FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Meta-Language and Framework for Aspect-Oriented Programming

Tiago Diogo Ribeiro de Carvalho

Master in Informatics and Computer Engineering

Supervisor: Prof. Doutor João Manuel Paiva Cardoso

14th July, 2011

© Tiago Diogo Ribeiro de Carvalho, 2011

A Meta-Language and Framework for Aspect-Oriented Programming

Tiago Diogo Ribeiro de Carvalho

Master in Informatics and Computer Engineering

Approved in oral examination by the committee:

Chair: Prof. Doutor Hugo Ferreira

External Examiner: Doutor José Gabriel Coutinho

Supervisor: Prof. Doutor João Manuel Paiva Cardoso

14th July, 2011

Abstract

Nowadays programmers have access to high-level programming artifacts, such as the use of APIs, functions, and classes. Each paradigm has its own decomposition criteria, but there are always parts of the code that do not align with system's primary decomposition (concerns). Those concerns are usually spread and repeated along the code and pollute the system main functionalities. An example is the insertion of code for debugging purposes that many times results in the insertion of prints and other functionalities in the middle of the code that only serve to debug the program.

To resolve this problem of decomposition exist the so called aspect-oriented programming (AOP), a recent programming paradigm that tries to raise system's modularity. The use of AOP consists in separating the concerns that do not align with program's main decomposition, creating separated aspects with those concerns and indicating their relationships with the main program. The aspect language weaver associates the main program with the aspects, inserting them in intended program join points. There are several AOP languages, such as AspectJ, an aspect-oriented extension for Java, and AspectC++, to apply aspects in C++ programs. Although these languages are pretty useful for their associated programming languages, the aspects cannot be reused between aspect languages. Thus, an aspect created for a certain target language would have to be recoded to be applied to a different target language. Other disadvantages of these languages are the points of interest (join points), that, e.g., do not consider loops and local variables.

This thesis presents an aspect-oriented meta-language and corresponding framework. Both allow the development of aspects that can be applied to different programming languages. The MATLAB and C programming languages are used as case studies. Another objective of this meta-language is to aid AOP with the possibility to specify/modify types of variables. This allows programmers to test different implementations, to easily try different data types, so it can help to optimize the performance of embedded computing programs. Our approach includes the specification of the join points of each programming language to adapt the front-end to different programming languages.

Resumo

Actualmente os programadores têm acesso a artefactos de programação de alto nível, tal como o uso de APIs, funções e classes. Cada paradigma tem o seu critério de decomposição, mas existem sempre partes do código que não alinham com a decomposição primária do sistema (chamados *concerns*). Esses *concerns* estão normalmente espalhados e repetidos ao longo do código para depuração do programa, que muitas vezes resulta na inserção de impressões e outras funcionalidades de depuração no meio do código.

Para resolver este problema de decomposição existe a chamada programação orientada a aspectos (AOP), um paradigma de programação recente que tenta aumentar a modularidade do sistema. O uso de AOP consiste em separar os *concerns* que não alinham com a decomposição principal do programa, criando para tal aspectos separados com esses *concerns* e indicando a sua relação com o programa principal. O *weaver* da linguagem de aspectos associa o programa principal com os aspectos, inserindo-os nos pontos de junção pretendidos. Existem várias linguagens de AOP, tal como AspectJ, uma extensão orientada a aspectos para Java, e AspectC++, para programas C++. Embora estas linguagens sejam eficazes para a suas linguagens de programação associadas, os aspectos não podem ser utilizadas entre linguagens de aspectos. Portanto, um aspecto criado para uma certa linguagem tem de ser recodificada para ser aplicada numa linguagem de aspectos diferente. Outras desvantagens destas linguagens são os pontos de interesse (*join points*) que, por exemplo, não consideram ciclos e variáveis locais.

Esta tese apresenta uma metalinguagem orientada a aspectos e a *framework* correspondente. Ambos permitirão o desenvolvimento de aspectos que podem ser aplicados a diferentes linguagens de programação. As linguagens de programação MATLAB e C são usados como casos de estudo. Outro objectivo desta metalinguagem é apoiar AOP com a possibilidade de especificar /modificar tipos de variáveis. Isto permite aos programadores testar diferentes implementações, para tentar facilmente diferentes tipos de dados, para assim optimizar a performance de programas de computação embebidos. A nossa abordagem inclui a especificação dos *join points* para cada linguagem de programação para adaptar o *front-end* para diferentes linguagens de programação.

Acknowledgments

First of all, I would like to thank Professor João Manuel Paiva Cardoso for all the support and knowledge transmitted to me which contributed to my growth both in terms of research and personal level. Thanks for his orientation, guidance and motivation when the results were difficult to find or did not match the expectation. And many thanks for being a great friend for all his team members that considered him a great leader and an example to be followed.

Thanks to all the SPeCS team members for receiving me with a warm welcome in their unit and for helping me during my thesis when I needed help, forcing me to go out to dinner when I was too lazy to go out: to Ricardo Nobre for helping me develop the language and supporting me with motivation and for our races (a few that have been); to Ali Azarian for receiving me with so much friendliness and happiness and kept me from freaking out innumerous times and being the most joyful friend anyone could have; to Adriano Kaminski for keeping me laughing when everyone needed and put up with me for this past months, as he kept me happy and cheerful, and for helping on my writings; to João Bispo for always having solution/class for all the problems encountered that no one could find a simpler solution. Thanks to you all, you have become not only my great colleagues but also great friends I can trust.

Many thanks to José Gabriel for helping me on the development of the meta-language and the IR and for teaching me the best way of creating them by pointing more to legibility and flexibility. Thanks for being patient when the results were not as expected and redirecting me on the right way.

I would also like to thank Luís Pedrosa which, although not being on the same team, he was one of the best colleagues that gave me possible, and many times easier, paths to work around a problem and thus being so flexible (!) during work to help whenever he was needed, and of course for being a pal for put up with me.

A special thanks to some of my greatest friends for all the "coffees" we took, the (not so many) football games we played, and the birthdays I unfortunately missed! To Micael "Bitaites" Pinho e Gustavo "Neco" Oliveira, thank you for being always by my side and helping me when

I most needed friends that would be always with me! Really, thanks to both of you, you really showed me that there are still awesome, trustful, helpful, funny (etc) people I can rely on. To Diogo "Jet7" Dias, João "Zé Caloiro" Soares, Miguel "Franjinhas" Seabra, Luís "Tails" Rocha and all the “B320 for the most enjoyable days on college with laughs and games. Also thanks to two special pairs that I considered my two best pair-friends: to Diogo "Kuwait" Nunes and Daniela Nunes, ups, I mean Daniela Dias, thank you for the concerts we saw together and that those holidays we had which thanks to you I met the most important person in the world. Thank you Diogo for your awesomeness in stage as a singer and as a friend when I needed most, and thank you Daniela for being a sister to me and putting up with me during the pasted works and nowadays helping me motivating to move forward in my work; To Renato "Mono" Cardoso and Patrícia Fernandes, for being such a funny couple and so supporting and for remembering me to go out to rest a little bit from work. Thanks Renato, if it was not for you and our conversations I would be much better! Just joking, you really are a great friend for me and your support was unique for me! Thanks Patrícia for putting up with Renato as if it would not been you we could not bare to listen to him all the time!

An enormous thanks to my family for always standing by my side, keeping interested on my work and helping me morally. Thanks mom for all the worries about me and my health and for always cooking me the plates I wanted when I arrived home and upset. Thank you dad for raising me to be the best person for me and everyone around me, and all the efforts you did to keep me healthy and to have all the things I needed, and even more than that. I really appreciate all you did for me and my sister and I told you I would not disappoint you. Also thanks for hearing me talk about my work, even knowing that you did not understand it. And last but not least, thank you Andreia Carvalho for being the best big sister a brother could ever have by supporting me unconditionally every time I needed and putting me in front of every task you had to do.

A special colossal thank you to Cátia Vieira, the best girlfriend one could ever have. Thank you for baring me during the thesis and always keeping me up and give me strength to continue forward, with happiness and energy. These past few months were hard, and if it wasn’t for you it would be worst. Your care, love and affection are the source of energy I need every single day, for my entire life. Thank you for being what you are: Everything to me!

Finally, I appreciate the financial support given by the AMADEUS Project (FCT (Portuguese Science Foundation) project- PTDC/EIA/70271/2006) and the support given by the REFLECT project (partially FP7 EU funded).

Contents

List of Figures

List of Tables

Abbreviations

@AJ	@AspectJ
AO	Aspect Oriented
AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CCC	Crosscutting Concerns
IR	Intermediate Representation
JSF	Join point’s Specification File
MAJ	Meta-AspectJ
OO	Object Oriented
OOP	Object Oriented Programming

1. Introduction

The evolution of programming languages raised a number of fundamental concepts in the development of programs. Nowadays, programmers are exposed to high-level programming artifacts such as the use of APIs and classes, among other concepts currently important for software development. Surrounded by a number of programming paradigms, object-oriented programming (OOP) has widespread. In OOP solutions are decomposed into classes and the software code is based on the creation and communication between class instances, i.e., objects (T. Elrad et al., 2001).

All programming paradigms have their decomposing criteria, but there are concerns that do not align with the primary decomposition. These concerns are spread along the decomposition unit, most of the times repeated along the code, and pollute system's core objective (R. E. Filman et al., 2004). It is possible to address these concerns with the use of Aspect Oriented Programming (AOP) (A.-O. S. A. O. Website, 2011). This paradigm is based on the idea that systems work better if the properties and areas of interest, named concerns, are specified separately and then the aspects describe the relationships between them. The weaving of the AOP will then bring these concerns together into the intended program (T. Elrad, et al., 2001).

With the growing of AOP, currently, there is a diverse set of Aspect programming languages, such as AspectJ (T. E. F. Website, 2011) and AspectC++ (AspectC++ Website, 2011), just to name a few. These AOP languages focus on specifying aspects for the corresponding target programming language. For instance, AspectJ is an aspect-oriented extension for Java^{TM} (G. Kiczales et al., 2001).

The main objective of this thesis is the development of a meta-language for AOP, to permit the use of aspects for different programming languages, i.e., a definition of an aspect in this supposed meta-language shall be generic enough so it can be applied to several problems, in a number of (specified) languages. The two central programming languages studied in this thesis are MATLAB and C, two of the most used programming languages in embedded systems. Our Approach intends to allow programmers to use this meta-language for code insertion purpose (e.g., for tracing and monitoring) and for changing the types and precision of variables (e.g., from double to single precision, from singe precision to fixed-point precision).

1.1. Aspect-Oriented Programming (AOP)

The limitation of traditional programming languages implies the *tyranny of the dominant decomposition* (P. Tarr et al., 1999), permitting only the modularization of programs one way at a time. As a result, the rest of the "secondary" concerns that do not align with the modularization get scattered and scrambled along the modules. These problems are normally called crosscutting concerns (CCC) and are essentially concerns that do not align with the primary decomposition. There are two types of symptoms that display the presence of CCC: code tangling and code scattering (R. E. Filman, et al., 2004). Code scattering can be seen when there are code fragments spread across the units of modularity, dealing with repeated code along the program. Code tangling can be observed in the crossing code modules, i.e., the primary concern appears tangled with code from other modules, becoming difficult to comprehend all the concerns in the unit (R. E. Filman, et al., 2004).

The class *Point* in Figure 1 represents a point in space with x, y and z coordinates. Each *set* method aims to update a coordinate of a point, and the *get* methods are used to obtain its coordinates. In each of those methods there is at least one print that is normally used for debugging or just as secondary functionality. For instance, with a graphical user interface the prints are only shown in the console (on second plan) just for debugging purpose or as feedback. All these prints, identified in Figure 1 with rectangles, pollute the code and make it more difficult to read and understand. Also, in a final version of the code those *prints* should be removed as they do not concern the main objective of the system and probably are used only during the system development.

```
public class Point{
    public int x,y,z;
    public int getX() {
            System.out.println("Returning X with value:" + this.x);
    return x;
    }
    public int getY() {
      System.out.println("Returning Y with value:" + this.y);
      return y;
    }
    public int getZ() {
      System.out.println("Returning Z with value:" + this.z);
      return z;
    }
    public void setX(int x) {
      System.out.println("Passing X value from" + this.x + " to " +x);
      this.x = x;
      System.out.println("The point is now: " + this.toString());
    }
    public void setY(int y) {
      System.out.println("Passing Y value from" + this.y + " to " +y);
      this.y = y;
      System.out.println("The point is now: " + this.toString());
    }
    public void setZ(int z) {
      System.out.println("Passing Z value from" + this.z + " to " +z);
      this.z = z;
      System.out.println("The point is now: " + this.toString());
    }
...
```

Figure 1: Java code, based on the Java^{TM} Class Point (J. P. Website, 2011), with some tangled code.

AOP is a programming paradigm whose objective is to increase modularity by separating CCC (G. Kiczales et al., 1997). By separating secondary concerns from program's core objective using aspects results in cleaner code, easier concern analysis (the concerns are separated), easier for monitoring, tracing, debugging, etc. (R. E. Filman, et al., 2004) As referred in (S. F. Website, 2011) "Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects."

Some important concepts of AOP (A.-O. S. A. O. Website, 2011) are:

- *Crosscutting concerns*: the concerns to be modularized. These are the program secondary functionalities that are scattered or tangled throughout the program;
- *Join point*: a point of interest in the execution. The point in the code where we want to apply an aspect;
- *Pointcut:* a set of join points, usually in a form of a predicate that matches join points;

- *Advice*: an action to be taken in a join point. An Advice is related to a pointcut expression and acts over a join point related to the pointcut. This action can be applied before, after or around the specified join point;
- *Weaving*: a tool for the linkage of aspects with the primary code. When applying aspects into OOP, the weaving links aspects with objects to create an "advised" object;
- *Join point shadow*: is a projection between the join point and the referenced point in the program code where the compiler works. In other words, it is the section within the code where the join point is met.

As mentioned before, there are several AOP languages. Those aspect languages try to aid programmers with mechanisms to achieve better modularity for their programs. Each AOP language addresses one or more purposes, being AspectC++ used for tracing purposes, and AspectJ for general purpose (G. Kiczales, et al., 2001; O. Spinczyk et al., 2005). Most AOP approaches consider the weaving at source-code, outputting code with the concerns defined with the aspect language. There are also AOP approaches that do bytecode weaving instead of source code weaving (T. E. F. Website, 2011).

Thus, an AOP language may be used to specify the CCC illustrated in Figure **1** and may allow the use of a single version of the code (Figure 2) which added with the aspects in Figure 3 can produce code similar to the code in Figure 1. As it is clearly seen from Figure 2, removing the code related to the secondary CCC results in more readable and understandable code.

```
public class Point{

    public int x,y,z;

    public int getX() {return x;}
    public int getY() {return y;}
    public int getZ() {return z;}

    public void setX(int x) {this.x = x;}
    public void setY(int y) {this.y = y;}
    public void setZ(int z) {this.z = z;}
    ...
}
```

Figure 2: Java code of class *Point*, including just the main concern.

The secondary CCCs are migrated to AspectJ as illustrated by the aspect shown in Figure 3. The two pointcuts (lines 2 and 4) create capture calls to the *set* and *get* methods in the class *Point* (Figure 2). The first advice (lines from 8 to 9) inserts a print before the calling to a *get*, informing which member is being called. The second advice (lines from 10 to 13) inserts a print informing that a member of the class *Point* is going to be altered to a certain value and, after the call, shows the updated position of the point (B. Griswold et al., 1998).

```
public aspect PointMonitor {

pointcut getters(Point p):
    target(p) && call(int get*());

pointcut setters(Point p, int value):
           target(p) &&
           call(void set*(int)) &&
           args(value);

before(Point p): getters(p) {
    System.out.println("Returning "+ thisJoinPoint.toShortString());
  }

void around(Point  p, int value): setters(p, value){
    System.out.println("Passing " + thisJoinPoint.toShortString()+
                          " member to " + value);
    proceed(p,value);
    System.out.println("The point is now: " + p.toString());
  }
}
```

Figure 3: Aspect concerning the CCC depicted above.

As previously mentioned, the weaving of the original code (Figure 2) with this aspect (Figure 3) will produce the code shown in Figure 1. Note that an important property of AOP is the reuse of aspects. In this case, the aspect of Figure 2 can be used, e.g., to monitor all the arguments passed to *set** methods, or to monitor the return value of *get** methods.

1.2. Meta-Language

Most AOP languages are tied to the target programming language. An AOP approach using a Meta-language would possibly untie that connection, by centering the AOP approach to the meta-language to be used to address an extensible set of programming languages.

In computer science, a meta-language is considered a language that is used to define or analyze another language or symbolic system (D. c. Website, 2011). A specific system is built in a programming language, which has a weaver that applies aspects to that programming language, This meta-language is used to represent the aspects for the programming language that its weaver can receive as input to apply the aspects, abstracting itself from the programming language representing the model (the programming method) and also from its referent aspect language (P. C. Website, 2011).

Meta-programming has the goal of generating other programs, possibly in a different language, having as base the program written in the meta-language (S. S. Huang et al., 2008). Hence, in this thesis, a meta-language is used to define aspects for other programming languages through abstractions related to the concerning programming language. The aspects are then structured into a representation which the weaver needs to recognize.

1.3. Context

This thesis is related to the AMADEUS R&D project (AMADEUS Website, 2011). AMADEUS is a research project partially funded by FCT[1] and with the participation of the following institutes:

- Faculdade de Engenharia da Universidade do Porto (FE/UP)
- Instituto de Desenvolvimento de Novas Tecnologias da Universidade Nova de Lisboa (UNINOVA/FCT/UNL)
- Fundação da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FFCT/FCT/UNL)
- Universidade do Minho (UM)
- Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa (INESC ID)

The AMADEUS project has the objective to enhance MATLAB development and implementation systems by using aspects. Specifically, the AMADEUS project focuses on optimizing programs in MATLAB by extending them with an AOP approach. This approach uses AOP to specify additional information such as type definition, to try different approaches to functions and variables, and to configure low-level representation of variables and expressions (J. M. P. Cardoso et al., 2010; J. M. P. Cardoso et al., 2006; R. Nobre et al., 2010).

The AMADEUS compilation flow is presented in Figure 4. This flow receives MATLAB code as input. The Front-End transforms the code into an intermediate representation (IR), based on TOM (T.-L. Website, 2011). This front-end is the MATLAB to IR (Mat2Tir). Then this IR can be passed to the weaver or it can be immediately passed to the back-ends. The weaver receives as input, apart from the IR, the aspects and the aspects strategies. These files are where the programmer specifies the desired actions using aspects to be mainly applied over the IR. Some of those aspects will not be applied in this stage. They are propagated throughout the files "Types and Shapes". These files represent the types and/or shapes to apply to the variables of the program. The weaving associated with the types and shapes is performed in an intermediate stage named *tir2tir*.

[1] Fundação para a Ciência e a Tecnologia: http://www.fct.mctes.pt

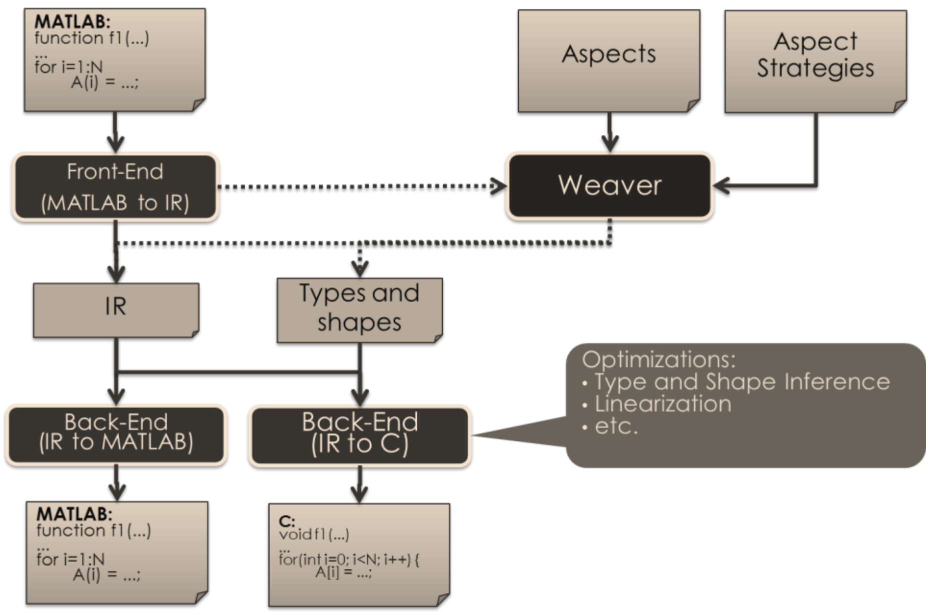

Figure 4: Flow of AMADEUS (J. M. P. Cardoso, et al., 2010).

The IR is then passed to the back-ends. The current back-ends include a tool to transform the IR into MATLAB code (the *tir2Mat* tool), and a tool to transform the IR into C code (the *tir2C* tool). Both of these results are representations of the same program.

As a way to increase the applicability and visibility of the AOP language, the work presented here also partially addresses its application to the REFLECT design flow (R. P. Website, 2011), which mainly consider C programs and the use of aspects and strategies to control and guide the compilation of input C programs to systems consisting of hardware and software components (J. M. P. Cardoso et al., 2011).

1.4. Objectives and Motivation

The possible creation of an aspect meta-language that can be used over a number of primitive languages could become most valuable for programmers as they will only need to learn one aspect language instead of learning one aspect language per programming language, such as AspectJ (T. E. F. Website, 2011) for Java and AspectC++ (AspectC++ Website, 2011) for C/C++.

Although with some aspect languages we are able to insert new members to classes, they cannot change the types of existing variables. The possibility of changing types of variables is to the best of our knowledge something recent in AOP (J. M. P. Cardoso, et al., 2006), and is an important issue if one needs to try different implementations for a variable for performance or precision purposes. Also, AOP languages such as AspectJ and AspectC++ are tied to their specific target programming language. Meaning that if there is an aspect created on AspectJ that could be useful for programs that are not created on Java, let's say C++, the aspect needs to be recoded to another aspect language, e.g., to AspecC++. Another problem of these aspect languages is that they do not influence many compiler flow stages, focusing only on source-to-source modifications for example.

The flexibility of this meta-language also will allow it to deal with a number of programming languages and also to some of their variants. For example, considering MATLAB it could be possible to deal not only with MATLAB programs but also with programs using programming languages variants of MATLAB, such as SciLab (C. Gomez, 1999) and Octave (J. W. Eaton et al., 2009; A. Quarteroni et al., 2010).

This thesis focuses on MATLAB and C programming languages as case studies. These two languages and two prototype tool chains will be used to illustrate the meta-language flexibility.

1.5. Thesis Structure

Beside the introduction, this thesis contains five additional chapters. Chapter 2 describes the most relevant related work. Chapter 3 describes the meta-language detailing its usage and utility. Chapter 4 describes the implementation of the meta-language and its respective front-end. Chapter 5 addresses the case studies used to validate the meta-language. Finally, in Chapter 6 we draw some conclusions and present possible avenues for future work.

2. Related Work

This chapter describes the most relevant related work to this thesis concerning aspect oriented software development. Each programming language follows a program paradigm being object-oriented programming one of the examples. However, there are always crosscutting concerns (CCCs) that are spread over multiple functions/classes. AOP (G. Kiczales, et al., 1997) aims at addressing CCCs, by modularizing them and eliminate their negative symptoms (M. P. Monteiro et al., 2010). Currently, there are several aspect-oriented programming (AOP) languages that extend traditional programming languages with aspect-oriented capabilities. Examples of AOP languages are AspectJ (T. E. F. Website, 2011) for Java, and AspectC++ (AspectC++ Website, 2011) for C++.

The following sections describe the most relevant approaches to AOP. Those approaches were used to analyze most common AOP features and to acquire inspiration for a Meta-language for AOP.

2.1. AspectJ

AspectJ (T. E. F. Website, 2011) is an AOP extension for Java^{TM} aimed at providing better modularity for developing Java programs. Developed by Xerox PARC and launched in August 1998, AspectJ contributes for a cleaner and better code by modularizing programs, providing solution for several CCCs such as monitoring, logging, debugging and synchronization. It is a general-purpose AOP language just like Java is a general-purpose OO language (B. Griswold, et al., 1998).

AspectJ supports two types of crosscutting implementations. One type is called dynamic crosscutting and allows adding implementations to act over defined points of execution on programs. The other type is called static crosscutting, and enables the definition inter-type members and other declaration forms (AspectJ Website, 2011).

AspectJ is faithful to its target language, Java, as it partially follows a similar structure of Java programs. For instance, the definition of an aspect is equivalent to a definition of a class, except that it is used the keyword *aspect* instead of *class* (B. Griswold, et al., 1998). The example below defines an empty aspect named “mainTesting”.

```
public aspect mainTesting {}
```

To be able to apply actions over the targeted code, AspectJ provides *join points*. Those join points, the points of execution in the program, are used to create *pointcut expressions* that define the points of interest within the code. There are two types of *pointcuts*: primitive *pointcuts*, the sets of *join points* and values on those points of interest; and user-defined *pointcuts*, named sets of *join points* and values (B. Griswold, et al., 1998). The following code defines a pointcut that captures every execution of the public main function, containing an array of Strings as argument.

```
public aspect mainTesting{
	pointcut mainMethod(String[] val):
		execution(public static void main(String[]))
		&& args(val);
}
```

After the pointcuts definition, certain additional actions can be applied over a certain join point in a pointcut, i.e., an advice. This action can be applied before, after or around that join point (B. Griswold, et al., 1998). The example below contains a simple aspect, named “mainTesting”, with a pointcut targeting the main function execution, and with an advice that inserts a print statement of “Hello from AspectJ” into the code, before the execution of the targeted join point, i.e., the main function execution.

```
public aspect mainTesting{
	pointcut mainMethod(String[] val):
		execution(public static void main(String[]))&&
		args(val);

	before(String[] val):  mainMethod(val) {
		System.out.println("Hello from AspectJ");
	}
}
```

2.1.1. **Join points and Pointcuts**

The join points are the points of interest within the code that one desires to capture so an action can take place on that concerning point.

> "A critical element in the design of any aspect-oriented language is the join point model" (AspectJ Website, 2011)

To capture the points of interest within the concerning code, AspectJ contains a preset of join points of the Java programming language syntax. They point to the desired place where the user requires applying a certain action.

The points of interest of where one wants to affect the program are the join points. For example, the following code:

```
int getX() {return this.x;}
```

Can be read as: "When the method named **getX()** that takes no arguments is called, it has to be executed the function's body **{return this.x;}**, which returns an integer". The execution of the function's body is an example of a specific point within the code that can be captured. Those specific points are the join points, and consist of (B. Griswold, et al., 1998):

- Method or constructor calls
- Method or constructor receptions
- Method or constructor executions
- Dynamic or static object initializations
- Field's Gets and Sets
- Exception's handlers execution

In the example below, the pointcut is associated to each call to **getX(), getY(),** or **getZ()**.

```
pointcut getters():
    call(int getX()) ||
    call(int getY()) ||
    call(int getZ());
```

Thus, with a pointcut definition it is possible to combine join points to specify points of interest. Some pointcuts can be (B. Griswold, et al., 1998):

- method execution
- method call

- method reception call
- execution of exception’s a handler
- this – object executed
- target – targeted class
- within – executing code belonging to a certain class
- cflow – the join point is in the control flow

AspectJ allows the combination of pointcuts with some logical members. Those logical members are (AspectJ Website, 2011):

- `!getters` – every join point not selected by the pointcut ***getters***
- `getters && setters` – the join points that are selected by both of the pointcuts
- `getters || setters` – the join points that are selected by the pointcut ***getters*** or by ***setters***

This way, AspectJ can accept multiple combinations of types of pointcuts, as illustrated in the example below. This aspect captures every call to Point’s *getX*, *getY* and *getZ* functions, when the targeted class is the class Point.

```
pointcut getters():
    target(Point) && (
      call(int Point.getX()) ||
      call(int Point.getY()) ||
      call(int Point.getZ())
    );
```

The pointcuts can also be defined, for example, within a class. Aspects can access that pointcut indicating the path to that pointcut, similar to: <ClassName>.<PointcutName>.

Instead of defining the complete name of what is planned to catch, AspectJ allows the use of wildcards in the pointcut expressions. With these wildcards, the aspects are more reusable and a broaden set of pointcuts can be picked with less repeated code. Some wildcards examples can be seen in Table 1. There is two types of wildcards: the ‘*’ symbol that represents any name, and the symbol ‘..’, which is a multi-part wild card, and can be seen as any set of values (B. Griswold, et al., 1998).

Table 1: Examples of wild card usage in AspectJ

Pointcut Expression	***Definition***
`execution(public * get*(..))`	Execution of any public method with name that starts with get, with any return type and any arguments
`handler(Exception+)`	Execution of an handler for an Exception or any of its subclasses
`org.eclipse..*`	Any declaration in a package that starts with "`org.eclipse.`"

Applying a wildcard in the example code above, we can obtain cleaner and reusable code similar to the example below. The example also captures every call to Point's get functions. The difference to the above example is in the use of a wildcard. When creating new get functions on Points, with no arguments and returning int, this pointcut does not need to be altered. On the contrary, the example above has to be updated so it can capture the newest get functions.

```
pointcut getters():
    target(Point) && call(int Point.get*());
```

All the examples given in this section do not have parameters on the pointcut's left side. This way the context from the join points is not published and so not accessible. It is not possible to obtain the context of the information that we may require. By including parameters in the pointcut it is possible to obtain that context, as illustrated by the following example:

```
pointcut getters(Point p):
    target(p) && call(int get*());
```

In this example the pointcut has one parameter of type **Point**. This way, the advices using this pointcut can access the Point **p** intercepted in each join point selected. (AspectJ Website, 2011).

2.1.2. **Advice**

After the definition of pointcuts, the action to be taken over the join points is called an advice. The advice defines the actions to take place on the pointcut expression, and those actions are used, for example, to insert new code into the primary program. The action to take place can be used, for example, to evaluate the values inserted (for example, overflow values), throw exceptions, debugging, synchronizing, among other uses. The advice can be applied (B. Griswold, et al., 1998):

- **before** the join point. The action takes place before passing the join point:

```
before(Point p): getters(p) {
      System.out.println("About to use a get*()");
    }
```

- **after returning** a value to the join point. The action will only be applied if the program does not throw an exception:

```
after(Point p) returning: getters(p) {
      System.out.println("Get successful");
    }
```

- **after throwing** on the join point. When an exception is thrown, the action is applied after throwing:

```
after(Point p) throwing: getters(p) {
      System.out.println("An error occurred");
    }
```

- **after** the join point, regardless of returning or throwing:

```
after(Point p): getters(p) {
      System.out.println("Left get function");
    }
```

- or **around** the join point. In this type of advice the aspect controls if the program proceeds into the join point and places actions before and after the join point, or takes another action different from the existing in the join point. To execute the captured join point, AspectJ holds a primitive function named *proceed* with the same signature as the around advice.

```
int around(Point p): getters(p) {                 Override of a getter
      return 0;
    }
```

```
int around(Point p): getters(p) {                 Advising around the pointcut
      System.out.println("About to use a get*()");
      int val = proceed(p);
      System.out.println("Left get function");
      return val;
    }
```

Sometimes the simple definition of pointcuts to define the points of interest can return undesired join points. The advice allows the implementation of conditions to clearly clarify the desired join points and therefore remove the undesired join points within a join point. It is possible to define pre-conditions like parameter validation, for example:

```
before(Point p, int x) throws Exception:
    target(p) && call(void setX()) && args(x){
      if(x > MAX || x < MIN)
            throw new Exception("Overflow: "+x);
    }
```

Before running the **setX** function, this advice verifies if the new value is within limits. If the new value exceeds a maximum value or is below a minimum value, the aspect throws an exception indicating that the value is invalid.

The following example shows how a post-condition can be implemented:

```
after(Point p, int x) throws Exception:
     target(p) && call(void setX()) && args(x){
        if(p.getX() != x)
                throw new Exception("setX Error: "+ p.x +" != "+x);
     }
```

This example verifies if the new value was properly placed in **p.x**. This way, it is possible to know, during the program execution, if the value was assigned correctly and no problem occurred during the **setX** execution.

Finally, it is also possible to manipulate values so the join point can meet the necessary conditions to be properly executed. For example:

```
void around(Point p, int x) throws Exception:
     target(p) && call(void setX()) && args(x){
        int limitedX = Math.max(MIN, Math.min(x, MAX));
                proceed(p,limitedX);
     }
```

This advice limits the new value to the defined range, avoiding possible overflow problems (B. Griswold, et al., 1998).

To manipulate a join point, AspectJ offers a reflective access to the join point by the special variable **thisJoinPoint** (B. Griswold, et al., 1998). This variable (AspectJ Website, 2011) can access static and dynamic information of the join points. Some of the methods that can be used are:

- **toString()** – method that return a string with a complete the join point definition
- **toShortString()** – returns a shorter definition of the join point
- **getArgs()** – the list of the join point's arguments
- **getStaticPart()** – the object with all the static information about the join point
- **getSignature()** – the signature of the given join point
- **getParameter()** – the join point's parameters

```
before(Point p): target(p) && execution(* *(..)){
        JoinPoint jp = thisJoinPoint;
        System.out.println("Execution "+jp.toShortString()+" with arguments: "+
jp.getArgs());
        }
```

This example prints in the console the joint point description and its arguments on every occasion of executions of any method of the class Point, with any arguments and any type of return. E.g.:

"Execution execution(Point.setX(..)) with arguments: 4"

2.1.3. **Abstract Aspects**

Just like Java can create abstract methods and classes, in AspectJ it is possible to define abstract aspects and aspect pointcuts. With an abstract aspect it is possible to reuse an aspect by defining a new aspect and then extend it with the abstract aspect. Abstract pointcuts are also important for the reusability given that the extended aspect can define those pointcuts as they fit better in each aspect case. If there are two aspects that extend the same abstract aspect, each one can have its own way to implement a pointcut.

In the example below we have an abstract aspect that contains an abstract pointcut and an advice to apply over that pointcut. Having the advice implemented, once an aspect extends this abstract aspect and implement the referent pointcut the aspect can be applied over the join points that this pointcut targets. The 2DPoint and 3DPoint aspects implement different pointcuts but both use the same advice.

```
abstract public aspect PointAspect{
abstract pointcut getters();

        before(): getters() {
                System.out.println("Get: "+thisJoinPoint.toShortString());
        }
}
public aspect 2DPoint extends PointAspect{
        pointcut getters():
                target(Point) && (
                        call(int Point.getX()) ||
                        call(int Point.getY())
                );
}
public aspect 3DPoint extends PointAspect{
        pointcut getters():
                target(Point) && (
                        call(int Point.getX()) ||
                        call(int Point.getY()) ||
              call(int Point.getZ())
                );
    }
```

Also, while an abstract aspect or pointcut is not defined, the aspect, or the advice using the referenced pointcut, will not influence the main code.

2.2. AspectJ as an extensible aspect language

The original AspectJ compiler is ajc (W. Cazzola, 2011a), created by Xerox, and is a static compiler, i.e., this is an inflexible compiler that can only be upgraded if the source is altered. The compiler takes Java source and .class files and AspectJ files, and the weaving process takes place on one of the three different types of weaving: compile-time, post-compile time and load-time. In the end, the resulting files are .class files altered regarding the aspects used, and those files are the same despite the type of weaving the user choose.

As response to the inflexibility of ajc, the abc compiler for AspectJ (abc Website, 2011) is an extensible version built in a way that the AspectJ can be more easily upgraded on its features. The compiler's main structure contains the AspectJ language and with its front-end, built using Polyglot framework (W. Cazzola, 2011a), the compiler can be extended with new features, such as join point model enrichment (W. Cazzola, 2011b).

Having so, AspectJ has some extensions for an upgraded language to support the user needs. Between those extensions, the ones that stand out are LoopsAJ, Meta-AspectJ and @AspectJ. Each one extends the aspect language to upgrade to the needs of the extension's objective, such as addition of new join points to the join point model to capture loops, for LoopsAJ.

2.2.1. **LoopsAJ**

LoopsAJ (L. Website, 2011) is an extension of AspectJ which implements a join point model for loops to be possible the capture of loops over Java code (L. Website, 2011). This approach permits the direct intervention over loops (B. Harbulot et al., 2006). As mentioned before, LoopsAJ uses AspectJ's abc (abc Website, 2011) compiler for the reason that abc is more extensible and relies on Soot framework (S. Website, 2011). The weaving is done in bytecode level. This is because Soot framework uses a three-address representation of bytecode (B. Harbulot & J. R. Gurd, 2006).

This extension adds a new pointcut to AspectJ, named *loop()*. It also includes contextual information that can be used for loop parallelization. So, it is possible to expose the context that a certain loop contains (B. Harbulot & J. R. Gurd, 2006). LoopsAJ can capture three categories of loop, being each one captured with a different pointcut expression (B. Harbulot & J. R. Gurd, 2006).

The first category groups the loops with several successors, also called general case (B. Harbulot & J. R. Gurd, 2006). These loops can be advised before and sometimes after, but not in all cases. They have one entry point (the header), and actions before the loop can be passed into a pre-header (node inserted before the loop header). The context of the loop cannot be obtained for this category. But it is possible to have more than one potential exit from the loop. This can be caused by using the *break* and *continue* statements inside nested-loop, which can result in different successor nodes[2]. So, weaving a possible advice as an after-action could mean a repetition of code in all possible exit nodes. The same problem is driven to the around option, for the reason that it is impossible to know if the *proceed()* action returns to the original join point. The second groups the loops with a unique successor. This is the default case where exit node branches always to the same successor node. Having a single successor, these loops can be advised before, after or around it. The after advice can be weaved in the end of each exit node or before the single successor. This category also does not support the context exposure. The last category groups the loops with a unique exit node. The loops in this category can be advised before, after and around the loop. In this category it is possible to obtain the loop information for the context exposure (B. Harbulot & J. R. Gurd, 2006).

LoopsAJ can expose the context relative to a loop. This implementation considers that, just like functions calls, loops depend on contextual information. This information can be important to access by the programmers (B. Harbulot & J. R. Gurd, 2006), having two types of contextualized loops considered (B. Harbulot & J. R. Gurd, 2006):

- Loops that iterate (regularly) between a range of integers;
- Loops iterating over an iterator;

To expose that information, the loop-invariant assignments are passed outside the loop into the pre-header. The next example, extracted from (B. Harbulot & J. R. Gurd, 2006), expose the process of separating the loop-invariant assignments from the loop to the pre-header.

The first step is the separation of the invariants from the loop, i.e., the variables that do not change during the loop. Normally, that invariant is the incremental value. The second step is the storage of the two boundaries (the minimum value and maximum value) on variables outside the loop. This way we obtain the context related to the loop, containing the minimum value, the maximum value that the iteration variable can assume, and the increment to apply (B. Harbulot & J. R. Gurd, 2006).

---

[2] Successor nodes are the nodes outside the loop that can create different branches depending on how/where the loop breaks/continue. It is possible to have multiple exits to the same successor.

```
/* Moving the invariants outside */
int i = 0 ;
while (i < 10) {
       /* ... */
       int stride = 3 ;
       i = i + stride;
}
/* First step: moving the invariants outside */
int i = 0 ;
int stride = 3 ;
while (i < 10) {
       /* ... */
       i = i + stride;
}
/* Second step: storing the boundaries in temporary variables */
int stride = 3 ;
int minimum = 0 ;
int maximum = 10 ;
int i = minimum ;
while (i < maximum) {
       /* ... */
       i = i + stride;
}
```

As mentioned before, the loops can be iterated by a range of integers or by an iterator. For each type of iteration, there are different forms to represent the pointcut. Those forms are shown in Table 2 (B. Harbulot & J. R. Gurd, 2006). In this table, it is possible to observe the basic pointcuts to capture loops. LoopsAJ uses the pointcut function *args* obviously not to verify the arguments list related to the loop but for context verification/attainment. For loops iterating in a sequence of integers, the limit values and the incremental value are passed to *args*. If an array can be found then a reference for that array can be bound to the pointcut parameter *array*. In case of an iterator the first argument should be the instance of the iterator. Here, a collection is the argument that can be obtained. When this obtained argument do not matter, the '..' wildcard is used instead of a possible variable (B. Harbulot & J. R. Gurd, 2006).

Table 2: Basic pointcut expressions to use in LoopsAJ

Pointcut expression	***Target***
`loop()&& args(min, max, inc)`	Loops that iterate over a range of integers and the compiler was unable to find an array
`loop()&& args(min, max, inc, ..)`	Loops iterating over an arithmetic sequence of integers
`loop()&& args(min, max, inc, array)`	Loops iterating over an array.

These two pointcuts captures two different types of loops. The first one captures every loop that is iterated by a range of integer values. If possible, the pointcut gives a reference to the array of the iterating values. In the second pointcut captures the loops that are iterated by an iterator. Also, if possible, it gives the collection of iterators that are iterated (B. Harbulot & J. R. Gurd, 2006).

```
pointcut iterationByArray
  (Object[] array, int min, int max, int inc):
      loop() && args(min, max, inc, array);

pointcut iterationByIterator
  (Object[] collection, Iterator it):
      loop() && args(it, collection);
```

Theoretically, this approach could be applied to other languages by integrating it in other aspect-oriented languages, but the fundamental objective of this model was to provide loops join point using AspectJ (B. Harbulot & J. R. Gurd, 2006).

2.2.2. **Meta-AspectJ**

Meta-AspectJ (MAJ) (M.-A. Website, 2011) is another extension for AspectJ that combines AspectJ files with code-generation and generates AspectJ programs with correct syntax, being a well-typed generator. A methodology that combines Aspect-Oriented programming with program generation, this meta-language minimizes the number of meta-programming operators to only two, the quote operator and the unquote operator. Also, it uses type inference to reduce the necessity of memorizing the type names of the syntactic entities (S. S. Huang, et al., 2008).

MAJ is a meta-language for generating AspectJ programs using code template, and can be used to implement (S. S. Huang et al., 2006; S. S. Huang, et al., 2008):

- generators using AspectJ;
- general-purpose aspect languages using generation.

The implementation of generators can be used to implement domain-specific languages translating domain-specific abstraction into AspectJ code; The general-purpose aspect language implementation is intentioned to create extensions to AspectJ of general-purpose, extensions that recognize different join point types (S. S. Huang, et al., 2008).

The problem this language tries to answer on AspectJ is that, even having syntax that enables the dynamic creation of aspect, like wildcards, it cannot support the adaptation of the application based on properties that is not possible to identify only with the syntactic name. And so, MAJ permits the verification of such information and with it is possible to create more complex and accurate join points. MAJ creators (S. S. Huang, et al., 2008) decided to group AspectJ with code-generation for the reason that AspectJ has good and appropriate vocabulary for the program transformation and therefore the code-generation can work over it to provide extra flexibility (S. S. Huang & Y. Smaragdakis, 2006).

MAJ can also be used to generation plain AspectJ code or even plain Java code, since it is a safe code generator that generates syntactically correct code represented explicitly on a typed structure (S. S. Huang, et al., 2008).

Two code template operators were added to Java (S. S. Huang, et al., 2008):

- ‘[…] - quote operator. Creates a representation of AspectJ code fragment
- #[Expr] or #IDENTIFIER - unquote operator. The representation of a variable inside a quote operator.

The quote operator allows the generation of an abstract syntax tree for a portion of AspectJ code and the unquote operator permits the usage of the content of a variable, with a piece of code, inside a quoted code (S. S. Huang & Y. Smaragdakis, 2006).

```
Pcd pcd1 = ‘[ call(* *(..)) ];
Pcd pcd2 = ‘[ call(* #methodName(..)) ];
```

The example above is a (quoted) declaration of a portion of AspectJ code that captures the calls to any method, being stored in a variable of type *Pcd* (Pointcut). In the pointcut *pcd1* there is nothing special about it, but when it is intended to insert some specific name (or expression) inside the declaration, the unquote operator enables such intention by substituting the name of a certain variable by its content: In *pcd2*, the usage of the variable *methodName* will be replace with its content. Also, inside the unquote operator can be used any expression, being always substituted by its final result. Being only useful inside a quote operation, the usage of an unquote operation can only be done inside it and never outside.

In Table 3 are depicted some of the types of entry points available in MAJ (M.-A. Website, 2011). These entry points correspond to a java class represented at the AST.

Table 3: Examples of entry points on Meta-AspectJ

Entry Point	***Example***
Single Identifier	IDENT classIdent = `[MyClass];
Identifier	Identifer vectorId = `[java.util.Vector];
Identifier Pattern	NamePattern anyPattern = `[*];
Modifiers	Modifiers staticPubModif = `[public static];
Import	Import vectorImport = `[import #vectorId;];
Expression	JavaExpr num8Int = `[8]; JavaExpre booleanExpr = `[2 >= 3];
Statement	Stmt countInc = `[count++];
Pointcut	Pcd allCalls = `[call(* *(..))];
Pointcut Declaration	Pointcut pcCalls = `[private pointcut #allCalls;];
Advice Declaration	AdviceDec advBeforeCalls = `[before(): #allCalls {}];
Aspect Declaration	AspectDec aspectForCalls = `[public aspect captureCalls {…}]

There are at least 20 entry points on MAJ, and so memorizing all those entry points could be very tedious and complex. To solve this problem, MAJ provides a special type named *infer*, a type that can be used to declare a variable of those types in place of their type name (M.-A. Website, 2011). This special type allows the declaration of variables being inferred by MAJ without the need of memorizing the desired type (M.-A. Website, 2011).

The example below shows some examples of different entry points declared with the special type *infer*. The two first lines are similar to be above example with the slight difference of not giving the immediate type of the entry point but the special type *infer*. The entry point on 3 is the declaration of an advice (*AdviceDec*), and line 5 infers the aspect declaration.

```
1. infer pcd1 = `[ call(* *(..)) ];
2. infer pcd2 = `[ call(* #methodName(..)) ];
3. infer adviceBefore = `[before: #pcd2 {}];
4. String aspectName = "MyAspect";
5. infer asp = `[ aspect #aspectName {}];
```

Thus the user does not need to worry about the type correctness as it is done by MAJ by inferring those types. The only limitation of its use is that a variable can only be declared as *infer* if it is immediately initialized, otherwise MAJ cannot resolve the variable's type. The example also shows the creation of an empty advice (*adviceBefore*) and an empty aspect (*asp*). Having no more information inside their scope, it is possible to add more code inside them, using the *addMember* method (D. Zook et al., 2004):

```
infer pcd1 = `[ call(* *(..)) ];
...
infer asp = `[ aspect #aspectName {}];
asp.addMember( `[public pointcut #pcd1] );
the same as: `[ aspect MyAspect {
                     public pointcut call(* *(..));
                     ...
                 }];
infer pc3 = `[ call * #methodName (#argsListTypes)];
```

The unquote operator can also be used with an array of expressions, called the unquote-slice. This mode of operation is used essentially to add arguments in a quoted-context that needs a flexible number of arguments (S. S. Huang, et al., 2008). If the variable *argsListTypes* is an array containing a set of types, the resulting value on the unquoted expression will be the set of those types listed as argument types in *pc3* on the above example (D. Zook, et al., 2004). For example, having: *{String, int, String}* inside the *argsListTypes* array, *pc3* will be a pointcut that captures every call to the method with name *methodName* and with 3 arguments of type, respectively, *String*, *int* and *String*.

One example of MAJ usage, given in (S. S. Huang & Y. Smaragdakis, 2006), is the creation of unimplemented methods for objects implementing interfaces. In Java, the implementation of an interface obliges the class to implement all of the interface methods. Some of those methods can be unnecessary for the class and Java always obliges the user to create them, even them being empty methods (returning a null value if a return is necessary).

```
public class MyPointer2D implements PointInterface {
       public void update2D(int x, int y){
         ... //Necessary method that does the update in 2D.
        }
       public int[] getPos2D(){
               ... //Necessary method that gets the position in 2D.
       }
      public void update3D(int x, int y, int z){} //Unnecessary Method
      public int[] getPos3D(){} //Unnecessary Method
}
```

With MAJ, is it possible to create the desired code, generating an AspectJ that implements the unimplemented (undesired) methods. The following example, created with help from (S. S. Huang & Y. Smaragdakis, 2006), represents how after the creation of a solution in MAJ code this can be used on the above example.

```
@Implements ("PointInterface");                                Original Java File
public class MyPointer2D {
       public void update2D(int x, int y){
             ... //Necessary method that does the update in 2D.
       }
       public int[] getPos2D(){
              ... //Necessary method that gets the position in 2D.
       }
    }
    public aspect implementsAspect1{                          Generated Aspect File
       void MyPointer2D.update3D(int x, int y, int z) {}
       int[] MyPointer2D.getPos3D() {}
       declare parents: MyPointer2D implements PointInterface;
}
```

The original java file only has to declare the *@Implements* annotation so MAJ compiler can be notified to create the corresponding aspect file that contains all the unimplemented methods. The MAJ code is *"less than 200 lines long, with most of the complexity in the traversal of Java classes, interfaces, and their methods using reflection"* (S. S. Huang & Y. Smaragdakis, 2006). The code process all input Java classes and retains the classes that contain Implements annotations. Then, finds all methods, pertaining to the interfaces passed as arguments in the Implement annotation, that are not implement in the class and then generates the AspectJ file with the implementation of such methods " (S. S. Huang & Y. Smaragdakis, 2006).

Hence, MAJ is a tool to generate Java and AspectJ programs and code templates for Java, and is a meta-language that does not have a heavy infrastructure, focusing on generating AspectJ code and assigning semantic matching issues to AspectJ. Using MAJ it is also possible to implement annotation-based domain-specific languages (just like the example above) that *"automate tedious and error-prone programming tasks"* (S. S. Huang, et al., 2008). With the type-checking that MAJ does to the generator, this meta-language extension assures correct syntax to the generated AspectJ/Java code (M.-A. Website, 2011).

2.2.3. **@AspectJ**

The @AspectJ (@AJ), developed by Marco Poggi (W. Cazzola, 2011a) and continued by Walter Cazzola, is *"A Fine-Grained AspectJ Extension"* that enables the capture of join points that cannot be addressed by the original AspectJ language because of its limited join point capture. For example, the capture of *if* and *while* join points and their inner join points, such as conditional expression and bodies. AspectJ is limited to the capture of method/constructor calls, object initializations and field accesses. This is caused by the heavy granularity of AspectJ's join point model and its dependency on the program syntax that only permits the capture of the available join points and does not considered the position and context of those join points (W. Cazzola, 2011b).

In the code below the *if* and *while* statement cannot be capture in the standard AspectJ version, as so the impossibility to take actions not only on them but also on their associated join points (conditional expression statement per example) (W. Cazzola, 2011a).

```
public static void main(String args){
        ...
        if(args.length < 5)
         usage();
        ...
        while(count < args.length){
         clone[count] = args[count];
         count++;
        }
        ...
}
```

@AJ tries to surpass those limitations by creating explicit marks to the desired join points in the code just like annotation-based pointcuts of AspectJ, with a finer granularity than AspectJ's. To do so, it takes advantage of the fine-grained annotation model provided by @Java[3] (W. Cazzola, 2011a)(annotation) programming language and allows the user to mark the desired join points statements, block of statements or even expressions, extending this way the join point model of AspectJ .

Just like @Java introduces some changes on standard Java, @AspectJ also introduces new primitive pointcuts: the *@block*, to capture block annotations and the *@expr* for the expression annotations. The following example contains the example above with a possible annotation to create a @AspectJ aspect aiming the conditions of *if* and *while* statements.

[3] @Java is also work of Walter Cazzola, and targets the increment of the granularity of the annotations in Java for an extended Java syntax.

```
aspect captureConditions{                                    @AspectJ
    pointcut conditions(): @expr(ifCond) || @expr(whileCond);
    ...
    }
    public static void main(String args){                         Java
    if( @ifCond{args.length < 5} )
      usage();
      while( @whileCond{count < args.length}){
             clone[count] = args[count];
             count++;
      }
    ...
}
```

This example shows the annotation on the code and a possible pointcut to capture those desired annotated join points. The aspect captures every expression with the corresponding tag and it can combine the new primitives with primitives of AspectJ. Also, it can be used as a regular pointcut for an advice command. In other words, the usage is similar to AspectJ, with the slight difference of the new primitives added and the annotations needed at the source code.

This is a good advantage for AspectJ evolution and aims for the expansion and flexibility of the join point model of AspectJ; nonetheless this forces the user to annotate the source code, eradicating the aspects separation with the java program and its obliviousness. Also, it means that the user needs to write those kinds of annotations whenever he wants to capture a certain point of interest. This also eliminates the generality of aspects since one aspect, using a certain pointcut created with a desired annotated join point, cannot be used over other source files if those do not contain that annotated join point.

2.3. AspectC++

The AspectC++ (AspectC++ Website, 2011) is an AOP extension to the C++ programming language. The motivation for the developers to create AspectC++ was that many programmers use C++ and the programs can become very extensive and difficult to maintain. To create better and cleaner C++ code, Dr. Olaf Spinczyk (O. S. Website, 2008) created an AOP extension for the C++ programming language. Code redundancy, reusability and maintainability are some of the advantages of using this aspect language (D. Lohmann et al., 2007).

"With the AspectC++ project we extend the AspectJ approach to C/C++" (AspectC++ Website, 2011)

To be able of applying aspects over C++ code, AspectC++ includes some extensions in the C++ grammar. Those extensions include (O. Spinczyk et al., 2002):

- **aspect** definition, similar to a class definition

- **pointcut** declaration, so it can be possible to define where the action should take place
- **advice** declaration, the action over a certain join point

AspectC++ relies in a source-to-source code weaving process. The application of aspects over C++ code results in another C++ code, modified with the desired aspects. The usage of AspectC++ is focused on the creation of tracing aspects. Even though, it is also used for profiling and constraint checks.

2.3.1. **Join points and Pointcuts**

The join points are the points of interest where an aspect can apply an action, and those join points can refer to a method, a class, a structure, an attribute or an object (O. Spinczyk, et al., 2002). The definition of pointcuts is done by combining join points. The pointcut expressions combine (O. Spinczyk, et al., 2002):

- match expressions – strings with search patterns
- pointcut functions – capture specific join points from a pointcut
- algebraic operators – combine pointcuts

The match expressions are strings that contain search patterns that use wildcards and named entities to express pointcuts. Those match expressions are primitive pointcut expressions that define a set of join points in the program's static structure. There are two types of wildcards in AspectC++: the '%' character is used to match any type or name; the '…' character stands for any sequence (O. Spinczyk, et al., 2005). Some examples can be seen in Table 4.

Table 4: Examples of wildcard usage in AspectC++

Wildcard example	***Description***
`"% *"`	Any pointer type
`"Foo%"`	Every class that begins with Foo
`% …::%(…)`	Any function of any class
`void …::get%()`	Any function that starts with get, of any class, with no arguments and returning void

AspectC++ also provides a set of pointcut functions that can be used to filter and select the join points that the aspect should influence. Those functions (O. Spinczyk, et al., 2002) are presented in Table 5.

Table 5: Provided pointcut functions

Pointcut function	***Target***
`call`	call of methods
`execution`	execution of methods
`callsto`	corresponding call join points to the execution join points
`get`	attribute access for reading
`set`	attribute update
`classes`	types of classes
`derived`	types of classes, structures or unions
`base`	types of classes, structures or unions
`instanceof`	types of classes, structures or unions
`within`	join points within the pointcut
`cflow`	join points of the dynamic execution, in the pointcut
`reachable`	all join points where the argument join points are reachable
`that`	join points referent to a certain object of a determined type
`target`	type of the target object of a call is compatible with the desired type
`args`	the given list of types matches with the arguments signature

The combination of pointcuts is also important for a better join point selection. Similar to AspectJ, in AspectC++ the pointcut expressions can be combined with set operators like intersection (`&&`), union (`||`) and negation (!) (O. Spinczyk, et al., 2005).

Some examples of pointcut definitions are presented below. This example contains three different pointcuts: one captures the execution of any get function with no arguments and any type of return; the other one captures calls to the set functions of class Point, with any number of arguments and returns void; the last one captures any function call with any arguments and any type of return, within the class Point.

```
aspect example {
      pointcut getters() = execution("% ...::get%()");
      pointcut setters() = call("void Point::set%(...)");
      pointcut allCallsInPoint() =      within(Point) &&
                                        call("% ...::%(...)");
    };
```

2.3.2. Advice

This is where the aspect's effect over the selected pointcut is defined. The actions to apply can occur, just like in AspectJ, before, after, or around the join point (O. Spinczyk, et al., 2005).

```
advice <pointcut_definition>: (before|after|around)(<args>) {
      ...
}
```

The above code represents the structure of an advice created in AspectC++. After indicating that it is an advice, the pointcut to capture needs to be inserted, and then inform when the advice is applied.

In the advice body are the actions that should take place on the join point. To manipulate and observe information on the join point, AspectC++ provides a pointer to access the join point's context information. The context information is provided by two ways, in compile-time and in runtime (O. Spinczyk et al., 2011). In compile-time we can obtain information like the type of object, target, result and arguments or the number of arguments. In runtime we can obtain pointers to the object, target, result and arguments. This pointer also provides the method to proceed the join point execution, just like in AspectJ, when inside an around advice (O. Spinczyk & D. Lohmann, 2011).

In the first advice of the example below, before every call of a function in class Point, the system prints the captured join point and the arguments for that function. On the second advice, every time that a set is called, before the call, the system should print the join point's signature, and after the call should print the termination of that call.

```
advice allCallsInPoint() : before(){
        cout << "Entering join point " << JoinPoint::signature();
        cout << " with "<< JoinPoint::ARGS << " arguments." << endl;
    }

advice setters() : around(){
        cout << "Before set: " << JoinPoint::signature() << endl;
        tjp->proceed;
        cout << "Ending set" << endl;
    }
```

Another type of advice is the introduction, an advice for static join points. This type of advice allows the inclusion of new members or methods inside a class or structure. The pointcut expression indicates where the information is going to be inserted, and the declaration is the information to insert (O. Spinczyk, et al., 2005). The advice, exampled below, will include a new method into the class **Point**. The introduced members are visible beyond the aspect. Although the main program is unaware of the existence of aspects that affect that program, that program can use this new function even not knowing if the function is created. It is some kind of obliviousness balance, for the reason that the main program, even not knowing the aspects affecting it, considers that the function will be created by some means different from its implementation (O. Spinczyk, et al., 2005).

```
advice "Point" : setPosition(int newX, int newY, int newZ){
        setX(newX);
        setY(newY);
        setz(newZ);
    }
```

2.3.3. **Virtual pointcuts**

The named pointcuts can be declared as virtual, the same as abstract in AspectJ. These virtual pointc∂uts only define how the crosscutting concern is implemented and does not advise where the effect should take place in the program (O. Spinczyk, et al., 2005). These two virtual pointcuts are just like abstract pointcuts in AspectJ. The getters and setters pointcuts are not created and are just used as abstraction.

```
pointcut virtual getters() = 0;
pointcut virtual setters() = 0;
```

The virtual pointcuts can be used by the advices, but the aspect will not affect the program while the virtual pointcut is not implemented. For that, an aspect can inherit this aspect with the virtual pointcut and implement that virtual pointcut. The inheritance is similar to the inheritance over classes, where the aspect that is inheriting another aspect implements the missing pointcut declarations (O. Spinczyk, et al., 2005).

With virtual pointcuts, we are able to define reusable aspects. Aspects with at least one virtual method or pointcut are considered abstract aspects (O. Spinczyk, et al., 2005).

2.4. AspectMATLAB

AspectMATLAB (AspectMatlab Website, 2011) provides AO extension for MATLAB (T. Aslam et al., 2010) and was created thinking on the needs on developing scientific applications. This aspect language supports the same notions of other aspect oriented languages, but it gives them other terminologies: the pointcuts are called patterns; and advice is called a named action. This choice of terminology was based to clarify the user which is the matching objective and which is the action to take place on the matching pattern. The definition of an aspect in AspectMATLAB is similar to a class in MATLAB (just like in AspectJ the definition of an aspect is similar to a class in Java). It is possible to define fields and methods in an aspect, enabling the addition of important information for the aspect. Additionally, it is also possible to specify patterns and actions (T. Aslam, et al., 2010).

This is a language based on the aspect-oriented language AspectJ, adding some distinctive features that are imperative for MATLAB programmers, such as: the ability to capture multidimensional array accesses and loops on MATLAB; and to exposure the join point shadow information and combine it with the action over that join point, in other words, use the join point information to recreate the action (T. Aslam, 2010).

The flow of the AspectMATLAB's weaver, the *amc*, is schematized in Figure 5 (T. Aslam, et al., 2010). The compiler was based on the *abc* compiler (abc Website, 2011) for AspectJ, and was built as an extension for the Natlab front-end (M. Website, 2011), extending its rules to include aspects as a program entity. The aspect file, during the compilation, is converted into a class and the action into the corresponding class methods.

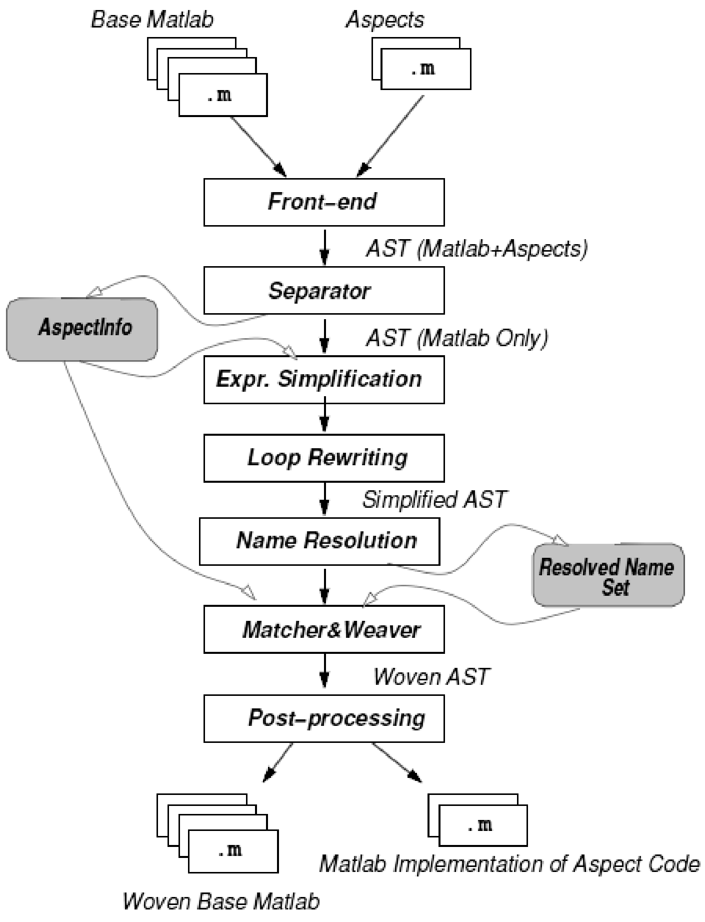

Figure 5: Weaving flow of AspectMATLAB, extracted from (T. Aslam, et al., 2010)

The aspects compilation, combined with the source code, always returns source code. In other words, AspectMATLAB files combined with MATLAB files are translated to MATLAB source files. These files contain the same source code of the MATLAB files, altered with the aspects defined on the AspectMATLAB file. The generated code can run on any MATLAB system. (T. Aslam, et al., 2010). An important matter of this aspect language is that the weaving needs to resolve the usage of a function call and an array access[4]. This is a difficult task for the reason that the capture of a function call can be tangled with the capture of an array access. This is caused by the language syntax, since that, per example, to make a function call with an argument can be similar to access a position of an array.

[4] An array access in a MATLAB system is identical to a function call: 'a(1)' in MATLAB could be an array access or a function call, even if 'a' is an input argument.

2.4.1. **Patterns and Actions**

AspectMATLAB provides some of the regular patterns (pointcuts) like *call*, *execution* and *within* and adds two new types of patterns: *get* and *set* patterns for array accesses and *loop* pattern to capture loops. The supported pointcuts on AspectMATLAB are explained in Table 6 (T. Aslam, et al., 2010).

Table 6: Existent patterns in AspectMATLAB

Pattern	***Target***
`call`	calls to functions/scripts
`execution`	Execution of function bodies
`get`	Array access
`set`	Array set
`loop`	Execution of loops
`loophead`	Loop's header
`loopbody`	Loop's body
`within`	Restricts the matching scope

The *call* and *execution* patterns are similar to those existing in AspectJ. To capture the access to arrays, AspectMATLAB includes the *get* and *set* patterns. These two patterns are very important for scientist because is commonly used by them for "its convenient and high-level syntax for arrays" (T. Aslam, 2010). The *loop* patterns are also very important as they are "key control system" for those programs (T. Aslam, et al., 2010).

The following code snippet shows an aspect with pattern examples. The first pattern gets all calls to functions that start with *get* and has no arguments. The second pattern captures all executions of functions that start with *set*. The next captures all function calls, and the last pattern captures all calls to functions with at least 2 arguments.

```
aspect PointerAspects

patterns
    getters: call(get*());
    setters: execution(set*(*));
    anyCall: call(*);
    atLeastTwoArguments: call(*(*,..));
end
...
```

The action in AspectMATLAB is identical to the advice in AspectJ and so the action can be applied before, after or around the join point. The difference to AspectJ is that an action contains a name. It is possible to apply many actions over the same join point. For that, this language contains default rules to indicate the action application order. The action that is applied around the join point takes precedence, and when there are multiple around actions, they are ordered in the same ordered as in the aspect source code. The before actions are applied after the around actions, and if multiple before actions occur is applied the same rule as before. Lastly, the after actions are applied and is applied the same rule as before if multiple before actions exist.

The final code snippet below shows the definition of actions that is intended to apply over MATLAB code. The first action inserts a print before any call to a get function. The second action gives information around a set's execution. In both actions is possible to see that they have arguments after the defining what pointcut to capture. Those arguments are the context exposure on AspectMATLAB, the static program context. In this example the first argument for both the actions gives the method's name and the second argument on *changingPosition* gives the method's arguments values.

```
...
actions
    printBeforeGets: before getters : (name)
       disp(['Using the get ', name]);
     end

     changingPosition: around setters : (name, args)
       disp(['Changing position  using', name,'with arguments', args]);
       proceed();
       disp(['Field changed']);
     end
end

end
```

2.5. Summary

AO is becoming a more mature paradigm with many AOP languages that can help programmers to modularize and simplify their programs with cleaner, (possibly) simpler and better structured code. The usage of AO paradigm can only bring advantages to its user if the program can be well modularized and each module can be created within an aspect, remaining in the main code the primary concern. Therefore, each time the user wishes to activate or deactivate a module (aspect), he does not need to change anything within the main code; he just needs to remove/comment the unwanted aspects.

The studied aspect languages and its extensions provides a vast list of advantages to their users, since this languages were made trying to resolve the programmers problems of modularization and repeated code that cannot be separated easily from the main program without an AO approach. Each language has its own focus; therefore they are limited to its selected language. Table 7 depicts some important properties of the two principal aspect languages studied (I. Aracic et al., 2006; T. Aslam, 2010; W. Cazzola, 2011a; B. Griswold, et al., 1998; B. Harbulot & J. R. Gurd, 2006; S. S. Huang, et al., 2008; O. Spinczyk, et al., 2002). None of the languages can define variable types and cannot be used over other languages. In other words, if there is an (extensive) aspect defined in AspectJ that can be very useful over other languages (for example, a tracing aspect showing class members accesses and executions), that aspect cannot be immediately used over the other languages. Instead, it has to be "translated" manually to the intended aspect language. Also, the reusability in these languages is basically the usage of abstraction and wildcards.

Table 7: Main characteristics of the AO Languages

Properties						
Aspect Language	AspectJ	AspectC++	AspectMATLAB	LoopsAJ	MAJ	@AJ
Class Member Introduction	Yes	Yes	N.A. [5]	Yes	Yes	Yes
Loops, arrays and Conditions	N.A.	N.A.	Loops and Arrays	Loops	N.A.	Yes
Reusability	Abstr. Wildcards	Virtual-Pointcuts Wildcards	Wildcards	Abstr. Wildcards	Wildcards Templates	Wildcards
Focused On	General-purpose	Tracing	Tracing	Loop-Tracing	General-purpose	General-purpose
Type Definition	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
Reusability with other languages	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
Target Languages	Java	C++ C (limited)	MATLAB	Java	Java	Java
Wildcards	'*' & '..'	'%' & '…'	'*' & '..'	'*' & '..'	'*' & '..'	'*' & '..'
Annotation Tags	N.A.	N.A.	N.A.	N.A.	Yes	Yes

This next table contains the join points that each studied programming language have and the ones that lacks (I. Aracic, et al., 2006; T. Aslam, 2010; W. Cazzola, 2011a; B. Griswold, et al., 1998; B. Harbulot & J. R. Gurd, 2006; S. S. Huang, et al., 2008; O. Spinczyk, et al., 2002).

[5] N.A. – acronym for **N**ot **A**vailable

Table 8: Join points supported by the AO Languages

Join point		*AspectJ*	*AspectC++*	*AspectMATLAB*	*LoopsAJ*	*MAJ*	*@AJ*
Class/ Structure	method	X	X		X	X	X
	field.get	X	X		X	X	X
	field.set	X	X		X	X	X
	constructor	X	X		X	X	X
Function	arguments	X	X	X	X	X	X
	return	X	X	X	X	X	X
	call	X	X	X	X	X	X
	reception-call	X			X	X	X
	execution	X	X	X	X	X	X
	body			X			X
Exception	handler	X	X		X	X	X
	throws	X	X		X	X	X
Variables	read			X			X
	write			X			X
Assignment	assignee			X			X
	assigned			X			X
If	condition			X			X
	body			X			X
	else body			X			X
Loops	arguments/ condition			X	X		X
	body			X	X		X
Arrays	access			X			X
	write			X			X

Table 8 shows the supported join points by the studied aspect languages. AspectJ and AspectC++ focus essentially on applying aspects over functions and classes, and do not care about local variables, statements, loops and conditional constructs. This is because the targeted programming languages are OO languages and so the aspect languages focus the interest in the classes that represent those objects and their parameters (B. Griswold, et al., 1998) (O. Spinczyk, et al., 2002). LoopsAJ extends the join points of AspectJ with the capture of loops, but still does not capture variables, statements and conditions (B. Harbulot & J. R. Gurd, 2006).

These aspects languages do not give attention to variables and other join points, internal to a function, and those join points could be an important matter to observe, change, or even optimize. For instance, having a program that is intended to observe what happens if every condition becomes true or the behavior of a certain loop or a variable's precision behavior, these aspect languages cannot create such advices since the join points for those aspects do not exist.

As for AspectMATLAB, this aspect language captures the join points that the other aspect languages do not capture [ASL10]. This aspect language does not have the same complexity of the other aspect languages. This is because AspectMATLAB tries to obtain the same type of usage as MATLAB, i.e., to be used by scientific programmers. And focus its attention to arrays access and loops for being important matters to the MATLAB's scientific programmer "community" (T. Aslam, et al., 2010).

@AspectJ can capture all of those join points if it is created an annotation on those places. A great advantage of flexibility on join point capture, with more freedom to choose which join points are selected and identify directly where to use it (W. Cazzola, 2011a). The main disadvantage resides on the reusability of aspects on other programs, for the reason that other possible target codes must have the same tags as the code that it was initially required to aspect.

3. Meta-Language and Framework for Aspect Oriented Programming

We present now the meta-language for AOP developed in the context of this thesis. This meta-language is extended with a set of files that includes: the join points referent to a specific programming language; attributes that can be used for each join point; models of action that can be applied over the pointcut. Having one set of files for each programming language, the weaver can combine the meta-aspects with the join points to obtain the specific aspect representation for the input program, validate the use of a join point attribute, and define the complete structure needed for the action to be applied.

3.1. Introduction to the Meta-Aspect Language and Framework

The meta-aspect language is an abstract front-end language to define abstract aspects that can be exported to a different tool that recreates the real aspect. The meta-language is transformed in an aspect representation, named Aspect-IR, for that target language, according to their requirements and delimitations. The meta-language, with an example given in Figure 6, allows the creation of meta-aspects with the keyword *aspectdef*. The aspect depicts a simple, yet helpful, aspect for function call debug purpose, where *select* pointcut expression identifies all the function calls on all functions of an application of input file. The *apply* represents the insert of a *print,* before the pointcut expression, that notifies which function was called.

```
aspectdef monitoringFuntionCalls                           Meta-Aspect code
   allFunctionCalls: select function{*}.call{*} end
   apply to allFunctionCalls
        insert before %{printf("call to <$function.name>\n");}%;
   end
end
```

Figure 6: Example of a meta-aspect concerning all function calls.

The first observable keyword is *aspectdef*, the definition of the aspect, and its descriptive name. In the definition of an aspect, declared using the *aspectdef* reserved keyword, the user can select the join points to capture and the actions to take place on the join point. Each meta-aspect file can contain one or more aspect definitions. The pointcut is defined with a *select* expression, being able to create a pointcut expression that symbolizes the point of interest within the code. This pointcut expression is common for every language if the join point model supports the join point chain in the select. To map the actions to the concerning join points, the meta-language has the *apply to* statement, which allows the mapping of one or more *select* expression to a set of actions to be applied over those pointcuts. To advice the actions, the user should name the selected pointcuts, and then describe each action to be applied on it.

Having as input the code in Figure 7 (a) which represents a MATLAB code, and (b), which represents a C code, the weaving using the aspect in Figure 6 results in the codes of Figure 7 (c) and (b), respectively.

```
function a = f1(b,c)

b = b+1;
if(b < 10)
      a= f1(b,c);
else
      a = f3(a,c);
end
a = a/f2(c);
```

(a) MATLAB code

```
void f1(int a){

      int count = 0;
      while( count < 10)
            a = f2(a);
      a = f3(count,a);
}
```

(b) C code

```
function a = f1(b,c)

b = b+1;
if(b < 10)
      printf("call to f1\n");
      a= f1(b,c);
else
      printf("call to f3\n");
      a = f3(a,c);
end
printf("call to f2\n");
a = a/f2(c);
```

(c) MATLAB code after weaving

```
void f1(int a){

      int count = 0;

      while( count < 10){
            printf("call to f2\n");
            a = f2(a);
      }

      printf("call to f3\n");
      a = f3(count,a);
}
```

(d) C code after weaving

Figure 7: Weaving the aspect in Figure 6 with two different programming languages.

The objective is to obtain in the end the input code transformed with the aspects. This meta-language aims at applying actions over such as loops, variables and conditions. The main concern of this meta-language is for monitoring and tracing code over different programming languages, with aspects that can be reused between them. Defining or changing types of variables is another feature of this language. An important topic when one needs to try different specializations for variables. By testing different implementations, the programmers can study the different program behaviors and choose an acceptable precision for the program variables.

Additional information is necessary to validate, organize and add more details to the Aspect-IR. The application of an aspect, as the one depicted in Figure 6, will depend always on the concerning programming language and its additional information.

As explained before, the Aspect definition can have *select*, *apply* or *condition* statements. These statements are related. For instance, an *apply* statement only has meaning if a *select* exist and this *apply* invokes that *select*. Also, a *condition* statement needs to be associated to an *apply*.

Figure 7, provides a simple flow to illustrate how the meta-language is used to deal with two different programming languages with the same meta-aspect file. The front-end parses the aspects and connects those aspects with the programming language input files and passes the information formatted in the structure required for each weaver, one for each language. The weaving process depends on the concerning language and usually, depending on the back-end, returns the resulting code in the same language.

The front-end creates the necessary preparations for the transition to the weaver for the targeting programming language. That preparation will depend on the requirements of the programming language. This thesis focuses on the design and development of the meta-aspect oriented programming language, and the construction of a weaver was never envisioned. The language can link the Aspect-IR with an existing weaver for the concerning programming language. Therefore, this language is considered a meta-language since it is a language that recreates the information to be passed onto another programming language, but in this case transformed into information for a set of weavers.

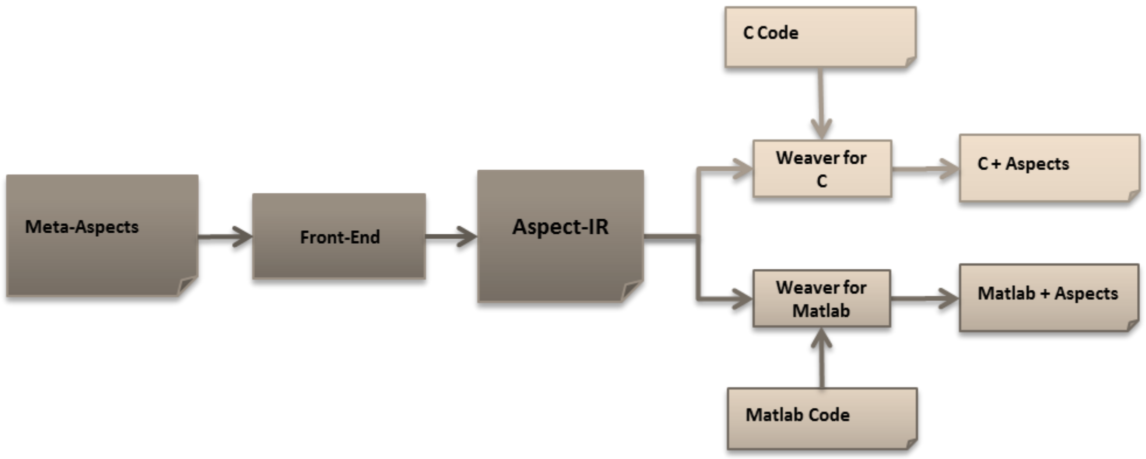

Figure 8: An aspect for two different languages is passed onto different weavers

3.2. Aspect-IR

The definition of an aspect in the meta-language will result in an aspect intermediate representation (Aspect-IR). That Aspect-IR contains the information depicted in the meta-language structured with the information needed to give to the weaver. Figure 9 depicts an Aspect-IR for the example in Figure 6, which contains the join point model referent to the programming language C. The Aspect-IR is structured as an AST, where each aspect contains the pointcut expressions and the applies. The pointcut expression in Figure 9 refers to every function call, with the complete hierarchy to the wanted join point. The apply statement contains an insert action where it is depicted the location of the file containing the code to be inserted.

```
aspects
  \_ aspect {name="monitoringFuntionCalls"}
    |\_ pointcut {name="allFunctionCalls"}
    |   \_ file {name=".*"}
    |     \_ function { name=".*"}
    |       \_ body
    |         \_ call { name=".*"}
    \_ apply { to="allFunctionCalls"}
      \_ actions
        \_ insert { code="monitoringFuntionCalls1_applyAllCalls_0.txt" position="before"}
          \_ parameter { target="function" attribute ="name" replace="<$function.name>"}
```

Figure 9: Resulting Aspect-IR of a meta-aspect.

3.3. Join Point Models and Attributes

The most critical element of an aspect-oriented language mechanism is the join point model, the model that represents the programming language's points of interest, like method execution, field gets, etc. (G. Kiczales, et al.).

"This model provides the common frame of reference that makes it possible for execution of a program's aspect and non-aspect code to be properly coordinated." (G. Kiczales, et al.)

The join point model can be different from programming language to programming language, and also depends on the capability of the weaver to support all the join points inside the chain. Having so, the join point model cannot be the same for all the concerning programming languages and each one should have its join point model that illustrates those supported points of interest. To obtain a more flexible meta-language, the front-end supports external file that represents the join point model of a certain programming language. This approach, illustrated in Figure 10, proves an important feature of the meta-language that represents its flexibility and expandability. The meta-language, left on the image, is transformed into the Aspect-IR with the assistance of the join point model and the join point attributes. This method allows the user to have a join point model more or less detailed for each language, with care of the weaver capability. Hence, the created pointcut expression is validated by the join point model.

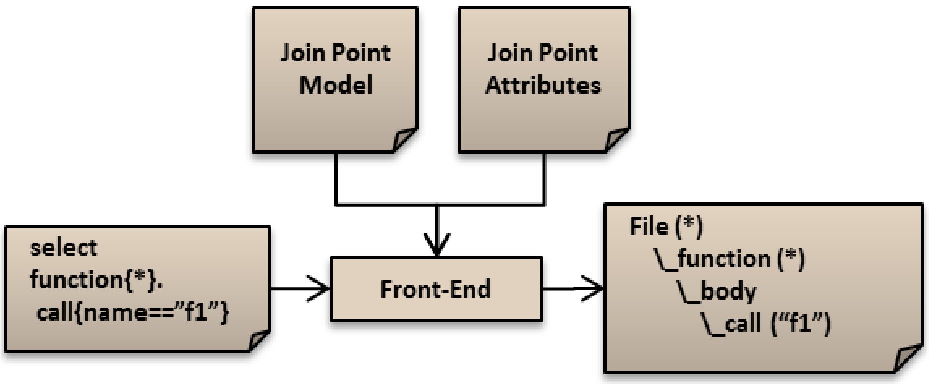

Figure 10: Select of function ("f1") calls to function and resulting hierarchy.

Also, a select statement can use join point attributes for a more accurate selection of the join points. This attributes can filter all the possible captures of a join points to only the join points with the desired attributes. They can be used to show to the user the current value of that attribute or to condition the use of a certain action on the aspect. With this attributes is possible to obtain information of each join point, and that information can influence the usage of actions over the program. For example, the join point *var* has properties like the name of the variable, the number of readings and writings, the value, etc. Another example is the information of a certain loop that we can obtain. With the number of cycles it is possible to choose whether the aspect should do loop unrolling or just check the properties of that loop.

A case study used on this thesis was the C programming language, and so its join point hierarchy was needed to validate the pointcut expressions. Figure 11 depicts an example of a join point hierarchy which contains a part of the content within a body pertaining to a function. The complete join point model can be seen in Appendix A.

```
file
 |\_var
 |\_declaration
  \_function
       |\_prototype
        \_body
             |\_var
             |\_call
             |\_if
             |   |\_condition
             |   |\_then
             |    \_else
              \_loop
                    |\_init
                    |\_condition
                    |\_counter
                    |\_body
                     \_control
```

Figure 11: Slice of C join point model that displays some join point hierarchy.

The join point model for MATLAB was also defined to be used as another case of study, and can be found in Appendix B.

3.4. Select a pointcut expression

The pointcuts defined on the meta-language depict the points of interest in the code, and can only be captured if the programming language contains the join points used on the pointcut. If the programming language contains those join points, they need to be specified on the join point model specification file (JSF). Otherwise the aspect cannot advice the code. The selection of a point of interest within a program is defined by a *select* statement. The *select* statement contains a unique identifier to be used by *apply* statements, and a pointcut expression to define the join points to be captured. The pointcut expression contains the information relevant for the capture of a specific group of join points. It is expressed by the declaration of join points combined with logical expressions and attribute limitation. The following example depicts a selection of all variables considering all functions and all files of a program.

```
allVars: select file{*}.function{*}.body.var{*} end
```

A single pointcut expression consists on a join point chain expressed by the hierarchy of the join point model, which states the route that the weaver should take to select the concerning join point. In other words, the join point chain is the exact scope to be searched in the weaving. To capture a join point within a scope, one has to define the pointcut by starting from the initial join point, usually the file join point, and adds the intermediate join points to be crossed, such as function and the body of the function, closing the chain with the said join point.

The user can then select the points in the code that are normally associated to loops and variables. Moreover, there is also possibility to select a specific point in the code by defining labels (also called pragma). Those labels can be selected the same way as the other type of join points and the labels can be filter by the name given to the defined label. Figure 12 depicts the label use in the code. The code contains a specific point that could not be selected with the usual join points, and so a label was used to define that point as a selectable join point. The aspect below the code represents the *selects* that captures those specific join points. This example uses three labels that could be used for programs that contain pre and post conditions in a specific point on the code. For the example, the conditions for the function are appointed as secondary CCC, leaving that CCC to be dealt by the aspects.

```
function a = f1(b,c)

a = 0;

%@preCondition

b = b+1;
a= f1(b,c);

%@middleCondition

a = f3(a,c);

%@postCondition

a = a/2;
```

```
firstCondition: select pragma{name==”preCondition”} end

middleCondition: select pragma{name==”middleCondition”} end

finalCondition: select pragma{name==”postCondition”} end
```

Figure 12: Code example with labels and the select statements for those labels.

More than one pointcut expression can be defined on the same selection by using logical expressions. The selection can be more detailed with equality (or inequality) expressions for the join point attributes. Those attributes contains the information within the join point, referring to relevant data such as name, value and type for a join point for a variable. The two examples of *select* statements below are equivalent and specify the variables *a* of type *float* in the functions *f1* and *f2*. Both *selects* argument the same pointcut expression.

```
A: select function{name==”f1”,name==”f2”}.var{(name==”a”, type==”float”)} end
B: select function{name==”f1”}.var{ (name==”a”, type==”float”) }
        || function{name==”f2”}.var{ (name==”a”, type==”float”) } end
```

The equalities between the curly brackets filter the join point search to only the attributes that match on those equalities. Those are examples of the use of joint point attributes. One can create pointcut expressions with the best option for him and use the attributes of each type of join point to refine the search.

The pointcut expression in a *select* statement can be used as a powerful joint point selection for the various join point/attributes combinations which the front-end accepts. The logical expressions can be defined in a way. For instance, the selection of a variable of type *float* and another of type *int*, or the selection of function calls that returns no value and contains zero or one argument, can be done as in the following code slice.

```
floatsAndInts:
  select function{*}.var{type==”float”, type=”int”}
end

funcCalls:
  select function{*}.call{(return_type==”void”, num_argins==0),

                          (return_type==”void”, num_argins==1)}

end
```

An importing notice is the use of round brackets and the comma between attributes. The brackets stand for the logical expression AND, which allows the combination of two different attributes to filter the selection to the join points that matches with all the attribute values defined in the selection. For instance, if a join point contains all matching attributes with the exception of one attribute, that join point is not selected, as the perfect match is not attained. The comma is used as an OR logical expression, allowing the selection of join points with a set of attributes that do not need to match in the same join points, i.e. each attribute is treated separately from the others. This allows the selection of join points that contains a certain attribute OR other attribute. The AND expression also contains commas separating the attribute equalities. However these commas are not considered OR expressions, but as AND expressions between the attribute equalities. In other words, the brackets change the OR expression mean on the commas to an AND expression. The code below shows equivalent select statements using explicit union and intersection operators:

```
floatsAndInts:
   select function{*}.var{type==”float”} ||
               function{*}.var{type==”int”}
end

funcCalls:
   select function{*}.call{(return_type==”void”, num_argins==0)} ||

               function{*}.call{(return_type==”void”, num_argins==1)}

end
```

Hence, these AND and OR operators are used as intersection and union, respectively. Some more options of combining and refining the selections are depicted in Table 9, which contains the expressions and their join point target. The possible combinations only depend on the available join points and their attributes. The meta-language allows those combinations to be defined in various ways.

Table 9: Examples of expressions for *select* statements

Expression	***Target***
`function{*}.var{*}`	Every variable in every function
`function{*}.var{name==”a”}`	Variable *a* within any function
`var{name==”a”, name==”b”}`	Variable *a* OR *b* (both are selected)
`var{(name==”a”, type==”float”)}`	Variable *a* of type *float*
`var{(name==”a”, type==”float”), name==”b”}`	Variable *a* of type *float* OR variable *b*
`var{(name==”a”, type==”float”)} || var {name==”b”}`	Same as above
`var{type!=”int”, (name==”count”, type==”int”)}`	Every variable that is not of type *int* OR variable *count* of type *int*
`loop{type==”for”}.var{name==”i”}`	Variable *i* used within a *for* statement
`if.then.loop{type!=”for”}`	Loops that are not of type *for*, which are inside a *then* body of an *if* statement

The pointcut expression is not confined to the capture of the join point in a linear way, i.e., the selection of a join point is made not only within scope of the previous join point, but also in the possible successive scopes within that join point. Instead, the research is done in depth in the program AST, allowing a set of all possible combinations of join points that can be found, including the join points that may be within the innermost scopes.

The schematic below in Figure 13 depicts an example of a program that contains 4 nested levels and two selections created in the meta-language. The *select* in line 10 aims to capture combinations for a single *loop* and the *select* in line 11 aims the capture of the nested *loops* within another *loop*. In the first *select* (line 1), the set of returned loop join points contains every loop within the code slice, as the selection is made not only in the first level of the previous scope of the loop join point (the body join point) but in every possible inner scopes within the parent scope of loop. So, if the AST of the program contains *if, scope, loop* and other types of statements that contains a scope within them will also be searched. The second *select* (line 2) is for every loop that is inside another loop, and so the only ones that are not selected are the loops in the first nested level, i.e., the loops that are not inside another loop.

```
A. for(int i=0; i < sizeX; i++)                                    Code Slice
B.   for(int j=0; j < sizeY; j++)
C.      for(int k=0; k < sizeZ; k++)
D.         for(int l=0; l < sizeW; l++)
               ...
E. for(int i=0; i < sizeX; i++)
F.   for(int j=0; j < sizeY; j++)
        ...
G. if(sizeZ < Z_OVERFLOW)
H.   for(int i=0; i < sizeX; i++)
I.      for(int j=0; j < sizeY; j++)
           ...
```

```
1. allLoops: select loop{*} end                                    Meta-Aspect
2. loops2Nested: select loop{*}.loop{*} end
3. loops1NestedTop: select loop{nested_level==1} end
4. loops1NestedTop: select loop{nested_level!=0}.loop{*} end
```

Figure 13: Code slice with nested loops and a meta-language selecting those loops

The join point for *loop* contains an attribute, the *nested_level*, which can be used to delimitate that selection to a threshold for the nested level. For instance, to capture a *loop* of one nested level the user can define the *select* as the meta-aspect code slice depicts in Figure 13, line 3, restraining the *loop* to level one of nesting with the *nested_level* attribute. To capture every *loop* in another *loop* with nested level superior to zero the user can condition the first loop not to be a level zero *loop*, similar to the *select* in Figure 13, line 4. Table 10 represents the pointcut expressions of the *select* statements above in Figure 13(lines 1 to 4) and how the exploration is done to find those pointcut expressions and which join points are selected.

Table 10: The selects depicted in Figure 13 and respective loop[6] selection

Expression	*Chain Selection*	*Loop Join Point Target*
`allLoops:` `select loop{*}` `end`	`A,B,C,D,E,F,H,I`	`A,B,C,D,E,F,H,I`
`loops2Nested:` `select loop{*}.loop{*}` `end`	`A->B, A->C, A->D` `B->C, B->D` `C->D` `E->F` `G->H->I`	`B,C,D,F,I`
`loops1NestedTop:` `select loop{nested_level==1}` `end`	`A->B` `E->F` `G->H->I`	`B,F,I`
`loops1NestedTop:` `select loop{nested_level!=0}` `.loop{*}` `end`	`A->B->C, A->B->D` `A->C->D`	`C,D`

After its definition, a join point can be referred as a variable. In the meta-language such variable is defined with a dollar (*$*) followed by the join point name, such as *$function*. When referring to a specific attribute of a join point reference, the variable is followed by the attribute name as *$function.name*. In this case we are referring to the name of each function in the pointcut. This syntax was chosen to be similar to a field access in Java and other programming languages and so the meta-language user can be more familiar with the attribute usage.

3.5. Advising an action to a pointcut

After selecting the desired join points, the *apply* statements take place. The *apply* statement as the same meaning as the advice in AspectJ. Thus its purpose is to define which actions should take place on a certain pointcut expression, i.e., a select statement. These statements contain a set of actions that will affect the selections. There are several possible actions that an *apply* can advise. The most common action is the *insert*, where one can insert a portion of code before, after or around the join point. The second action available is the *optimize* action that allows some code optimizations by unrolling loops or inlining a call to a function. The last action is *define*, where one can define the values of a join point if those values are known by the user. The *define* action also allows the definition of types. The same *apply to* statement can be used for more than one *select*. Hence, the *apply to* statement will associate their actions with all the listed *select*.

[6] Each loop is illustrated with a letter

A default action available is the direct insertion of a slice of code directly into the source code. It is a basic slice of code with the option to integrate values of join point attributes. This is an option, e.g., for monitoring and tracing purpose. This type of action is a normal feature available in all AOP languages that enables the modularization of a programming, retaining the secondary functionalities within an aspect that will insert them in the correct positions. This insertion can be done *before*, *after* or *around* the occurrence of the join point. However, the *around* occurrence will substitute the join point with the inserting code. The following example represents a simple insertion of code before every function call. Note that the code to be inserted uses a parameter (*$function.name*) that will be defined by the weaver for each join point.

```
allCalls: select function{*}.call{*} end
apply to allCalls insert before %{printf("Call to <$function.name>"}%; end
```

The code to be inserted is placed between brackets after the *insert* command, or in a separated structure called *codedef* (explained on 3.7). This feature enables the common usage of aspects-oriented programming: insertion of secondary code into the primary source code. The front-end does not take actions over the code by itself and therefore the insertion action is passed to the language's associated weaver. The front-end informs the weaver the use of join point attributes that it should replace those join point attribute usage with the attributes corresponding value.

Another possible action is *optimize*. This feature allows actions over the source code like *unrolling* a loop a settled number of times, the *inlining* of a certain function, among other possibilities. Its primary concern is, as the name suggest, optimize programs by changing the source code partially to avoid low-level jumps and conditions. The code bellows shows two simple examples. The first example selects every *loop* of type *for* and optimizes the code by unrolling those loops with a factor of ten, i.e., the *loop* should unroll ten iterations. For the second example, the selection is done on every function call to function "f1*"*, and the apply replaces those calls with the body of the function which was called.

```
    forLoops: select function{*}.loop{type=="for"} end
    apply to forLoops optimize loop_unroll(k=10); end

    f1Call: select function{*}.call{name=="f1"} end
    apply to f1Call optimize inline; end
```

The *loop unrolling* action enables the unroll of the selected loops the number of times the user desires. This helps the compilation by reducing the number of conditional verifications. Also, the inline option inserts the body of the function to inline on the location of the function's call, and so there will be not the need of jumping statements to the called function.

Also on the action model is the *define join point attributes* action. With this action the user can define the values of join point attributes, information that the front-end cannot know by itself. However, if the information can be obtained by the user or it is of his/her knowledge, that information can be transmitted to the front-end. Thus, the front-end can do more than just pass the Aspect-IR to the weaver and process contexts such as condition evaluation and automatic parameters substitution on a code definition. The example below depicts a definition use that adds the type of the selected variable and how many times the variable was read on the code. With this definition, every time the attribute *type* of variable "a" is requested, the front-end would substitute this reference by the proper (known) value.

Passing the attributes information to the front-end would reduce the work and effort of the succeeding weaver and augment its aspect productivity and, therefore give more power of use to the meta-language. The weaver would then receive the Aspect-IR with the final value, and would not need to find that reference every time that is requested, becoming only responsible for the main objectives that are requested. The definitions that are declared within an *aspectdef* are also spread to every other *aspectdef* declared on the same meta-aspect file. This spread allows a better way to structure the aspects, in which the definitions could be defined on an *aspectdef*, separated from the other types of aspects.

```
A: select function{name=="main"}.var{name=="a"} end
apply to A define var(type="float", num_read=10,...); end
```

This feature also allows the definition of type of variables. For each variable type definition within the aspect file, the front-end will generate within the Aspect-IR the request of type change/definition. The example depicted in the following code snippet has a MATLAB code that it is requested to define a group of variables as *integers*, avoiding unnecessary *double* usage. The user can then create an aspect defining those variables as the desired types.

```
function max = Max(vals)                                    MATLAB function
max = vals(1);
count = 2;
while count < length(vals)
        if(vals(count) > max)
                max = vals(count);
        end
        count = count +1 ;
end
```

```
maxVar: select function{name==”max”}.var{name==”max”} end          Meta-Aspect
apply to maxVar define var(type=”int32”); end
countVar: select function{name==”max”}.var{name==”count”} end
apply to countVar define var(type=”uint8”); end
```

Figure 14: Definition of types for a MATLAB function.

With the created aspect the variable *count* will have a maximum of 255, a range that the user knows that can be used safely. The user knows that it is only working with integers, and so the *max* variable can be defined as an integer. This definition is an important feature for a MATLAB system as this programming language does not have variable type definition. Also, any of the known AOP languages does not support type change or definition, and this can be an important issue when trying to get the best precision/performance for a program. Since the definitions are spread along all the *aspectdefs*, those type definitions can be declared separately from the other concerns and thus allowing a definition list that can be easily found and modified if necessary.

For all the available actions, the user can use join point attributes information inside the actions. The inclusion of such information can benefit the usability and understanding of the desired action. For instance, creating an insert with only plain text is not as powerful as an insert containing join point attribute values.

3.6. Conditional Expressions

The conditional expression (*condition*), as the name says, conditions the *apply to* statement which can only be applied if the condition evaluates true. The *apply* statement always takes place on the selected pointcut expressions, and so all the applies defined are going to be present on the code, if the selects are encountered. However, sometimes the use of an *apply* makes only sense when some requirements are met. The *apply* could only be necessary if some important issue is within the code or if some information does not delimitate that use. For instance, when the user requires to:

- Apply a loop unroll only if it is of type “for” and has at least 5 iterations;

- Insert code before a function call, if it returns a float;
- Inline a function if it has more than 3 input arguments;
- …

To limit the *apply* with those requirements, the user can define condition statements, which have as targets a set of *apply* statements, and predicates them with logical expression. It is also possible to include join point attributes to condition the *applies* with that information. The examples above are declared as conditions in the meta-aspect below. The first apply only unrolls a loop if the condition evaluates true, i.e., if the loop is of type "for" and the number of its iterations is superior to five. The second condition only evaluates true if the function called returns a float value. The last condition only lets the apply to inline a function if the number of its input arguments is larger than 3.

```
optUnroll: apply to A optimize loop_unroll(k=5); end
condition for optUnroll: $loop.type=="for" && $loop.num_iterations >= 5 end
insertCode: apply to A
      insert %{printf("function <$function.name> returns a float")}% .before;
end
condition for insertCode: $call.return_type=="float" end
optInline: apply to A optimize inline; end
condition for optInline: $call.num_argin > 3 end
```

If the values of the join point attributes were already known, the front-end would validate the condition and in case of positive match, the *apply* would be passed without any condition and the weaver would only need to carry out the *apply*. On negative match, the *apply* would not be passed to the weaver and so it would not need to try to perform the *apply* as it was already known in the front-end that the triggering condition evaluates false.

3.7. Code definition

The *code definition* provides code to be used by the insert action. A code definition contains code that the user can reference from an aspect declaration, with the purpose of inserting that code in the source code file. That code definition is usually a slice of code that the user desires to insert on the concerning core code and is available for the set of aspects within the file, i.e., it is a reusable code slice. For each code definition a text file is created containing the defined code. That file can then be used by the weaver to insert the code whenever an apply statement uses the code definition. The code insertion has a particular property that enables the injection of join point attributes, such as function name and variable value or type, called parameters. Whenever the code contains parameters within it, the front-end retains those parameters to advise the weaver that it should replace them with the correct values. However, if the front-end contains that information, it will replace the parameters with the corresponding value and does not need to request the weaver to do the task.

The example below contains a code definition to be inserted in a C program. It is clearly a slice of code to print a message. In this case the *codedef* uses an attribute related to the function, the function name. This way the monitoring of a system related to their function calls is more accurate about which function is actually called, instead of just saying that a certain (unknown) function was called.

```
codedef code1
%{
	printf("Call to function <$function.name>\n");
}%
end
```

One can then use the join point attributes to create better and more complex code to insert, instead of inserting just plain text with minimum connection to the main code. The usage of an attribute of a certain join point is called a code parameter. That parameter is used by the form: *$<join_point>.<attribute>*. The code is saved on the file with the parameter, and the weaver is informed of the existence of parameters on that code. To advice the weaver of the replacement, as already depicted before, the Aspect-IR contains the list of parameters that needs to be replaced and the attribute that the user desires to be the replacement value. Evidently, the weaver can only replace the parameters to which are possible to find the values for the requested attributes. Thus, the information passed to the weaver of the code definition is the name of the file containing that code and the list of parameters used in the code.

3.8. Import

The *import* declaration enables the introduction of information of other aspect definitions declared externally to the aspect file. The possible imports are join point attributes definition and the Aspect-IR of the imported aspects. With it, the user can structure the aspects definitions just like he/she can do for a class division, where it is possible to divide the context on several classes and structure them according to the problem to be solved.

As shown in the example below, the import feature allows the external definition of aspects and the normal use of its information. For this purpose, the final resulting output files, beyond the XML containing the Aspect-IR for Harmonic, the front-end creates two additional files: the defines XML and another Aspect-IR XML, this last one different from the first for having a different construction structure, being exactly identical to the Aspect-IR used on the front-end for import facilitation.

```
aspectdef defineVariableA                                    defineVariableA.lara
   A: select function{name=="f1"}.var{name=="a"} end
   apply to A
                define var(type="float");
                define function(latency=120,resources=1000);
   end
end
```

```
import defineVariableA.defines;                             Meta-Aspect with import
aspectdef varA_Aspect
   A: select function{name=="f1"}.var{name=="a"} end
   B: apply to A insert before %{printf("Variable <$var.name> is of type
<$var.type>");}%; end
   condition for B: $function.latency < 200 && $function.resources < 500 &&
$var.type=="float" end
end
```

3.9. Summary

This chapter described the AOP meta-language and its framework (also referred as front-end) proposed in this thesis. The aspect meta-language has been designed bearing in mind code simplicity and readability. The selection of a pointcut expression is done with a *select* statement. The apply statements specify the set of actions to be applied over the join points identified by that pointcut expression. Apply statements can be conditioned with a conditional expression. This allows actions to be executed based on certain criteria.

The framework is able to process the input aspects and to generate a representation of the aspects in a form named as Aspect-IR (aspect intermediate representation). The framework is also able to interpret condition expressions and to generate simplified Aspect-IR representations when the conditions are resolved at this stage.

4. Implementation

The aspect language was designed having code compactness, legibility and flexibility, as priority goals. The aspects in the meta-language are transformed in an intermediate representation to be processed by a weaver. To process this meta-language, the front-end receives a meta-aspect input file and, with help of some additional files, generates the intended Aspect-IR. This chapter describes the approach regarding both the implementation of the aspect meta-language and the front-end.

4.1. Meta-Aspect Language and the Aspect-IR/ XML

A front-end tool process the meta-aspect language to obtain the Aspect-IR. The grammar of the meta-aspect language has been specified using the JavaCC parser generator (J. C. C. Website, 2011). To obtain an annotated AST (abstract syntax tree) containing the representation of the input aspect file, we have used the JTB syntax tree builder[7] (J. Website, 2011).

Part of the grammar of the meta-language is depicted in Figure 15 (Appendix C includes the complete grammar). An aspect file can have three types of sections: the import declaration, the aspect definition, and the code definition. The import declaration allows the inclusion of external information to the aspect. The aspect definition specifies the join points to capture and the actions to perform when certain requirements are fulfilled. The code definition includes code to be injected onto source code.

[7] JTB takes a JavaCC grammar and generates a set of syntax tree classes, referent to the grammar, two visitors and an annotated JavaCC grammar.

```
Start ::= (Import)* ( AspectDef )* ( CodeDef )*

Import ::= <IMPORT> Identifier <DOT> Identifier <SEMICOLON>

AspectDef ::= <ASPECTDEF> Outputs Identifier Inputs Variables

                  ( Select | Apply | ConditionalExpression ) *

              <END>

CodeDef ::= <CODEDEF> Identifier <CODE> <END>
```

Figure 15: Part of the grammar for the aspect meta-language.

The front-end accepts the aspects specified within a meta-aspect file according to this grammar. Figure 16 depicts the processing flow of the front-end for the aspect meta-language, including its inputs and outputs. The front-end parses the aspect specification file and, by merging with the three input files (right to the front-end in Figure 16), creates an intermediate representation (Aspect IR) of it and finally, exports this Aspect-IR (as XML code) to the possible weavers.

The aspect specification file holds the declaration of transformations described by aspects the user desires to apply to the source code. The XML representation of the Aspect IR can be used as the communication vehicle between the front-end and the possible compilation stages (as in the case of the REFLECT design flow (J. M. P. Cardoso, et al., 2011)).

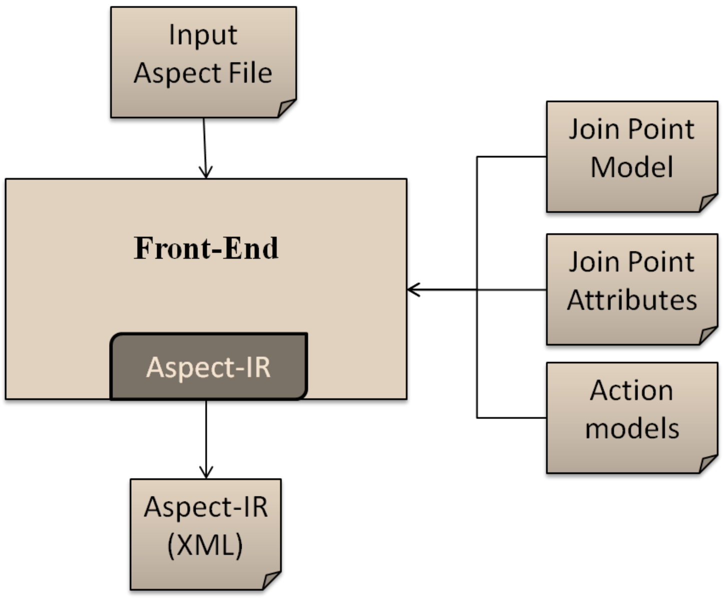

Figure 16: Aspect Meta-Language Front-End.

To support multiple programming languages, we consider three input files which represent important information for the aspect meta-language:

- the **join point model**, representing the points of interest in the addressed programming language;
- the **join point attributes**, such as variable name and value of the points of interest in the code;
- and the **action models**, describing each possible action an aspect can perform to the code, such as code insertions and specific optimizations (e.g., loop unrolling).

All these files (described in detail in Section 4.2) are input in XML format, as it allows storing, accessing and updating properties of interest outside the core of the program, which is useful as the complexity and the flexibility of the front-end rises. Also, XML is a standard representation format which is easy to parse and to change. With this framework, a compilation flow can take advantage of an unique aspect meta-language.

As the definition of the join points varies between programming languages, the JSF (Join point Specification File) of a programming language provides the weaver the knowledge of how and where to apply a certain advice in a join point. The Aspect-IR is then obtained by combining the aspects defined in the meta-language with the join point specification. That representation will depend not only on the programming language to be advised, but also on the compiling stage to influence.

4.2. Specification Input Files

The additional information consists in three XML files, each one with a specific type of information: the first contains the join point model referent to the language; the second one contains the attributes of each one of those join points; and the last file contains the possible actions the weaver can apply. The front-end will then associate the meta-language with that information, validating the defined aspects and completing the information that the weaver requires and it is not present on the meta-aspect code. That association will result in an aspect intermediate representation (Aspect-IR) which contains all the necessary information for the weaver for that language.

These external files allow more flexibility for the front-end and one can change them so they can be up-to-date. These files are vital to have a flexible and easy to use front-end. By creating a relation between the input aspect file and the XML input files, the front-end creates the Aspect-IR. This implementation allows the expandability of our aspect meta-language to be used for additional programming languages..

4.2.1. **Joint point model**

Every programming language needs one join point model that clearly characterizes the points of interest. The meta-language weaver needs to acquire the join points of each language. For that, the weaver receives a join point specification file (JSF), a file that represents the join points of a certain programming language and how they can be captured. One JSF should exist for each programming language that is intended to apply the aspects. In this work two JSF files were partially specified: one for MATLAB and another for C.

Figure 17 shows an excerpt of the joint point model (using an XML representation) currently used for the C programming language. In the excerpt in Figure 17, the join point loop has as its predecessor a body, which pertains to a function, followed by the file that they belong to. Hence, this file is used to verify and/or return the complete hierarchy to a certain join point so the weaver can be advised with the complete path to the desired join point.

```
...
<file type="file">
       <var type="var"/>
       <function type="function"/>
</file>
<function type="function">
       <prototype type="prototype"/>
       <body type="body"/>
</function>
<body type="section">
       <var type="var"/>
       <loop type="loop"/>
       <call type="call"/>
       <if type="if"/>
</body>
<loop type="loop">
       <init type="expr"/>
       <condition type="expr"/>
       <counter type="expr"/>
       <body type="body"/>
       <control type="var"/>
</loop>
...
```

Figure 17: Excerpt of the XML representation of the join point model hierarchy for the C programming language.

The user can be oblivious of the complete join point hierarchy, as the front-end auto-completes the faulting middle hierarchy with the help of the join point model specified in the JSF. The meta-language allows the use of a shorten version of the join point chain if the front-end can recreate the absent middle path, otherwise the join point chain is considered incorrect and the user should add some more information, i.e., add one or two middle join points. There are cases where the hierarchy can include two different paths, as illustrated in Figure 18. The front-end, as default, uses the first match of a possible path and if this choice does not return the desired group of join points, the user should refine the join point chain.

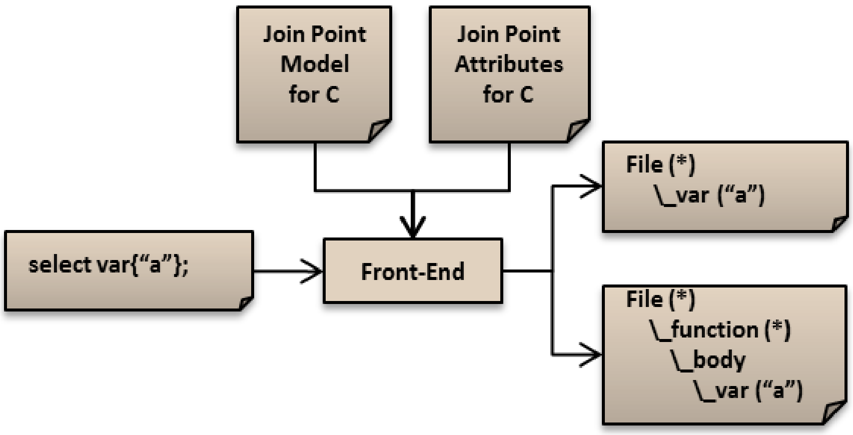
Join Point Model for C
Join Point Attributes for C
select var{"a"};
Front-End
File (*) _var ("a")
File (*) _function (*) _body _var ("a")

Figure 18: Example of two possible paths for a join point hierarchy.

4.2.2. Join Point attributes

Each join point has attributes with the information of the join point that can be obtained by the weavers, as depicted in Figure 19. For instance, the join point *var* contains some attributes that can be used by the programmer to see how many times the variable was read and also to limit the join point captures using the name of the variable. In addition, there are global attributes that are common to all join points, such as the name of the file, and the total number of array references and pointer references.

```
<artifacts> <!-- Attributes for each join point -->
       <artifact name="var">
       <attribute name="type"  type="String"/>
       <attribute name="name" type="String"/>
       <attribute name="value" type="Value"/>
       ...
</artifact>
       <artifact name="loop">
       <attribute name="num_iterations"  type="int"/>
       <attribute name="increment_value"  type="int"/>
       <attribute name="nested_level"  type="int"/>
       ...
</artifact>
...
</artifacts>
```

Figure 19: Artifact list, containing the attributes for each join point.

A select statement can use join point attributes for a more accurate selection of the join points. These attributes can filter all the possible captures of a pointcut to only the join points with the desired attributes. For a correct use of join point attributes, the select exploit the join point attribute XML file to observe the correct usage. The attributes that each join point contains are specified over that XML file. These are also important properties of the join point. They can be used to show to the user the current value of that attribute or to condition the use of a certain action on the aspect. With these attributes it is possible to obtain information of each join point, and that information can influence the applicability of actions over the program. For example, the join point *var* includes as attributes the name of the variable, the number of readings and writings, the value, etc. Another example is the information of a certain loop that we can obtain.

The front-end uses the XML files to verify the pointcut selection and return the complete join point hierarchy and also the existence of the join point attribute as well as the correct type when comparing to a certain value. The final result is a detailed pointcut expression with every attribute used approved.

This file with attributes can be specific to a programming language and/or to the weavers in the compilation flow.

4.2.3. **Action model**

The third file is related to the action model. This file contains the definitions for each type of possible action that can be used over a certain join point. For instance, to optimize code by using an optimize action such as loop unrolling, the tool should specify which stage and engine in the compiler flow is responsible to apply it. An action in AOP is the act of changing the program in the join point. That action is the aspect to be applied on that specific join point. The usual action in AOP is the code insertion before, after or around the join point. Normally, this is the predefined action in AOP languages.

For instance, the *define* action available for MATLAB programming language allows the definition of types of variables. As the weaver used in AMADEUS allows the definition of variables, this action is listed as a possible action to be used on MATLAB programs.

The XML example depicted in Figure 20 shows some of the possible actions and their properties to be passed onto the Aspect-IR when considering the REFLECT compilation flow (J. M. P. Cardoso, et al., 2011). The optimize action is processed by a CoSy[8] compiler instance (J. M. P. Cardoso, et al., 2011) and the front-end should notify this stage about the engine to use and about the parameters passed as arguments to that engine.

With an insert action, the REFLECT compilation flow can insert code into the source code by using Harmonic (W. Luk et al., 2009).

```
<actions>
    <optimize>
        <inline stage="cosy" engine="inliner"/>
        <loop_unroll stage="cosy" engine="loopunroll">
              <parameter name="k" type="int"/>
        <loop_unrool/>
        ...
    </optimize>
    <insert>
        <before stage="harmonic"/>
        <after stage="harmonic"/>
        <around stage="harmonic"/>
    </insert>
    ...
</actions>
```

Figure 20: Action models containing some available actions for C programming language.

[8] CoSy is a compiler framework of ACE Associated Compiler Experts by: http://wwww.ace.nl

The number of available actions can be expanded since those actions are listed on the action model XML, just like the join point model and the available attributes, along with their properties and possible parameters needed.

4.3. The Front-End Architecture

The front-end architecture follows the intermediate representation of the aspects, as its main objective focuses on the interpretation of the meta-language and the generation of the Aspect-IR, according to the target programming language.

The main portion of the front-end architecture is depicted in Figure 21. The front-end starts the program with the parser for the meta-language, which is depicted in Appendix C. The parser generates an annotated syntax tree of the input meta-aspect file which is then analyzed and merged with the programming language specification files as the Aspect-IR is created. After the Aspect-IR creation, the front-end does an interpretation of the Aspect-IR data and resolves the expressions which can be interpreted by the front-end. This interpreter resolves the conditions that can be resolved at this stage and rewrites the Aspect-IR according to the interpreter changes.

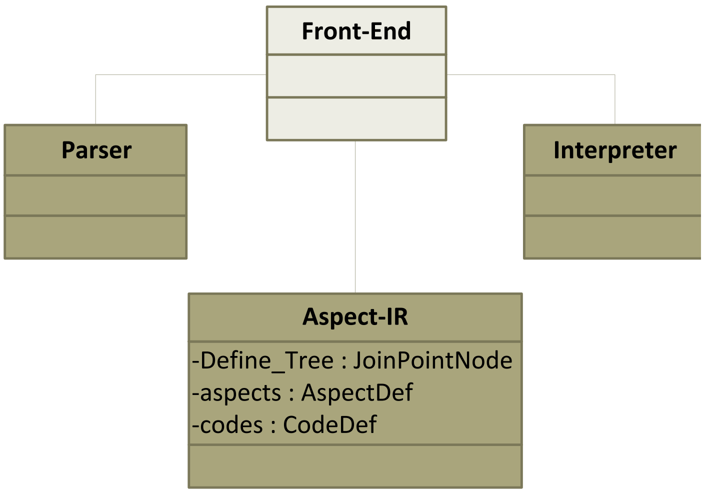

Figure 21: Main portion of the front-end architecture.

The Aspect-IR contains a similar structure to the annotated syntax tree created by the meta-language parser merged with the targeting programming language specifications. That structure is depicted in Figure 22, where each Aspect-IR contains the aspect definitions, the code definitions and the definitions and attributes used in a tree structure. This tree is used by the interpreter to solve the expressions which it can resolve. The code definition contains its name and the corresponding code slice. Also, it stores the parameters used in the code to notify the weaver. The aspect definition stores its name along with a set of selects, applies and conditions defined within the aspect.

In the apply, the Aspect-IR stores its name, which can be omitted if no condition is to be defined for it, the selects where the actions will advise and the list of actions for that apply statement. Moreover, if a condition occurs for this apply, the Aspect-IR will also store it within the Apply Class, where it can only be declared one condition. If multiple conditions exists for one apply, only first occurrence is considered. The actions for the applies are defined according to the action model, and so they include the type of action, such as insert and define, the descriptor of the action (e.g. unroll and inline) and the parameter required by the action, if any.

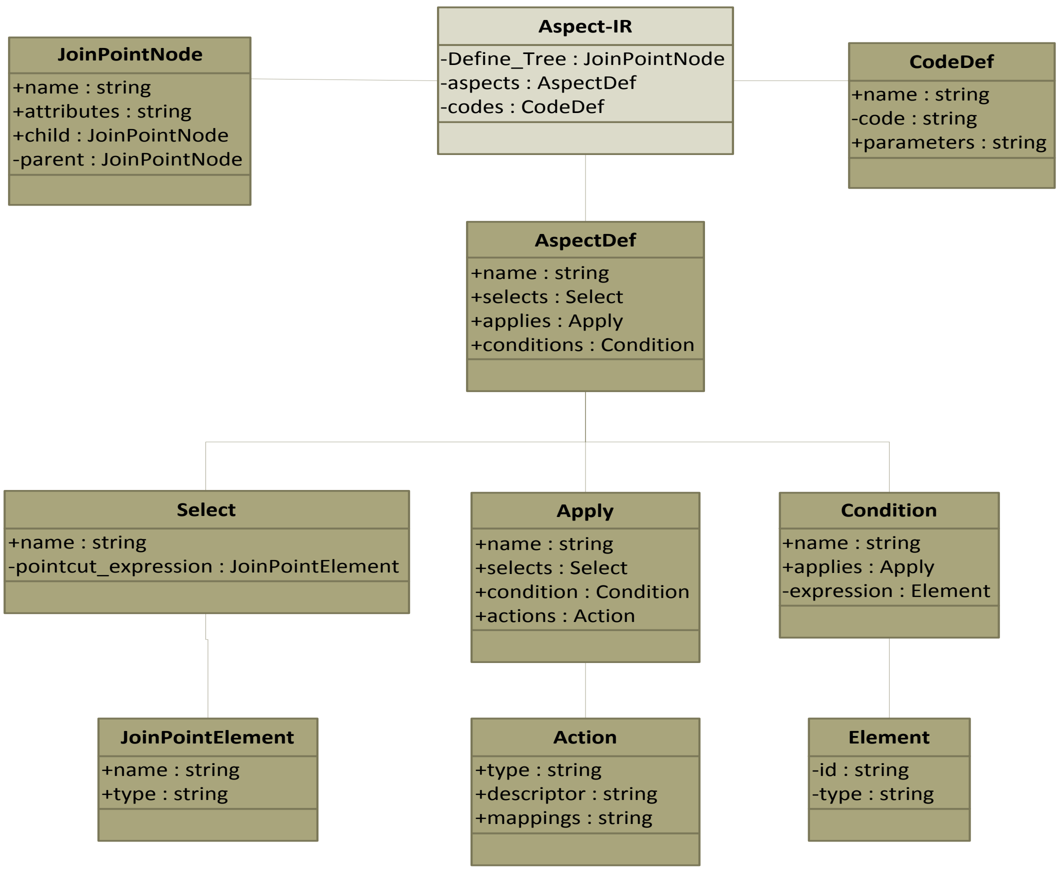

Figure 22: Aspect-IR Architecture.

Each select, schematized in Figure 23, has an unique identifier and the pointcut expression declared on the meta-language for that select. This pointcut expression can combine join points and logical expressions. A default type was defined, considered a JoinPointElement, which can be a logical operation or a join point with its attributes, simplifying the pointcut expression for an interpretation.

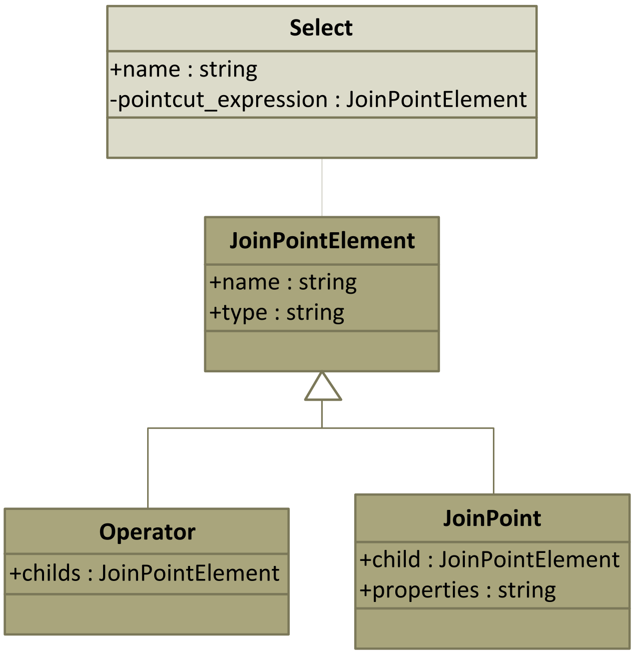

Figure 23: Select Class and its pointcut expression.

For instance, having a pointcut expression as the example in Figure 24 (a), which captures the variables “a” of type “float” within the functions “f1” and “f2”, its representation is depicted in Figure 24(b). As these types extend the JoinPointElement class, any one of them can be used as the pointcut expression for the select.

```
B: select function{name==”f1”}.var{ (name==”a”, type==”float”) }
        || function{name==”f2”}.var{ (name==”a”, type==”float”) } end
```

(a)

```
Operation: OR
    JoinPoint: file {name=".*"}
        JoinPoint: function {name="f1"}
            JoinPoint: body
                JoinPoint: var {name="a", type="float"}
    JoinPoint: file {name=".*"}
        JoinPoint: function {name="f2"}
            JoinPoint: body
            JoinPoint: var {name="a", type="float"}
```

(b)

Figure 24: Join point expression example and its Aspect-IR representation.

Figure 25 depicts the architecture for the condition statements, where it is possible to define an expression to condition an apply statement. Each condition may contain its identifier, although not required; contains a set of identifiers which represents the set of applies that will be conditioned, and an expression which can contain literal values, attribute references and logical expressions. Thus, similar to the pointcut expression, this expression is divided in three types of Elements: the Reference, Literal and Operation. The Reference class is used whenever a join point attribute is used within the expression. The Literal corresponds to constant values such as integer values and strings. The Operation symbolizes the usage of any type of operation, such as an equal, adding and bigger than operations.

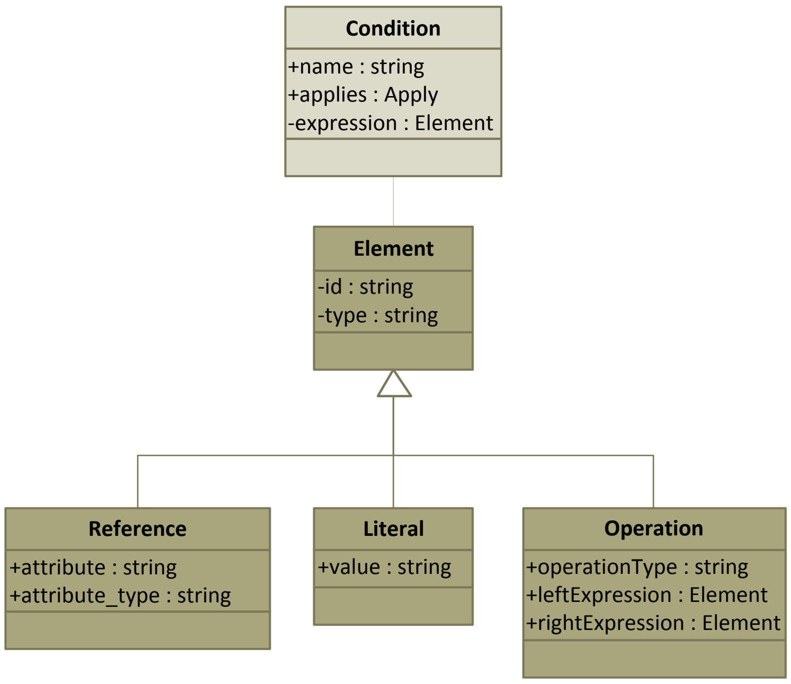

Figure 25: Condition Class and the logical expression.

As in the pointcut expression any of these Elements can be used in the conditional expression. For instance, the conditional expression in Figure 26(a) evaluates true if the loop is a zero nested_level loop and its body contains more than fifty statements. The conditional expression will be similar as depicted in Figure 26(b).

```
condition for applyUnroll:
$loop.nested_level == 0 && $body.num_statements > 50
end
```

(a)

```
Operation: AND
Operation: EQ
            Reference $loop nested_level int
            Literal int 0
          Operation: GT
             Reference $body num_statements int
             Literal int 50
```

(b)

Figure 26: Conditional expression example and its Aspect-IR representation.

4.4. Expressions Interpreter

As the meta-language is parsed and the Aspect-IR is created, the attribute use within a conditional expression delimits the condition evaluation to be done by the weaver. To give more use power to the meta-language front-end, and since one can define the attributes for the join points, the front-end integrates an interpreter which can resolve the expressions and join point attributes use, if the required information is provided. This interpreter, receiving the Aspect-IR as input, analyzes that IR and assesses the possible expressions, and leaves intact those expressions which do not have the necessary information available.

For instance, if a same condition is addressed to two different apply statement and only one of those can be evaluated, for its necessary attribute data is accessible, the other apply statement remains intact. As for the apply statement which can be evaluated, the statement remains in the Aspect-IR with no condition if this condition is evaluated true. Otherwise, if the condition is evaluated false then the apply statement is removed from the Aspect-IR as it would never be invoked in the weaver for having a conditional expression evaluated false. Hence, the examples depicted in Figure 27 express some evaluation cases, in which the required information of the function call "f1" for the condition is appropriate and for function call "f2" the information is limited.

```
...
functionF1: select function{*}.call{name==”f1”} end
apply to functionF1: define call(num_argin=3,return_type=”void”); end

functionF2: select function{*}.call{name==”f2”} end
apply to functionF2: define call(return_type=”void”); end

inlineFunction: apply to functionF1, functionF2
       optimize inline;
end

debugFunBefore: apply to functionF1, functionF2
       insert before %{printf(“call to function <$call.name>”)}%;
end

debugFunAfter: apply to functionF1, functionF2
       insert after %{printf(“function return a <$call.return_type> value”)}%;
end

inlineCondition: condition for inlineFunction: $call.num_argin > 2 end
debugCondition: condition for debugFunBefore: $call.return_type == “void” end
debug2Condition: condition for debugFunAfter: $call.return_type!= “void” end
...
```

Figure 27: Aspect example holding conditioned applies.

For the first condition, *inlineCondition*, the information is only known for the *functionF1* select, and thus only for this select the condition is evaluated. The condition is true and so the *inlineFunction* apply remains in the Aspect-IR with no condition for the select *functionF1*, but with condition for the select *functionF2*. The *debugCondition* requires that both *functionF1* and *functionF2* call selects return no value. Knowing that one defined both calls with a void return, both evaluates the condition true and thus the *debugCondition* condition is no longer required in the Aspect-IR, being removed from so. The last condition, *debug2Condition*, requires both selects to have a return type different from "void". As seen in the condition *debugCondition,* both have "void" as return type, and so both evaluates the condition false. Thus, the *debugFunAfter* apply should not be handed to the weaver as it will never be applied. The *debugFunAfter*, and respective condition, will then be removed from the Aspect-IR.

After passing through the interpreter, the Aspect-IR will be simpler for the weaver to analyze as some of the conditions were already evaluated and a minor effort is needed for the weaver. The front-end has then a more important role than just translate the meta-language into the Aspect-IR, becoming responsible to solve the possible conditions. The following example translates the Aspect-IR for the example in Figure 27 after being interpreted. The only remaining applies are the *inlineFunction* with condition for the *functionF2* select, a new *inlineFunction* apply containing no condition the *functionF1* select and the *debugFunBefore* apply where both of the conditions were evaluated true and thus no condition is kept in this apply.

```
Apply inlineFunction
     Targets: [functionF2]
     Conditioned by: inlineCondition
     Actions:
        Optimize inline
Apply inlineFunction_2
     Targets: [functionF1]
     Actions:
        Optimize inline
Apply debugFunBefore_3
     Targets: [functionF1, functionF2]
     Actions:
            Insert before simples_debugFunBefore_0
```

4.5. MATLAB Flow

Figure 28 shows the use of aspects specified in the meta-language in a MATLAB code. As already referred, to apply aspects over MATLAB code, the MATLAB's JSF is needed. The front-end will then combine the aspects and the JSF, creating an Aspect-IR that then can be used to apply the aspects to the MATLAB program.

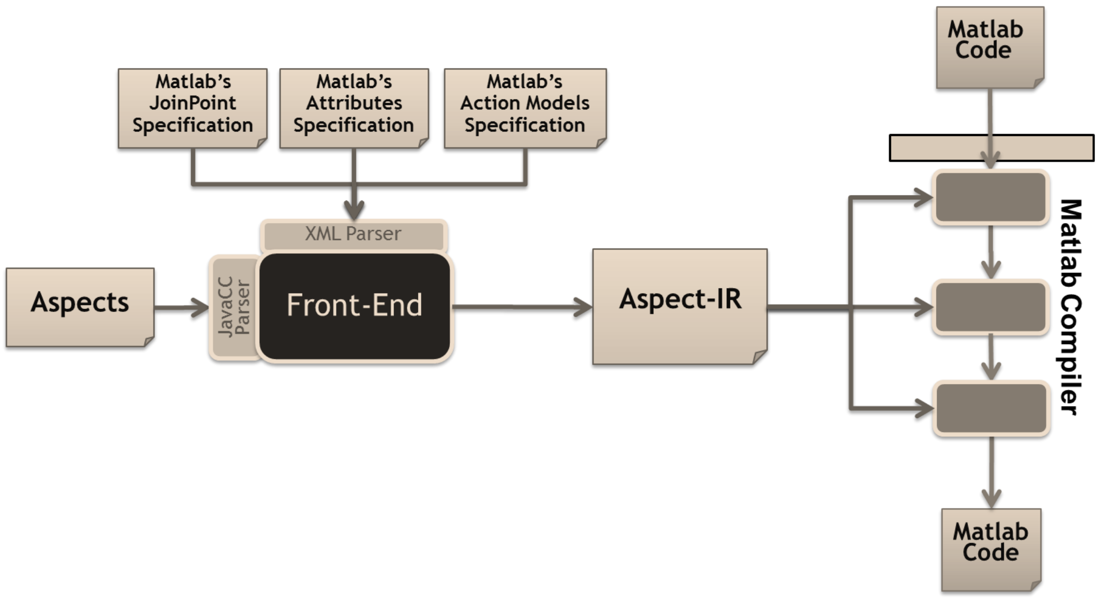
Matlab's JoinPoint Specification
Matlab's Attributes Specification
Matlab's Action Models Specification
XML Parser
Aspects
JavaCC Parser
Front-End
Aspect-IR
Matlab Code
Matlab Compiler
Matlab Code

Figure 28: Application of aspects on MATLAB code.

With respect to MATLAB, one can specify aspects in the meta-language which influence different stages of the MATLAB Compiler (Ricardo Nobre et al., 2010). The first level in the MATLAB Compiler is the "MATLAB to IR" (*Mat2tir*) which parses the MATLAB program into a Tom intermediate representation (T.-L. Website, 2011), named as Tom-IR. Tom-IR consists in a tree representation of the program and in which the aspects are applied. This intermediate stage enables the manipulation of the program structure and the selection of the join points according to the pointcut expressions. An example of the transition from a MATLAB program to a Tom-IR is depicted in Figure 29. As the Tom-IR takes the form of an XML structure, it is easier to manipulate and to add actions into the MATLAB program.

```
function a = changeVarA(b, c)                                 MATLAB function
    a = b+c;
    for i = 1:1:c
      a = a + b*c;
end
```

```
<Start>                                                       Tom-IR
     <FunctionMFile> <!—input and output declaration and function name-->
      <ConcStatement>
       <Statement>
        <Assign>
         <Expression>
    <Id> <Identifier>terminal = "a"</Identifier></Id>
         </Expression>
         <Plus>
          <ConcExpression>
      <Id><Identifier>terminal = "b" </Identifier></Id>
      <Id><Identifier>terminal = "c" </Identifier></Id>
          </ConcExpression>
         </Plus>
       </Statement>
       ...
     </FunctionMFile>
</Start>
```

Figure 29: Use of a MATLAB program in *Mat2tir*: MATLAB code (on the top) and partial Tom-IR (on the bottom).

To change types of variables in the code according to aspects, an extra tool is used[9]. It is used between the "MATLAB to IR" stage and the "IR to MATLAB" or "IR to C" stage, and was named as *tir2tir*. This tool uses Tom strategies (T. S. Website, 2011) to search for the variables identified by a specific pointcut and apply the type defined in the aspect. This definition is passed into the *tir2tir* tool by an input text file, exemplified in Figure 30, where the types are listed for each variable. The variables that are not listed are considered with type double. This input file for *tir2tir* is interpreted by the tool and associates every variable used within the code with the variable's corresponding type.

Not only *tir2tir* do an initial definition of the variable, but also performs a semantic analysis to the program expressions and adjusts the operations to a precision of the maximum type between the two expressions of the operation, e.g., with an *int8* variable type and an *int16*, the operation may return its value with an *int16* precision. This type of semantic analysis can be perceived in Figure 30, which depicts the type definition for variables "a", "b" and "c" and shows the final result of the type definition merged with the Tom-IR.

Figure 30 shows the default semantic analysis that was used during the definition of types for MATLAB programs with the meta-language. The tool contains three additional types of

[9] This tool has been partially developed in the context of a previous schoolarship involving the author of this thesis and Luis...

semantic analysis. One semantic analysis considers the maximum precision needed, where, in the same example as above, between an *int8* and an *int16*, for add and sub operations, the maximum precision needed is the *int16* plus an extra bit, which results in an *int17*. This type is a non-existent type in MATLAB, which is why the next closer type is chosen, the *int32*. For multiplications, the sum of the bits of both types is the maximum precision needed: an *int24*. The same rule applies for this inexistent type, and so the *int32* type is selected. Other semantic analysis considered is the result-precision driven, which is similar to the default option, thought the maximum type allowed is the type of the assignable variable. The last type of semantic analysis considers the user-custom precision, where one can define which type to use between two operands with specific types and considering a specific operator.

```
scope typeDef {                                               Type definition
       c: int8
       a: int32
       b: int16
     }
```

```
function a = simples(b, c)                                    Final MATLAB function
     c = int8(c);
     b = int16(b);
     a = int32(b+c);
     for i = 1:1:c
     a = int32(a+int16((b*c)));
end
```

Figure 30: Variable type definition (top) used in *ti2tir* and the final result (bottom).

After applying the aspects and defining the types of the variables, the Tom-IR is passed into the back-end, in which the Tom-IR is converted into MATLAB code or C code. The code example in Figure 30 is the result of the conversion of the Tom-IR in Figure 29, merged with the type definition.

4.6. Summary

This chapter described some details about the implementation of the framework (front-end). With the goal to apply aspects to MATLAB code, we also presented the MATLAB compilation flow. The meta-language grammar is flexible to accept any name for join point, its attributes, and also the identification of actions. All these statements are accepted with identifiers which do not limit the syntax of the meta-language to a limited number of keywords. Thus, the set of keywords are small and the meta-language is flexible to accept any set of words, if such words are defined in the respective specification file.

The front-end developed during this thesis, is a flexible front-end which enables the propagation of aspects into other programming languages by accepting the programming language specification externally. The external specification also allows the update of the specification as long as the structure is kept. The join point model is specified in a hierarchical way to solve the join point chain correctness.

To envision more usability for the front-end, an interpreter was developed. This interpreter enables the condition evaluation during the front-end stage, releasing some work for the weaver and keeping the main concern of capturing join points and applying actions.

5. Case Studies

This chapter presents the evaluation of the approach with a number of examples and with two compiler flows. While the system was in development, some unitary tests were done to verify system's consistency, and to prevent possible present or future errors related to development. After obtaining a stable release of the front-end and tested the system, the meta-language was used in real programs to confirm its utility and reusability.

The case studies used on this meta-language were a weaver for C programming language, provided by REFLECT (R. P. Website, 2011), which has focus, among others, on applying aspects on C programs, and the MATLAB Compiler, provided by AMADEUS (Ricardo Nobre, et al., 2010), and extended with some aspect weaving features. These weavers are different and each one requires different structure and information. The program codes were provided by the research project AMADEUS (Ricardo Nobre, et al., 2010) and the European research project REFLECT (J. M. P. Cardoso, et al., 2011). We show some examples using the aspect meta-language in the context of two different programming languages: MATLAB and C.

5.1. MATLAB Examples

The meta-language was used in a set of MATLAB examples to demonstrate its usage as a meta-language for AOP, focusing on type definition and monitoring.

5.1.1. LATNRM

The *latnrm* function, depicted in Figure 31, is a MATLAB function included in the benchmark repository of the AMADEUS project. The benchmark mainly consists of the *latnrm* function which implements a 32nd-order Normalized Lattice filter (the MATLAB version presented here was obtained by a manual translation from the original C code version presented in (C. G. Lee, 2011)). Each variable uses double precision floating-point representation.

```
function [outa] = latnrm(data, coefficient, internal_state, NPOINTS, ORDER)
    bottom=0;
    for i = 1:1:NPOINTS
       top = data(i);
       for j = 2:1:ORDER
          left = top;
          right = internal_state(j);
          internal_state(j) = bottom;
          top = coefficient(j-1) * left - coefficient(j) * right;
          bottom = right*coefficient(j-1) + coefficient(j) * left;
       end
       internal_state(ORDER-1) = bottom;
       internal_state(ORDER) = top;
       sum = 0.0;
       for j = 1:1:ORDER
           sum = sum + internal_state(j) * coefficient(j+ORDER);
       end
       outa(i) = sum;
    end
```

Figure 31: The *latnrm* MATLAB function.

A possible implementation of this *latnrm* function may take advantage of the use of fixed-point data types. Using the following simple aspect to define types of variables one can declare all the variables with twenty bits of integer part and sixteen bits for the decimal part.

```
aspectdef latnrmTypeDef
    vars: select var{*} end  //select all variables
    apply to vars
      define var(type="fixed(20,16)"); // assign a fixed-point type to them
    end
end
```

The result of applying the above type definition to the *latnrm* function is presented in Figure 32. The type definition allows the operations to be done in the variables type, instead of doing the operations with double precision. Although the precision was limited to a fixed(20,16) type and thus the operations do not need to be done with double precision, the right hand side of the assignment statements become more complex. As can be seen by this example, the use of a simple aspect and the associated weaving process may eliminate the need of manual code modifications, and consequently the elimination of possible errors and/or omissions.

```
function outa = latnrm(data, coefficient, internal_state, NPOINTS, ORDER)

    tir2tir_quantizer_0 = quantizer('mode', 'fixed', 'format', [20 16]);
    bottom = quantize(tir2tir_quantizer_0, 0);

    for i = 1:1:NPOINTS
         top = quantize(tir2tir_quantizer_0, data(i));

    for j = 2:1:ORDER
         left = quantize(tir2tir_quantizer_0, top);
         right = quantize(tir2tir_quantizer_0, internal_state(j));
         internal_state(j) = quantize(tir2tir_quantizer_0, bottom);
         top = quantize(tir2tir_quantizer_0,
                 quantize(tir2tir_quantizer_0,
                    (quantize(tir2tir_quantizer_0,
                    (coefficient((j-1))*left))-
                        quantize(tir2tir_quantizer_0,
                            (coefficient(j)*right)))));
         bottom = quantize(tir2tir_quantizer_0,
                    quantize(tir2tir_quantizer_0,
                        quantize(tir2tir_quantizer_0,
                            (right*coefficient((j-1))))+
                              quantize(tir2tir_quantizer_0,
                                 (coefficient(j)*left)))));
         end

         internal_state((ORDER-1)) = quantize(tir2tir_quantizer_0, bottom);
         internal_state(ORDER) = quantize(tir2tir_quantizer_0, top);
         sum = quantize(tir2tir_quantizer_0, 0);

         for j = 1:1:ORDER
             sum = quantize(tir2tir_quantizer_0,
                    quantize(tir2tir_quantizer_0,
                        (sum+quantize(tir2tir_quantizer_0,
                            internal_state(j)*coefficient((j+ORDER)))))));
         end

         outa(i) = sum;
end
```

Figure 32: The *latnrm* function using variables with fixed(20,16) precision.

5.1.2. **Harris**

Harris is a function used in an application provided by Honeywell to the REFLECT project.It is an image corner detector, known due to its invariance to rotation, scale, illumination variation and image noise (C. Harris et al., 1988). Figure 33 shows part of the MATLAB code of the Harris function.

```
function [cim, r, c, rsubp, csubp] = harrisExtFilt(im, filter, thresh, radius)

    error(nargchk(2,5,nargin));

    if ~isa(im,'double')
         im = double(im);
    end

    subpixel = 1; % rcm - force use of subpixel,

    ... % code omitted

    cim = (Ix2.*Iy2 - Ixy.^2)./(Ix2 + Iy2 + eps);

    if nargin > 2
            if subpixel
                 [r,c,rsubp,csubp] = nonmaxsuppts(cim, radius, thresh);
            else
                 [r,c] = nonmaxsuppts(cim, radius, thresh);
        end
    end
```

Figure 33: Partial MATLAB code of Harris function.

The function can be simplified and cleaned if the secondary cross cutting concerns are moved into an aspect as the following code depicts.

```
    aspectdef conditionalExpressions
      beginning: select function{*}.body.first end
      apply to beginning
              insert before
                      %{if ~isa(im,'double')
                             im = double(im);
                        end);}%;

              insert before %{error(nargchk(2,5,nargin));}%;
       end
      lastStat: select function{*}.body.last end
      apply to lastStat
              insert after
                      %{if nargin > 2
                             if subpixel
                                    [r,c,rsubp,csubp] =
                                          nonmaxsuppts(cim, radius, thresh);
                             else
                                    [r,c] = nonmaxsuppts(cim, radius, thresh);
                             end
                       end }%;
       end
    end
```

The code presented in Figure 34 contains only the main concern which contains a simpler body function.

```
function [cim, r, c, rsubp, csubp] = harrisExtFilt(im, filter, thresh, radius)

    subpixel = 1; % rcm - force use of subpixel
    ... % code omitted
    cim = (Ix2.*Iy2 - Ixy.^2)./(Ix2 + Iy2 + eps);
```

Figure 34: Partial MATLAB code of Harris function containing only the main concern.

The weaving of the code in Figure 34 with the aspect presented above results in the original code presented in Figure 33.

A more elaborated aspect can be created to reduce the quantity of condition verification on if statements by separating the chained if statements into conditioned applies, as the following example depicts:

```
aspectdef conditionalExpressions2
      beginning: select function{*}.body.first end
      apply to beginning
             insert before
                    %{if ~isa(im,'double')
                           im = double(im);
                      end);}%;

             insert before %{error(nargchk(2,5,nargin));}%;
      end
      lastStat: select function{*}.body.last end
      lastApply1: apply to lastStat
             insert after
                    %{if subpixel
                           [r,c,rsubp,csubp] =
                                nonmaxsuppts(cim, radius, thresh);
                      else
                           [r,c] = nonmaxsuppts(cim, radius, thresh);
                      end}%;
       end
      condition for lastApply1:
             $function.nargins > 2
             && $function.nargouts != 4 && $function.nargouts != 2
      end

      lastApply2: apply to lastStat
             insert after
                    %{[r,c,rsubp,csubp] =
                           nonmaxsuppts(cim, radius, thresh);}%;
      end
      condition for lastApply2:
             $function.nargins > 2 && $function.nargouts == 4 end

      lastApply3: apply to lastStat
      insert after
        %{[r,c] = nonmaxsuppts(cim, radius, thresh);}%;
      end
      condition for lastApply3:
             $function.nargins > 2 && $function.nargouts == 2 end
end
```

5.2. C Examples

The meta-language was tested with some C program examples to demonstrate its usage as a meta-language for AOP, focusing on monitoring and code optimizations.

5.2.1. Timing an application

Developers often need to measure the execution time of their applications, and that requires extra code to be inserted on every application. One can describe this concern using the meta-language, so that the aspect can be applied on every accessible application.

Figure 35 demonstrates an aspect description which instruments the main function, and adds the necessary code to measure the execution time of an application. It contains three pointcut definitions. The first, mainFunction, selects the main function declaration and adds the #include directive to use the required time functions. Next, we select the first statement (firstStmt) of the main function, and add code before that statement to initialize the structures and start monitoring time. Finally, we select the last statement (lastStmt), which we assume to be the return statement[10], and insert code to stop the timer and print the elapsed time.

```
aspectdef timer
  mainFunction: select function{name == "main"} end
  apply to mainFunction
        insert before %{#include <time.h>}%;
  end
  firstStmt: select function{name == "main"}.body.first end
  apply to firstStmt
        insert before  %{time_t start,end;
                         printf("starting timer!\n");
                         time (&start);}%;
  end
  lastStmt: select function{name == "main"}.body.last end
  apply to lastStmt
        insert before %{time (&end);
                       printf("ellapsed time: %.2lfs\n",
                                difftime(end,start));}%;
  end
end
```

Figure 35: Aspect description which instruments the code to time the application

An example of the result of the weaving process after having applied the aspect is shown in the following code. The inserted code by the weaver will advise the user that a timer will start when the application starts and at the end the code advises the user the time spent during the program run.

```
#include <time.h>
int main()   {
   printf("starting timer!\n");
   time(&start);
   …
   f();
   …
   time (&end);
   printf("ellapsed time: %.2lfs\n", difftime(end,start));
   return 0;
}
```

[10] A more complete aspect will have to deal with multiple return statements in the function main, with the absence of a return statement, and with a return statement including a function call.

5.2.2. Counting the number of loop iterations

Loops can often induce hotspots in the application, and finding the number of iterations can be useful to determine the applicability of certain transformations. The example in Figure 36 depicts two nested loops where one requires identifying the number of iterations for each one of the loops.

```
... //code omitted
{
   for (int i = 0; i < N; i++) {
     ...
     for (int j = 0; j < M; j++) {
         ... //code omitted
     }
     ... //code omitted
   }
   ... //code omitted
}
```

Figure 36: Example code with two nested loops.

Figure 37 illustrates an aspect description which instruments every *for* loop using two selection points. The first selection point picks the loop itself, in which two actions are applied, namely initializing the loop count before the loop construct and printing the count result after loop execution, respectively. The second selection point (C) picks the first statement of the loop body, where an increment loop count is inserted before that statement. The loop count *uid* attribute returns a unique identifier for every join point, and in this case ensures that every variable declared is unique.

```
aspectdef demoA4
  A: select function{*}.body.loop{type=="for"} end
  apply to A
      insert before %{int count_<$loop.uid> = 0;}%;
      insert after %{printf("loop [%s(), %d, %s]: %d\n",
                       "<$function.name>", <$loop.level>,
                       "<$loop.loc>", count_<$loop.uid>);}%;
  end
  C: select function{*}.body.loop{type=="for"}.body.first end
  apply to C insert before %{count_<$loop.uid>++;}%; end
end
```

Figure 37: Aspect to add loop count for every loop in the application

An example of the weaved code is shown in Figure 38. This example shows the introduction of counting variables that accumulates in each iteration of the corresponding loop and displays its value in the end of the loop.

```
...
{
   int count_01 = 0;
   for (int i = 0; i < N; i++) {
      count_01++;
      ...
      int count_02 = 0;
      for (int j = 0; j < M; j++) {
         count_02++;
         ...
      }
      printf("loop [%s(), %d, %s]: %d\n", "main.c", 1, "(10,5)", count_02) ;
      ...
   }
   printf("loop [%s(), %d, %s]: %d\n", "main.c", 0, "(7,2)", count_01) ;
   ...
}
```

Figure 38: Weaved code after applying the aspect shown in Figure 37 to the code in Figure 36. Statements in bold identify code inserted by the weaver.

5.2.3. **Loop Unrolling**

Although loops are useful for programmers to iterate a set of values with them with no repeated code, the computational costs of a loop can be considerably high. A way of avoiding the overhead associated to the control of loop iterations and to increase the potential to other compiler optimizations is by unrolling loops (partial or completely). For instance, the following example depicts a simple loop. The iterations of this loop consists on a jump statement, followed by a conditional expression verification, and another possible jump, each time an iteration comes to an end. Knowing that the number of loop iterations is always higher than a certain amount, the loop could be easily unrolled a number of times.

```
... //code omitted
int main() {
    int acc = 0;
    int i;
    for(i=0; i<10; i++) {
       acc += factorial(i);
    }
    return 0;
}
```

To do so, one can define an aspect to *unroll* the loops a specific number of times. Let that value be 2. In that case, the example below depicts that aspect, in which the weaver is instructed to *unroll* the loop two times and update the rest of the loop.

```
      aspectdef loop_unrolling
         forLoops: select function{*}.body.loop{type=="for"} end
         applyUnroll: apply to forLoops optimize loopunroll(2); end
      end
```

In the REFLECT design flow (J. M. P. Cardoso, et al., 2011), this aspect is processed in a low level stage and not in the source-to-source weaving. Therefore, the resulting code is

Assembly code, representing the C code depicted in the example above, plus the advising of the loop unroll aspect. The result, being an extensive Assembly, is depicted in Appendix D.

5.2.4. **Function Inlining**

Function calls may have a large impact in the execution time and may prevent specialization and optimizations. Function inlining is a compiler optimization that replaces a function call with a copy of the body of the function. The example below depicts a code slice, used in the context of REFLECT, which contains a set of functions calls that, within each of those functions, has function calls.

```
unsigned int factorial(int n)
{
      int f = 1;
      int i;
      for(i = 1; i <= n; i++)
      {
        f *= i;
      }
      return f;
}

int main() {
      int acc = 0;
      int i;
      for(i=0; i<10; i++) {
        acc += factorial(i);
      }
      return 0;
}
```

Another and more practical solution to *inline* those function call is the definition of an aspect (see below) that instructs the weaver to do function inlining for a set of functions.

```
     aspectdef function_inlining
      funcCalls: select
                 function{name=="main"}.body.call{name=="factorial"}
        end
       apply to funcCalls optimize inline; end
     end
```

Thus, the main function would be changed to contain the functions bodies and no jump statement to the function that was called will be needed. Similar to the case study in Section 5.2.3, the resulting code is Assembly and thus being an extensive code the result of weaving the code example with the aspect above would result in the Assembly code depicted in Appendix E.

5.3. Conditional expressions and the Front-end

Usually, when using the condition expression for *apply* statements, the condition is evaluated by the weaver, as the meta-language does not contain the necessary information to evaluate the condition. The example in Figure 39 depicts an aspect that selects the calls to function usage and update and *inlines* those function calls if they do not have input arguments.

```
aspectdef aspectFunctionCalls

mainCalls:
   select file{name=="main.c"}.function{*}.call{name=="usage",name=="update"}
end
pointCalls:
   select file{name=="point.c"}.function{*}.call{name=="usage",name=="update"}
end
applyToCalls:
    apply to mainCalls, pointCalls
       optimize inline;
    end
condition for applyToCalls: $call.num_argin == 0 end
end
```

Figure 39: Aspect to inline function calls with no input arguments.

The decision of applying the action to the pointcut is entirely the work of the weaver. Although, if one knows the number of input arguments, that decision can be moved to the front-end, which can evaluate the conditions if one notifies the front-end of the join point attributes values. So, if the front-end contains the necessary information to solve some join point attribute uses, that request is not necessary to be passed onto the weaver. For instance, if one creates an aspect file defining some attribute values to be used in the aspect of Figure 39, the condition could be resolved for the apply. Thus, the Aspect-IR is created depending on the quantity of information the front-end obtains. Using the example in Figure 39 as reference, one of the following Aspect-IR can arise.

5.3.1. **Condition not possible to evaluate by the front-end**

In this case the Aspect-IR is generated as usual, with no changes in the structure of the aspect. The example below is the resulting Aspect-IR in XML containing all the information of the aspect presented in Figure 39. This case happens whenever the front-end does not have the necessary information to evaluate the condition.

```
    <aspects>
      <aspect name="aspectFunctionCalls">
        <pointcut name="pointCalls">
          <file name="point.c">
            <function name=".*">
              <body>
                <call name="usage"/>
                <call name="update"/></body></function></file>
        </pointcut>
        <pointcut name="mainCalls">
          <file name="main.c">
            <function name=".*">
              <body>
                <call name="usage"/>
                <call name="update"/></body></function></file>
        </pointcut>
        <apply to="mainCalls">
          <actions>
            <optimize kind="inline" stage="cosy-front-end"/>
          </actions>
          <condition>
      <EQ>
              <identifier name="$call" attribute="num_argin"
      id_type="joinpoint" attr_type="int"/>
              <literal value="0" type="int"/>
            </EQ>
       </condition>
        </apply>
        <apply to="pointCalls">
          <actions>
            <optimize kind="inline" stage="cosy-front-end"/>
          </actions>
          <condition>
      <EQ>
              <identifier name="$call" attribute="num_argin"
      id_type="joinpoint" attr_type="int"/>
              <literal value="0" type="int"/>
            </EQ>
    </condition></apply></aspect></aspects>
```

5.3.2. **Condition evaluated true**

The aspect below contains the same aspect as the one depicted in Figure 39 and adds a join point attribute definition. The user defines the *num_argin* function call attribute as zero for all function calls.

```
aspectdef aspectFunctionCalls
    ...
    apply to mainCalls, pointCalls:
         define call(num_argin=0);
    end
    ...
end
```

Thus, the condition for the apply *applyToCalls* is evaluated true and so the condition does not have to be passed to the weaver, being removed from the Aspect-IR, as depicted in the following example.

```
    <aspects>
       ...
        <apply to="mainCalls">
          <actions>
            <optimize kind="inline" stage="cosy-front-end"/>
          </actions>
        </apply>
        <apply to="pointCalls">
          <actions>
            <optimize kind="inline" stage="cosy-front-end"/>
          </actions>
    </condition></apply></aspect></aspects>
```

5.3.3. Condition evaluate false

This example is similar to the one in Section 5.3.2, the difference resides in the condition being evaluated as false, and so the action should not take place. For instance, the following aspect defines the attribute *num_argins* as one, i.e., all the function call selected has one argument.

```
aspectdef aspectFunctionCalls
    ...
    apply to mainCalls, pointCalls:
         define call(num_argin=1);
    end
    ...
end
```

As referred, in this case the condition is evaluated false and the *apply* should not take place on those function calls. Thus, neither the *condition* nor the *apply* sections are sent to the weaver. The Aspect-IR will then have the structure as the following example shows, where it is only visible the selection of the function calls as depicted in Figure 39.

```
...
<pointcut name="pointCalls">
      <file name="point.c">
        <function name=".*">
          <body>
            <call name="usage"/>
            <call name="update"/></body></function></file>
    </pointcut>
    <pointcut name="mainCalls">
      <file name="main.c">
        <function name=".*">
          <body>
            <call name="usage"/>
            <call name="update"/></body></function></file>
    </pointcut>
...
```

5.4. Summary

The meta-language and respective front-end were tested with some case studies. Those case studies were used to evaluate the meta-language practicability and reusability. Each case study final result was dependent on the capability of the weaver to apply the desired aspects. For instance, the *optimize* action, available in the weaver for C programming language was not available in the weaver for MATLAB, and likewise the type definition available for MATLAB is not yet available for C. Thus, each aspect was created knowing the capability of the weaver.

The result of the weaving depends of what type of action is applied. For an insertion of code the weaving is done at an earlier stage while the type definition or the unroll is created on a later compilation stage.

The final results were similar to the ones expected and the meta-language proved to be useful for both AMADEUS and REFLECT projects. The meta-language examples created for the case studies presented in this chapter can be also used with other code programs as the aspects exploit plenty reusability.

6. Conclusions

The AOP paradigm has evolved but still is not very used when developing embedded system programs. An aspect meta-language may expand the interest of programmers of embedded applications to AOP. Systems to use AOP in their daily programming will help them in concerns that are not part of the program main objective, such as tracing, monitoring, and precision studies. We presented an aspect meta-language and a tool able to process that input language. With this meta-language it is possible to specify the aspect modules and apply them over different programming languages. Furthermore, our approach is able to deal with aspects that may influence several compilation flow stages.

We considered as case studies the application of aspects programmed in our meta-language to MATLAB and C codes. Targeting two different programming languages such as MATLAB and C is important for our approach. In the development of many embedded systems MATLAB is used as an early model language and C as the default standard implementation language.

The creation of a meta-language has high value for programmers needing to monitor programs written in different programming languages. This aspect oriented meta-language also aims at helping programmers to resolve problems like variable type definition for precision purposes. For instance, in MATLAB all variables are by default represented in floating-point using double precision. For acceptable precision, the programmer could start by using double precision but as this may become computationally heavy, the programmer usually needs to specialize data types and analyze its impact. Using this meta-language, the programmer can easily test different precisions for each variable. Our approach provides tracing and type definition functionalities that can be reused over several programs of one programming language, or, depending on the aspect, pointcuts and advices used, can be reused over other programming languages.

Future work will consider extensions to the aspect meta-language presented in this thesis. The following step is related to the extensions to the apply section in order to include other type of statements beyond actions, such as if and loop statements. This will allow programmers to develop more advanced aspects. The import statement is another point of evolution as it can be used to import single aspects or sections of an aspect to be used by aspects within the importing meta-aspect file. Furthermore, a meta-code definition is another possible extension. With a meta-code definition, the aspects can be more reusable as the same code slice could correspond to the different addressed programming languages, without the requirement of changing the code to correspond to a different programming language and/or to use specific functions with implementations in all the target programming languages to which a given aspect may be applied.

References

Aracic, I., Gasiunas, V., Mezini, M., & Ostermann, K. (2006). An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 135-173.

Aslam, T. (2010). *AspectMatlab: an aspect-oriented scientific programming language.* Master, McGill University, Montréal.

Aslam, T., Doherty, J., Dubrau, A., & Hendren, L. (2010). *AspectMatlab: an aspect-oriented scientific programming language*. Paper presented at the Proceedings of the 9th International Conference on Aspect-Oriented Software Development, Rennes and Saint-Malo, France.

Cardoso, J. M. P., Diniz, P., Monteiro, M. P., Fernandes, J. M., & Saraiva, J. (2010). *A Domain-Specific Aspect Language for Transforming MATLAB Programs*. Paper presented at the Domain-Specific Aspect Language Workshop (DSAL'2010), part of the 9th International Conference on Aspect-Oriented Software Development (AOSD'2010), Rennes & Saint Malo, France.

Cardoso, J. M. P., Diniz, P. C., Petrov, Z., Bertels, K., Hübner, M., Someren, H. v., Gonçalves, F., Coutinho, J. G. d. F., Constantinides, G., Olivier, B., Luk, W., Becker, J., Kuzmanov, G., Thoma, F., Braun, L., Kühnle, M., Nane, R., Sima, V.-M., Krátký, K., Alves, J. C., & Ferreira, J. C. (2011). REFLECT: Rendering FPGAs to Multi-Core Embedded Computing. In J. M. P. Cardoso & M. Huebner (Eds.), *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*: Springer.

Cardoso, J. M. P., Fernandes, J. M., & Monteiro, M. (2006). *Adding Aspect-Oriented Features to MATLAB*. Paper presented at the SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies, A workshop affiliated with AOSD 2006, Bonn, Germany.

Cazzola, W. (2011a, 22th February). @AspectJ: a Fine-Grained AspectJ Extension Retrieved 30th May, 2011, from http://cazzola.dico.unimi.it/ataspectj.html

Cazzola, W. (2011b, 22th February). @Java: a Java Annotation Extension Retrieved 30th May, 2011, from http://cazzola.dico.unimi.it/atjava.html

Eaton, J. W., Bateman, D., & Hauberg, S. (2009). *GNU Octave Manual Version 3*.

Elrad, T., Filman, R. E., & Bader, A. (2001). Aspect-oriented programming: Introduction. *Communications of the ACM, 44*(10), 29-32.

Filman, R. E., Elrad, T., Clarke, S., & Ak it, M. (2004). *Aspect-oriented software development*.

Gomez, C. (1999). *Engineering and scientific computing with Scilab*: Birkhauser.

Griswold, B., Hilsdale, E., Hugunin, J., Isberg, W., Kiczales, G., & Kersten, M. (1998). Aspect-oriented programming with AspectJ. *Copyright Xerox Corporation, 2001*.

Harbulot, B., & Gurd, J. R. (2006). *A join point for loops in AspectJ*. Paper presented at the Proceedings of the 5th international conference on Aspect-oriented software development, Bonn, Germany.

Harris, C., & Stephens, M. (1988). *A combined corner and edge detector.* Paper presented at the Proceedings Fourth Alvey Vision Conference, Manchester, UK.

Huang, S. S., & Smaragdakis, Y. (2006). *Easy language extension with meta-aspectJ*. Paper presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China.

Huang, S. S., Zook, D., & Smaragdakis, Y. (2008). Domain-specific languages and program generation with meta-AspectJ. *ACM Trans. Softw. Eng. Methodol., 18*(2), 1-32. doi: 10.1145/1416563.1416566

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). An overview of AspectJ. *ECOOP 2001—Object-Oriented Programming*, 327-354.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, V., C., Loingtier, J.-M., Irwin, & J. (1997). *Aspect Oriented Programming*. Paper presented at the ECOOP'97, Jyväskylä, Finnland.

Lee, C. G. (2011). The UTDSP Benchmark Suite. Retrieved from http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.tar.gz

Lohmann, D., & Spinczyk, O. (2007). Aspect-Oriented Programming with C++ and AspectC++.

Luk, W., Coutinho, J., Todman, T., Lam, Y., Osborne, W., Susanto, K., Liu, Q., & Wong, W. (2009, Sept). *A high-level compilation toolchain for heterogeneous systems.* Paper presented at the Int'l SOC Conf. (SOCC'09).

Monteiro, M. P., Cardoso, J. M. P., & Posea, S. (2010). Identification and Characterization of Crosscutting Concerns in MATLAB Systems1.

Nobre, R., Cardoso, J. M. P., & Diniz, P. C. (2010). Leveraging Type Knowledge for Efficient MATLAB to C Translation: Technical Report, Portugal.

Nobre, R., Cardoso, J. M. P., & Diniz, P. C. (2010). *Leveraging Type Knowledge for Efficient MATLAB to C Translation*. Paper presented at the 15th Workshop on Compilers for Parallel Computing (CPC'10), Vienna University of Technology, Vienna, Austria.

Quarteroni, A., Saleri, F., & Gervasio, P. (2010). *Scientific computing with MATLAB and Octave*: Springer Verlag.

Spinczyk, O., Gal, A., & Schröder-Preikschat, W. (2002). *AspectC++: an aspect-oriented extension to the C++ programming language*.

Spinczyk, O., & Lohmann, D. (2011). AspectC++ Quick Reference.

Spinczyk, O., Lohmann, D., & Urban, M. (2005, May, 2005). Aspect C++: an AOP Extension for C++. *Software Developer's Journal,* 7.

Tarr, P., Ossher, H., Harrison, W., & Stanley M. Sutton, J. (1999). *N degrees of separation: multi-dimensional separation of concerns*. Paper presented at the Proceedings of the 21st international conference on Software engineering, Los Angeles, California, United States.

Website, A.-O. S. A. O. (2011). Aspect-Oriented Software Development Retrieved 13th February, 2011, from http://www.aosd.net/

Website, a. (2011). abc: The AspectBench Compiler for AspectJ Website Retrieved 13th February, 2011, from http://www.aspectbench.org/

Website, A. (2011). AMADEUS Website Retrieved 15th June, 2011, from http://www.fe.up.pt/~specs/projects/AMADEUS

Website, A. (2011). The AspectJ^{TM} Programming Guide Retrieved 4th February, 2011, from http://www.eclipse.org/aspectj/doc/released/progguide/index.html

Website, A. (2011). AspectMatlab Retrieved 13th February, 2011, from http://www.sable.mcgill.ca/mclab/aspectmatlab/index.html

Website, A. (2011). The Home of AspectC++ Retrieved 4th February, 2011, from http://www.aspectc.org/

Website, C. (2011). Cosy Retrieved 13th June, 2011, from http://www.ace.nl/compiler/cosy.html

Website, D. c. (2011). Dictionary.com, LLC Retrieved 11th February, 2011, from http://dictionary.reference.com/browse/meta-language

Website, J. (2011). Java Tree Builder Website Retrieved 10th June, 2011, from http://www.cs.ucla.edu/~palsberg/jtb/

Website, J. C. C. (2011). Java Compiler Compiler Retrieved 19th June, 2011, from http://javacc.java.net/

Website, J. P. (2011). API specification for the Java 2 Platform: Class Point Retrieved 11th February, 2011, from http://download.oracle.com/javase/1.3/docs/api/java/awt/Point.html

Website, L. (2011). LoopsAJ: a join point for loops in AspectJ Retrieved 13th February, 2011, from http://intranet.cs.man.ac.uk/cnc/projects/loopsaj.php

Website, M.-A. (2011). Meta-AspectJ Retrieved 28th May, 2011, from http://www.cc.gatech.edu/~yannis/maj/

Website, M. (2011). McLAB Retrieved 13th February, 2011, from http://www.sable.mcgill.ca/mclab/

Website, O. S. (2008, 28th March). Olaf Spinczyk Retrieved 13th February, 2011, from http://www4.informatik.uni-erlangen.de/~spinczyk/

Website, P. C. (2011). Valentin Turchin in Principia Cibernetica Retrieved 11th February, 2011, from http://pespmc1.vub.ac.be/METALARE.html

Website, R. P. (2011). Reflect Project Website Retrieved 13th June, 2011, from http://www.reflect-project.eu/

Website, S. (2011). Soot: a Java Optimization Framework Retrieved 13th February, 2011, from http://www.sable.mcgill.ca/soot/

Website, S. F. (2011). Spring Framework (Vol. 2011).

Website, T.-L. (2011). Tom Retrieved 13th February, 2011, from http://tom.loria.fr/

Website, T. E. F. (2011, 2011). AspectJ: crosscutting objects for better modularity Retrieved 27th January, 2011, from http://www.eclipse.org/aspectj/

Website, T. S. (2011). Tom Strategies Retrieved 18th June, 2011, from http://tom.loria.fr/wiki/index.php5/Documentation:Strategies

Zook, D., Huang, S., & Smaragdakis, Y. (2004). Generating AspectJ Programs with Meta-AspectJ. In G. Karsai & E. Visser (Eds.), *Generative Programming and Component Engineering* (Vol. 3286, pp. 583-605): Springer Berlin / Heidelberg.

Appendix A

Join point Model for C

The following code is the join point model, represented in XML, considered for the C programming language. This join point model is being used in the REFLECT project (J. M. P. Cardoso, et al., 2011) and includes a vast set of points of interest in C code. The join point hierarchy is represented here using the hierarchical structure of the XML

```
<joinpoints>
      <roots> <!-- starting join points for the join point model hierarchy -->
            <file/>
      </roots>
      <file type="file">
            <var type="var"/>
            <decl type="decl"/>
            <function type="function"/>
            <label type="label"/>
            <pragma value="label" type="label"/>
      </file>

      <var type="var">
            <index type="expr+"/> <!-- in the case of an array variable -->
      </var>

      <function type="function">
            <prototype type="prototype"/>
            <body type="body"/>
      </function>

      <prototype type="prototype">
            <args type="var"/>
            <return type="type"/>
      </prototype>

      <body type="section">
            <begin type="{"/>
            <end type="}"/>
            <decl type="decl"/>
            <first type="statement"/>
```

```xml
            <last type="statement"/>
            <statement type="statement"/>
            <return type="type"/>
            <in type="var*"/>
            <out type="var*"/>
            <var type="var"/>
            <loop type="loop"/>
            <assignment type="assignment"/>
            <call type="call"/>
            <pragma value="label" type="label"/>
            <section type="section"/>
            <if type="if"/>
            <switch type="switch"/>
            <label type="label"/>
      </body>

      <loop type="loop">
            <init type="expr"/>
            <condition type="expr"/>
            <counter type="expr"/>
            <body type="body"/>
            <control type="var"/>
      </loop>

      <assignment type="assignment">
            <lhs type="var"/>
            <rhs type="expr"/>
      </assignment>

      <call type="call">
            <args type="expr"/>
      </call>

      <if type="if">
            <condition type="expr"/>
            <then type="body"/>
            <else type="body"/>
      </if>
</joinpoints>
```

Appendix B

Join point Model for MATLAB

The following code is the join point model, represented in XML, considered for the MATLAB programming language. This join point model was based on the grammar of the AMADEUS MATLAB compiler (R. P. Website). Not every possible join point is mapped in this model. The join point hierarchy is represented here using the hierarchical structure of the XML.

```
<joinpoints>
	<roots> <!-- starting join points for the join point model hierarchy -->
		<file/>
	</roots>
	<file type="file">
		<function type="function"/>
		<label type="label"/>
		<pragma value="label" type="label"/>
	</file>
	<function type="function">
		<argins type="var+">
		<argouts type="var+">
		<body type="body"/>
	</function>
	<body type="section">
		<first type="statement"/>
		<last type="statement"/>
		<statement type="statement"/>
		<return type="type"/>
		<var type="var"/>
		<loop type="loop"/>
		<assignment type="assignment"/>
		<call type="call"/>
		<pragma value="label" type="label"/>
		<section type="section"/>
		<if type="if"/>
		<label type="label"/>
	</body>
```

```
       <loop type="loop">
              <init type="expr"/>
              <condition type="expr"/>
              <body type="body"/>
       </loop>

       <assignment type="assignment">
              <lhs type="var+"/>
              <rhs type="expr"/>
       </assignment>

       <call type="call">
              <args type="expr"/>
       </call>

       <if type="if">
              <condition type="expr"/>
              <then type="body"/>
              <else type="body"/>
       </if>
</joinpoints>
```

Appendix C

Meta-language Grammar

The following grammar corresponds to the current version of the meta-language grammar.

TOKENS

```
<DEFAULT> SKIP : {
" "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
| <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
}


<DEFAULT> TOKEN : {
<INTEGER: <DECIMAL_LITERAL> (["l","L"])? | <HEX_LITERAL> (["l","L"])? |
<OCTAL_LITERAL> (["l","L"])?>
| <#DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])*>
| <#HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+>
| <#OCTAL_LITERAL: "0" (["0"-"7"])*>
| <FLOAT: (["0"-"9"])+ "." (["0"-"9"])+>
}


<DEFAULT> TOKEN : {
<IMPORT: "import">
| <ASPECTDEF: "aspectdef">
| <CODEDEF: "codedef">
| <SELECT: "select">
| <APPLY: "apply">
| <TO: "to">
| <COMPUTE: "compute">
| <CONDITION: "condition">
| <FOR: "for">
| <BEGIN: "begin">
| <END: "end">
| <INSERT: "insert">
| <BEFORE: "before">
| <AFTER: "after">
| <AROUND: "around">
| <MAP: "map">
| <HARDWARE: "hardware">
| <OPTIMIZE: "optimize">
| <DEFINE: "define">
| <CODE: "%{" (~["}"])* "}" ("}" | ~["}","%"] (~["}"])* "}")* "%">
}

<DEFAULT> TOKEN : {
<COLON: ":">
| <SEMICOLON: ";">
| <DOT: ".">
| <COMMA: ",">
| <LBRACE: "{">
| <RBRACE: "}">
| <LBRACKET: "(">
| <RBRACKET: ")">
| <NOT: "!">
| <PLUS: "+">
| <MINUS: "-">
| <TIMES: "*">
| <SLASH: "/">
| <LTHANE: "<=">
| <LTHAN: "<">
| <GTHANE: ">=">
| <GTHAN: ">">
| <EQUAL: "==">
| <UNEQUAL: "!=">
| <ASSIGN: "=">
| <OR: "||">
| <AND: "&&">
| <LSBRACKET: "[">
| <RSBRACKET: "]">
}


<DEFAULT> TOKEN : {
<STRING: "\"" (~["\"","\\","\n","\r"] | "\\"
(["n","t","b","r","f","\\","\'","\""] | ["0"-"7"] (["0"-"7"])? | ["0"-"3"]
["0"-"7"] ["0"-"7"] | ["\n","\r"] | "\r\n"))* "\"">
| <IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)*>
| <VARIABLE: "$" <IDENTIFIER>>
| <#LETTER: ["_","a"-"z","A"-"Z"]>
| <#DIGIT: ["0"-"9"]>
}
```

NON-TERMINALS

```
    /** Start Node
     * This Node contains the aspects and code definitions
     */
    Start    ::=    ( Import )* ( AspectDef )* ( CodeDef )*
    /** Import
     * The import statement allows one to add information from another aspect
file such as attribute definitions
     */
    Import   ::=    <IMPORT> Identifier <DOT> Identifier <SEMICOLON>
    /** Aspect Definition
     * the aspect definition can contain outputs, inputs and variables
     * the body of the aspect consists of selects, applies and conditions
     */
    AspectDef::=    <ASPECTDEF> Outputs Identifier Inputs Variables ( ( Select
| Apply | ConditionalExpression ) )* <END>
    /** Outputs Definition **/
    Outputs  ::=    ( <LSBRACKET> Arguments <RSBRACKET> <ASSIGN> )?
    /** Inputs Definition **/
    Inputs   ::=    ( <LBRACKET> Arguments <RBRACKET> )?
    /** Variables Definition **/
    Variables::=    ( VariableDeclaration )*

    /** Select creation

     * the select contains an identifier of itself, and a joinPointExpr

     */

    Select   ::=    Identifier <COLON> <SELECT> JoinPointExpr <END>

    /** JoinPoint Using OR

     * joinpoints where is used OR expressions, like function{*}.var{name="i"}
|| function{*}.loop

     */

    JoinPointExpr   ::=    AndJoinPoint ( <OR> AndJoinPoint )*

    /** JoinPoint using AND

     * joinpoints where is used AND expressions

     */

    AndJoinPoint    ::=    NotJoinPoint ( <AND> NotJoinPoint )*

    /** JoinPoint using NOT

     * joinpoints where is used NOT expressions to represent where not to
apply: !function{*}.call{name="f1"}

     */

    NotJoinPoint    ::=    ( <NOT> )? JoinPoint

    /** JoinPoint

     * it has an identifier and a possible property expressions and each join
point can have a child, the join point down to

     * its hierarchy

     */

    JoinPoint::=    ( Identifier ( JoinPropertiesExpr )? ( <DOT> <IDENTIFIER>
( JoinPropertiesExpr )? )* | <LBRACKET> JoinPointExpr <RBRACKET> )

    /** Join Point properties

     *  it can be any property (*) or a set of properties

     */

    JoinPropertiesExpr    ::=    <LBRACE>  (  (  <TIMES>  |  PropertiesExpr  )  )
<RBRACE>

    /** The comma is used here as an OR symbol (like ||) **/

    PropertiesExpr  ::=    JoinPointProperties ( <COMMA> JoinPointProperties )*
    /** To combine properties like as an AND, include every properties wanted inside brackets **/

    JoinPointProperties   ::=   ( Property | <LBRACKET> Property ( <COMMA> Property )* <RBRACKET> )

    /** The property is defined by comparing to a value **/

    Property ::=   Identifier ( ( <EQUAL> | <UNEQUAL> ) ) Value

    /** Apply Definition

     * The apply can contain (or not) an identifier and it need a list of identifiers corresponding to selects

     * The body of the apply can be a list of insertions of code, mappings and optimizations

     */

    Apply   ::=   ( Identifier <COLON> )? <APPLY> <TO> Identifier ( <COMMA> Identifier )* ( ( ( Insert | MapTo | Optimize | Define ) ) <SEMICOLON> )+ <END>

    /** Insert

     * The insertion of code is made by indicating a codedefinition's identifier, or writting the desired code

     * It is also needed to say if the code is inserted before, after or around the pointcut

     */

    Insert   ::=   <INSERT> ( ( <BEFORE> | <AFTER> | <AROUND> ) ) ( ( <CODE> | <LBRACE> Identifier <RBRACE> ) )

    /** Map to

     * mappings to the hardware, recieves list of assignments as argument

     */

    MapTo   ::=   <MAP> <TO> <HARDWARE> <LBRACKET> Assignment ( <COMMA> Assignment )* <RBRACKET>

    /** Optimize

     * optimize has an identifier that is the action to be taken

     * the action can also have assignments as arguments

     */

    Optimize ::=   <OPTIMIZE> Identifier ( <LBRACKET> AssignmentOpt ( <COMMA> AssignmentOpt )* <RBRACKET> )?
    Define    ::=    <DEFINE>   Identifier   <LBRACKET>   Assignment   (   <COMMA>
Assignment )* <RBRACKET>

    /** Assigment of a property with a certain value **/

    Assignment      ::=    Identifier <ASSIGN> Value

    AssignmentOpt   ::=    ( Identifier <ASSIGN> )? Value

    /** Conditions

     *  can  have  an  identifier,  and  needs  a  list  of  identifiers  representing
the applies.

     * the body of the condition is a logical expression

     */

    ConditionalExpression ::=    (  Identifier  <COLON>  )?  <CONDITION>  <FOR>
Identifier ( <COMMA> Identifier )* <COLON> Expr <END>

    /** Declaration of variables

     * it has a type (the Identifier) and a variable identification (Variable)

     * A value can be assign to the variable (the Expr)

     */

    VariableDeclaration    ::=    Identifier   Variable   (   <ASSIGN>   Expr   )?
<SEMICOLON>

    /** Arguments

     * list of the inputs or outputs

     */

    Arguments::=    Identifier Variable ( <COMMA> Identifier Variable )*

    /** Code Definition

     * a code definition needs an identifier

     * and the body is C code that can contain a tag like  or

     * to insert information on the code like a variable value or a join point
property

     */

    CodeDef   ::=    <CODEDEF> Identifier <CODE> <END>

    /** Expression

     *  The  last  level  on  the  expression  is  represented  the  OR,  the  last
operation to be executed

     */

    Expr      ::=    SixthOp ( <OR> Expr )?
    /** AND operations **/

    SixthOp  ::=   FifthOp ( <AND> SixthOp )?

    /** LOGICAL operations: >, >=, <, <=, ==, != **/

    FifthOp  ::=   FourthOp ( ( <LTHAN> FifthOp | <LTHANE> FifthOp | <GTHAN>
FifthOp | <GTHANE> FifthOp | <EQUAL> FifthOp | <UNEQUAL> FifthOp ) )?

    /** '+' and '-' operations **/

    FourthOp ::=   ThirdOp ( ( <PLUS> FourthOp | <MINUS> FourthOp ) )?

    /** '*' and '/' operations **/

    ThirdOp  ::=   SecondOP ( ( <TIMES> ThirdOp | <SLASH> ThirdOp ) )?

    /** NOT, POSITIVE and NEGATIVE Operators **/

    SecondOP ::=   ( ( <NOT> | <PLUS> | <MINUS> ) )? FirstOp

    /** Inline or variable or join point use, or brackets use  **/

    FirstOp  ::=   ( Value | Identifier | <LBRACKET> Expr <RBRACKET> )

    /** a Parameter of a join point **/

    Parameter::=   Variable  (  <LBRACE>  <INTEGER>  <RBRACE>  )?  (  <DOT>
Identifier ( <LBRACKET> Assignment ( <COMMA> Assignment )* <RBRACKET> )? )?

    /** Variable token **/

    Variable ::=   <VARIABLE>

    /** Identifier token **/

    Identifier      ::=   <IDENTIFIER>

    /** Value **/

    Value    ::=   ( <INTEGER> | <FLOAT> | <STRING> | Parameter )
```

Appendix D

Loop Unroll

The following Assembly code (x86 processor) represents the code obtained for the case study in Section 5.2.3, after applying the loop unroll aspect to the input C code using CoSy (C. Website, 2011).

```
  .text
      .align      4
      .global
      factorial
  factorial:
  // -- bb0 --
  .L1:
      // outargsize
64
      pushl %ebp
      movl
      %esp,%ebp
      pushl %edi
      pushl %esi
      pushl %ebx
      movl
      8(%ebp),%edx
      movl  $1,%ecx
  // -- bb_la1 --
  .L2:
      leal  -
1(%edx),%eax
      movl  $1,%esi
      andl  $3,%eax
      incl  %eax
      movl
      %eax,%edi
  // -- bb1 --
  .L3:
  // -- bb2 --
  .L4:
```

```
    cmp
    %edx,%esi
    setle %al
    movzbl
    %al,%eax
    cmp
    %edi,%esi
    setle %bl
    movzbl
    %bl,%ebx
    andl
    %ebx,%eax
    testl
    %eax,%eax
    jz    .L7
// -- bb3 --
.L5:
    imull
    %esi,%ecx
// -- bb4 --
.L6:
    incl  %esi
    jmp   .L4
// -- bb5 --
.L7:
// -- bb6 --
.L8:
    jmp   .L20
// -- bb7 --
.L9:
    imull
    %esi,%ecx
// -- bb8 --
```

```
.L10:
// -- bb9 --
.L11:
// -- bb10 --
.L12:
    leal
    1(%esi),%eax
    imull
    %eax,%ecx
// -- bb11 --
.L13:
// -- bb12 --
.L14:
// -- bb13 --
.L15:
    movl
    %esi,%eax
    addl  $2,%eax
    imull
    %eax,%ecx
// -- bb14 --
.L16:
// -- bb15 --
.L17:
// -- bb16 --
.L18:
    movl
    %esi,%eax
    addl  $3,%eax
    imull
    %eax,%ecx
// -- bb17 --
.L19:
      addl  $4,%esi
  // -- bb18 --
  .L20:
      cmp
      %edx,%esi
      jle   .L9
  // -- bb19 --
  .L21:
  // -- bb20 --
  .L22:
      movl
      %ecx,%eax
  // -- bb21 --
  .L23:
      popl  %ebx
      popl  %esi
      popl  %edi
      popl  %ebp
      ret
      .align 4
      .global
      main
  main:
  // -- bb0 --
  .L24:
      // outargsize
96
      pushl %ebp
      movl
      %esp,%ebp
```

```
      pushl %esi
      pushl %ebx
      xorl
      %esi,%esi
  // -- bb_la1 --
  .L25:
      xorl
      %ebx,%ebx
  // -- bb1 --
  .L26:
  // -- bb2 --
  .L27:
      cmp
      $10,%ebx
      jge   .L33
  // -- bb3 --
  .L28:
      // sp offset
at call 64
      pushl %ebx
      call
      factorial
      addl  $4,%esp
      addl
      %eax,%esi
  // -- bb4 --
  .L29:
  // -- bb5 --
  .L30:
  // -- bb6 --
```

```
  .L31:
      leal
      1(%ebx),%eax
      // sp offset
at call 64
      pushl %eax
      call
      factorial
      addl  $4,%esp
      addl
      %eax,%esi
  // -- bb7 --
  .L32:
      addl  $2,%ebx
      jmp   .L27
  // -- bb8 --
  .L33:
  // -- bb9 --
  .L34:
      xorl
      %eax,%eax
  // -- bb10 --
  .L35:
      popl  %ebx
      popl  %esi
      popl  %ebp
      ret
```

Appendix E

Function Inlining

The Assembly code (x86 processor) below depicts the resultant code for the case study in Section 5.2.4, after applying to the original C code an aspect to perform function inlining using CoSy (C. Website, 2011).

```
  .text
      .align 4
      .global
      factorial
  factorial:
  // -- bb0 --
  .L1:
      //   outargsize
64
      pushl  %ebp
      movl
      %esp,%ebp
      pushl  %ebx
      movl
      8(%ebp),%ebx
      movl   $1,%eax
      movl   $1,%ecx
  // -- bb1 --
  .L2:
      cmp
      %ebx,%ecx
      jg     .L5
  // -- bb2 --
  .L3:
      imull
      %ecx,%eax
  // -- bb3 --
  .L4:
      incl   %ecx
      jmp    .L2
  // -- bb4 --
  .L5:
  // -- bb5 --
  .L6:
```

```
      popl   %ebx
      popl   %ebp
      ret
      .align 4
      .global
      main
  main:
  // -- bb0 --
  .L7:
      //   outargsize
64
      pushl  %ebp
      movl
      %esp,%ebp
      pushl  %esi
      pushl  %ebx
      xorl
      %eax,%eax
      xorl
      %ebx,%ebx
      jmp    .L12
  // -- bb1 --
  .L8:
  // -- bb2 --
  .L9:
  // -- bb3 --
  .L10:
      addl
      %edx,%eax
  // -- bb4 --
  .L11:
      incl   %ebx
  // -- bb5 --
  .L12:
```

```
      cmp    $10,%ebx
      jge    .L18
  // -- bb6 --
  .L13:
  // -- bb7 --
  .L14:
      movl   $1,%edx
      movl   $1,%esi
      movl
      %ebx,%ecx
  // -- bb8 --
  .L15:
      cmp
      %ecx,%esi
      jg     .L8
  // -- bb9 --
  .L16:
      imull
      %esi,%edx
  // -- bb10 --
  .L17:
      incl   %esi
      jmp    .L15
  // -- bb11 --
  .L18:
      xorl
      %eax,%eax
  // -- bb12 --
  .L19:
      popl   %ebx
      popl   %esi
      popl   %ebp
      ret
```