanism to extend the AADL language. We consider that aspects can be specified in a language other than AADL, and then integrated inro AADL models as annexes. We define then a grammar for the new proposed language to rigorously describe the aspects in the AADL descriptions.
Aspects are used to describe the non-functional properties. We classified such properties into two classes: the first one includes the non-functional properties that represent the quality of software as liveness properties (like performance, scalability, etc.), while the second class presents the non-functional properties that are directly related to the behavior of the running system, like safety properties (like access control, security, consistency, etc.). In our work, we are interested only in the second class (non-functional safety properties). These properties ensure the proper functioning of software systems against undesirable behavior. We propose in our approach to specify as aspects the non-functional safety properties that crosscut among multiple modules of the system (crosscutting non-functional safety properties). The designer can also specify crosscutting functional concerns as aspects but we do not consider this in our work.
We also propose a code generation process from the aspect-oriented architectural description of the system. This process is composed of two main phases: the first phase is to generate functional code from the AADL description of the system. This type of code generation is available in the Ocarina tool suite. The second phase focuses on the generation of aspect code which is one of our contributions as part of this work. For this purpose, we define the necessary transformation rules for generating aspects written in an aspect-oriented programming language from aspects described in AO4AADL. To facilitate the work of the designer, we offer as part of this work a graphical editor called "AADL Graphical Editor" as an Eclipse plug-in. Our own editor allows modeling aspect-oriented architectures by integrating all AADL and AO4AADL concepts. This graphical editor provides a global view of the modeled system with some level of abstraction that facilitates its management.
We illustrate and evaluate our approach using two case studies (Automated Teller Machine (ATM) and Health Watcher (HW)) that exhibit some traditional crosscutting concerns at the architectural level.
The remainder of this paper is organized as follows. Section 2 presents background concepts related to AADL and AOSD. Section 3 introduces the case study of the ATM system used throughout the paper to explain our approach. Section 4 presents the syntax and the semantics of AO4AADL, our proposed aspect oriented language for AADL. Section 5 details the AO4AADL compiler by presenting the Ocarina tool suite and some aspect generation rules. Section 6 presents the development process from the architectural description until code generation. Section 7 presents an additional case study (HW) to validate our approach. Section 8 details the related work and Section 9 presents the evaluation of our approach and the associated tool. Section 10 presents the conclusion and the future work.

2. Background

In this section, we present the basic concepts of AADL through some examples and the main ideas of the AOSD paradigm.

2.1. AADL : Architecture Analysis & Design Language

AADL [30] (Architecture Analysis & Design Language) is an architecture description language for describing embedded systems. The building blocks of AADL, as the majority of architectural description languages, are components and connections. Components are used to define the structural system description, while connections are the focus of interaction.
A component in AADL corresponds either to a hardware entity of the system such as a bus, a memory, a processor, etc. or a software entity such as a thread, a process, a subprogram, etc. Each component is described by a type and several implementations or none. A type defines an externally visible interface (i.e., ports) that allows the interaction between components. Each type consists mainly of two parts : features, and properties. An implementation specifies the internal structure of a component as an assembly of subcomponents and connections. Connections are established between the

subcomponents features as well as between the features of the component and those of its own subcomponents.

```
----------
-- Data --
----------

data Simple_Type
end Simple_Type;

-----------------
-- Subprograms --
-----------------

subprogram Do_Ping_Spg
features
  Data_Source : out parameter Simple_Type;
end Do_Ping_Spg;

subprogram Ping_Spg
features
  Data_Sink : in parameter Simple_Type;
end Ping_Spg;

-------------
-- threads --
-------------

thread P
features
  Data_Source : out event data port Simple_Type;
end P;

thread implementation P.Impl
calls
Mycalls: {
  P_Spg : subprogram Do_Ping_Spg;
};
connections
  parameter P_Spg.Data_Source -> Data_Source;
end P.Impl;

thread Q
features
  Data_Sink : in event data port Simple_Type;
end Q;

thread implementation Q.Impl
calls
Mycalls: {
  Q_Spg : subprogram Ping_Spg;
};
connections
  parameter Data_Sink -> Q_Spg.Data_Sink;
end Q.Impl;

---------------
-- Processes --
---------------

process A
features
  Out_Port : out event data port Simple_Type;
end A;

process implementation A.Impl
subcomponents
  Pinger       : thread P.Impl;
connections
  port Pinger.Data_Source -> Out_Port;
end A.Impl;

process B
features
  In_Port : in event data port Simple_Type;
end B;

process implementation B.Impl
subcomponents
  Ping_Me      : thread Q.Impl;
connections
  port In_Port -> Ping_Me.Data_Sink;
end B.Impl;

------------
-- System --
------------

system PING
end PING;

```

```
89 system implementation PING.Impl
90 subcomponents
91   Node_A : process A.Impl;
92   Node_B : process B.Impl;
93
94 connections
95   port Node_A.Out_Port -> Node_B.In_Port;
96 end PING.Impl;
```

Listing 1. Connections between AADL components

AADL consists of both a textual and graphical language. Listing 1 presents the AADL description of the software part of the PING system inspired from the Ocarina tool suite [31]. This AADL description illustrates how to model a simple interaction between threads. Figure 1 corresponds to the graphical representation of this system according to the AADL language.

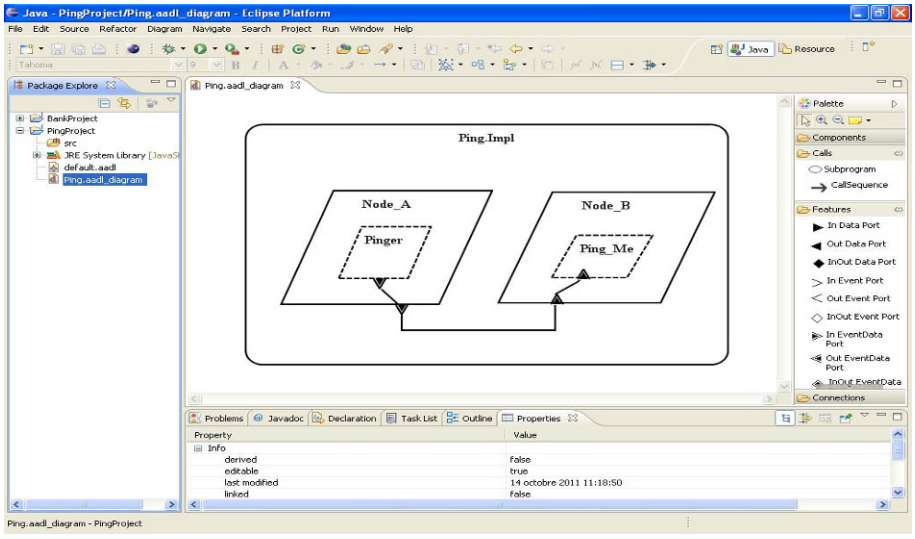

Figure 1. The graphical representation of the PING system

The type of the PING system is presented in lines 86 and 87. Its implementation is given in lines 89–96. The system is composed of two interconnected subcomponents `Node_A` and `Node_B` (lines 90–95). `Node_A` is an instance of the process `A.Impl` presented in lines 58–68. `Node_B` is an instance of the process `B.Impl` presented in lines 70–80. Each node is composed of a thread : a `Pinger` thread for the `Node_A` (lines 64–65) an a `Ping_Me` thread for the `Node_B` (lines 76–77). The `Pinger` thread pings the `Ping_Me` thread, sending a data. The type of the used data is called `Simple_Type` and defined in lines 5–6.
Each thread calls a subprogram that describes its behavior : the `P_Spg` subprogram called by the `Pinger` thread (line 34) presents an instance of the `Do_Ping_Spg` subprogram (lines 12–15), and the `Q_Spg` subprogram called by the `Ping_Me` thread (line 48) is an instance of the `Ping_Spg` subprogram (lines 17–20). Each thread is connected to its corresponding process : the output port of the `Pinger` thread is connected to the output port of the process `A` (line 67), while the input port of the process `B` is connected to the input port of the `Ping_Me` thread (line 79).
To support crosscutting properties such as the nature of the thread (periodicity, priority, etc.), AADL defines two main extension mechanisms : properties and annexes.

- Properties are characteristics which can be associated to different entities (components, connections, features, etc.). They are used to specify characteristics of components or constraints : processor frequency, worst case execution time of a thread, etc. Their specifications are gathered into named sets which are used in analysis tools in order to verify characteristics of the modeled system. A standard set of properties is already defined in AADL but it's possible to define specific properties for a given application.

- Annexes can contain annotations of components in a language different from AADL. Two types of annexes are used in AADL : (1) Annex subclauses that allow annotations expressed in a sublanguage to be attached to component types, component implementations, and feature group types, and (2) Annex libraries that contain reusable declarations expressed in a sublanguage and which are declared in packages. These reusable declarations can be referenced by annex subclauses.

2.2. Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) [8] is a development approach based on aspect oriented programming concepts. The main idea of AOSD is to provide new architectural abstractions and composition mechanisms in order to identify and model concerns that crosscut multiple modules in a software system. AOSD influences various phases of the development process (requirement, modeling and development) within different application domains.
In traditional software development approaches, these concerns are treated inside the basic program code which leads to a tangled and scattered code. The use of new abstractions in AOSD allows encapsulating these concerns into separated modular units (aspects) which will be integrated later with other system modules at specific points of the code (joinpoints). This separation of concerns improves modularization, reusability (reuse the same aspect in different system units) and allows having a clean code easy to understand.
Using AOSD, an application is composed of two parts : the base program which implements the core functionality, and aspects, which implement the crosscutting concerns. At architectural level, the core functionalities are presented with basic components while aspects are presented with aspectual components. Aspects are new units of modularity, which encapsulate crosscutting concerns in complex systems using joinpoints, pointcuts, and advices.
Joinpoints are well-defined points in the execution of a program. In AspectJ [18], which is an aspect-oriented extension of Java, joinpoints correspond to method calls, constructor calls, etc. The pointcut allows selecting a set of joinpoints, where some crosscutting functionalities should be executed.
The advice is a piece of code that implements a crosscutting concern, which is associated with a pointcut. This code is executed whenever a joinpoint in the set identified by the pointcut is reached. There are three types of advices, before, after, or in the place of the joinpoint at hand; this corresponds respectively to the advices types ***before***, ***after*** and ***around*** in AspectJ. While before and around advices are relatively not ambiguous, there can be three interpretations of after advice: After the execution of a join point completes normally (***after returning***), after it throws an exception (***after throwing***), or after it does either one (***after***).
Listing 2 presents an example of aspect in the AspectJ language. The aspect logs calls to objects of the class `Pinger`.

```
1  public aspect Logging
2  {
3      //where?
4      pointcut logPinger(Pinger p) : call(* Account.* (..)) && this(p);
5      //when?
6      after (Pinger a) : logPinger(p)
7      {
8          //what?
9          System.out.println("An activity was executed ... " );
10     }
11 }
```

Listing 2. Example of aspect in AspectJ

In this aspect, the pointcut `logPinger` (line 4, Listing 2) specifies where exactly the logging concern will be executed. This pointcut captures the call to any method of the class `Pinger` without specifying the return types and the input parameters. The advice (lines 6–11) answers the questions about when and what behavior of the concern will be executed. The advice will run after executing any jointpoint that is matched by the defined pointcut (line 6). In addition, this advice prints a logging message (line 9) to show the value of the variable `amount` defined in the class `Pinger`.
In case the logging concern is required in other classes or modules, the developer can just modify the pointcut. If this concern was not implemented as an aspect, the developer would have to localize all places where logging is required and add the corresponding code there.
Similar to the aspect notion at the programming level, we denote the so-called *architectural aspects* to model concerns that crosscut the basic components at the architectural level.

3. Case Study: Automated Teller Machine (ATM)

In this section, we introduce the Automated Teller Machine (ATM) system that will be used throughout the paper to illustrate and evaluate our approach. An ATM is an automated telecommunications device that allows customers to access their bank accounts to make withdrawals in cash (or cash advances using a credit card), check their balances and purchase prepaid credit for mobile phones. On most modern ATMs, the customer is identified by inserting a card with a chip that contains a unique card number (*NumCard*) and some security information as an expiration date. Authentication is performed by entering manually a personal identification number (*code*).
The architecture of the ATM is composed of three interconnected processes : the *Customer* process, the *Account* process and the *AccountData* process. The *Customer* process is intended to manage the authentication step. It is composed of three threads : *PingerTh*, *GUITh* and *ValidationTh*. *PingerTh* detects the inserted card, reads and sends the card number to the *ValidationTh* thread to check the validity of the inserted card.
If the card has expired, it will be rejected and an explanatory message will be displayed to the customer. In the other case (the card stills valid), the *GUITh* thread prompts the customer to enter his code. This thread presents the GUI of the ATM allowing interaction between the customer and the banking system. It enables the customer providing the requested information (code, how much to withdraw, etc.).
After that, the *ValidationTh* thread checks whether the code is wrong. We present in Listing 3 a part of the AADL description of the *ValidationTh* thread. The full code of this description can be retrieved in[1]. Checking is performed by calling the `Check_Code_Spg.Impl` subprogram (line 12, Listing 3). If the code is wrong, the subprogram sends the information to the *ValidationTh* thread throw the `RestoreCode_V_out` port (line 18), to ask the customer to reenter the code. The customer has only three authentication attempts. At the failure of the third attempt, the card is rejected by the system and an explanatory message is displayed to the customer.
When the customer succeeds the authentication, the *Customer* process will connect to the *AccountData* process. This process consists in a single thread *AccountDataTh* that manages the entire database of customers and their bank accounts. Then, the customer is prompted to choose the operation he wants to perform and the amount to withdraw (in the case of a withdrawal operation). The operation is performed by the *AccountTh* thread of the *Account* process. Finally, a message is displayed to inform the customer if the transaction is successful or not.

```
1  thread ValidationTh
2  features
3    NumCard_in_V : in event data port Integer_Type;
4    Code_in_V : in event data port Integer_Type;
5    RestoreCode_out_V : out event data port String_Type; // Port that to send a message to the
6                                                         // customer when the code is wrong
7    ...
8  end ValidationTh;
9
10 thread implementation ValidationTh.Impl
11 calls {
12   CC_Spg : subprogram Check_Code_Spg.Impl; // subprogram that checks if the code is wrong.
13   ...
14 };
15 connections
16   parameter NumCard_in_V -> CC_Spg.NumCard;
17   parameter Code_in_V -> CC_Spg.Code;
18   parameter CC_Spg.RestoreCode_V -> RestoreCode_out_V;
19   ...
20 end ValidationTh.Impl;
```

Listing 3. AADL description of the *ValidationTh* thread

In the automated teller machine, we identify three non-functional properties :

1. **Code Verification** : This property checks the number of authentication attempts made by the customer. If it reaches the third attempt, the card will be rejected by the system and an explanatory message will be displayed to the customer.

   The authentication action is performed before the withdrawals in cash and before checking the balance. Hence, the *Code Verification* property should be checked at these points.

[1] http://www.redcad.org/projects/AO4AADL/code/Bank.aadl

2. **Shrinkage stresses** : The withdrawal of money is determined by the two following constraints:
   - A customer can not withdraw more than 1000 Euro / day
   - A customer can not withdraw more than 4000 Euro / week

   This property is checked only when performing the withdrawal operation.
3. **Traceability** : This property is defined to ensure a better quality of service : an SMS is delivered to the customer containing the details of the performed operation on the account and the current balance in his bank account. This SMS is sent after each operation performed on the account of the customer (debit money, withdraw money,...).

4. The AO4AADL language

Considering aspect concepts at the beginning of software life cycle is considerably valuable since it improves comprehensibility, evolution and reuse in the development of complex software systems. In this paper, we propose a new aspect oriented ADL, called AO4AADL [14], an extension of AADL with aspect concepts.
The choice of AADL was driven by many reasons. First, AADL is a standard and the resulting architecture enables simulation and analysis of architectural characteristics using precise execution and communication semantics. Second, AADL is a concrete ADL in which all elements of a description correspond to concrete entities allowing the description of both hardware and software parts of the system. AADL enables rapid system evolution for complex, real time, safety critical systems. Third, compared with other ADLs, AADL introduces two extension mechanisms : properties and annexes. These two mechanisms make the language much easier to extend. Moreover, they offer a good foundation for additional capabilities in analysis, automated system integration, distribution, and dynamics. Based on the annex extension mechanism, we propose to enrich AADL specifications with aspect concepts. In this context, we defined a rigorous grammar of our proposed language AO4AADL.

4.1. AO4AADL: hybrid approach

The proposed approach, contrary to the existing ones, is neither a symmetrical nor an asymmetrical approach. It is rather a hybrid approach (half-symmetrical half-asymmetrical). Specifically, our approach is based primarily on defining a new language called AO4AADL for describing the crosscutting non-functional safety properties at architectural level. This new language gives an asymmetrical aspect to our approach because it corresponds to the definition of a new formalism to model aspects specifying the non-functional safety properties.
These aspects will be included as annexes in the basic architecture described using AADL. The use of this extension mechanism provides a symmetrical appearance to our approach since annexes are considered basic abstractions of AADL. Thus, to integrate aspects at the architectural level, there is no need to define new abstractions or to add new interfaces to basic abstractions in order to ensure the coordination between basic and aspectual components. Moreover, the obtained aspect oriented models still remain compatible with other tools manipulating AADL models.

4.2. Syntax of AO4AADL

The syntax of our proposed grammar is inspired from the aspect-oriented programming language AspectJ [18]. We present in this section the different syntactic rules used in our language AO4AADL to describe aspect concepts at the architectural level. A part of the proposed grammar[2] is presented in Listing 4.

4.2.1. Aspect specification

Three types of aspects can be modeled using our AO4AADL language (line 1, Listing 4) :

[2] *The full version of the grammar is available at* http://www.redcad.org/projects/AO4AADL

- *Behavioral aspect* (lines 2–5, Listing 4) : an aspect that describes the behavior of a crosscutting non-functional safety property.
- *Precedence aspect* (lines 6–8, Listing 4) : an aspect that declares the precedence of the already defined behavioral aspects.
- *Affected_Components* aspect (lines 9–11, Listing 4) : an aspect that defines the list of components which are supposed to be affected by a defined behavioral aspect.

The aspect is a modular unit designed to describe the behavior of a concern, which will be executed at a specific moment during system execution. The architectural description of an aspect using AO4AADL is composed of two main parts : definition part (lines 2, 6 and 9, Listing 4) specified with the keyword ***aspect*** followed by the aspect identifier and (ii) the behavioral description part of the aspect. While the first part of an aspect is common for all the three types of aspects, the structure of the second part is not. In fact, the *Behavioral aspect* is composed of two subclauses :

Pointcut specification : This subclause is supposed to define the conditions under which the aspect is invoked.

Advice specification : This subclause encapsulates the aspect behavior.

For the *Precedence aspect*, it is composed of one subclause (**Precedence_Specification**). This subclause defines the order of the execution of aspects when two or more aspects intercept the same point at the same moment. The format of this subclause is given in line 13. It is specified with the keyword ***precedence***. The priority of execution is in the order of the declared list in this subclause. For example : `precedence Aspect1, Aspect2;` means that `Aspect1` have more priority than `Aspect2`. If only one aspect is defined then this subclause will be ignored by the analyzer.
The *Affected_Components aspect* is composed of one subclause (**Components_applies_to**): The format of this subclause is given in lines 14–17. It is defined to specify the name of software components that are supposed to be affected by the aspect. Actually, an aspect is supposed to influence either only one component, or many components (two or more). To each defined behavioral aspect corresponds a line in the *Affected_Components aspect* to specify the affected components.
All the aspects are declared as an annex library in an AADL package outside the components to be visible throughout the system.

```
1  Aspect_Expression := Behavioral_Aspect | Precedence_Aspect | Affected_Components_Aspect
2  Behavioral_Aspect := aspect {Aspect_Identifier} {
3                       {Pointcut_Specification;}+
4                       {Advice_Specification}+
5                   }
6  Precedence_Aspect := aspect {Aspect_Identifier} {
7                       {Precedence_Specification;}+
8                   }
9  Affected_Components_Aspect := aspect {Aspect_Identifier} {
10                      {Components_applies_to;}+
11                  }
12
13 Precedence_Specification ::= precedence Aspect_Identifier { , Aspect_Identifier }*
14 Components_applies_to ::= Aspect_Identifier applies to ListComponents
15 ListComponents ::= Component { ,Component }*
16 Component ::= Component_Category Component_Identifier
17 Component_Category ::= thread | process | subprogram
18 ...
```

Listing 4. Part of the aspect grammar in AO4AADL

In the following, we will detail how to design AO4AADL pointcuts and advices.

4.2.2. Pointcut specification

As shown in Listing 5 (lines 1–2), pointcut definitions consist of a left-hand side (line 1) containing the specification of the pointcut name and parameters (the data available when the events happen) and a right-hand side (line 2) consisting in the pointcut expression itself.
In real architectural configuration, aspect behavior may be executed by several architectural joinpoints. Hence, an architectural pointcut should be defined as an expression that specifies the set of joinpoints to which the behavior of an aspect is applicable. A joinpoint (called `Pointcut_Primitive` in Listing 5) specifies a well-defined point of the

aspect behavior execution. In order to express the architectural quantification mechanism, we introduce the operators "and" ("&&"), and "or" ("||") (lines 4–6, Listing 5) to describe sets of joinpoints invoking the same advice.
We propose also three primitives (line 8, Listing 5) for defining a joinpoint : (i) a ***call*** primitive (line 9) to capture the fact of calling a port to send or receive a message or calling a subprogram in the AADL specification; (ii) an ***execution*** primitive (line 10) to intercept the exchanged messages through the ports or intercept the execution of a subprogram, or (iii) ***args*** primitive (line 23) to intercept the type or the value of the intercepted parameters.

```
1  Pointcut_Specification ::= pointcut Pointcut_Identifier ( [ ParamList ] ) :
2                             Pointcut_Expression
3  ...
4  Pointcut_Expression ::= Pointcut_Primitive | ( Pointcut_Expression )
5                        | Pointcut_Expression && Pointcut_Expression
6                        | Pointcut_Expression || Pointcut_Expression
7
8  Pointcut_Primitive ::= Call | Execution | Args
9  Call ::= call Callee
10 Execution ::= execution Callee
11 Callee ::= subprogram ( Subprogram_Identifier ( [ Subprogram_Parameter_Types ] ) )
12             | inport ( Input_Port_Identifier ( [ Data_Type ] ) )
13             | outport ( Output_Port_Identifier ( [ Data_Type ] ) )
14             | inoutport ( InOutput_Port_Identifier ( [ Data_Type ] ) )
15
16 Subprogram_Parameter_Types ::= Type {,Type }*
17 Data_Type ::= Type
18 Type ::= .. | Type_Identifier
19 Subprogram_Identifier ::= Identifier | *
20 Input_Port_Identifier ::= Identifier | *
21 Output_Port_Identifier ::= Identifier | *
22 InOutput_Port_Identifier ::= Identifier | *
23 Args ::= args (Arguments)
24 Arguments ::= Argument { ,Argument }*
25 Argument ::= .. | Argument_Identifier { ,Argument Identifier }*
```

Listing 5. Part of the pointcut grammar described in AO4AADL

AO4AADL explicitly defines the places where the effect of aspect annex can occur (lines 11–14, Listing 5). They include : (i) the subprograms already declared in the AADL specifications (line 11), (ii) the incoming data into an AADL component (line 12), (iii) the outgoing data flow emerging from an AADL component (line 13), and (iiii) both the incoming and the outgoing data through an inoutput port (line 14).
When defining a joinpoint, the designer should specify the identifier of the intercepted subprogram or the identifier of the intercepted port. He can also use the wildcard "*" (lines 19–22), which means that all the subprograms or all the ports are intercepted whatever are the identifiers.
Moreover, the designer can specify the type identifier of the data available when intercepting the joinpoint. However, he can use the wildcard ".." (lines 18 and 25) to specify that he is not interested in the type of this data. In other words, whatever the type of the data available at this moment, the joinpoint will be intercepted.
Listing 6 presents a part of the **CheckCode** aspect described in AO4AADL. This aspect belongs to the first non-functional property of the automated teller machine (**code verification**) presented in Section 3. This aspect checks the number of the authentication attempts made by the customer. In fact, he has only three attempts. For this purpose, the `Verification` pointcut presented in this aspect (line 6, Listing 6) seeks to intercept the fact of calling the output port `RestoreCode_out_V` of the *ValidationTh* thread, which is responsible of sending a message to the customer to reenter his code again. To avoid displaying this message at the failure of the third authentication, this output port should not be called to send the message. Consequently, the pointcut intercepts the `RestoreCode_out_V` with a ***call*** primitive. We notice here the ".." passed as a parameter to the intercepted output port. This means that we are not interested in the data available when the events happen. In other words, whatever the type of the data sent through this output port, it will be intercepted. Therefore, no parameters are specified in the pointcut parameters.
The verification code that will be executed when invoking this aspect will be presented in the advice subclause.

```
1 system implementation Bank.others
2   ...
3 end Bank.others;
4 annex ao4aadl {**
5    aspect CheckCode {
6       pointcut Verification(): call outport (Restore_Code_out_V (..));
7       ...
8    }
9 **}
```

Listing 6. Example of AO4AADL annex

This aspect is declared as an annex library inside the AADL specification (lines 4–9, Listing 6). Since this aspect is supposed to intercept an output port of the `ValidationTh` thread, a line is added in the *Affected_Components aspect* describing the list of components that are supposed to be affected by the **CheckCode** aspect (line 2, Listing 7).

```
1 aspect Affected_Components{
2    CheckCode applies to thread ValidationTh.Impl;
3 }
4 end Bank.others;
```

Listing 7. Example of the Affected_Components aspect

4.2.3. Advice specification

The advice is a piece of code associated with the pointcut. It is executed whenever a joinpoint in the set identified by the pointcut is reached. For each pointcut, we can associate one or more crosscutting behavior which is expressed in an advice subclause allowing one or more advice subclauses to be associated to the same pointcut. The syntax of the advice specification in AO4AADL is displayed in Listing 8. The structure of the advice subclause is given in line 1 of Listing 8. AO4AADL defines three types of advice (lines 2–5, Listing 8) listed by the keywords : ***before*** used when the advice action runs before the joinpoint, ***after*** used when the advice code runs after each joinpoint and ***around*** used to integrate the further execution of the intercepted joinpoint in the middle of some other code using the keyword ***proceed*** (line 35) to progress in the execution of the functional code. To better reference the corresponding pointcut, we specify all its parameters in the advice declaration (lines 3–5). Then, the pointcut will be called as if we call a method in the object oriented programming (line 6).
The advice code has the structure given in lines 8–12 of Listing 8. It is specified in three subclauses :

- *Variables declaration subclause* (lines 13–14) is defined to declare a set of local variables used in the action part. It is specified using the keyword ***variables***.
- *Initialization subclause* (line 15) is defined to initialize the local variables already declared in the previous subclause. It is specified using the key word ***initially***.
- *Action subclause* (lines 16–19) is used to define the advice behavior. It includes all the instructions to execute. The syntax of these instructions is inspired from the AADL Behavior Annex [29] since this annex satisfies the majority of our requirements. Yet, we assigned minor modifications to the syntax of the Behavior Annex to express other requirements such as adding the `Proceed_Action` as well as the addition of the `Character` and the `String_Literal` variables to the `Behavior_Expression` which are not used in the AADL Behavior Annex. Moreover, we removed some things that are not useful in our case. For example, we do not use in our language `state_variable_identifier` or a `data_access_identifier` in the definition of the `Reference_Expression`. However, we added the `Parameter_Identifier` and the `Argument_Identifier`.

  Different instructions can be used to specify the advice behavior such as the conditional statement ***if***, the ***for*** and ***while*** loops (lines 17–19) as well as four basic actions (lines 21–24) : (i) the assignment action (line 26) which allows manipulating the arithmetic operations, (ii) the communication action (lines 28–30) that uses two types of operators : the operator ! which enables to send a message through an output port or to execute a subprogram while the operator ? is used to receive a message through an input port, (iii) the times action (lines 32–33) used to put some temporal conditions, and (vi) the ***proceed()*** action (lines 35) used to progress in the execution of the basic functional code.

```
1  advice Advice_Declaration: Pointcut_Reference { Advice_Action }
2  Advice_Declaration ::= Before_Advice | After_Advice | Around_Advice
3  Before_Advice::= before ( [ Paramlist ] )
4  After_Advice::= after ( [ Paramlist ] )
5  Around_Advice::= around ( [ Paramlist ] )
6  Pointcut_Reference ::= Pointcut_Identifier ( [ Parameters ] )
7  ...
8  Advice_Action ::= {
9     [ Variables_Declaration ]
10    [ Initialisation ]
11    { Action }+
12    }
13 Variables_Declaration::= variables { { Variable }+ }
```

```
14 Variable ::= Variable_Identifier { , {Variable_Identifier} }* : Type_Identifier;
15 Initialisation ::= initially { { Assignment }+ }
16 Action ::= Basic_Action
17         | If_Statement
18         | For_Statement
19         | While_Statement
20 ...
21 Basic\_Action ::= Assignment
22               | Communication
23               | Timed_Actions
24               | Proceed_Action
25 ...
26 Assignment ::= Reference_Expression := Behavior_Expression
27 ...
28 Communication ::= Required_Subprogram_Identifier ! [(Parameter_Profile)]
29                  | Output_Port_Identifier ! (Data_Identifier)
30                  | Input_Port_Identifier ? (Data_Identifier)
31 ...
32 Timed_Actions ::= computation ( Behavior_Time [ , Behavior_Time] );
33                | delay ( Behavior_Time [ , Behavior_Time] );
34 ...
35 Proceed_Action ::= proceed ([Parameter_Profile]);
```

Listing 8. Part of the Advice specification grammar described in AO4AADL

Listing 9 presents the advice code (lines 3–14) associated to the pointcut presented in Listing 6.
We define an **around** advice to check some conditions before executing the aspect code. In fact, when invoking the corresponding pointcut, the aspect checks if the customer has remaining attempts. If he has already used three attempts, the functional code will be blocked at the point specified in the pointcut and another code will be executed. In our case, the card will be rejected through the output port `RejectedCard_out_V` and an explanatory message will be displayed to the customer (lines 8–10, Listing 9). In the other case (lines 11–14, Listing 9), we call the ***proceed()*** instruction (line 12) to progress in the execution of the functional code. This means that a message is displayed to the customer to restore his code again through the output port `RestoreCode_out_V` and then the counter of the number of attempts is incremented (line 13). All the variables used in the advice action are declared in the variables subclause (line 5) and initialized in the initialization subclause (line 6).

```
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

5. AO4AADL compiler

We begin this section by presenting 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.

5.1. Ocarina tool suite

Ocarina [31] presents a free tool suite written in Ada to manipulate AADL models. This tool suite includes three code generators that are able to generate some programming languages such as Ada [32], C [7] and RTSJ (Real Time Specification for Java) [1] from an AADL specification.
The architecture of Ocarina is composed of three main libraries that are easily extensible :

1. A central library (**libocarina**) which presents a low abstraction level to build and manipulate syntactic trees.

2. A set of frontends that allows analyzing the syntax and semantics of files written in AADL language using routines of the central library.

3. A set of backends whose role is to automatically produce code. It is based on trees resulting from the frontends.

The code generators offered by the Ocarina tool suite allow the generation of functional code from the basic components of the AADL description. As an example, The RTSJ code generator allows generating RTSJ code from AADL specifications. It allows translating 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].

5.2. Aspect code generation

We present in this section our extension of 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 the 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.
In our work, we chose to start generating aspects written in AspectJ [18] language from the architectural aspects described in AO4AADL. In fact, the study of the various existing aspect-oriented programming languages has proven that AspectJ is the most popular aspect language due to its widespread use with an emphasis on simplicity and ease of work for end users. To apply these AspectJ aspects, we need a base system described in the Java language in order to get a complete Java prototype. For this reason, we adopt in our approach the RTSJ code generator 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 one example of the transformation rules from AO4AADL to AspectJ. This example details how to generate AspectJ joinpoint intercepting a subprogram from AO4AADL specifications (Table 1). The full version of these transformation rules is available in [13].
As shown in Table 1, the interception of a subprogram at the 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. On the one side, according to the syntax of AspectJ, we have to specify the returning type of the intercepted method. On the other side, all generated methods using the RTSJ generator 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 11 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 10.

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

Listing 10. Example of an AO4AADL pointcut intercepting a subprogram

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

Listing 11. Generated AspectJ code from Listing 10

We have to note that the aspect and the pointcut names as well as the keywords '***aspect***', '***pointcut***' and '***call***/***execution***' are kept as they are without any modification.

5.3. Implementation

We present in this section the main extensions made to the Ocarina tool suite to implement our new language as well as the aspect code generator from AO4AADL language to AspectJ language. We present also a short description of our graphical editor "AADL Graphical Editor".

5.3.1. Extension of Ocarina

The development of the AO4AADL compiler consists mainly in three steps :

1. The implementation of the file describing the AO4AADL tree : This step consists in the translation of the AO4AADL grammar rules into a file written in a pseudo language similar to the IDL syntax. This file describes the tree of AO4AADL which will be later used by the frontends and the backends parts. Listing 12 presents a part of the code of this file. It corresponds to the translation of the grammar rules that define the structure of the aspect (lines 5–27), the structure of a pointcut (lines 31–39) as well as the structure of an advice (lines 43–52).

```
1  // AO4AADL Tree
2
3  module Ocarina::ME_AO4AADL::AO4AADL_Tree::Nodes {
4
5    /*
6       Aspect_Annex ::= { Aspect_Expression }+
7    */
8
9    interface Aspect_Annex : Node_Id {
10     List_Id Aspect_Expressions;
11   };
12
13   /*
14     Aspect_Expression := Behavioral_Aspect | Precedence_Aspect | Affected_Components_Aspect
15     Behavioral_Aspect := aspect {Aspect_Identifier} {
16                          {Pointcut_Specification;}+
17                          {Advice_Specification}+
18                       }
19     Precedence_Aspect := aspect {Aspect_Identifier} {
20                          {Precedence_Specification;}+
21                       }
22     Affected_Componenets_Aspect := aspect {Aspect_Identifier} {
23                          {Components_applies_to;}+
24                       }
25    */
26
27   interface Aspect_Expression : Definition {
28     List_Id Components_Applies_Tos;
29     List_Id Precedence_Specification;
30     List_Id Pointcut_Specification;
31     List_Id Advice_Specification;
32   };
33   ...
34   /*
35     Pointcut_Specification ::= pointcut Pointcut_Identifier ( [ ParamList ] ) :
36                                Pointcut_Expression
37    */
38
39   interface Pointcut_Specification : Definition {
40     List_Id  Parameters;
41     Node_Id  Pointcut_Expression;
42   };
43   ...
44   /*
```

```
45    Advice_Specification ::= advice Advice_Declaration : Pointcut_Reference
46                             Advice_Action
47   */
48
49   interface Advice_Specification : Node_Id {
50     Node_Id Advice_Declaration;
51     Node_Id Pointcut_Reference;
52     Node_Id Advice_Action;
53   };
54   ...
55 };
```

Listing 12. A part of the implemented file describing the AO4AADL tree

2. The implementation of the frontends : In order to make our language understandable by the Ocarina compiler, we extended the existing frontends part by adding a new module. This module allows checking the vocabulary, the syntax and the semantics of our language based on the file describing the tree of AO4AADL already defined in the previous step. For this purpose, we implemented first the required files that enable to make a lexical analysis of the input file. It consists in parsing the AO4AADL annex included in the AADL file and try to recognize all the lexemes (identifiers, operators, delimiters, etc.). Second, we developed the necessary files for the syntactic and the semantics analysis based on the output of the lexical analysis to build a hierarchical structure called abstract syntactic tree. This tree presents the relations between all the found lexemes. The secondary outputs of this step consist of any warnings or error messages. Finally, a semantic analysis is applied to the obtained tree in order to lead to an analyzed syntactic tree.

3. The implementation of the backends : This step consists in implementing all the defined transformation rules (presented in section 5.2) in order to automatically generate AspectJ code from AO4AADL descriptions. This needs to develop another module in the backends set of the Ocarina tool suite. This module takes as input the syntactic tree generated from the previous module (frontends) and try to build the corresponding syntactic tree for the target language, which is AspectJ in our case. Finally, this tree is used to generate the AspectJ code.

The implementation of the frontends and the backends modules has taken a long time to adapt to the Ocarina tool suite and be able to extend it especially when we have to develop a long program (2833 lines of code for the frontends part and 5532 for the backends one). Finally, we have to mention that the frontends part is generic for all the generators since it depends only on the file describing the AO4AADL tree. While the backends part, which implements the transformation rules to the programming language, is specific to each generator. So to transform an AO4AADL specification into an aspect-oriented programming language different from AspectJ, we have simply to define and implement the corresponding transformation rules.

5.3.2. AADL graphical editor

By exploring the different existing editors enabling AADL modeling, we found that these editors are few, immature and do not meet our needs.
Really, we tested the *ADELE*[3] and *OSATE-TOPCASED*[4] editors that are open source projects but they cause many problems: blocking of the editor, unexpected errors, a sudden closure of the editor, etc.. For the *STOOD* editor, which is a stable product developed in C++, it supports several modeling languages such as AADL and UML2 but it is a paid product. Only a demo version of 30 days is available.
For this, we thought to develop our own editor that meets our needs. We implement this editor called *AADL Graphical Editor*, as an Eclipse plug-in. The graphical interface of our editor is presented in Figure 2.

[3] http://www.topcased.org/index.php?idd_projet_pere=73
[4] http://www.topcased.org/index.php$?$idd_projet_pere=61

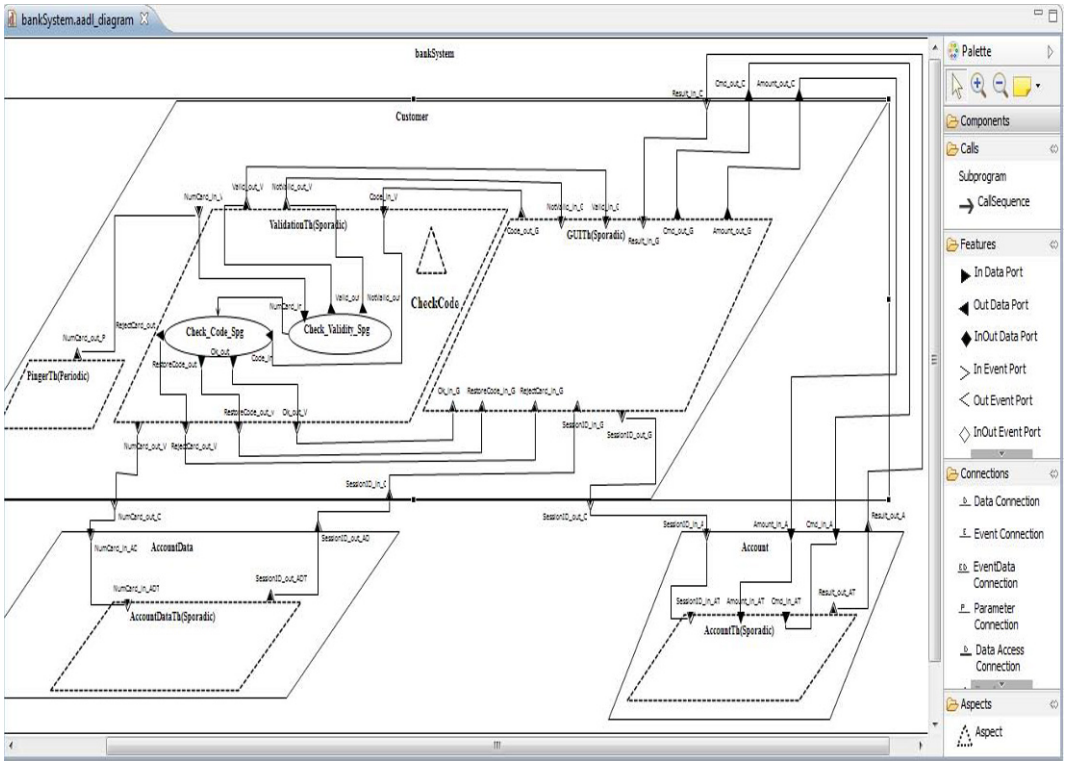

Figure 2. A graphical description of the ATM system

This editor is developed using three main frameworks *EMF* (*Eclipse Modelling Framework*), *GEF* (*Graphical Editing Framework*) and *GMF* (*Graphical Modeling Framework*).
EMF is essential for the implementation of a graphical editor. We use this framework in order to create a meta-model describing the data to capture in the AADL application. Once the meta-model is set up, we use the *GEF* framework to manage the graphical side. In fact, we use *GEF* to create the palette tool of our graphical editor as well as the tools of creation and selection. Based on these two frameworks, we can finally generate our graphical editor using the *GMF* framework that provides basic functionality for creating menus, an area to view the figures, a palette tool,etc..
The AADL concepts are graphically presented according to the standard of the AADL language. We propose to use a discontinuous triangle to graphically model the AO4AADL aspects. For this purpose, we use a Palette tool which is divided into five groups **Components**, **Calls**, **Features**, **Connections** to model AADL components as well as their connections and **Aspects** to model AO4AADL aspects.
After the graphical modeling of the system, we can generate an AADL textual description of this system as well as the AO4AADL annexes describing the aspects.
Most elements of the editor have a set of properties available through a *Properties Pane* in our graphical editor. We do not present all the information of these elements in the graphical editor. We present only some information; for example an aspect is characterized by its name, the type of its advice, the name of its pointcut and the type of the primitive used in its pointcut expression. Other information should be accomplished by the user himself in the generated textual description.

6. The development process

In this section, we outline our approach for designing aspect oriented architectures. We propose a complete process, starting with a rigorous specification until the enforcement upon the generation and the execution of aspect code. As shown in Figure 3, our approach consists mainly in three phases :

1. **Implementing Functional code** : In this phase, the designer should focus on the main functionalities of his application without considering any crosscutting properties. First, he specifies, at the architectural level, his applications in terms of AADL elements (components, connectors, threads, ports, etc.). Then, the designer generates the corresponding functional code using a code generator. In fact, many compilers translate an AADL specification into several languages such as Ada [32], C [7] or RTSJ (Real Time Specification for Java) [1].

2. **Designing crosscutting non-functional safety properties** : At this phase, the designer defines the non-functional safety properties and states the conditions under which his application operates correctly, such as security, availability, etc. To express these crosscutting concerns at the architectural level, the designer should use our aspect language called AO4AADL. We proposed a complete grammar of our AADL language as well as its different syntactic and semantic rules (cf. Section 6). These aspects will be integrated as annexes in the AADL specification to form an aspect oriented architectural description of the modeled system.

3. **Generating crosscutting non-functional safety properties code** : Since our proposed language is generic, AO4AADL aspects can be translated into different aspect languages such as AspectJ [18] or JAC [21] for Java language, AspectAda [22] for Ada language, AspectC [6] for C language, etc. This generation requires an aspect generator and well defined transformation rules from A04AADL to the considered aspect language. These transformation rules should take into consideration the transformation rules already existing for generating the functional code (phase 1). The conformity of the generators ensures the consistency between the functional code and the aspects. These aspects will be automatically integrated into the generated functional code, which will lead to a complete prototype.

In our approach, we focus on the generation of RTSJ code from AADL specifications using Ocarina tool suite [31]. Therefore, we implemented a prototype of AspectJ generator based on a set of transformation rules.

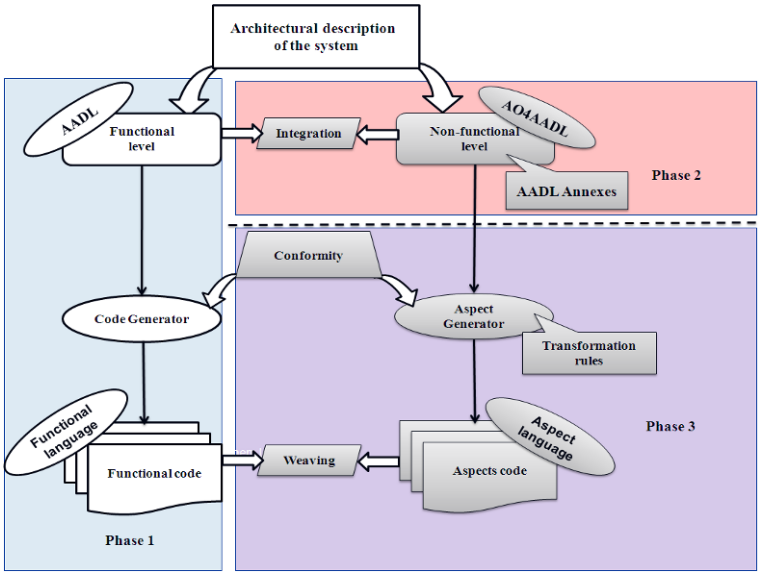

Figure 3. The proposed development process

7. Validation with a second case study: Health Watcher (HW)

In this section, we introduce the Health Watcher system (HW), another case study to illustrate and evaluate our approach. The HW system is a Web-based information system. The purpose of this system is to collect then manage public health related complaints. The system is also used to notify people of important information regarding the Health System.

There are two kinds of users: citizens and employees. A citizen is a person who wishes to interact with the system to query information and to specify (register) a complaint. An employee has special permissions and can access the various operations of the system such as managing the complaints, update the health unit's data, update the status of the employee, etc. Figure 4 illustrates a part of the AADL graphical representation of the HW architecture. To ensure better visibility of the architecture, we did not represented all the connections between different processes. The HW is composed of two main architectural processes: (i) the *GUI* (Graphical User Interface) process provides a Web interface of the system, for the two kinds of users (citizens and employees) and (ii) the *Business* process defines all business operations.
The *GUI* process is composed of a single thread *User_GUI* while the *Business* process is composed of a thread *BusinessT* and a data component *DataStore* that stores the information about complaints, employees, health units, symptoms and so on.

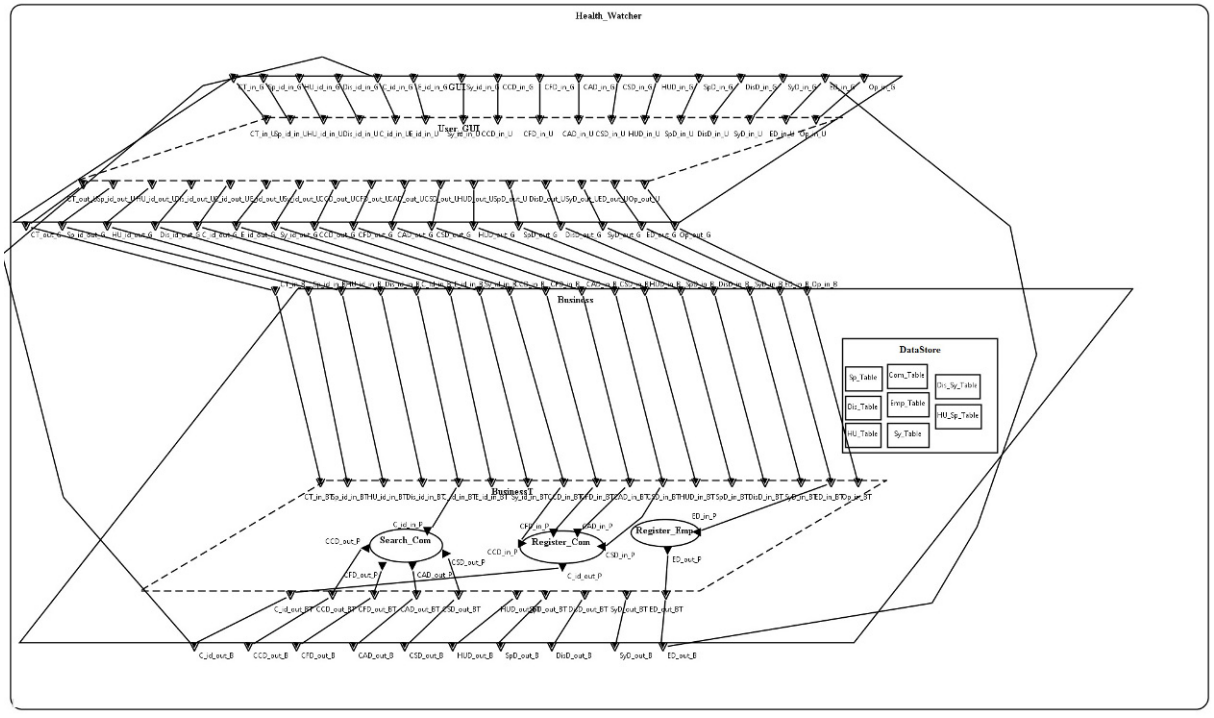

Figure 4. A graphical representation of the HW architecture

Apart from the main functionality of the system, three crosscutting concerns have been identified: (i) *Security*: The security concern is divided into two concerns :

- *Encryption and Decryption* : The system should use a secure protocol when sending/receiving data over the internet,
- *Authentication*: The employee has access to restricted operations on the system. He should then provide his login and password to perform the corresponding operation,

(ii) *Replication*: The system should store a copy of the data that the *Business* process sends to the *DataStore* component, and (iii) *Consistency*: During the interaction among the *GUI* and the *Business* processes, every time the *Business* process has to check the consistency of the received data.
Using only AADL, the *Encryption* concern, for example, is represented by the call to the subprogram `Encrypt_Data` in all the used threads (*User_GUI* and *BusinessT*) in order to encrypt the outgoing data from these threads (lines 9 and 23, Listing 13). Hence, this concern crosscuts all the used threads. However, using our aspect-oriented language

AO4AADL, this concern can be represented as an aspect at the architectural level. In our case, it is represented inside the aspect `Security` (lines 7–16, Listing 14). In fact, this aspect is represented by two pointcuts (one for the encryption concern and one for the decryption one) and their corresponding advices.
For the encryption concern, its pointcut (line 8, Listing 14) intercepts then all the data sent through all the out ports of all the threads described in the system. For the advice (lines 10–12), it will run before executing any joinpoint that is matched by the defined pointcut (line 10). This advice calls the `Encrypt_Data` subprogram to encrypt the data to send. This subprogram takes as parameters the value of the intercepted outport (`this.value`) and returns as an out parameter the encrypted data (`Enc_Data`).
Similarly, the decryption concern is represented by a pointcut (line 9) that intercepts all the data received through all the in ports of all the threads. Its corresponding advice (lines 13–15, Listing 14) calls the `Decrypt_Data` subprogram to decrypt the received data.
For the authentication concern, it is represented in a separate aspect called *Authentication* (lines 18–29, Listing 14) since it does not affect the same components that are affected by the encryption and decryption concerns. This aspect is supposed to intercept all the incoming data through the in port `Op_in_U` of the *User_GUI* thread (line 19–21) and checks if the requested operation is different from the query information and the register complaint operations that are related to citizens (line 25). If so, the `Authentication_Spg` subprogram is called to invite the employee to enter his parameters (line 26).

```
1  thread GUIT
2  ...
3  end GUIT;
4
5  thread implementation GUIT.Impl
6  calls {
7   EED_Spg : subprogram Enter_Emp_Data.Imp;
8   ...
9   ED_Spg : subprogram Encrypt_Data.Impl;
10 };
11 end GUIT.Impl;
12
13 thread BusinessT
14 ...
15 end BusinessT;
16
17 thread implementation BusinessT.Impl
18 calls {
19  RU_Spg : subprogram Register_User.Impl;
20  RC_Spg : subprogram Register_Com.Impl;
21  UC_Spg : subprogram Update_Com.Impl;
22  ...
23  ED_Spg : subprogram Encrypt_Data.Impl;
24 };
25 ...
26 end BusinessT.Impl;
27 ...
```

Listing 13. Part of the AADL description of the HW system

```
1  ...
2  system implementation Health_Watcher.others
3  ...
4  end Health_Watcher.others;
5  annex ao4aadl {**
6
7     aspect Security{
8        pointcut Encryption (Enc_Data:String_Type): execution outport (* (..));
9        pointcut Decryption (Desenc_Data:String_Type): execution inport (* (..));
10       advice before(Enc_Data:String_Type ) : Encryption(Enc_Data){
11          Encrypt_Data! (this.value, Enc_Data);
12       }
13       advice before(Desenc_Data:String_Type) : Decryption(Desenc_Data){
14        Decrypt_Data! (this.value, Desenc_Data);
15        }
16    }
17
18    aspect Authentication{
19       pointcut Authenticate (op:String_Type, login:String_Type,
20                 password:String_Type, result:String_Type):
21                     execution inport (Op_in_U (..)) && args (op);
22       advice after (op:String_Type, login:String_Type,
23                 password:String_Type, result:String_Type) :
24                        Authenticate (op, login, password, result){
25          if(op != "Register_Com") and (op != "Query"){
26              Authentication! (login, password, result);
```

```
27            }
28        }
29     }
30
31     aspect Replication{
32        pointcut Replicate(): execution subprogram (Register* (..)) || execution subprogram (Update* (..));
33        advice after() : Replicate(){
34        Replicate_Data! (this.value);
35        }
36    }
37
38    aspect Consistency{
39        pointcut Consistent(consistent:Boolean_Type): execution inport (* (..));
40        advice after(consistent:Boolean_Type) : Consistent(consistent){
41            if(this != Op_in_BT) and (this != CT_in_BT){
42                Check_Consistency! (this, this.value, consistent);
43              if (consistent=false){
44                    send_Message! ("Provided data is not consistent");
45                    }
46            }
47        }
48    }
49
50    aspect Affected_Components{
51        Security applies to thread User_GUI.Impl, thread BusinessT.Impl;
52        Authentication applies to thread User_GUI.Impl;
53        Replication applies to thread BusinessT.Impl;
54        Consistency applies to thread BusinessT.Impl;
55    }
56 **}
```

Listing 14. AO4AADL description of the *Security* aspect

In addition, a line is added in the *Affected_Components aspect* (lines 50–55, Listing 14) describing the list of components that are supposed to be affected by the *Security* aspect. Line 51 specifies that the `Security` aspect is supposed to affect the *User_GUI* thread and the *BusinessT* thread, while line 52 indicates that the `Authentication` aspect affects only the *User_GUI* thread.

The *Replication* concern is presented by the `Replication` aspect described in lines 31–36 of Listing 14. This aspect intercepts all the incoming data to the *Data* component. Since the incoming data is provided from all the registration and the update subprograms called by the *BusinessT* thread, the pointcut (line 32) intercepts all the subprograms whose names begin with Register and Update (`subprogram (Register* (..))` and `subprogram (Update* (..))`). The advice (lines 33–35) will run after executing any joinpoint that is matched by the defined pointcut (line 33). This advice calls the `Replicate_Data` subprogram that takes as parameters the incoming data. The *Replication* aspect is supposed to affect only the *BusinessT* thread (line 53, Listing 14).

For the *Consistency* concern, it is presented by the `Consistency` aspect described in lines 38–48 of Listing 14. This aspect intercepts all the incoming data through the in ports of the *BusinessT* thread (lines 39) except the data that comes from the `Op_in_BT` and the `CT_in_BT` ports that reference the type of the operation to perform and the type of the complaint, respectively. The advice (lines 40–47) will run after the execution of any joinpoint that is matched by the defined pointcut (line 40). This advice checks first if the intercepted port is different from the `Op_in_BT` and the `CT_in_BT` ports (line 41). If so, it calls the `Check_Consistency` subprogram that checks the consistency of the incoming data (line 42). This subprogram takes as parameters the intercepted input port (`this`) and the value of the incoming data (`this.value`) through this port and returns as an out parameter a boolean (`consistent`), which indicates if the data is consistent or not. If the data is not consistent a message is displayed to the user to inform him that the provided data is not consistent (lines 43–45).

All the aspects are defined as an annex library (lines 5–56, Listing 14).

In the following, we represent the AspectJ code generated from the AO4AADL aspects defined in Listing 14. Due to lack of space, we represent the generated AspectJ code of only two aspects. Listing 15 corresponds to the generated code from the `Consistency` aspect and Listing 16 corresponds to the `Authentication` aspect.

```
1  aspect Consistency{
2    pointcut Consistent(GeneratedTypes.Boolean_Type consistent):
3                execution (* Activity.getValue(..));
4    after(Generated_Types.Boolean_Type consistent) : Consistent(consistent){
5      if((((InPort)thisJoinPoint.getArgs()[1]).getPortNumber() !=
6             Deployment.NODE_Business_BusinessT_OP_IN_BT_K) &&
7                 (((InPort)thisJoinPoint.getArgs()[1]).getPortNumber() !=
8                     Deployment.NODE_Business_BusinessT_CT_IN_BT_K)){
9        SubPrograms.Check_ConsistencyImpl
10           (((InPort)thisJoinPoint.getArgs()[1]).getPortNumber(),
```

```
11            ((GeneratedTypes)thisJoinPoint.getArgs()[2]).value, consistent.value);
12        if(consistent.value==false){
13            SubPrograms.Send_MessageImpl("Provided data is not consistent");
14            }
15        }
16    }
17  }
```

Listing 15. Generated AspectJ code from the *Consistency* aspect

The generated pointcut is presented in lines 2–3 of Listing 15, whereas the generated advice code is presented in lines 4–16. In the generated AspectJ code, all the generated types are included in the `GeneratedTypes` class. Lines 5–8 of Listing 15 checks if the intercepted ports are different from the `NODE_Business_BusinessT_OP_IN_BT_K` and the `NODE_Business_BusinessT_CT_IN_BT_K` ports, which corresponds respectively to `Op_in_BT` and the `CT_in_BT` ports defined at the architectural level. These ports are already declared in the *Deployment* class generated using the RTSJ generator. The call to the subprogram `Check_Consistency` is transformed to a simple call of the method `Check_ConsistencyImpl` of the generated *SubPrograms* class. The same rule is applied to the `Send_Message` subprogram.

```
1  aspect Authentication{
2     pointcut Authenticate (GeneratedTypes.String_Type op, GeneratedTypes.String_Type login,
3         GeneratedTypes.String_Type password, GeneratedTypes.String_Type result) :
4               execution (* Activity.getValue(..)) && args (op);
5     after (GeneratedTypes.String_Type op, GeneratedTypes.String_Type login,
6          GeneratedTypes.String_Type password, GeneratedTypes.String_Type result) :
7             Authenticate (op, login, password, result){
8         if(((InPort)thisJoinPoint.getArgs()[1]).getPortNumber()==
9                       Deployment.NODE_GUI_USER_GUI_OP_IN_U_K){
10            if((op.value != "Register_Com") && (op.value != "Query")){
11                SubPrograms.AuthenticationImpl (login.value, password.value, result.value);
12                }
13            }
14        }
15    }
```

Listing 16. Generated AspectJ code from the *Authentication* aspect

Lines 2–4 of Listing 16 present the generated pointcut. The generated advice code is given in lines 5–14. Lines 8–9 checks if the intercepted port corresponds to the `Op_in_U` input port of the *User_GUI* thread (`Deployment.NODE_GUI_USER_GUI_OP_IN_U_K`). Line 10 checks then if the value of the intercepted port is deffernt from the operations "`Register_Com`" and "`Query`" to call in line 11 the subprogram `AuthenticationImpl` in order to perform the operation of authentication.

8. Related work

Although [19] proposed a set of requirements that must be met by the ADL to allow the management of aspects and [2] offers the features that should be supported by these languages, there is no consensus on how to approach the description of the architectural aspects and integrate the aspect concepts at the architectural level. But most of existing Aspect-Oriented architectural approaches agree on that the semantics of the composition should be somehow extended in order to ensure the connection between the aspects and the basic components.
As stated by [10], extending a component based formalism to AOSD is performed either symmetrically or asymmetrically. Some existing implementations of AOSD in an ADL used the asymmetrical approach. In this sense, DAOP-ADL [26], a new language that does not extend any existing ADL, defines aspects as first-order entities which affect the component's interfaces by means of an evaluated interface and a target interface. The composition between components and aspects is supported by a set of aspect evaluation rules that defines when and how to apply aspects to components in order to extend the behavior of the system with aspectual properties (weaving between components and aspects). These rules are described in a separate section, apart from the components, using XML language. Contrary to our language, DAOP-ADL is completely a new ADL which needs much more effort to implement a platform to support it. Similar to our approach, a shortcoming of DAOP-ADL is the distinction between components and aspects which decreases the chance of reusability of architectural elements in the sense that aspects cannot be reused as basic components and basic components cannot

be reused as aspects. Besides, this language does not support any quantification mechanism and no code generator has been implemented so far.

Similarly, FAC [24], which is an extension of Fractal [25], proposes a new kind of component named Aspectual Component (AC) to specify crosscutting concerns in software architecture. Each AC contains a special interface to intercept regular components and define the composition between aspects and components using XML language. Furthermore, the software architecture described in FAC can be translated into Java. Similar to our approach, and like in DAOP-ADL, a shortcoming of FAC is the distinction between components and aspects, which decreases the chance of reusability of architectural elements.

In AspectLEDA [20], aspects are LEDA [5] components. The support of aspect concepts is provided by including a special kind of components called "coordinator". This component defines the weaving process that synchronizes and coordinates the aspects and components. AspectLEDA architectural description is translated into LEDA to be able to translate it into Java using a code generator and thus a simulating prototype implementation of the expanded system can be obtained. However, the integration of a new abstraction increases the complexity of the architectural description since the architect is forced to learn a new element different from the basic abstractions (components and connectors). In addition, the integration of a new architectural abstraction requires to define a new platform to support descriptions of architectures using AspectLEDA because existing tools do not allow more.

In AC2-ADL [11], a new Aspect-Oriented ADL, aspects are modeled using several architectural elements : First, it defines an Aspectual Component (AC) with a new kind of interfaces to encapsulate the behavior of a crosscutting concern. Second, it uses an Aspectual Connector (AC) (a special type of connector) to capture the crosscutting interaction of certain architectural elements and finally, this language defines architectural joinpoints to ensure the composition between base components and aspectual components. As in the case of all languages developed from scratch, AC2-ADL requires more effort for its implementation. Moreover, the distinction between the basic components and the aspectual ones decreases the reusability of components. Similarly, the use of an Aspectual Connector with a new interface reduces the reusability of connectors. For code generation, AC2-ADL has no type of code generator.

Some other implementations use the symmetrical approaches. They use components to model both functional components and aspects.

In AspectualACME [3], an extension of ACME [9], aspects are defined as ACME components. The only extension required to integrate aspect concepts is the introduction of the concept of Aspectual Connector. Aspectual Connector is an ACME connector with a new interface that is defined on the one hand, to distinguish between the basic components and aspects and, on the other hand, to capture how are interconnected the different types of components. However, aspectual connectors cannot be reused as basic connectors and basic connectors cannot be reused as aspectual ones since they did not have the same structure and the same representation. AspectualACME inherits the property for the construction of executable configurations but without bearing the code generation.

Similarly, AO-ADL [27] considers that components model either crosscutting or non-crosscutting behavior. The crosscutting nature of a component only depends on the connections with other components, which are specified in connectors. In this sense, AO-ADL extends the semantics of traditional connectors to represent the crosscutting effect of aspectual components. As in DAOP-ADL, the composition between components and aspects is defined by a set of composition rules using XML. Contrary to our AO4AADL language, AO-ADL is completely a new ADL which needs much more effort to implement a platform to support it. Besides, no code generator has been implemented so far.

In PRISMA [23], the structure or behavior of architectural elements (components and connectors) is modeled using a new abstraction called aspect. Then aspects are used to model either crosscutting or non-crosscutting concerns. The composition specification is also specified inside both components and connectors. Contrary to our AO4AADL language, PRISMA is completely a new language that does not extend an existing ADL. This needs much more effort to implement a platform to support it. Moreover, it is difficult to visually distinguish the aspects from the basic components since they are represented using the same architectural abstraction.

In our case, we integrated the aspect code in the model (in the same document) while keeping our model compatible with tools that do not support AO4AADL. For this purpose, we used the AADL annex extension mechanism. This allows us having a whole new formalism to describe the aspects (benefit of the asymmetrical approach) while keeping a single model which can be reusable among different tools (benefit of the symmetrical approach).

Thus, we followed a hybrid approach that brings together the advantages of both symmetrical and asymmetrical approaches.

9. Evaluation

In this section, we evaluate our approach and the developed Eclispe plug-in based on the most used criteria and metrics for evaluating methods and tools. These criteria are inspired from [12, 17, 27, 28]. This section provides the designer with a fine-grained understanding the quality of the AO4AADL and the associated plug-in.

- The *generality* criterion states that the proposed method can cover most important aspects of the application domain. For evaluating this criterion, we used our aspect language in two case studies (Automated Teller Machine and Health Watcher) for specifying and implementing different crosscutting non-functional safety properties such as: security, replication, consistency and authentication. We are currently evaluating our approach on a third case study ("auction system") which is inspired from http://caosd.lcc.uma.es/aoadl/index.htm. We are trying in this case to model more and different non-functional properties such as security (authentication, access control), concurrency, validation, enrollment, monitoring, transaction and response time.

- The *usability* criterion expresses that the application should be easy to use by the class of user for whom it was intended. Our proposed approach simplifies the task of the architect by separately defining crosscutting non-functional safety properties and automatically generating the corresponding AspectJ aspects. In addition, we provide to the architect a GUI as an Eclipse plug-in that allows easily defining AO4AADL aspects and specifying their characteristics. Moreover, some researchers in our team (no co-authors), working on AADL, have already used our editor to model simple systems.

- The *applicability* criterion expresses that the method meets the needs of the designer and covers the development phases and it is applicable in any context. In addition to the definition of the syntax and the semantics of AO4AADL, we propose an approach that covers the whole development process for implementing crosscutting non-functional safety properties. First, the designer can easily and graphically define AO4AADL aspect at the architectural level. Second, the designer can refine his specification and add more details about the aspect (pointcut, precedence, etc.). Third, the designer can use our generator to automatically generate AspectJ code, which can be integrated into the functional application code (which is automatically generated from AADL specifications). Moreover, our approach was applied to two different case studies and we are currently evaluating it on a third one.

- The *scalability* criterion expresses the ability of the language to model both small and large-scale systems equally well. For evaluating this criterion, we consider the ability of our language to easily add new architectural artifacts to the architecture. In AO4AADL, the quantification mechanism offers the possibility to add new non-functional safety properties to many functional components by simply adding a single architectural aspect (for example, the Security aspect in Listing 14). Moreover, our language has strong scalability for the small systems (less than 15 components). For large systems, we can improve the scalability of our language by offering the user the opportunity of selecting a view of the designed system. Hence, the large system can be seen as a set of views (small systems).

- The *refinement* criterion expresses the ability of the approach to provide a correct and consistent refinement. Thanks to the mechanism of separation of concerns, our approach requires, as a first step, to specify the components related to functional concerns. Then, in a second step, the designer can easily add non-functional safety properties and associate them with the basic functional components. Depending on system requirements, the architect can then easily add or remove other non-functional safety properties. We consider this as an iterative process until all the non-functional safety properties are integrated into the architecture.

- The *traceability* criterion states the ability of the approach to trace the aspects from requirements to implementation. According to our development process, the aspects describing the non-functional safety properties are maintained separate from the functional concerns, from requirements to implementation. In fact, the aspects are specified as annex libraries at the design phase and then they are transformed into modular units (aspects) written in aspect-oriented programming language (AspectJ) separately from the generated functional code (RTSJ). It will then be easy to check if an aspect specified at the architectural level is also an aspect at the implementation level. Also, it is possible to check if new aspects appear in the architecture, or how an architectural aspect is expressed at the architectural level or at the implementation one.

- The *evolvability* criterion expresses the ability to easily accommodate new architectural artifacts or to modify existing ones, without undue effort by the developer. In AO4AADL, adding a new crosscutting non-functional safety property amounts to add an aspect inside an annex library without troubling the existing components or connections.

- The *tool support* criterion expresses that our approach is implemented and validated using an already existing tool support called the Ocarina tool suite. This tool which already supports the AADL specifications is simply extended in our work to be able to integrate the AO4AADL concepts. Although we have integrated new concepts, the obtained aspect-oriented model is kept compatible with tools that manipulate AADL models and do not support the AO4AADL concepts due to the use of the annex mechanism. The Ocarina tool suite is incorporated into the developed Eclipse plug-in to facilitate its use.

10. Conclusions and future work

In this paper, we presented two main contributions in our work. The first contribution consists in the definition of AO4AADL, an aspect-oriented architecture description language, which extends the AADL language using the annex extension mechanism to capture crosscutting non-functional safety properties at the architectural level. Thus, for a given application, the functional concerns are described in AADL components while the crosscutting non-functional safety properties are described in AO4AADL aspects. We defined a rigorous grammar that supports all basic aspect-oriented concepts. Our work is then enclosed in the AOSD discipline. The syntax of this grammar is inspired from the AspectJ one, which offers a simple syntax to define these concepts and from the Behavior Annex of AADL, which is very rich and enables to capture several behavior instructions.
The second important contribution is the definition of a code generation process allowing to obtain a prototype ready to be executed. This code generation process is composed of two main phases : In the first phase, we used the RTSJ generator available at the Ocarina tool suite to be able to generate RTSJ code from the basic components described in AADL. The second phase consists in generating AspectJ aspects from architectural aspects described in AO4AADL. This last type of code generation is based on a set of transformation rules that we defined in this work. The generated AspectJ aspects are later weaved with the Java classes generated by the RTSJ generator to obtain a complete Java prototype.
AO4AADL allows having a clear design and highly cohesive components because functional components and aspects can remain separated until the code generation. It ensures modularity, reusability and maintainability.
In addition, the automatic code generation makes our approach especially user-friendly and bridges the gap between the implementation of the application and its architectural description.
However, our approach has some limitations, which we will address in future work. First, the AspectJ generator works only for some prototypes since the used RTSJ generator that is very basic in our work presents several limits. Actually, this generator allows only manipulating data event ports and it supports only integer data. To address this problem, we plan to improve the RTSJ generator. Second, we adopted in our work the AspectJ code generation, despite our proposed language could be translated in several aspect-oriented programming languages such as AspectC and AspectAda, particularly that we already have a C and Ada code generators in the Ocarina tool suite.
As future work, we plan, first, to extend the backends part of Ocarina to support the AspectC and AspectAda code generation in order to make our language more generic. Second, we plan to consider the definition of abstract aspects to add more generalization to our approach. Finally, we plan to use our language for defining an approach for modeling at runtime the AADL specifications (Model@ Runtime [4]) which allows the management, at runtime, of the real-time applications for their adaptation or reparation.

References

[1] Autret T.,Code Generation of Real-Time Java for Real-time Systems, Master's thesis, Pierre & Marie Curie University, Paris VI, 2009

[2] Batista T. et al., Reflections on architectural connection: Seven issues on aspects and adls, In: Proceedings of the ICSE Workshop on Early Aspects, 3-10, 2006

[3] Batista T. et al., Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs, In: Proceedings of the 20th Brazilian Symposium on Software Engineering, ACM, 2006

[4] Blair G., Bencomo N., France R.B., Models@Run.Time, IEEE Computer, 42, 22-27, 2009

[5] Canal C., Pimenteland E., Troya J.M.,Compatibility and Inheritance in Software Architectures,Sci. Comp. Program., 41, 105-138, 2001

[6] Coady Y., Kiczales G., Feeley M., Smolyn G.,Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, In: Proceedings of the 8th European Software Engineering Conference, 88-98, 2001

[7] Delangea J., Hugues J., Pautetand L., Zalila B., Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain, In: Proceedings of the 4th European Congress ERTS, Embedded Real-Time Software, 2008

[8] Filman R.E., Elrad T., Clarke S., Akşit M., editors Aspect-Oriented Software Development, Addison-Wesley, 2005

[9] Garlan D., Monroe R.T., David Wile D., ACME: Architectural Description of Component-Based Systems, In: Foundations of Component-Based Systems, Cambridge University Press, 47-68, 2000

[10] Harrison W.H., Ossher H.L., Tarr P.L., Harrison W., Asymmetrically vs. symmetrically organized paradigms for software composition, Technical report, IBM Research Division, Thomas J. Watson Research Center, UY, USA, 2002

[11] Jing W., Shi Y., LinLin Z., YouCong N., AC2-ADL: Architectural Description of Aspect-Oriented Systems, Int. J. Software Eng. Its Appl., 3, 1-10, 2009

[12] Kitchenham B., Pickard L., Pfleeger S.L., Case studies for method and tool evaluation, IEEE Soft., 12, 52-62, 1995

[13] Loukil S., AO4AADL Compiler, Technical report, ReDCAD, University of Sfax, Tunisia, 2011, http://www.redcad.org/projects/AO4AADL/pdf/AO4AADLcompiler.pdf

[14] Loukil S., Kallel S., Zalila B., Jmaiel M., Toward an Aspect Oriented ADL for Embedded Systems, In: Proceedings of the 4th European Conference on Software Architecture (ECSA), Lect. Notes Comput. Sci., 6285, 2010

[15] Martínez A.N, Pérez M.A., Murillo J.M., *AspectLEDA:* extending an adl with aspectual concepts, In: Proceedings of the First European Conference on Software Architecture (ECSA), Lect. Notes Comput. Sci., 4758, 330-334, 2007

[16] Martínez A.N., Marco de trabajo para el desarrollo de arquitecturas software orientado a aspectos, PhD thesis, 2008

[17] Michelsen C.D., Dominick W.D., Urban J.E., A methodology for the objective evaluation of the user/system interfaces of the madam system using software engineering principles, In: Proceedings of the 18th annual Southeast regional conference, 103-109, 1980

[18] Miles R., AspectJ Cookbook. O'Reilly Media, Inc., 2004

[19] Navasa A., Pérez-Toledano M.A., Murillo J.M., Hernández J., Aspect oriented software architecture: a structural perspective, In: Proceedings of the AOSD Workshop on Early Aspects AspectOriented Requirements Engineering and Architecture Design, ACM, 2002

[20] Navasa A., Pérez-Toledano M.A., Murillo J.M., An ADL Dealing with Aspects at Software Architecture Stage, Information Software Technololy, 51, 306-324, 2009

[21] Pawlak R., Seinturier L., Duchien L., Florin G., JAC: A Flexible Solution for Aspect-Oriented Programming in Java, In: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, 2192, 1-24, 2001

[22] Pedersen K.H., Constantinides C., AspectAda: Aspect Priented Programming for ADA95. In: Proceedings of the annual ACM SIGAda International conference on Ada, 79-92, ACM, 2005

[23] Pérez J., Ramos I., Jaén J., and Patricio Letelier. Prisma: Towards quality, aspect oriented and dynamic software architectures, In: Int. Conf. On Quality Software, 59-66, 2003

[24] Pessemier N., Seinturier L., Duchien L., Components, ADL & AOP: Towards a Common Approach, In: Proceedings

of the ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution, 61-69, 2004

[25] Pessemier N., Seinturier L., Coupaye T., Duchien L., A Model for Developing Component-Based and Aspect-Oriented Systems, In: Proceedings of the 5th International Symposium of Software Composition, 259-274, 2006

[26] Pinto M., Fuentes L., Troya J.M., DAOP-ADL: an architecture description langage for dynamic component and aspect-based development, In: Proceedings of the 2nd international conference on Generative programming and component engineering, 118-137, 2003

[27] Pinto M., Fuentes L., Troya Linero J.M., Specifying aspect-oriented architectures in ao-adl, Inf. Soft. Tech., 53, 1165-1182, 2011

[28] Sacha K., Evaluation of software quality, In: Proceeding of the 2005 conference on Software Engineering: Evolution and Emerging Technologies, IOS Press, 381-388, 2005

[29] SAE. Architecture Analysis & Design Language: Annex Behavior, 2008

[30] SAE. Architecture Analysis & Design Language (AADL), 2003, http://www.sae.org/technical/standards/AS5506A

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

[32] Zalila B., 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, 2008