Universidade do Minho

Pedro Miguel Ribeiro Martins

A Domain Specific Aspect Language for Matlab

Extending Matlab with Aspects

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Prof. Dr. João Alexandre Saraiva

Novembro 2010

Acknowledgements

First, I would like to thank my supervisor, Ph.D João Saraiva due to his amazing skill of always knowing the right words for every moment. I really hope that I can benefit from those words for a long time, both as a student and a friend.

Then, of course, I must thank my parents, Isabel and Zeca, for being the two pillars of myself as a student and as a human being.

My friends also played an important role, namely Helder, Pinto, Mitra, Carlos, Pedrinho and Mariana. If what Erik Van Wyk told me once is true, that you must sometimes let your subconscious lead the way, than I must really thank them for providing such a great environment for that to happen.

I would also like to acknowledge Jácome Cunha and João Paulo Fernandes due to their great help while receiving me as a new researcher in the laboratory where they already worked. They made my residence there easier and more comfortable than I could possibly imagine.

Miguel Monteiro, João Cardoso and João Miguel Fernandes were always available to review my work and help when I had problems. They always had something important and interesting to point while answering to my questions, and because of that I am truly thankful.

I would also like to thank Don Batori for helping me with **AspectJ** and allowing me to use his code in some of the examples.

To my brother Luís, which is my best friend, I dedicate this work.

Finally, I would like to thank the AMADEUS project, about Aspects and Compiler Optimizations for Matlab System Development, under FCT contract PTDC/EIA/70271/2006.2008-2010 for funding and supporting this research.

Resume

This MSc thesis's goal is to develop an aspect-oriented language extension for the **Matlab** language, and to experiment different implementation features.

Although **Matlab** supports some useful features, such as allowing certain variables to represent different data types during runtime or enabling the same function to be called with different number of variables and types of arguments, truth is when a user wants to monitor variables or implement handlers to control certain behaviors, all these options are tedious, difficult to carry and make errors hard to control and avoid. Furthermore, every time a user needs to implement these features, he needs to both create new code and make intrusive changes in the old one.

This project aims to embed aspect oriented features in **Matlab**, for help modeling and exploring certain features in a controlled manner, without intrusive code changes.

Resumo

O objectivo deste projecto é desenvolver uma linguagem orientada a aspectos para o programa **Matlab** e respectivo ambiente de programação, e experimentar diferentes características de implementação da mesma linguagem.

Embora o ambiente de programação **Matlab** suporte técnicas que permitem a fácil criação de software neste paradigma, como o facto de determinadas variáveis poderem representar diferentes tipos de dados durante a execução de um programa ou permitir que a mesma função possa ser chamada com um numero diferente de argumentos (que podem, inclusive, ter tipos diferentes), a verdade é que quando é necessário fazer tarefas de manutenção ao código, como manter um registo da utilização de variáveis ou controlar o comportamento de ciclos, estas tarefas são obrigatoriamente executadas de forma rudimentar e com alterações intrusivas no código original, que o tornam desorganizado e são susceptíveis de criar erros de execução e compilação.

O objectivo deste projecto é criar, usando o paradigma de programação orientada a aspectos, um mecanismo que permita explorar características e comportamentos de programas **Matlab** de forma organizada e estruturada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A big effort has been put into programming machines easier, and software technology has evolved quickly, from a state where humans had to write direct, technical commands to modern languages, more focused in what the user really wants to do rather then how the machine will exactly proceed. What all this evolution tries to do is to shorten the distance between programming languages to the ones we use daily, i.e., the natural languages. Modern programming languages contain a fair amount of constructors and are built in such a modular and concise way that makes information transmission easy and clear.

To increase software's productivity, a number of features had been developed, such as data type systems, abstraction, high order functions and modularity, which increase the software developers productivity. And even these features had seen some improvements. While in the 80's modularity was all about separating the code into different files, that concept had evolved to Objects Oriented Programming and, more recently, to Aspects Oriented Programming (AOP).

The popularity of Aspects Oriented Programming is motivated by the state of art programming technologies, which although represent a big progress of evolution into building software as modular as possible, fail to create perfectly sectioned and structured software because they assume a complete system is made entirely out of hierarchical compositions of smaller units, forgetting system's properties. It is possible to roughly separate a system into two parts: the functional part, that represents the exact functionalities of the system, and another part that represents important requirements and properties, not directly related to the main function. Most modularity technologies are related only to the organization of the functional part, being the rest of the code intermingled in the functional modules. AOP

aims to clearly modularize this 'secondary' code.

This lack of a powerful modular system in modern programming languages has motivated the embedding of AOP in different programming languages [SLU05, LDS05, VSS09] and programming paradigms [RM07, Mar10]. In this thesis, we present an AOP extension to the **Matlab** programming language.

Matlab is a numerical computing environmental heavily adopted by both industry and academia, and used in many areas such as scientific computing, control systems, signal processing, systems engineering, simulation, etc. **Matlab** is furnished with a complete integrated environment to develop projects, which also includes debug features.

There a big number of features available in **Matlab**, such as operator overloading, function polymorphism and dynamic type specialization. Operator overload enable the same operator to be called with different data types. For example, in **Matlab** one can call the operator '+' either with integers or with arrays. Functions polymorphism allows the same function to be called with different arguments and data types. And by supporting dynamic type specialization, variables can represent different data types during execution.

However, despite allowing fast code production, code maintenance and edition is still very similar to other programming environments and languages. To use **Matlab** features, the programmer is forced to explore typical behaviors such as direct code changes, the use of handlers and other tedious, cumbersome and consequently error-prone tasks.

Next, we present a picture (Fig. 1.1) that shows the addiction of control code to part of a **Matlab** function. In this function, we can see lines of code (highlighted) whose only function is to check if a variable exceed a certain value and warn the user about it. The problem is that there are so many of this checking code that we can barely see the main function. This checking pollutes the main, functional part of the code. And this process gets worse as we grow both the system or the quantity of 'auxiliary' information, that is independent from the main function procedures. This problem is not related to **Matlab** itself, but rather to a big number of programming paradigms.

To solve this problem, AOP allows us to separate the functional part of the system to the parts of code that are only there to serve 'auxiliary' functionalities. If applied to **Matlab**, one could create the main functional code in one file and on another file could create the code related to the second functionalities, and let a simple software, known as 'weaver', combine the two files and create our final solution. In this paradigm, auxiliary and supporting functions are isolated from the main program logic. A program is broke

```
...
for j = 1:1:N
        if sum >= 10000 warning('sum too big! %f',sum);
          end
        if sum >= 10000 warning('A(j) too big! %f',A(j));
          end
       sum = sum + A(j) * B(j+N);
end
 if sum >= 10000 warning('sum too big! %f',sum);
   end
outa(i) = sum;
...
```

Figure 1.1: ***Matlab*** *function with control code, taken from [CDM+10]*

into distinct parts (the aspects), through the implementation of cross-cutting expressions that detect the parts of the main code where auxiliary functions will be inserted to create the final executable code.

An AOP mechanism would allow the programmer to write (abstract) statements in the form of:

Find where the 'for' cycle is
    Insert test to 'sum' (insert test inside the cycle)
    Insert test to 'A(j)' (insert test inside the cycle)

Find where 'out(i)' is assigned to 'sum'
    Insert test to 'sum' (insert test before the assignment)

If we connect, using a software called 'weaver', this statement with the following **Matlab** program (which does not have tests or 'polluting' code and is composed only by the important, functional lines):

```
...
for j = 1:1:N
      sum = sum + A(j) * B(j+N);
end
```

```
outa(i) = sum;
...
```

we would obtain the program that we see in Fig. 1.1. Through this mechanism, the programmers can now cleanly and efficiently structure the software without having to create disorganized and confusing functions.

By using AOP, we can clearly and easily organize the whole system by clearly separating the main code from secondary code. It is easy to understand how this whole process helps in creating, editing and maintaining code. Implementation becomes easier because the creation of code can be independent from the implementation of additional features. Editing and maintaining the code becomes easier because it is easier now to detect the implemented code, and to find the pieces we want to edit/maintain.

This paper presents the main concepts of a domain-specific aspect language (DSAL) for specifying transformations of **Matlab** programs. By using the proposed aspect extensions, the programmer is able to specify program transformations and optimizations without having to manually edit the original code. Here, we present an expressive language whose aim is to allow the production of **Matlab** code to be more easy and elegant.

1.1 Contents

The main body of this report is divided as follows.

- Chap. 2 describes **Matlab** and explains the main features and characteristics of programming in such environment.
- Chap. 3 introduces Aspect Oriented Programming by showing how the whole process of producing, editing and maintaining code becomes easier when this paradigm is available
- Chap. 4 studies **AspectJ** a famous implementation of AOP, and presents some examples and particularities of this language.
- Chap. 5 represents the main objective of this work, and shows a DSAL that can detect point cuts in **Matlab** and perform code alterations.

- Chap. 6 shows bibliography that is somehow related to this work, either by introducing AOP or by introducing transformations or more general work around **Matlab**.
- Chap. 7, summarizes what has been achieved and what could be done next.

You could also use an appendix for listings of **AspectJ** examples (App. A).

Chapter 2

Matlab

> *"MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using the MATLAB product, you can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran." in* [1]

In this chapter we introduce the **Matlab** language and environment, as well as some exclusive features and programming particularities when developing in **Matlab** such as function polymorphism, logging and versioning.

The last section shows some of the main difficulties when translating and using **Matlab** in embedded systems.

2.1 Introduction to Matlab

Matlab is a numerical computing language and interactive environment that includes a large number of pre-built functions that support a wide range of applications [HH05], from signal and image processing, to communications, control design, test and measurement, financial modeling and analysis, and computational biology[2]. Moreover, there is support to a number of special purpose functions, called toolboxes, that extend the original environment to solve particular problems related to the application areas described above.

[1] http://www.mathworks.com

[2] http://www.mathworks.com/products/matlab/description1.html

There is a big focus on the **Matlab** programming language on mathematical calculus. Indeed, the pre-programmed mathematical functions, that go from linear algebra and statistics to Fourier analysis and numerical integration, plus other useful features described below, justify why it is such a widespread tool, differing from C, C++ and Java, for example, where predefined data types and functions only represent real numbers and work only with real arithmetic.

The basic data type in **Matlab** is a multidimensional array of complex numbers, with both real and imaginary parts stored in double precision floating point arithmetic conforming to the IEEE standard [IEE85]. When dealing with real numbers the imaginary part is not stored. Almost all computation is performed in floating point arithmetic, and **Matlab** automatically uses complex arithmetic when needed. Variables, both vectors and scalars, are declared in the same time they are assigned, and the size of arrays is dynamic and automatically expanded every time a new assignment is made so it makes sense.

Matlab includes a large and user extensible collection of functions, which take zero or more inputs and return zero or more outputs. However, there is a clear, enforced distinction between input and output when programming in **Matlab**: input parameters appear on the right of the function name, surrounded by parenthesis and the output on the left, surrounded by square brackets. Furthermore, the number of variables a function supports is not static, as the same function supports different calls, where the input changes both in number of variables and their content, and returns different outputs. In fact, for a given function, the number of variables and type or arguments changes the behavior while keeping the statement valid. This feature is called method overloading.

A built-in debugger is included, with a tool set that helps isolate and control variables during runtime. These tools include breakpoints, the program execution controls in the Debug menu, variable datatips and the Editor Context menu [Fot09].

Matlab supports Object Oriented Programming (OOP), which introduce the concepts of class, object and inheritance, but apart from architecture and designing, there is no useful new advantages to code control or debugging, which still implies direct changes in the source code. Garbage collection [Fot09] is not supported, first because of the complexity associated with managing object lifecycles, and lastly because it makes testing and debugging an application more difficult [Fot09]. For instance, one can stop the workspace and see all the activities that took place, while a garbage collection engine could destroy steps that cannot be repeated.

There are more than 1200[3] books that, alone, prove the wide usage of **Matlab** and associated functions and tools.

2.2 Programming in Matlab

Matlab is a mathematical-oriented programming language that uses high-level operations, which means that the language does not work with data types as they exist in other programming languages. Operations in **Matlab** can be made in two ways: with a direct sequence of instructions called scripts (Fig. 2.1) or by structuring the code in functions (Fig. 2.2). Recently the language has been extended to support classes. Fig. 2.2 shows the code of a **Matlab** script that takes the sum of two numbers and finds if it is bigger than a third number:

```
% MATLAB script
a=5;
b=6;
comp = 16;
if(a+b > comp)
        disp('The sum is bigger!')
else
        disp('The sum is smaller!')
end
```

Figure 2.1: *A simple **Matlab** script*

There are a few particularities in this **Matlab** code. The first thing to notice is that types are not declared. From the assignment to variable 'a' in the code, one may conclude that 'a' can, in this case, be stored as a scalar of integer type. However, internally, the variable 'a' will be stored as a single-element array of type double. Array variable shapes are inferred when the code is executed. Hence, there are the assignments to variables that expose in runtime the shape of those variables. In a certain point of a **Matlab** the existence of 'c=5;' implies that 'c' can be a single-element array, while in other point of the code, the existence of 'c=[1 2 3; 3 4 5; 6 7 8];' implies that 'c' refers now to a 3x3 matrix. This dynamic features of **Matlab** helps development, but complicates the translation to non-

[3] http://www.mathworks.com/support/books/index.html

dynamic languages, and especially to implementations where the overhead to maintain this dynamic behavior is not acceptable.

The definition of functions in **Matlab** is very similar to C. In Fig. 2.2, we present the **Matlab** function with name 'sumvals' taken from [CBHV10]:

```
function s = sumvals(start, step, stop)
i = start;
s = i;

        while i < stop
                i = i+step;
                s = s + i;
        end
end

a = sumvals(1, 1, 10^6);
b = sumvals([1 2], [1.5 3], [20^5 20^5];
```

Figure 2.2: *Sumvals function in **Matlab***

The function 'sumvals' is designed to sum numbers within a range of values. There are a few differences from **Matlab** and C functions. First, the variable returned by the function (in this case is 's'), is declared in the function definition, instead of having a special primitive to do this, like the 'return()' in C. In fact, **Matlab** functions may return more than one variable. Secondly, functions in **Matlab**are polymorph, as we see in Sec. 2.2.1.

2.2.1 Polymorphic Functions in Matlab

Functions in **Matlab** are polymorphic, which means functions can handle values of different data types using a uniform interface. The function 'sumvals' (Fig. 2.2), for example, can be applied to both scalar and arrays types of values. Specifically, to the variable 'a' will be assigned the scalar `5*10e11`, and to 'b' will be assigned the value `1.0e12`, which is a `1x2` floating point array. **Matlab** functions can even be called without passing all the list of parameters.

This implies that, when translation a specific behavior to C, **Matlab** introduces a few new challenges (Table. 2.1).

Table 2.1: Challenges when passing from **Matlab** to C

Issue	Solution
Function 'sumvals can either accept scalars or vectors as arguments	The solution is to write two functions in C, where each one accepts one type of argument
The '$*$' operator is also polymorphic, which means it works with scalars and vectors as well	C already includes a '$*$' operator, but it only works with scalars. The solution is to re-implement the operator so it can handle vectors

2.2.2 Logging in Matlab

Logging is a specialized method to record a program's execution information. This is typically used by programmers to trace and debug a program during the development cycle. Imagine that, for the function 'sumvals' described above, is important that the variable 'start' is never negative. And by the way, we would like to know, for debugging purposes, the behavior of the variable 'i' during the program flow. Such changes can be implemented by introducing logging instructions in the original program, as seen in Fig. 2.3.

```
function s = sumvals(start, step, stop)

if(start < 0)
display('Attention, start is negative');

i = start;
disp(i);
s = i;
        while i < stop
                i = i+step;
                disp(i);
                s = s + i;
        end
end
```

Figure 2.3: *Logging in **Matlab***

These alterations are easy to perform manually for small-size programs. But for medium and large size applications this can be a tedious and error prone task. In addition, the code

for logging may not be considered to be included in the final implementation, and in this case the programmer needs to eliminate that code from the original **Matlab** code. This is also a tedious and error prone task.

2.2.3 Matlab in Embedded Computational Systems

With a higher-level of abstraction than, e.g., the C programming language, **Matlab** allows the programmer to invest more time on the problem-solving than on implementation issues. However, such higher abstraction level makes more difficult the job of compilers as to produce efficient code (such as C) they need now sophisticated analysis and optimizations, especially for data type and shape inference. Shape and data type inference is not easy and in many cases it is not feasible to be performed by the compiler. In addition, embedded systems have multiple target architectures, different embedded processors and co-processors, which expose the need to generate implementations aware of the specificities of the target architecture.

One possible avenue to solve the problem of data type and shape resolution and the generation of different implementations according to the target domain and architecture is the use of aspect- oriented programming. AOP can be used to extend **Matlab** programs with transformation and specialization rules that help the compiler to achieve more efficient code considering a certain target system.

Let us consider the function 'sumvals' again. This functions works with either scalars and arrays, and the values operated on could be either integer, real or complex. Consequently, transforming this code to other programming language can be challenging, since type information is not explicit. To generate efficient code we not only need type information, but also we may need different versions of the function specialized to each of the possible types.

In Fig. 2.4, we present a redefinition of the function 'sumvals', where the data types are explicitly defined by the programmer, which will make the generation of code more efficient when we are working on an integer domain.

```
function s <scalar int> = sumvals(start <scalar int>,
 step <scalar int>, stop <scalar int>)

i <scalar int> = start;
s <scalar int> = i;

        while i < stop
                i = i + step;
                s = s + i;
        end
end
```

Figure 2.4: *Function sumvals with type information*

2.3 Summary

Matlab is a very complete and powerful development environment however, as we have seen, some of the particularities that make the language practical and easy to use such as direct variables attributions or the advantage of not having to declare types also makes translating it to for example C, very difficult because most languages are more focused and static.

The powerful features of **Matlab** have the disadvantages of making the language hard to translate and embedded in other systems, while traditional operations such as logging and tracing still have to be made the traditional, hard-coding way.

Chapter 3

Aspect Oriented Programming (AOP)

Aspects-Oriented Programming was developed to make it possible to clearly express those programs that OOP (or any of the procedural-based approaches) fails to perfectly support.

3.1 Introduction

Dijkstra [Dij97] suggested separation of concerns as a problem-solving technique, although he never specified how to achieve it. To support this concept, most approaches to organize and create software systems use some form of composition or modularity [Par79]. The solution is to split the original problem into subparts that can be solved in a way more or less independent from the original, simplifying the solution to programmers, that can address the subparts rather than have to directly construct the final solution. These parts can be completely independent between them or have complex inter-relations, but the main idea is that the composition of all those parts represents the final software system.

There are different programming paradigms that have different mechanisms to support this idea of modular software units (such as the modular system of the functional languages ML or Haskell, or the Objects in OOP), with a large number of design practices and notations. Some of these mechanisms even support the split of those units in sub-units (such as subclasses, superclasses in Java), coping with the idea of splitting a big problem into sub-problems and sub-sub-problems, decreasing the complexity by each iteration. In short, programming technologies had evolved to support abstraction, modularization and reuse of code.

Objects-Oriented Programming (OOP) is one example of a paradigm designed to aid in software engineering by creating an object model that provides a more capable way of handling the construction and implementation of software, by better fitting domain problems. The ambients of Objects-Oriented languages are highly sophisticated and complex, with a number of technologies such as refactoring, structure views, profiling, code completion, quick fixes, UML diagram generations, etc that provide tools to create perfectly sectioned software.

So far, as long as a software system is made entirely out of the hierarchical composition of smaller units, we are able to create perfectly sectioned software by using the paradigms and technologies described before. But software systems are more complex than that. They are not made entirely out of smaller components with inheritance relations. They also have properties, who can not be sectioned so easily and destroy the programming model provided by OOP.

But what are these problems, and how do they corrupt the perfectly sectioned software created in OOP?

3.2 Cross Cutting Concerns

Software systems are full of pieces of code whose aim is to fulfill some requirements, even if they are not important to the main function of the module they are implemented in. On an ATM system, for example, a module that is responsible for making deposits on an account may have pieces of code that keep checking for security and network requirements, even if they are not related to the main objective of making a deposit. They are not a functional part of the system, but they are needed nevertheless.

These properties, which can be application specific, such as defining financial products or configuring network services [CE00], or more general, such as scheduling, resource allocation or performance optimizations [CBE+00] do not emerge randomly, but appear during execution, and do not necessarily align with the functional parts of the system. In the Objects Oriented Paradigm (OOP), for example, these are not objects or classes.

To this properties, that appear in different parts of the system and are required in various operations but can not be cleanly decomposed in both design and implementation we call crosscutting concerns (CCC).

Crosscutting concerns are not a modular part of a system in OOP (or most of the other

programming paradigms), and must be inserted along the system modules. When a developer implements a module, he must not only implement the program logic, he must also know all the concerns related to it and implement them as well, instead of single-mindedly fulfill a particular problem. This incapacity for the Object Oriented Paradigm to address a separation of concerns was studied before, and was called "tyranny of dominant decomposition" [KLM+97, OT01] due to the fact that OOP provides only a dimension along where concerns can be separated.

Fig. 3.1 shows an example often used to illustrate the effects of crosscutting concerns in Java, including on the seminal paper on **AspectJ** [KHH+01], although this particular example was taken from [MJS10]. Here, we see two classes, **Point** and **Line**, that comprise an abstract declaration - **FigureElement**. In this diagram there is also included an functionality to display figure elements in a graphical environment. This graphical representation must be updated upon any alterations to figure elements or respective subclasses.

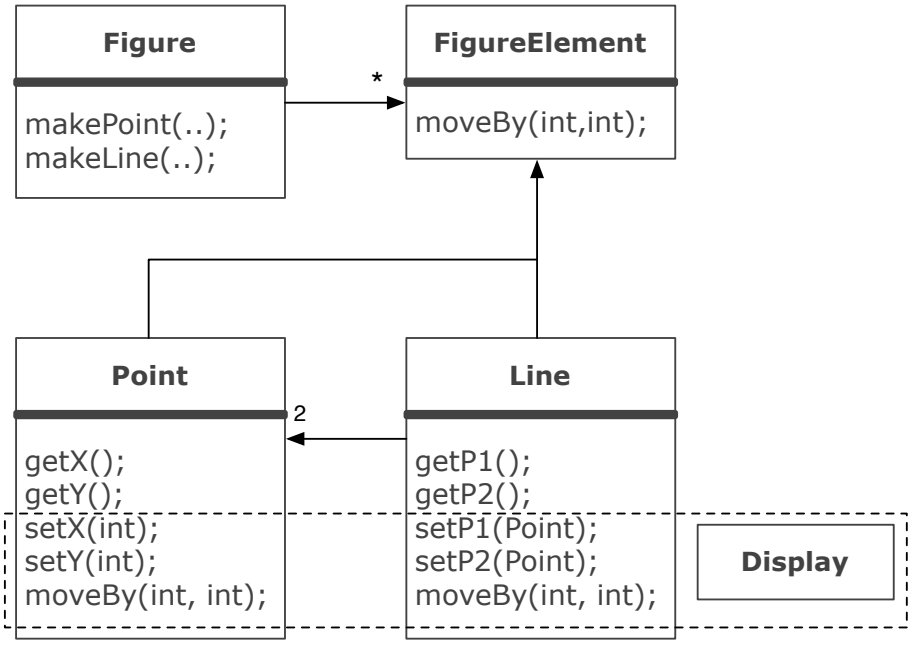

Figure 3.1: *Crosscutting concerns in Java, taken from [MJS10].*

The problem is that all operations that have an impact on the display of the figures must undergo invasive changes, which represent references do the class *Display*.

When we try to decompose the modular units of the system (in this example, the classes **Point** and **Line**) we see that this classes's structure is not only composed by functions

```
public class Point implements FigureElement{
    private int _x, _y;
    private Display _display;
    public Point(int x, int y) {
    _x = x;
    _y = y;
    }
    public Point(int x, int y, Display display) {
    this(x, y);
    setDisplay(display);
    }
    public void setX(int x) {
    _x = x;
    _display.update(this);
    }
    public void setY(int y) {
    _y = y;
    _display.update(this);
    }
    public void setDisplay(Display display) {
    _display.update(this);
    }
    public void moveBy(int dx, int dy) {
    _x += dx;
    _y += dy;
    _display.update(this);
    }
}
```

Figure 3.2: *Example of CCC's in Java, taken from [MJS10].*

directly related to the class itself, but also to functions with code related to a concern, which in this case is a graphical representation. Fig. 3.2 show the source code of one of this classes - **Point**, with the invasive code highlighted in gray. In this system, although not represented, there are also invasive code changes in the class **Line** and in the class **Figure**.

In Sect. 3.3 we present a more complete example on how the creation of a software system from source gets more and more complex by the constant addition of CCC's who gradually pollute the code and make it more confusing..

As requirements are introduced along the code, wherever they are needed, it begins to be illegible as more and more concerns are implemented into each modular unit. And poor understandability leads to poor maintainability and makes enhancements harder to introduce, which is specially true for large systems.

As we have seen, a crosscutting concern is a property of the system that can not be cleanly encapsulated [KLM+97]. But how extreme can this process be on the final system, and how, by each production iteration does this situation gets worse?

3.3 Code Tangling and Code Scattering

Tangling and Scattering occur when the interactions between the modules of the system and the concerns are not well defined (the concern is not well modularized). In this cases, one of two things may happen:

- A concern can be implemented in various parts of the code. For example, returning to the ATM system example, all the main modules require network tests before doing the required operation. What happens than is that the same code (in this case, network testing code) spreads across various parts of the system. If there is a network requirement change, for example, one has to search all the system for all the places where that code appears and change it accordingly. When the same concern affects multiple parts of the system and is implemented several times we call it **code scattering**.

- The same module may require several concerns implemented. Sometimes the same software module may have different concerns, related to network requirements, research allocation or concurrency, for example. This requires higher abstraction from the programmer that has to know very well all the concerns when he is implementing the module. Then the same system modules needs several concerns implemented to correctly work, we call **code tangling**.

3.3.1 How code becomes tangled and scattered

Next, we present an illustrative example (Fig. 3.3) in order to show how the tangling and scattering process appears, how it evolves and how it ends up increasing the efforts to

maintain and edit the code. This example shows a small program to manage the location of books within different spaces in a company. It is composed by two methods: (i) **register**, to insert new books in the system and associate them with a locale, and (ii) **locate**, where we provide a key string which is matched with the information in the system and shows the first successful result. In addition to these basic functionality, the system must also be accessed through a network and process requests concurrently. This example, which is represented in Fig. 3.3 was adapted from [Lop97], is presented in Java and shows a complete definition of the classes and methods required, although it is not ready to deployment yet.

This system's specification also implies that BookLocator should perform several requests concurrently. Since the methods **register** and **locate** use and update the same instance objects, additional code is necessary in order to avoid inconsistencies. Therefore, concurrency can be applied by temporarily blocking all "write" accesses anytime a "read" is executed within the system (various "reads" can be performed at the same time), while all "write" accesses to the system must completely block all other services. In this case, to avoid inconsistencies various instances of **locate** can be performed at the same time, while blocking all **register** methods, and each **register** method must be performed alone.

Fig. 3.4 shows an implementation of BookLocator with concurrency, where all the new code necessary to avoid inconsistencies is highlighted in grey. Only the class BookLocator is shown since the other two classes remain unchanged. A few steps are needed in order to avoid multiple conflicting accesses, which imply inserting peaces of code in specific points of the original code: 1) define extra variables; 2) insert a number of checks at the beginning of each entering method to ensure exclusive access to the target object and 3) alter all exit points of the methods to change the state of the target object. These peaces of code inserted into the methods could be structures calls, using modularity to avoid code repetition, as we present below:

```
protected synchronized void after_write()
    --activeWriters; notifyAll();
```

This way some of the alterations could be simpler to introduce and the final program would result in less lines of code. The methods that return objects are particularly dangerous since the return expression may affect the state of the object and should be taken in consideration (see the method *locate*). The complexity of the final program has increased with this alterations, as it consists now of many more different and connected parts, diluting the code. Two original concerns: maintain a books database and ensure concurrency

```
public class Book {
   // possible implementation of Books
   String title, author;
   int isbn;
   Project owner;
   OCRImage firstpage;
  public Book (String t, String a, int n) {
      title = t; author = a; isbn = n;
   }
  public String get_title() {
      return title;
   }
  public String get_author() {
      return author;
   }
   public int get_isbn() {
      return isbn;
   }
}
public class Location {
   //possible implementation of Locations
   public int building, room;

   public Location (int bn, int rn) {
      building = bn; room = rn;
   }
}
```

```
public class BookLocator
{
   //One possible implementation
   //books[i] is in locations[i]
   private Book books[];
   private Location locations[];
   private int nbooks = 0;
  public BookLocator (int dbsize) {
      books = new Book[dbsize];
      locations = new Location[dbsize];
   }
   public void register (Book b, Location l)
   throws LocatorFull {
      if (nbooks > books.length)
         { throw new LocatorFull(); }
      else {
         // Just put it at the end
         books[nbooks] = b;
         locations[nbooks++] = l;
      }
   }
   public Location locate (String str)
   throws BookNotFound {
      Book abook = books[0];
      int i = 0;
      boolean found = false;
      while (i < nbooks && found == false) {
        if(abook.get_title().compareTo(str)==0 ||
        abook.get_author().compareTo(str)==0)
        {
           found = true;
        }
        else { abook = books[++i]; }
      }
      if (found == false)
         { throw new BookNotFound (str);}
      return locations[i];
   }
}
```

Figure 3.3: *Simple Implementation of BookLocator.*

between processes resulted in an unique block of code that became harder to understand. And we have not yet applied all the desired features, since the last piece of specification implied the system would be available through a network.

Unlike the coordination issue, there is not a common understanding of what is the best

```
public class BookLocator
{
   // One possible implementation
   // books[i] is in locations[i]
   private Book books[];
   private Location locations[];
   private int nbooks = 0;
   protected int active_readers = 0;
   protected int active_writers = 0;
   public BookLocator (int dbsize) {
      books = new Book[dbsize];
      locations = new Location[dbsize];
   }
   public void register (Book b, Location l)
   throws LocatorFull{
   while (active_readers > 0
   || active_writers > 0)
   {
      try { wait(); }
      catch (InterruptedException e) {}
   }
   ++active_writers;
   if (nbooks > books.length)
   {
      --activeWriters;
      notifyAll();
      throw new LocatorFull();
   }
      else {
         // Just put it at the end
         books[nbooks] = b;
         locations[nbooks++] = l;
      }
   --activeWriters;
   notifyAll();
}
```

```
// Continuation of class BookLocator
public Location locate (String str)
throws BookNotFound {
   Book abook = books[0];
   int i = 0;
   boolean found = false;
   location l;
   synchronized (this) {
      while (active_writers > 0)
      {
         try { wait(); }
         catch (InterruptedException e) {}
      } ++active_readers;
   }
   while (i < nbooks && found == false) {
      if (abook.get_title().compareTo(str)==0
      || abook.get_author().compareTo(str)==0)
      {
         found = true;
      }
      else { abook = books[++i]; }
   }
   if (found == false)
   {
      synchronized (this)
      {
      --active_readers; notifyAll();
      }
   throw new BookNotFound (str);
   }
   l = locations[i];
   synchronized (this);
   {
      --active_readers; notifyAll();
   }
   return l;
   }
}
```

Figure 3.4: *Implementation of BookLocator with concurrency.*

way to insert the network features [Lop97], but similarly to the concurrency implementation, inserting network capabilities in the system would increase the code dilution and the amount of information on a single block of code. As we can see, OOP can not keep the code organized by requirement - even with its modulation, abstraction and reuse capabilities the final result is a sequence of instructions as the language provides only a dimension along which concerns can be implemented.

Ideally, the separation and decomposition of concerns should create a structure that perfectly matches the whole system. However, this structure must be composed to create the final system, and such composition must be supported by the tools used. Consequently, failing to do so makes separation unpractical, leading to unorganized and confusing code since these aspects can not be modularized with traditional mechanisms.

3.3.2 Crosscutting concerns in Matlab

The presence and description CCCs in **Matlab** was provided by [CDM+10]. Because **Matlab** has a procedural nature, the decomposition units are typically functions or groups of functions, which are a much more limited mechanism for modularizing concerns than classes. Concerns are not as well organized in **Matlab** as in other programming systems, but they exist nonetheless.

```
function z = expo(x,n)
y = 1;
for i = 1:n
   y = y + x^i/factorial(i);
end
z=y;
```

```
function z = expo(x,n ,p )
P(1) = 1; Y(1) = 1;
for i = 1:n
   P(i+1) = P(i) + 1;
   Y(i+1) = Y(i) + x^i/factorial(i);
end
z = Y(n+1);
if (p) plot(P,Y)
end
```

Figure 3.5: *An example of crosscutting domain in **Matlab**.*

Figure 3.5 show an example of CCCs in **Matlab**, taken from [MJS10]. The extra code

presented on the right side of the image has the function of building a graphical representation. Note that in real-world examples usually this implementation is even more polluting to the code, since the programmers usually define extra parameters to the 'plot', such as legends and a title.

What is usually different from this example to the previous examples of CCC's in Java is that in this example we are not only in the presence of additional code intermixed with the original, functional code. In **Matlab** it is very frequent that a concern has a direct impact in the original code, creating different, modified code. In the example of Fig. 3.5, the code on the right (with the concerns implemented) no longer uses simple, scalar types in his computation, but rather uses vectors to produce the data that feeds 'plot'. The alterations required for such transformation are highlighted.

In their work, Monteiro et al. [MJS10] identified a potential list of CCC categories in **Matlab**:

- Messages and monitoring: messages to the user, warnings, errors, graphics visualization, monitoring, etc.;
- I/O data: reading data from file, writing data to file, saving an image, loading an image, etc.;
- Verification of function arguments and return values: default shapes and values for the arguments that may not be passed in certain function calls;
- Data type verification and specialization: check whether a variable is of certain type, configuring the assignment of data types to variables, etc.
- System: code that verifies certain system environment properties, to pause execution, etc.
- Memory allocation/deallocation: The use of the 'zeros' function is most of times used to allocate a specific array size. This avoids the reallocation for each new item to be stored in an array. Use of the 'clear' instruction that appears in some **Matlab** functions is another example.
- Parallelization: use of parallel primitives such as 'parfor';

- Design space exploration: code to explore different specializations, different algorithms to solve the same problem, to find the number of iterations needed (e.g., to be above a certain precision).

- Dynamic properties: constructing inline function objects (inline), executing a string containing **Matlab** expressions ('eval'), etc.

```
function EO = gaborconvolve(im, nscale, norient, minWaveLength,
        mult, sigmaOnf, dThetaOnSigma, feedback )

        if nargin == 7
        feedback = 0;
        end
        ... original code removed
        if ~isa(im,'double')
                im = double(im);
        end
        ... original code removed
        clear x; clear y; clear theta; % save a little memory
                ... original code removed
        for o = 1:norient, % for each iteration
                if feedback
                fprintf('Processing orientation %d ', o );
        end
        ... original code removed
        end
        if feedback, fprintf('          ');
        end
```

Figure 3.6: *Examples of CCC's in **Matlab**, taken from [MJS10].*

The illustrative example of Fig. 3.6, from the same authors, show some examples of potential CCC's that are in some of the categories described. In this example we can see: messages to the user ('fprintf'), argument verification ('nargin'), freeing memory ('clear') and class verification ('isa'). In this particular example, the code highlighted in gray indicates code which has the potential to be extracted into aspects.

3.4 Aspects Oriented Paradigm

By selecting an appropriate group of domain-related concerns to a problem, AOP achieves a good balance between the localization and management of relevant issues, complexity and redundancy. Aspect Oriented Programming gives to crosscutting concerns what OOP has given to objects: encapsulation and inheritance, language mechanisms that clearly capture the software structure [KHH+01].

3.4.1 Aspects

In the Aspects Oriented Paradigm, cross-cutting concerns are called aspects, which do not usually belong to the system's functional decomposition, but are properties that affect the performance or requirements of the various components. Aspects are, similarly to classes in Java, a unit of modularity, encapsulation and abstraction [CCHW04] although unlike classes, aspects can be used to implement crosscutting concerns.

Aspects are independent from the other modular units of the system. If we are talking of a system in Java for example, an aspect can be presented in part of a class, in all the methods of a class or in various classes of the whole system, as seen in Fig. 3.7.

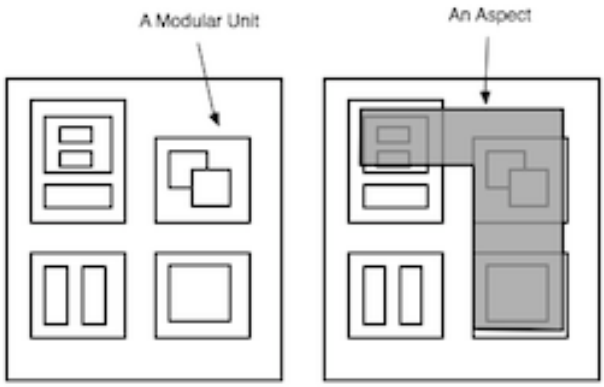

Figure 3.7: *Modular versus aspectual decomposition, taken from [CE00].*

3.4.2 Definition of AOP

With the terms explained above it is now possible to clearly define the goal of AOP: support the clean separation of components and aspects in the code. In AOP [KLM+97], by

selecting an appropriate group of domain-related concerns to a problem, a good balance between the localization and management of relevant issues, complexity and redundancy is achieved. Aspect Oriented Programming gives to crosscutting concerns what generalized procedure languages had given to components: encapsulation and inheritance, language mechanisms that clearly capture the software structure [KHH+01] and compose them to produce the overall system.

In AOP it is possible create instructions such as: "Realize a specific concern is needed and perform the necessary alterations to do it". In the BookLocator example on Fig. 3.3, a possible statement would be: "Whenever a method writes information in BookLocator, lock all other writes". This new approach is completely different from traditional programming paradigms because it breaks with the local demands, and programs organization can now be made in the most appropriate form for maintenance and edition instead of a single block. What's more, the original code does not need flags to mark these new instructions - the AOP language is self-capable of localize and perform the alterations.

> *"AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers." in [FF00]*

Statements in the form of [FF00]:

> In programs P, whenever condition C arises, perform action A.

are now available to the programmers, where P is a traditional coded program. The introduction of this new statement introduces three new concerns to programmers and designers: i) which conditions C can we specify, and if they are static or dynamic (happen at run-time), ii) how do the action A interacts with the original program and iii) how is this new statement and the original program P intermixed (this is called the weaving process, performed by a "Weaver").

AOP not only contributes to the organization of the code, it can also contribute to it's safety since aspects hide not only how something is done, but also when. Error handling in a Graphical user interface (GUI) [CCHW04] for example, has some critical implementation differences using aspects or OOP. The control flow of an user interface must handle errors in a manner that if they are originated by the user, they should be flagged to a dialog box. On the other hand, internal errors should only be kept in a log. Aspects not only make

this simple to implement, but their absence during programming means that everywhere an error might occur function calls must be made to handle it and error controlling policy is leaked. This is specially critic in other systems, such as security or banking systems.

3.5 Instances/Incarnation of AOP

3.5.1 CaesarJ

> *"**CaesarJ** is an aspect-oriented language which unifies aspects, classes and packages in a single powerful construct that helps to solve a set of different problems of both aspect-oriented and component-oriented programming."* [AGMO06]

CaesarJ [AGMO06] represents a new aspect-oriented programming language based on abstraction, information hiding and minimization of dependencies. In **CaesarJ**, aspects are designed as components, which improves abstraction and reuse, rather than on the physical separation that appears on other AOP implementations. **CaesarJ** compiles to conventional java bytecode, so it can be used everywhere java can, and can be used to improve the seeding of existing Java projects or in the development of new ones.

Modularization in **CaesarJ** is made through the use of multiple collaborating classes. After the definition of collaboration interfaces, the developer implements the component using the pre-defined collaboration interface. The binding of the component to the application is made in a different module. Information hiding is achieved because the application specific concepts do not know the component implementation (it is abstract), while the component implementation is hidden by the collaboration interface.

On the other hand, the developer can define base class collaborations and improve them with new features in its sub collaborations with the use of virtual classes.

Features from different developers can be easily merged by applying mix-in composition on the sub collaborations.

3.5.2 AspectJ

AspectJ was the first well known implementation of AOP and is considered a quasi standard language for Aspects Oriented Programming. Due to it's relevance we will introduce it in detail in the next chapter and explain it's main characteristics.

3.5.3 LoopsAJ

LoopsAJ [HG06] is not another language implementation based in AOP. **LoopsAJ** is an improvement to the **AspectJ**'s capacity to capture pieces of a program control flow.

Loops are natural places to improve performance, yet, **AspectJ** does not have a join point for loops capture, and even if it is sometimes possible [HG06], the method resorts to refactoring of the base-code. To eliminate this inconvenience, **LoopsAJ** allows direct parallelization of loops without the need to use refactoring.

Based on a bytecode representation that recognizes the behavior, not the way the code is written, this new join point can expose data related to the data processed and the iterations. Iterator, the integer bounds, collections and arrays can be passed to the arguments if they exist and if it is possible to do so.

Since loops have no names, loop selection can not be based on the name, although, this selection is made according to a type pattern matching the data processed.

3.6 Aspects Oriented Software Development

Because AOP is not an independent paradigm, independent of any other, but rather a paradigm that aims at extending the features (mainly the modularity) of other paradigms, we have the problem of how to implement AOP on an already existing software system (e.g. legacy system). Because of this, we have the problem of dissecting programs in order to find where to cross cut the implementation. This problem, of course, does not exist when we are starting the construction of a new system.

3.6.1 Using an Aspect Oriented Language

The normal way to use AOP in the software development cycle. On this project, the cross cutting concerns were previously identified and characterized, and the programmers have a perfect idea of which part of the code should be implemented in the functional part of the system, and which pieces of code should be implemented in the form of aspects, on a different modular container.

3.6.2 Aspect Mining

When we try to apply the Aspects Oriented Paradigm to existing software, the big problem is to find which parts of the system should be dissected and transformed into aspects. To do so, there is a technique called **Aspect Mining** that aims at finding cross cutting concerns in the already written code. There are a big number of techniques to do so [Cc07, KMT07, CHJvdB10], but their main objective is always to find where code tangling and code scattering appear more in the code and how it can be modularized in the form of aspects.

After the identification of promising candidates to be extracted into aspects, one of the ideas is to use a refactoring process that can automatically extract code and transform it in modular units that represent aspects [MF06].

Aspect Mining in Matlab

To our knowledge, there is only one project that tries to automatically identify and characterize cross cutting concerns in **Matlab** [MJS10]. In this project, the authors use a 'token-based' detection technique to perform a tokenization of the source code and then use the tokens as a base to clone detection. The idea of this tool is to decompose **Matlab** code, by reading a number of *.m files (*.m is the **Matlab** source files extension) and compute a number of metrics, such as:

- Number of times a given function name appears in a given **Matlab** file
- Number of different function names appearing in a given **Matlab** file

The early results of applying this technique to a big number of **Matlab** files (a total of 19 repositories with 209 **Matlab** files) gave some interesting results. This results included

the usage of specific functions, such as the function 'size', with 15% of the global uses of the functions identifying aspects candidates, followed by 'error' (11.4%), 'zeros' (9.5%) and 'nargin' (6.9%). A more interesting result is that, if we measure the "pollution" of **Matlab** code by the number of lines of code that use all the functions captured, around 16.64% of code is a potential candidate to be modularized as an aspect.

This aspects-mining technique enabled the identification of a promising number of candidates to be extracted into aspects and, although this is not the aim of this thesis, it clearly shows **Matlab** has a good potential and there is a necessity to use the aspects oriented algorithm in this ambience.

3.7 Summary

Although traditional programming styles allow the modularizations of system's methods, the programmer is obliged to introduced system's properties by hard-coding in the main block of code. This leads to code tangling which results in increasing efforts to understand, edit and maintain the software. AOP introduces to the programming paradigms the availability to make quantified statements. This way, system's components and properties can be separated into two dimensions and code organization becomes more flexible.

While in this examples AOP is applied to OOP, it is an independent paradigm capable of being used in other programming styles.

Chapter 4

AspectJ

In recent years, Aspect Oriented Programming (AOP) has been a very active research field. AOP has been proposed in different programming paradigms (objects oriented, imperative, attribute grammars, etc), and we can find several aspect oriented programming languages and several extensions to existing programming languages in the literature [KHH+01, AGMO06, SLU05, LDS05, VSS09].

There are various instances of the Aspect Oriented Programming, being **AspectJ** the widely-used fe-facto incarnation of AOP, which supports the modularization of cross cutting concerns, in the form of aspects.

In this section, we present a brief introduction to **AspectJ**.

4.1 Introduction

AspectJ is a general purpose, aspect-oriented extension for the Java programming language, developed at Xerox PARC, which introduces the new concept of join point.

A join point is a well defined point in the program flow. Every time a method is used within a program flow, whether it is a constructor, a simple print or any other method represents events that occur during a function runtime and are, consequently, join points. Direct access to read and alter variables, however, is not a possible join point in **AspectJ** and must be done through a method.

A join point appears in any code and is used in **AspectJ** to specify the position where a certain aspect is to be implemented. These join points create interceptions between the

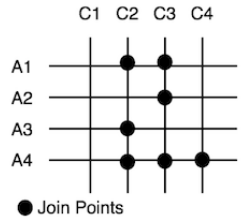

Figure 4.1: *The same aspect can be in more than a component (A4), and a component can have more than one aspect (C3). Each of this interceptions represents a join point, taken from [CBE+00].*

components of the original program (since we are talking about Java, this components represent mainly classes) and the aspects. These interceptions can happen in multiple ways, since an aspect can be part of a class or part of various classes and a class can have more than one aspect (Fig. 4.1).

AspectJ adds two new primitives: one to detect join points, called pointcut, and another to make alterations in the original program in the position marked by a pointcut, called advices.

4.2 Pointcuts

A pointcut detects and picks out certain join points and values at those points, acting as a filter that catches parts of the original program that match a certain criteria and blocks all the other. The pointcut **call(int increase())**, for example, matches all the methods called **increase()** that take no arguments and return an integer.

A pointcut can be anonymous or can be named (which is always preferred, expect on the simplest examples). Naming pointcuts not only makes the code clearer and easier to read, but also provides abstraction, encapsulation and reuse. It provides abstraction because you can call the pointcut by name rather than by the related expression, which you can even be unaware of (encapsulation). It provides reuse because we can call the same pointcut in other pointcuts.

A pointcut is generally declared as follows [CCHW04]:

$$[\text{visibility-modifier}]\ \text{pointcut name}\ (\text{ParameterList})\ :\ \text{PointcutExpr}\ ;$$

where PointcutExpr is the joint point defined to this pointcut. The parameter list is used

to publish information important for the action associated with the join point captured (we will see examples of this process in the next section).

A pointcut can be composed by other pointcuts by the use of logic operators, as shown in the next example:

```
pointcut arraychanges() :
        call(void add(int)) ||
        call(void remove(int));
```

This pointcut is composed by two smaller pointcuts, **call(void add(int))** and **call(void remove(int))**, connected by the use of the logic operator ||. In this example, a call to the pointcut **arraychanges()** is the same as calling the composing pointcuts individually. **AspectJ** supports the use of more logic operators, such as and/conjunction and negation.

A pointcut can be based on the explicit enumeration of a set of methods, but **AspectJ** allows this specification to be property-based, either by the use of wildcards or of special primitives.

The pointcuts:

```
call (public * account.*(..));

cflow (arraychanges ());
```

are both property-based. The first one, **call (public * account.*(..))** catches all the calls to the class **account's** public methods, independent of their return type (first '*'), of their name (second '*') or their arguments (represented by the '..' between the arguments brackets). The second pointcut picks all the join points that occur in the dynamic context of the join point **arraychanges()**, which means it picks out each join points that occurs between when an **arraychanges()** method is called and when it returns (either normally or by throwing an exception).

There are three possible pointcut categories in **AspectJ**. The first and most important is based on the type of joint point, whatever it is a method call, the execution of an exception handler, the initialization of a class, and so on. The second category filters join points based on their scope. These pointcuts can detect if the code is within a certain package or if it occurred during the control flow of a given operation. A third category matches join points

based on the context information of the join point itself. For example: is the currently executing object an instance of a given type?

4.3 Advices

A pointcut is an important primitive that allows us to detect join points within a program flow. But what to do after that join point is detected? That is the function of the advice primitive.

Each piece of advice is associated with a pointcut (named or anonymous) and specifies the behavior to be implemented in that join point. To support this, an advice declaration may contain parameters, whose references are in the body of the advice, although all parameters values must be provided by the related pointcut. For example if the advice for the pointcut **arraychanges()** from Sect. 4.2 had to insert the name of the array in a warning message, that information needs to be declared in the pointcut (we will give examples of this process later).

The implicit declaration of an advice determines how it interacts with the joint point. **AspectJ** separates advices into those who are executed before the join point, those who are executed after the join point and those who are executed in place of (or "around") the join point. Using such advices, the pointcut **pointcut call(void add(int))** could insert a warning message before the call, after the call or could change that call to something different, such as a warning message to a log.

The flags *after* and *before* cannot change contextual information like, for example, return values, arguments, etc, they can only read it. The around advice is not only able to read contextual information, but also to change it. Since it runs in place of the join point it operates over and is allowed to return a value, it must be declared with a return type, like a method.

The declaration of advices generally follows one of the following examples [CCHW04]:

```
before (ParameterList) : PointcutExpr { ... };
after (ParameterList) : returning PointcutExpr { ... };
after (ParameterList) : throwing PointcutExpr{ ... };
Type around (ParameterList) : PointcutExpr{ ... };
```

The first thing to notice is that since advices are implicitly related to a pointcut, there is

no need to name them and there are no visibility modifiers for advices.

Secondly, before the list of parameters the advice needs, which must also be declared in the pointcut (this list can be empty, as we will see) always comes the flag who determines if the advice is to be inserted after, before or around the pointcut.

After the list of parameters comes the *PointcutExpr*, which is the name of the pointcut the advice is advising. This is necessary because the advice does not need to be declared immediately after the pointcut on the **AspectJ**'s source code (except on the pointcuts that do not have a name) and because there can be various advices to one pointcut. Since Java programs can leave a join point 'normally' or by throwing an exception, the *after* advice can be of one of three types: after returning, after throwing, and plain after (which runs after returning or throwing, like Java's 'finally').

In the end of an advice declaration, always comes an expression which represents the actual piece of code who is to be inserted into the original program.

An example might clear it out [1]:

```
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe) && args(x, y);

after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");}
```

The pointcut is defined as seen in Sect. 4.2, only this time it takes arguments and two new flags: *target* and *args* are introduced. These flags (there is one more: *this*) are used to tell that pointcut which values to publish, so they can be used by the advice. This pointcut publishes three arguments, the object **fe** and two integers, **x** and **y**.

In this example the advising is made after the join point, takes the arguments **FigureElement fe**, **int x** and **int y** published by the pointcut **setXY** and, after the return on the join point, prints where **fe** was moved to.

It is important to notice that in the definition of a pointcut and advice the flags *target*, *args* and *this* are not mandatory, and the advice does not need to take arguments. By using the pointcut **arraychanges()** defined in Sect. 4.2, we can make a simple advice:

[1] Example taken from: http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/progguide/starting-aspectj.html

```
after() returning: arraychanges()
        { System.out.println("Array was changed"); }
```

AspectJ supports the declaration of the pointcut inside the advice, so the following code:

```
after() returning:
        call(void add(int))
        || call(void remove(int)); {
System.out.println("Array was changed"); }
```

is also correct.

4.4 Inter-type declarations

AspectJ also supports inter-type declarations, that allow the modification of classes and their relations. While advices are declarations written in an aspect, that change certain behaviors within a join point, inter-type declarations are statements that an aspect takes complete control and responsibility from certain characteristics on behalf of other types. Inter-type declarations cut across classes and their declaration, and may alter a system's classes and their inheritance relationships. These alterations are, unlike advices, always operated statically, at compile time.

In Java, when a capability is shared by some existing classes that already extend a class (have inheritance relations), the solution is to create an interface that adds to all the affected classes a method that extends this interface. **AspectJ** can express this concern using an inter-type declaration, on one place only. One only needs to declare the methods and fields necessary to express the concern and **AspectJ** alters the methods and fields of the classes accordingly.

Suppose we want a Librarian class to observe changes on Book objects, and both are existing classes. One way to implement this in **AspectJ** is by writing an aspect declaring that the object Book has an instance field, librarians, that tracks the Librarian objects that are observing Books.

We first start the inter-type declaration by defining the needed variables (in this case, an Arraylist):

```
Aspect BookTracking {
        private Arraylist Book.librarians = new Arraylist();
        ...
}
```

Secondly, we had the new methods. In this case, the methods **addLibrarian()** and **removeLibrarian()** are needed because only **BookTracking** can see this new field, **librarians**, since it is private, and therefore we need to declare methods to add and remove information from it:

```
Aspect BookTracking {
        private Arraylist Book.librarians = new Arraylist();

        public static void addLibrarian(Book b, Librarian l) {
        b.librarians.add(l); }

        public static void removeLibrarian(Book b, Librarian l) {
        b.librarians.remove(l); }
        ...
}
```

The next step is to create a pointcut **rents** that catches the join point that represents the execution of a rent, and an advice to define what we want to do after the join point, which in this case is to notify the librarians.

```
Aspect BookTracking {
        private Arraylist Book.librarians = new Arraylist();

        public static void addLibrarian(Book b, Librarian l) {
        b.librarians.add(l); }

        public static void removeLibrarian(Book b, Librarian l) {
        b.librarians.remove(l); }

pointcut rents(Book p): target(p) && call(void Book.rent());
```

```
after(Book b): changes(b) {
        Iterator iter = b.librarians.iterator();
        while ( iter.hasNext() )
        { updateObserver(b,(Librarian)iter.next()); }        }
}
```

Note that in this process we do not have to change the code from none of the classes implied to extend this functionality.

4.5 Aspects

AspectJ introduces a new modular unit: aspect. An aspect is defined very much like a class in Java, and can have methods, fields and initializers in addition to pointcuts, advices and inter-type declarations. In **AspectJ**, these primitives can only be included in the modular unit of an aspect, so their declaration is localized.

Aspects may be instantiated, just like normal classes, but **AspectJ** limits this instantiation so one can not use Java's 'new' form to build new aspect instances. Because typically each aspect is a singleton, one aspect instance is created which means that advices can use non-static fields of the aspect (as seen on the aspect **BookTracking** from Sect. 4.4).

Next, we present an example of an aspect:

```
aspect ArrayAccess {
   int n_accesses = 0;

   before(): add() && remove() && edit() {
      n_accesses++;
      System.out.println("Array accessed
                                    by the" + n_accesses + "time.");
   }
}
```

Notice here that the advice uses the local variable, **n_accesses** to maintain important information to advise the join point (int this case, the number of accesses to an array).

4.6 Examples

Having introduced the **AspectJ** language, let us see now how we can use it in real examples.

Next, we present some examples of **AspectJ** and the output they produce, as well as some particular features from the language.

4.6.1 Tracing and Context Exposure

Let's look at the following aspect:

```
aspect asp {

  pointcut mypc(foo c, Object o, int i) :
        args(i) && call( void foo.bar(int) ) && target(c) && this(o);

  before(foo c, Object o, int i) : mypc(c,o,i) {
    System.out.println( "calling " + c.name + "(" + i + ") from "
                        + o.getClass().getName());
              System.out.println("Entering: " + thisJoinPoint);
  }
}
```

This aspect does a few things. First, it declares a pointcut: **mypc**. This pointcut's join point is quite simple: it not only detects calls to the method *bar* from the object **foo**, but also captures information important to the advice, such as the arguments, the objects and the classes related to the join point. In **AspectJ**, context exposure must be declared in the pointcut and in the advice.

The advice acts before the join point, and captures all the information published by the pointcut and prints it into the screen. The output, when we run the class *test*, in the package *teste*, the object's name is **first** and is being called with the argument **1**, is the following:

```
Entering: call(void teste.foo.bar(int))
calling first(1) from teste.test
bar called with 1
```

The first line, "Entering: call(void teste.foo.bar(int))", is produced using a special primitive from **AspectJ**: *thisJoinPoint*. In all advice bodies this variable is bound to an object that describes the current join point.

The complete code to this example is in App. A.1.

4.6.2 Two aspects

It is possible to have two aspects capturing the same join point and giving the same advice. In this example, the pointcut is the same in both aspects:

```
pointcut mycut() : within(foo) && execution(* * (..));
```

This pointcut captures, within an object **foo**, all the execution of methods, no matter what their name, their arguments or their returns are. In this example we have two aspects, **asp1** and **asp2**, with exactly the same source code, apart from the name of the aspect.

Although these two aspects work on the same program on the same time, we can control which one acts first. There is a special command in **AspectJ**, called *precedence*, where we can define which aspect is executed first.

So, the line:

```
declare precedence : asp1, asp2;
```

is actually defining that **asp1** is to be executed before **asp2**.

The output is the following:

```
before asp1
around1 asp1
before asp2
around1 asp2
bar called
after asp2
around1 asp2
after asp1
around1 asp1
```

And if we change the precedence order:

```
before asp2
around1 asp2
before asp1
around1 asp1
bar called
after asp1
around1 asp1
after asp2
around1 asp2
```

The complete code to this example is in App. A.2.

4.6.3 Advice to advice

Is it possible to give an advice to another advice? Sure, but a few precautions must be taken.

Imagine an aspect whose join point are calls to the function "bar" and has the following advice:

```
pointcut barcut(foo c) : call(void foo.bar()) && target(c);

before(foo c) : barcut {
        c.bar(); }
```

This aspect captures calls to the method **bar()** from the class **foo** and calls the same method from the same class before the join point. This aspect, however, cannot be execute because it generates a Stack Overflow. This is because the advice generates a join point that the pointcut *barcut* is programmed to capture (Fig. 4.2).

For this aspect to work, we have to tell the pointcut to capture all the calls to the method **bar()** except the ones generated by his own advice. This is made using the **within()** primitive from **AspectJ**:

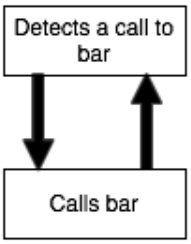

Figure 4.2: *Infinite loops in aspects.*

```
public aspect asp2 {

        pointcut barcut(foo c) :
                call(void foo.bar()) && target(c)
                && !within(asp2) ;

        before(foo c) : barcut {
                c.bar(); }
}
```

This aspect now correctly captures only the calls of the method **bar()** made within the object **foo**.

It is now possible to define an aspect:

```
        pointcut barcut() : call(void foo.bar());

        before () : barcut() {
                System.out.println( "advice by asp1"); }
```

That captures all the join points that represent a call to the method **bar()**, independently of the origin of those calls, that can be the object **foo** or the aspect **asp2**.

When we run these two aspects, the output is:

```
advice by asp1
advice by asp1
bar called
bar called
```

Notice that the the first call to the method **bar()** was not from the object **foo** but from the aspect **asp2** and then, consequently, the first advice (which comes from **asp1**) is advising other advice (which comes from **asp2**).

The complete code to this example is in App. A.3.

4.7 Summary

AspectJ presents three new primitives: pointcuts, advices an inter-type declarations to modularize cross-cutting concerns. Pointcuts are used to detect join points and publish important information from those join points. Advices are used to write the actual alterations in the original code, which can be inserted after, before or around a join point. Inter-type declarations allow the modification of classes and their relations. **AspectJ** introduces a modular unit, aspects, to modularize the implementation of this three primitives.

Chapter 5

Domain Specific Aspects Language (DSAL)

In previous works [CDM+10], aspect-oriented features were proposed to support triggering conditions and monitoring variable values, as well as a draft of an Aspect-Language to support these features. In this chapter, we describe our aspect language specification with mechanisms to detect join points and perform transformations in **Matlab** source code.

5.1 Introduction

> *"Aspect-oriented programming provides powerful ways to augment programs with information out of the scope of the base language while avoiding harming code readability and thus portability.* ***Matlab*** *is a popular modeling/programming language that will strongly benefit of aspect-oriented programming features. For instance,* ***Matlab*** *programmers could use aspects to provide information such as restrictions on allowed data types and/or values, monitoring specific aspects of the execution such as the effective dataset sizes or if a given variable ever assumes a specific value, without "polluting" the code with "check code"."* [CDM+10]

As seen before, the flexibility of the interpretative language of **Matlab** also hinders performance, forcing programmers to develop reference versions of the program functionality

in languages such as C and C++. What is more, when it comes to evaluate specific features, such as logging (Section 2.2.2), exploiting non-uniform fixed-point representations or including handlers to watch certain behaviors, the programmer is overwhelmed by cumbersome, error-prone and tedious tasks, which imply invasive code in the original **Matlab** sources.

The original base program is free of language enhancements and sources remain legal **Matlab**. The proposed DSAL enables programmers to retain the obvious advantages of a single source program representation while allowing the implementations to explore a wide range of specific solutions at reduced programming and maintenance costs.

This language is the basis for an empirical study of AOP. By creating this language, the respective weaver and an online tool, we expect to understand what happens when a community of users uses aspects extensions in **Matlab**. More concretely, we want to know:

- What kind of aspects do they write?
- Are these aspects domain-specific (do different communities that work in different domains have different requirements about each concerns are important to capture)?
- What kind of patterns do they create and what kind of style guidelines emerge?
- Can the the community understand each others aspects?
- Can the aspects be reused by different parts of the community?
- But above all: is it possible to create code that is more modular, more reusable, easier to implement and to understand?

The creation of this domain specific language is actually just part of the big goal of the AMADEUS project: to understand if the **Matlab** community can really utilize aspect oriented concepts while programming in **Matlab**.

5.2 Organization of an Aspect Module

Our language treats an aspect as an independent modular unit. An aspect module can only represent one instruction (remember, or join point capture mechanism is 'Instruction

Based'), although it can have more than one action to be executed in that instruction. It is started by the constructor *aspect*, followed by the name of the aspect, and ended by *end* (similarly to **Matlab**). Inside the aspect we define the join point and the actions necessary, as shown next:

```
aspect aspect_name
        capture join_point
        action to join_point
        action to join_point
        ...
end
```

The aspects are organized in source files, that may contain more than one aspect. Each source file from our DSAL must have, in the beginning, a strategy for applying the aspects. This strategy is mandatory and represents the sequence in which the programmer wants the aspects to be implemented. This strategy is composed by the disjunctions (&&) and conjunctions (||) (disjunctions have priority).

So, at the beginning of each DSAL source file, it is mandatory to right a strategy before the aspects, as shown next (strategy is highlighted):

```
a1 && a2 && a4 || a3

aspect a1
aspect a2
aspect a4
aspect a4
```

In this particular example, the aspects 'a1', 'a2' and 'a4' run sequentially and, if any of them can not be executed, than the aspect 'a3' is executed. It is possible to use parentheses and create very powerful strategies. If no Aspect Combinator is defined, the Weaver applies the aspects in the order they are defined in the file. Using the disjunction (&&), if the any of the aspect fails, the others can not be applied, whereas using the conjunction (||) the failing of an aspect does not interrupt the sequence.

It is important to notice that it is possible for two aspects to advise the same join point or, more important, for an aspect to advise another. One aspect might, for example, introduce

a new variable 'a', and another aspect search for the declaration of the same variable. For this to happen, it is important that the strategy is constructed in a manner that the second aspect runs after the first one.

With such a powerful mechanism, finding the correct and appropriate strategy is an interesting research topic by itself, although in this work the focus in on the programming support for aspects.

5.3 Join Point Capture

5.3.1 The Joint Point Model for Matlab

When designing an AOP language, the join point model is a critical element. This model is the basis of the mechanism that allows the perfect coordination between the original source code and the aspects.

There are many different kinds of joint point model, such as primitive application nodes in a dataflow graph [ILG+97,Lop97,MMK+97,OT01] and method bodies [OT01,OKH+95]. In this language, we have created a join point model that defines a precise instruction in the original program.

To clearly define a joint point in **Matlab** we have separated the context in which join point appears into three groups: *declarations*, *assignments* and *functions*. Anything captured by out language is within one of these three groups, which are unitary and can not be divided. For example, if the programmer wants to catch an assignment to a variable 'a', such as in: '**a = errors[5];**', he cannot separate the line into two parts. In practice, what this means is that when a programmer catches a join point, and executes an action on that join point (wether it appears after, before or around), the whole line must be advised[1], and not only part of the assignment. We say our join point capture is 'Instruction Based', and what it means is that the advising on the join point must be something like: "introduce the whole instruction, but with a scalar as an alternative to an array" instead of "introduce a scalar in the right side of the assignment".

Next, we present an example of correct and incorrect advising:

[1] In some of the explanations we use **AspectJ**-related expressions, such as *advice*. Although this language is quite different, such expressions are used to clearly explain to the reader a concept or an idea. Later on we will see that such expressions disappear when we start introducing our own primitives.

Capture the writing to 'a'
a = array[5];

Advice: change the whole line to a = 10;
instead of
Advice: change right of the assignment to 10;

In this example we have, highlighted in gray, the actual join point we want to capture, which is an assignment to the variable 'a'. What is important to notice is that, the correct advice is to rewrite the whole assignment, instead of only the part that we want to change. Out language is not capable of extracting or dealing with a smaller amount of information than these groups, as shown in the next figure.

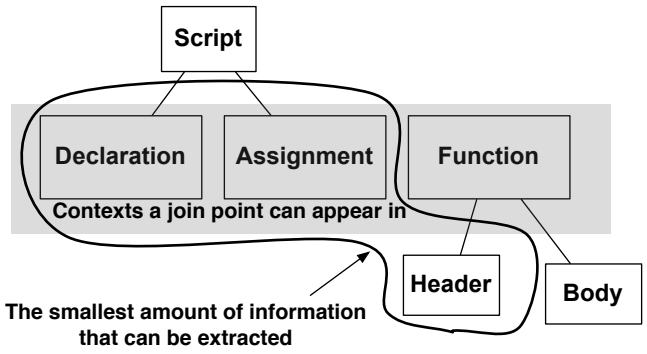

Figure 5.1: *Join point model.*

Fig. 5.1 shows all the contexts a join point can appear in. Declarations, assignments the header of a function follow the 'Instruction Based' idea, and represent the smallest granular unit our aspect language can deal with. It is also possible to write a **Matlab** program in the form of a script, but all the lines within a script are typically or assignments or declarations[2].

Functions, however, are a different context for join points. This happens because, in **Matlab**, functions are the modular unit of the language, and so programs are usually organized in groups of functions. As so, this is the only context a join point can appear in that can be separated between his body or his header. On the particular case of functions, we can specify if the join point is the header of the function, or if it appears in it's body. If we want to capture the head of a function, this particular join point is also 'Instruction Based',

[2]We also have loops, which are very important on mathematical calculus, but are not extracted by our language. See Chap. 7.1

and must be advised entirely. The body of the function is not exactly a join point, but rather a 'context' the join point can appear in, as we explain in the next section.

5.3.2 Join Point Capture Primitives

In any Aspects Oriented language, an important part of the system is the mechanism that crosscut the modular structure of the host language. For example, when AOP was applied to the Objects Oriented Paradigm, constructors had to be created to cross the hierarchical modularity of OO programs. And the same is to true to the block mechanisms of procedural languages [FF00, CKF00].

In **Matlab** we do not have access to the complex modular mechanisms of, for example, OO. A **Matlab** program is typically composed or by a script or by a function or a group of functions. Because this composition represents a poor mechanism to modularize concerns, we might expect that concerns do not appear as well organized in **Matlab** as they do in other programming environments.

The limited composition of programs in **Matlab** is directly related to the dynamic join point we want our language to capture. We no longer have to worry with inheritances or subclasses of code, for example. Instead, the important information to 'capture' is the one related to mathematical calculus, such as variables attributions, declarations, operations related to arrays and, of course, functions.

In this DSAL, the primitive for capture dynamic join points is *select*. This primitive is just an indication that we want to capture information on the source code, and is followed by the specific part of the code we want to 'capture', as shown next:

```
select: " any time the variable 'a' is read ";
      or
select: " any call to the function 'matrices_sum' ";
```

To specify the exact dynamic join point we want to capture, we have created a set of primitives that represent a big number of potential concerns we want to capture. It is not possible to say these primitives capture all the potential concerns in **Matlab**, or the most important, but they surely capture the most common[3] We have separated the potential

[3]It is hard to clearly define all the important join points on **Matlab** code, and to our knowledge, no work has ever developed a systematic and complete capture of **Matlab** crosscut concerns. This might represent an excellent research (more on future work).

concerns into three groups, each with potential join points.

As we see in Fig. 5.2, we have separated **Matlab** code into three major groups: **variables**, **arrays** and **functions**. It might seem strange why variables and constants are on the same group but we will get to that later.

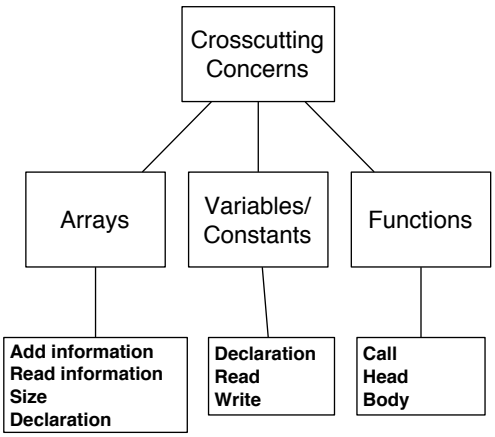

Figure 5.2: *Structure of the dynamic join points captured.*

For any of these groups, we have selected the properties that have more potential to be important join points on a program. So, for arrays for example, we found the most important concerns related with this data structure are:

- Anytime information is written to an array
- Anytime information is written to an array
- The size of the array (useful, for example, to boundaries control)
- When the array is declared

The variables and constants are in the same group because, anytime we have an expression like: '**a = 6;**', we actually have two join points here: one that is the writing on the variable 'a', and the other which is the reading of a scalar, in this case, '6'. So one can capture this line of code as: "the writing of 'a'" or "the reading of '6'".

The functions that capture join points are in the Table. 5.1. For each kind of structure - arrays, variables/constants and functions there are a number of primitives that help in capturing information. These primitives are directly related to the structure of potential

parts we want to capture, which is present in Fig. 5.2. All these primitives 'Instruction Based', which means they extract the whole instruction they appear in. This is not true only for the primitive *within*, as we explain after the table.

It is possible to use logic operators, such as conjunctions (||), disjunctions (&&) and negation (!) to construct more complex join joints.

Table 5.1: Primitives for join point capture

Arrays	Variables/Constants	Functions
add()	read()	call()
get()	write()	header()
size()	declare_var()	within()
declare_array()		

For arrays, we have four primitives - *add()*, *get()*, *size()* and *declare_array()*, which capture the addition of information to an array and the reading of information from an array, the size of an array and his declaration, respectively. Similarly to some **Matlab** functions, the primitives *add()* and *get()* are polymorphic, because some of them can take one or two arguments. The primitives related with arrays can have two arguments. For example, can be used as '**add(a)**', which represents the addition of information to an array, or can be used as '**add(a,5)**', which represents the addition of information to the array 'a', on the index '5'. The same is true for *get()*.

For variables, we have only three primitives that capture the reading and the writing of information to a variable and his declaration. The primitive *read* can receive as an argument an integer, which symbolizes the attribution of a constant to a variable.

For functions, things are a bit different, because functions constitute the modular units in **Matlab**. So, we have primitives to capture when a function is called, and to capture the head of the function. Sometimes, it is useful to understand if a join point occurs inside a specific function, and that is what the primitive *within()* is for. Using it, we can define join points such as:

```
select: anytime variable 'a' is read inside function 'sumvals';
```

Using these primitives, we can now clearly express the join points defined earlier using only the primitive *select* and primitives from Table. 5.1. Next, we present how some join points can be captured using such mechanism:

```
select: " any time the variable 'a' is read " ;
        becomes
select: read(a) ;

select: " any call to the function 'matrices_sum' " ;
        becomes
select: call(matrices_sum) ;

select: anytime variable 'a' is read inside function 'sumvals' ;
        becomes
select: read(a); && within(sumvals) ;
```

These primitives represent a huge potential and a big number of join points, either by used alone or together with logical connectors. Table 5.2 shows some examples of how certain join points can be captured from **Matlab** source code using this mechanism.

Sometimes, the same line of code can be captured by different forms, because there are more than one join point present there. For example, in the line from the second column: '**a[i]=z[x];**', we can capture information related to:

- The addition of information to 'a'
- The addition of information to 'a' on the index 'i'
- The reading of information from 'z'
- The reading of information from the index 'x' on the array 'z'
- Any combination of the previous join points

It is important to remember that although these are different join points, an assignment is a granular unit our 'Instruction Based' capture mechanism can not divide. So all of these join points capture exactly the same thing: the instruction '**a[i]=z[x];**'.

It is important to remember that, even if any of this **Matlab** statements have various join points, the overall amount of information captured by *select* is the same: the primitives for capture join points are 'Line based', which mean they catch all the statement the join point appears in.

Using the logical operators, we can have very complex, user-defined *select's*, such as:

Table 5.2: Examples of join point captures

Join Points	a[i]=z[x];	a[i]=d;	c=z[i];	d=soma(x,y);	x=5;
select:	get(z);	add(a);	write(c);	write(d);	write(x);
select:	get(z,x);	add(a,i);	get(z);	call(soma);	read(5);
select:	add(a,i);	read(d);	get(z,i);	write(d) && call(soma);	write(x) && read(5);
select:	add(a) && read(d);	add(a) && get(z);			

```
select: (read (a) && within(sumvals)) || (!add(z) && within(sumvals_arrays);
```

What this example does is: or captures any read of the variable 'a' on the function 'sumvals' or makes sure the array 'z' is not edited inside the function 'sumvals_arrays'. Such mechanisms represent a big potential for content capture and crosscut concerns by our DSAL.

We are convinced that these powerful mechanism can be used in most cases to capture code logging, tracing or control, for example. Although, as any AOP language, it is very hard to clearly define the exact characteristics we want to modularize, since it depends of factors such as the host programming paradigm, the host language or even the domain the programmers are working in. We will get back to this issue in the conclusions (Chap. 7). However, it is very important to notice that this language, and the associated weaver (Sect. 5.8) are constructed in a way that they can be quickly adapted to support new primitives and join points. Once again, we will get back to this on Sect. 5.8.

5.4 Action Description

The action description, which in this language is represented by the primitive *apply* is a method-like mechanism that is used to declare that a certain amount of information should appear in the content extracted by the *select*. The *apply* primitive is directly associated with three flags: *after*, *around* and *before*, which have the same functions as the equal flags of **AspectJ**: to declare if the information in inserted after, before or around (substituting) the content captured by the aspect.

The argument of the primitive *apply* is a string with the exact information we want to put in the source code. This string is directly inserted in the final file, followed by the

primitive *execute*, that takes as argument a flag indicating the position where it will be inserted, related to the join point (after, before and around).

A join point can have more than one action. It makes perfect sense to support more than one action per aspect becouse it is very common that a programmer wants to do two or more operations per concern crosscutted. The programmer might want, for example, to change the value of a variable and keep a log of all the alterations to the same variable.

Below we see an example of a very simple aspect (it could take even more *apply's*):

```
aspect log_a
        select: write(a)
        apply: display('Variable will be written.'); :: execute before
        apply: display('Variable was written.'); :: execute after
        ...
end
```

The part of the action description immediately after *apply:*, in this case 'display('Variable will be written.')' and 'display('Variable was written.')' is directly inserted in the joint point.

If the original source had the line of code '**a = 5;**', and we applied the previous aspect, we would get the following result:

```
... % some Matlab code
display('Variable will be written.');
a = 5;
display('Variable was written.');
... % some Matlab code
```

The fact that the argument of the primitive *apply* is an exact instruction in **Matlab** makes it very easy to implement an aspect: any **Matlab** programmer can easily implement an action because he knows the exact line of code he wants to be present in the original code, and no abstraction is needed. These arguments are direct **Matlab** instructions.

If an aspect has lots of *apply's*, all on the same aspect, and all are supposed to be performed before the join point, the alterations in the source code appear in the exact same order they appear in the aspect. For example, the aspect:

```
aspect log_a
        select: write(a);
        apply: display('1'); :: execute before
        apply: display('2'); :: execute before
        apply: display('3'); :: execute before
end
```

Creates the following output:

```
... % some Matlab code
display('1');
display('2');
display('3');
a = 2;
... % some Matlab code
```

The same is also true for the *after* flag, and any aspect can only have an 'around' application, otherwise it generates an error.

The function-related primitives *head()* and *within()* are an exception to these semantic rules. The primitive *head()* must always be advised only once, and that advising is mandatorily 'around' the captured content. The primitive *within()* must never be declared alone, and works only with conjunction with other join point capture primitives. So, the following aspects:

```
aspect big_mistake
        select: within(sumvals);
        ...
end

aspect big_mistake_2
        select: head(sumvals)
        apply: disp('first ups'); :: execute before
        apply: disp('second ups'); :: execute after
end
```

are both incorrect. The first one ('big_mistake') is incorrect because the primitive *within()* can never be used alone, independent of the *apply's* of the aspect. The second one ('big_mistake_2') is incorrect for two reason: first, the apply of the primitive *head()* must be necessarily 'around' the content captured (which is the head of the function), and secondly because this primitive can only have one *apply*.

5.5 Content Exposure

Sometimes, it is very useful to pass information from the execution context of the join point to the action the aspect will perform. An advanced aspect for logging, for example, might want to save the information of the value that was assigned to a variable, rather than simply saving the information that the variable was written.

Our DSAL provides a mechanism that makes it possible for the join point capture to see a set of values in the execution context of the *apply*. We call this content exposure.

Content exposure uses a form of binding similar to the one in [LDS05] to expose the content of the join point. For each information from the join point capture functions, there are primitives that capture content from that join point. These primitives are described in table 5.3.

Table 5.3: Primitives to capture content

Arrays	Variables/Constants
size index value(index)	value

These primitives represent dynamic content that might be necessary for the action related to the captured content (all the static content, such as variables names, is known by the programmer and needed for programming the join point capture). So, for arrays, we have access to the values of his size, his value on a given index and the index the array is when the join point was captured. For the dynamic context of variables, we have access to his value.

On the primitive *apply*, the argument (which is always a string) can be composed on the aspect. For example, writing: ''**apply: display('The value is 9');**'' is exactly the same (assuming 'a=9' and the *select* captured 'a') as writing: ''**display('The is ++ a.value ++**

');''.

Functions do not need content exposure because, with the exception of the primitive *call*, because their primitives do not really catch a join point (except *call()*), they are rather used or to control a domain of execution (*within*) or for very specific content capture (*header*).

This 'content exposure' might seem complicated, so let us see if a small example can clear things out. How do we create an aspect that keeps a log, on the screen, of all the insertions in the array 'a'?

Easy. First, we initialize the aspect and declare the join point capture (which is the insertion of an element in a):

```
aspect array_a_tracing
        select: add(a)
        ...
...
```

Next, we declare the action and, since we want to print in the screen the array name and the element being inserted, this context is passed as arguments in the *apply*:

```
aspect array_a_tracing
        select: add(a)
        apply: display(['The value ++ a.value ++
                        will be inserted into a'])
               :: execute before
end
```

The string is composed and inserted in the final code, and this aspect prints outputs similar to the ones on table 5.4.

Table 5.4: Outputs from the aspect

Action	Result
a[34] = 23;	The value 23 will be inserted into a
a[2] = soma(x,y);	The value soma(x,y) will be inserted into a
a[4] = z[5];	The value z[5] will be inserted into a

It is important to notice, in the Table. 5.4, that the capture description never does any calculation. For example, if one asks an aspect for the value that is being assigned to the variable 'a' in the following line of code: '**a = sum(b,c);**', the output is '**sum(b,c);**'. This is important so the programmer can keep the exact tracking of what is happening within the code. But, and if the programmer really wants to know the value assigned to 'a'? It is quite easily actually, we just delegate that calculation to **Matlab**. The following aspect:

```
aspect log_a
        select: write(a)
        apply: int aux = sum(b,c); :: execute before
        apply: display(aux); :: execute before
        apply: a = aux; :: execute around
end
```

creates the fllowing **Matlab** code:

```
... % some Matlab code
int aux = sum(b,c);
disp(aux);
a = aux;
... % some Matlab code
```

that prints to the screen the actual value that will be written to 'a', without changes in the behavior of the original function Such mechanism makes it very easy to know both the behavior of the **Matlab** function, how it was implemented and the exact values that are being used within it.

5.6 Aspects calling Aspects

It is possible, in our DSAL, to have an aspect calling another one. Such mechanism is very useful when we want, for example, to capture a join point in a piece of code limited by another join point. We might want, for example, that 'aspect_2' is only executed after the joint point 'jp1'. To do so, we can declare the 'aspect_2' as the action to be executed after the capture of 'jp1', as shown next:

```
aspect first
        select: jp1
        apply: aspect_2 :: execute after
end

aspect aspect_2
        ... some instructions
end
```

When we have a set of aspects in a source file, an aspect can call another one, independently of the strategy defined. For example, in the following file:

```
a1 && a3 && a4

aspect a1
        select: some join point
        apply: a3
end
aspect a3
        select: some join point
        apply: a2
end
aspect a2
aspect a4
```

we see a number of correct ways of interrelate aspects. We can have a chain of aspects advising other aspects ('a2' advises 'a3' which advises 'a2+1'), and we can have aspects that are only used in the context of another aspect (such as 'a2'). A practical example of an aspect calling another one is in Subsection. 5.7.1

5.7 Language in Practice

5.7.1 Capturing a global variable

Imagine the simple example of capturing the value of a variable, but only after a call to a function. This example is purely academic and works only to show the potential of our content capture mechanism. We have a **Matlab** source file such as shown next:

```
... % some Matlab functions
function sum_matrices()
        ... %some Matlab instructions
        c = 0;
        ... % some Matlab instructions
        res = sum_ints(a,b);
        ... %some Matlab instructions
end
... % more Matlab functions
```

What we want to do is to know the value of the variable 'c' after the call of the function 'sum_ints()'. To do so, we need to define two aspects: one that captures the value of the variable 'c', and another one that captures the call to the function 'sum_ints()'. One possible solution is shown next:

```
aspect catch_sumints
        select: call(sum_ints)
        apply: value_c :: execute after
end

aspect value_c
        select: read(c) || write(c)
        apply: disp('Value of c: '); disp(c); :: execute before
end
```

These aspects have some particularities, that show the expressive power of our language. The first aspect ('catch_sumints') is quite simple, and only finds the call to the function

'sum_ints' and applies the aspect 'value_c' after. The aspect 'value_c' is more interesting. First, his join point is: anytime the variable 'c' is accessed or written. Because we don't exactly know where this variable appears, a good method to discover it's value is to find the first time it is used. Secondly, the *apply* is constituted by two **Matlab** instructions. Such is possible because, between the primitive *apply* and the primitive *execute*, everything is directly inserted into the original **Matlab** file. The 'apply' of the aspect 'value_c' is exactly the same as writing:

```
apply: disp('Value of c: '); :: execute before
apply: disp(c); :: execute before
```

After the aspects and the original code are combined, we obtain the following valid **Matlab** code:

```
... % some Matlab functions
function sum_matrices()
    ... %some Matlab instructions
    c = 0;
    ... % some Matlab instructions
    res = sum_ints(a,b);
    disp('Value of c: '); disp(c);
    ... %some Matlab instructions
end
... % more Matlab functions
```

Note that the **Matlab** primitives 'disp' appear in the same line because this was the result of weaving the code with the aspect that has only one *'apply'*. If we applied the aspect with the two *'apply'*, we would obtain a slight different result, where the 'disp' primitives were in different lines. Despite this small layout difference, the final code would perform exactly the same.

Aspects are automatically combined to the original code in order to create a new, valid **Matlab** program. This means the alterations performed by the programmer are easy to perform and after using them, he can always revert to the original **Matlab** function since it was untouched during the whole process. More information about the weaving process and the program that implements this transformation is presented in Section 5.8.

5.7.2 Logging

In Chap. 2 (Fig. 2.3) we have manually changed the original 'sumvals' function in order to perform logging instructions. This process implied manually insert intrusive pieces of information on the original function to be able to log parts of the function execution. Next, we present how to concisely specify such program using our DSAL.

First, we define an aspect with name 'variable_tracing' that is responsible for tracing a variable received as argument. The join point 'write' detects when a value is written to the argument variable. The execute primitive introduces the invasive code before the join point.

```
aspect variable_tracing
        select: ( write(i) || read(i) ) && within(sumvals)
        apply(variable.name): display( i );        :: execute before
end
```

Next, we present an aspect that not only traces a variable, but also does a test. This aspect reuses the aspect 'variable_tracing' described before.

```
aspect sumvals_logging
        select: within(sumvals) && read(star)
        apply: if(start < 0)
                    display('Attention, start is negative');
               end :: execute before
        apply: variable_tracing :: execute after
end
```

In this aspect, we select the definition of the function 'sumvals' and we perform two transformations: First, we introduce a condition to test if 'start' is ever negative. Second, we apply the aspect 'variable_tracing' to the body of the selected function. After both the aspects and the original **Matlab** program are 'weaved', we obtain the code on Fig. 2.3.

5.7.3 Type Specialization

In **Matlab** types are not mandatory. Thus, variables and functions can be defined and used without specifying their types. When compiling **Matlab** into an embedded system, how-

ever, type information is crucial in order to produce efficient code. Transforming generic function definitions to type-specific functions is easily done with our aspect oriented language for **Matlab**. Next, we present an aspect that performs type specialization.

```
aspect sumvals_types
        select: header(sumvals)
        apply: function s <scalar int> = sumvals(start <scalar int>,
                        step <scalar int>, stop <scalar int>)
        :: execute around
end

aspect change_types_i
        select: declare_var(i)
        apply: i <scalar int> = start; :: execute around
end

aspect change_types_s
        select: declare_var(s)
        apply: s <scalar int> = i; :: execute around
end
```

These aspects are all quite simple. The first one uses the special primitive *'header'*, that receives as argument the name of a function and allows us to edit his header. This is a special aspect in the sense that it can uses a primitive (*'header'*) that can only be used with one *'apply'* and this apply must perform alterations that substitute ('around') the original content. The second and the third aspects are quite simple, they detect the declaration of a variable and change that declaration (in these cases, the transformation is to force the type ('scalar int') on the variables).

By weaving this aspect to the original **Matlab** program included in Fig. 2.2, we automatically obtain the **Matlab** program in Fig. 2.4.

5.8 Implementation

After the aspect program is written, it needs to be connected to the original source in order to generate aspects-extended code. This process is called Weaving and is done using a compiler which connects the two pieces.

This Weaver is being done in parallel to the language development, by other members of this project [Mac10].

The weaver uses strategic programming, which includes a fair amount of transversal schemes which allows to perform basic actions to the right data type in the right order to obtain the expected result.

A parsing technology that is worth mentioning and that is used is the ToOne Matching (TOM)[4] framework that helps in controlling data structures. TOM includes generic traverse schemes for crossing Abstract Syntax Trees (AST).

ANTLR[5] will be the responsible for semantics check, and then the AST tree produced is converted, using a special TOM tool called Gom Antlr Adapter, to a a Gom tree which TOM can manipulate.

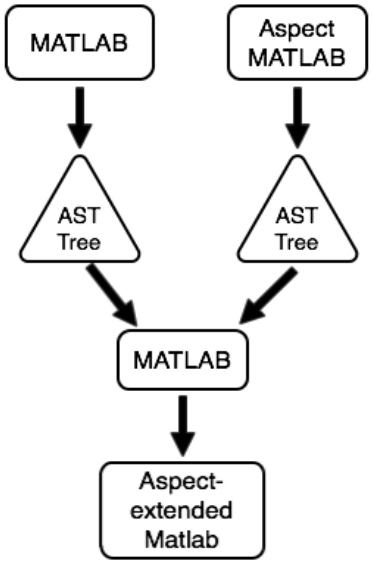

Figure 5.3: *Weaver diagram.*

Figure 5.3 shows the weaving process. It can be split in three main stages. The first two

[4]http://tom.loria.fr/
[5]http://www.antlr.org/

refer to the compilation and parsing of both the original **Matlab** source and of the aspects language to generate the respective AST trees. The third step is the most important, and is related to the weaving of the trees. Embedding the aspects into the original code is done by applying the correct strategies.

5.8.1 AMADEUS Toolbox

In the context of the AMADEUS project we have constructed a set of libraries and tools to process **Matlab** and the aspect extension. The overall architecture of our toolbox is shown in Figure 5.4.

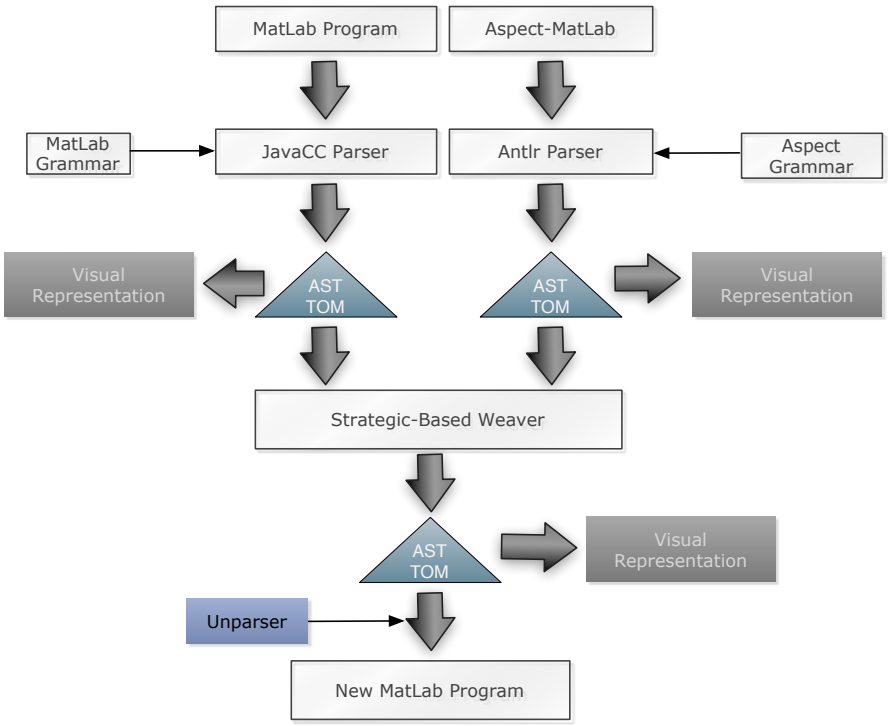

Figure 5.4: *AMADEUS Toolbox Architecture.*

The toolbox includes the following components developed in this project:

- A context-free grammar defining the **Matlab** language, a parser and an unparser. The parser is built using the JavaCC parser generator, which builds an abstract syntax tree as a TOM tree.

- A context-free grammar defining our aspect language and its parser built by the ANTLR [Par07] parser generator. Like for the **Matlab** parser, we use TOM trees to model ASTs.

- A strategic-based weaver implemented as explained in Section 5.8.

- An online version of our tool, that provides all the functionalities described above[6].

5.9 Benchmarks

In previous section we have presented several versions of **Matlab** programs, namely with our without logging instructions and by explicitly define types in **Matlab** functions. In this section we present a set of benchmarks of running different versions of type specialized programs. We consider four different **Matlab** programs: the original 'sumvals' program (as presented in [CBHV10]), a variation of the 'sumvals' program (called cycles), Euclid's GCD Algorithm, and a primality test (naive) algorithm. The function of each program is not relevant to this case. What is important is that, for each of these programs we consider it's generic, untyped definition, and the type specialized version obtained after weaving.

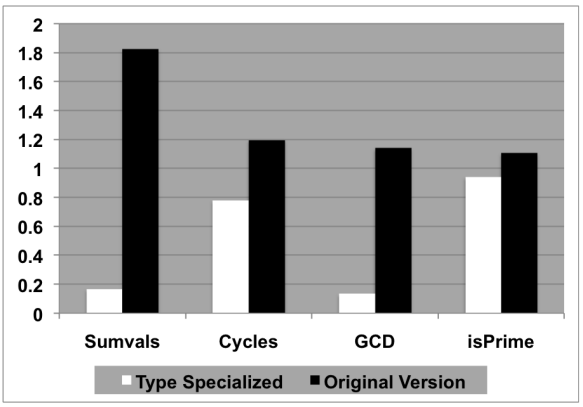

Figure 5.5: *Generic versus type specialized functions (transformed to C and compiled by gcc).*

Moreover, we consider the C version of each of the programs, obtained by manually translating the **Matlab** code into C, but bearing in mind a possible automatic translation stage. Figure 5.5 includes the runtime (in seconds) of the executable programs produced

[6]The online version of our tool is available at `http://amadeus.di.uminho.pt/`.

by *gcc* (version i686-apple-darwin10-gcc-4.2.1). The times were obtained in a Intel Core 2 Duo, 2.2 GHz, running MacOS X (version 10.6.3).

As the results clearly show, the type specialized programs are more efficient in terms of execution time than the generic ones: the speedup we obtain ranges from `1.2` (program isPrime) to `9` (program sumvals). In average, the specialized programs run `5` times faster than the generic ones. The gains obtained are mainly due to the box and unbox of values in a C implementation of a generic **Matlab** function. As expected, for a type specialized **Matlab** function, the C program does not need to perform such operations.

To get such improvements in an **Matlab** program, one has to manually perform type optimizations in the source, and backtrack those alterations when they are not needed. Using our DSAL, this process is not only easier and faster, but it is also very simple to alter the optimizations according to the environment the program will run in. Our language provides a very simple and easy mechanism but that might produce enormous speedups.

Chapter 6

Related Works

There are numerous projects based on AOP, both to implement this paradigm in other paradigms to to improve already existing AOP paradigms.

The most famous implementation of AOP, which was described on Chap. 4, is **AspectJ**, which aims at the Objects Oriented Paradigm, more concretely to Java. Although this project is usually a reference, and the considered the stat of the art AOP implementation, there are some projects whose aim is to extend **AspectJ** with additional features. **LoopsAJ** [HG06] is an example of a try to improve **AspectJ**, by defining a method to capture loops in Java and the related content exposure.

Other AOP related projects include the study of the addition of aspects to other paradigms. In AspectC++ [SLU05], the authors try to integrate AOP concepts with the philosophy of the language, which is very different to Java. Other project [LDS05] studies the impact of AOP in COBOL, and tries to identify potential crosscutting concerns, join points and advices in such paradigm. There is also a project that aims at extending AOP to Haskell [VSS09], through the use Attribute Grammars (AG). In this project the authors present a typed embedding of Attribute Grammars in Haskell, that relies on extensive polymorphic records and expresses AG conditions as type-level predicates.

On the context of attribute grammars, aspects are an important feature whose support is shown in AspectLISA [RM07]. AspectLISA is a tool that uses incremental and reusable aspect oriented attribute grammar extensions to automatically generate compilers, interpreters and other language related tools. This project is an example of how attribute grammars, domain specific aspect languages and aspect oriented specifications can work together. In fact, some research projects [Mar10] study the implementations of various

extensions, aspects included, to attribute grammar specifications.

Sometimes, AOP is not studied with the main objective of improving modularization in different paradigms, but also to study if well known techniques can be refactored into aspects. One example of such project is [ADMTH09], where the authors try to understand how the concerns and patterns managed by conditional compilation can be express in the form of aspects.

In the context of **Matlab** optimizations, there are numerous works on this subject, which work either by source level transformations [JB03, MP99] or by the use of compiler techniques [DeR96, AP02].

To the best of our knowledge, [CFM06] was the first approach considering aspect oriented programming extensions to **Matlab**.They have shown how aspect rules can be used to separate concerns such as monitoring, logging, handling, function configuration, and type specialization from the **Matlab** source code.

The work presented in [CDM+10] addresses a DSAL language for transforming **Matlab** source code. Although the work in this paper has clear roots in the ideas presented in [CDM+10], it advances that work by considering a number of extensions to the DSAL and by presenting an implementation of the weaver. Furthermore, we show in this paper the importance of aspects to specialize and obtain more efficient C implementations, an important issue when targeting embedded systems.

Translation from **Matlab** to C code is also an important goal in the AMADEUS project. Recent work is presented in [RN10], where type declarations are used to assist and make more effective and efficient a **Matlab** to C translation process.

Recently, there has been another effort, called AspectMatlab, to extend **Matlab** with aspects [ADDH10a]. AspectMatlab is geared towards scientific codes using aspect modules that define patterns-actions that support constructs such as loops, loop bodies, array accesses, and function calls. the AspectMatlab approach mainly focuses on monitoring aspects and does not consider embedded systems features.

However, the approaches followed in [ADDH10a] are quite different. A big difference in (lets call it AspectMatlab) is the language used. AspectMatlab defines a new language with new constructors to detect and insert information on join points. This language needs a compiler (called amc) to compile and woven the aspects source code to the **Matlab** code.

This compiler is currently in production and there is a beta version available[1]. Because **Matlab** supports some exclusive mathematical characteristics in the source code, compiling presents new challenges, different from less focus languages such as Java. One example is the check of the expression test(i), which can be either a function call with the argument i, or a get to the i'th element of the array test. What is more, simple weaving on **Matlab** requires lots of checks to detect expressions matches, as rules to look up for names differs in functions, inner functions and scripts [ADDH10b].

The language in AspectMatlab is very similar in design to **AspectJ**, although the syntax was developed in order to be similar to classes in **Matlab**. There was an effort in order to detect join points in loops, crosscut **Matlab** array accesses and to bind the context information from the join point in the action declaration. Currently this language supports features such as tracking array sparsity, measuring floating point operations and adding units to computations, however their patterns support much more possibilities.

[1] http://www.sable.mcgill.ca/mclab/aspectmatlab/index.html

Chapter 7

Conclusion

This document presents a language to add aspect oriented mechanism to **Matlab**, which helps programmers in implementing and experimenting strategies to program in programming environments. We believe this technology will, ultimately, help in creating better structured and modular **Matlab** code, through the addition of aspect software modules.

After reading the Aspects Oriented Programming bibliography cited in this paper, we are convinced that this language greatly improves the programmer performance. We have seen that AOP improves the creation of code by better modularizing all the system, and finding and organizing parts of code becomes so easier that editing and maintaining the code is now faster and simpler.

We have also seen how source code transformations in **Matlab** programs might bring such a great improvement in productivity, sometimes creating upgraded and specialized versions up to nine times faster than the original ones. We only show a few examples of this optimizations, but the bibliography is full of them, creating an enormous necessity for a mechanics such as ours, that allows those implementations to very easy to be used when necessary and discarded when not necessary. This tests quantified the overall performance gains by performing transformations on the source code, transformations supported by our system.

With our language, we not only gain in performance and running time, we also gain in modularity and organization. We have seen with very expressive examples how simple is to implement crosscut concerns in **Matlab** and how the aspects implementation is well organized, with simple *primitives* and code organization that allow for the fast use of this technology when creating numerical programs.

The simplicity of our language hides, however, the powerful mechanism and strategies we support. From aspects strategies to aspects calling aspects, our language allows for complex programming strategies that provide complex applications using our system.

Finally, we have presented an embedding of the aspect language in Java via the TOM strategic programming system, and create an online tool so the community can use our system and experience different techniques and programming strategies.

7.1 Future Work

Being this a recent project, there has not been a study of the full improvements made by this system, either in the creation of new **Matlab** systems, either in the management or edition of such code. Such study would constitute an excellent future project and would quantify the true gains of our domain specific language.

As we have shown, it is very hard to clearly define concerns in **Matlab** because, due to it's simplicity in code modularization, one can not find concerns as well organized as they are found, for example, in other programming algorithms, such as OO. What is more, it is an empirical work only that allows us to predict the parts of code that are really important to crosscut - such information would only be obtained with a study of how the community that uses **Matlab** in different domains requires the code to be organized and partitioned.

Our language captures a fair amount of information and allows us, as we have seen, to perform important transformations - it already has that explicit power. However, there are more potential upgrades, fine-tuning and studies possible this language:

- The aspects could take parameters. This way, instead of having an aspect that captures, for example, all reads to the variable 'a', we could have an aspect that captured all reads to any variable, being this variable a parameter. The aspects could even be polymorphic, in the sense that they could take an argument (the variable, for example), or zero arguments, and this last option would capture any read to any variable. Such feature would make the code on our language more reusable.

- Cycles are an important part of any mathematical calculus, however our language does not have a primitive to capture them. Study potential ways of captures any kind of cycles and the related exposure content would constitute a good case study and an important upgrade to this DSAL.

- Another potential upgrade is the introduction of condition within aspects. We have already created an exposure content mechanism that allows us to dynamically check values that occur in the context of the joint point, but the aspects could only be performed if certain conditions within that dynamic context would occur, such as variables being negative or cycles never achieving a certain value.

- This DSAL supports a very powerful mechanism of strategies within aspects, supported by the reuse of aspects by another aspects and by the strategies presented in any DSAL source code. The study of such a complex mechanism by itself would constitute a good case study. Which concerns can be crosscut? Which parts of code can be modularized? And which complex transformations does this system support?

Based on our experience programming with the proposed DSAL for **Matlab**, we believe that the productivity of programmers improve. However, to confirm this improvement, we would like to perform a detailed empirical study to evaluate the impact on the productivity of **Matlab** programming when using our AOP extension. We have seen that AOP eases the production of software when applied to other paradigms, but his implications on an numerical programming environment are, to our known, a subject never studied and therefore a good research topic.

There have been some evolution on **Matlab** in order to support OO mechanisms when producing programs. Although such support is not widely used in current **Matlab** software production and OO also suffers from the "tyranny of dominant composition", failing to clearly structure the software, a study on how our language can be used under such mechanisms or, if it can not, how it needs to be adapted constitutes interesting future work.

All this potential upgrades constitute good studies on their impact in the overall language and on what needs to change to support them but above all what were the implications of such additions to create **Matlab** programs.

Appendix A

AspectJ Examples

This Appendice contains **AspectJ** code. Here we can find the examples with compilable code and testing environments.

A.1 Example 1 - Tracing and Context Exposure

A.1.1 asp.aj

```
package teste;

aspect asp {

       pointcut mypc(foo c, Object o, int i) :
          args(i) && call( void foo.bar(int) ) && target(c) && this(o);

       before(foo c, Object o, int i) : mypc(c,o,i) {
           System.out.println("Entering: " + thisJoinPoint);
          System.out.println( "calling " + c.name + "(" + i + ") from "
                      + o.getClass().getName());
       }
}
```

A.1.2 foo.java

```
package teste;
```

```
public class foo {

    String name;

    public foo( String name ) { this.name = name; }

    public void bar(int i) {
        System.out.println("bar called with " + i);
    }

    public static void main(String args[]) {
    }
}
```

A.1.3 test.java

```
package teste;

public class test {
     public static void main( String args[] ) {
         test t = new test();
         t.doit();
     }
     public void doit() {
         foo f1 = new foo("first");
         f1.bar(1);

         foo f2 = new foo("second");
         f2.bar(2);
     }
}
```

A.2 Two Aspects

A.2.1 asp1.aj

```
package teste;
```

```
public class test {
package e8;

public aspect asp1 {

    pointcut mycut() : within(foo) && execution(* * (..));

    after () : mycut() {
        out( "after asp1");
    }

    before () : mycut() {
        out( "before asp1");
    }

   void around () : mycut() {
        out( "around1 asp1" );
        proceed();
        out( "around1 asp1" );
   }

   void out( String x ) { System.out.println( x ); };
} }
```

A.2.2 asp2.aj

```
package e8;

public aspect asp2 {

    pointcut mycut() : within(foo) && execution(* * (..));

    after () : mycut() {
        out( "after asp2");
    }

    before () : mycut() {
        out( "before asp2");
    }

   void around () : mycut() {
```

```
        out( "around1 asp2" );
        proceed();
        out( "around1 asp2" );
   }

   void out( String x ) { System.out.println( x ); };
}
```

A.2.3 foo.java

```
package e8;

public class foo {

    public static void main(){}

    void bar() { System.out.println("bar called"); }
    void baz() { System.out.println("baz called"); }

}
```

A.2.4 order.aj

```
package e8;

public aspect order {
    declare precedence : asp2, asp1;

}
```

A.2.5 test.java

```
package e8;

public class test {
    public static void main( String args[] ) {
          foo f = new foo();
            f.bar();
        }
```

```
    }
```

A.3 Advice to advice

A.3.1 asp1.aj

```
package e20;

public aspect asp1 {

    pointcut barcut() : call(void foo.bar());

    before () : barcut() {
        out( "advice by asp1");
    }

   void out( String x ) { System.out.println( x ); };
}
```

A.3.2 asp2.aj

```
package e20;

public aspect asp2 {

    pointcut barcut(foo c) : !within(asp2) &&
        call(void foo.bar()) && target(c);

    before(foo c) : barcut(c) {
        c.bar();
    }
}
```

A.3.3 foo.java

```
package e20;

public class foo {
```

```
    public static void main(){}

    void bar() { System.out.println("bar called"); }

}
```

A.3.4 test.java

```
package e20;

public class test {
    public static void main( String args[] ) {
          foo f = new foo();
            f.bar();
        }
    }
```

Bibliography

[ADDH10a] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. Aspectmatlab: an aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[ADDH10b] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. Aspectmatlab: an aspect-oriented scientific programming language. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 181–192, New York, NY, USA, 2010. ACM.

[ADMTH09] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 243–254, New York, NY, USA, 2009. ACM.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. *An Overview of CaesarJ*. http://caesarj.org/, 2006.

[AP02] George Almási and David Padua. Majic: compiling matlab for speed and responsiveness. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, New York, NY, USA, 2002. ACM.

[CBE+00] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Comput. Surv.*, page 41, 2000.

[CBHV10] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In Rajiv Gupta, edi-

tor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 2010.

[Cc07] Grigoreta Sofia Cojocar and Gabriela Şerban. On some criteria for comparing aspect mining techniques. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 7, New York, NY, USA, 2007. ACM.

[CCHW04] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.

[CDM+10] João Cardoso, Pedro Diniz, Miguel P. Monteiro, João M. Fernandes, and João Saraiva. A domain-specific aspect language for transforming MATLAB programs. In *Fifth Workshop on Domain-Specific Aspect Languages*, March 2010.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CFM06] João Cardoso, João Fernandes, and Miguel Monteiro. Adding aspect-oriented features to matlab. In *workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT! 2006)*, March 2006.

[CHJvdB10] José M. Conejero, Juan Hernández, Elena Jurado, and Klaas van den Berg. Mining early aspects based on syntactical and dependency analyses. *Sci. Comput. Program.*, 75(11):1113–1141, 2010.

[CKF00] Y Coady, Gregor Kiczales, and M. Feeley. Exploring an aspect-oriented approach to operating system code. In *OOPSLA '00: Position paper for the Advanced Separation of Concerns Workshop at the conference on Object-oriented programming, systems, languages, and applications*, page 163, New York, NY, USA, 2000. ACM.

[DeR96] Luiz A. DeRose. Compiler techniques for matlab programs. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1996.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[Fot09] Dave Foti. Matlab digest. Technical report, Mathworks, 2009.

[HG06] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[HH05] Desmond J. Higham and Nicholas J. Higham. *Matlab Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005.

[IEE85] IEEE. Ieee standard for binary floating-point arithmetic. Technical report, IEEE, 1985.

[ILG+97] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, London, UK, 1997. Springer-Verlag.

[JB03] Pramod G. Joisha and Prithviraj Banerjee. Static array storage optimization in matlab. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 2003. ACM.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[KMT07] Andy Kellens, Kim Mens, and Paolo Tonella. Transactions on aspect-oriented software development iv. pages 143–162, 2007.

[LDS05] Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to cobol? In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 99–110, New York, NY, USA, 2005. ACM.

[Lop97] Cristina Isabel Videira Lopes. D: A language framework for distributed programming, 1997.

[Mac10] Helder Nuno Ribeiro Macedo. A strategic-based weaver for aspect-matlab, 2010.

[Mar10] Pedro Martins. Zipper-based embedding of modern attribute grammar extensions. In *Doctoral Symposium of the 3rd International Conference on Software Language Engineering (SLE10)*, October 2010.

[MF06] M. Monteiro and J.M. Fernandes. Towards a catalogue of refactorings and code smells for aspectj. In *Transactions on Aspect-Oriented Software Development (TAOSD), Springer LNCS vol. 3880/2006, p. 214 - 258*, March 2006.

[MJS10] Monteiro M.P., Cardoso J., and Posea S. Identification and characterization of crosscutting concerns in matlab systems. In *Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2010)*, September 2010.

[MMK+97] Anurag Mendhekar, Anurag Mendhekar, Gregor Kiczales, Gregor Kiczales, John Lamping, and John Lamping. Rg: A case-study for aspectoriented programming. Technical report, 1997.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in matlab. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, New York, NY, USA, 1999. ACM.

[OKH+95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. *SIGPLAN Not.*, 30(10):235–250, 1995.

[OT01] Harold Ossher and Petri Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 821–822, Washington, DC, USA, 2001. IEEE Computer Society.

[Par79] D. L. Parnas. *On the criteria to be used in decomposing systems into modules*, pages 139–150. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[RM07] Damijan Rebernak and Marjan Mernik. A tool for compiler construction based on aspect-oriented specifications. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 11–16, Washington, DC, USA, 2007. IEEE Computer Society.

[RN10] Pedro C. Diniz Ricardo Nobre, João M.P. Cardoso. Leveraging type knowledge for efficient matlab to c translation. In *15th Workshop on Compilers for Parallel Computing, Vienna University of Technology, Vienna, Austria (to appear)*, 2010.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in aop with aspectc++. In *Proceeding of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques*, pages 33–53, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press.

[VSS09] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 245–256, New York, NY, USA, 2009. ACM.