Helder Nuno Ribeiro Macedo

A Strategic-Based Weaver for Aspect-MatLab Design and Implementation

UMinho | 2010

Universidade do Minho
Escola de Engenharia

Helder Nuno Ribeiro Macedo

A Strategic-Based Weaver for Aspect-MatLab Design and Implementation

Outubro de 2010

Universidade do Minho
Escola de Engenharia

Helder Nuno Ribeiro Macedo

A Strategic-Based Weaver for Aspect-MatLab Design and Implementation

Dissertação de Mestrado
Mestrado em Informática

Trabalho efectuado sob a orientação do
Prof. Dr. João Saraiva
e do
Dr. Jácome Cunha

Outubro de 2010

Universidade do Minho

Helder Nuno Ribeiro Macedo

A Strategic-Based Weaver for Aspect-MatLab
Design and Implementation

Tese de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Prof. Dr. João Saraiva e Dr. Jácome Cunha

Novembro 2010

Acknowledgements

I am grateful to Professor João Saraiva and Jácome Cunha for their guidance and helpful comments during the preparation and deployment of this dissertation.

My parents, Joaquim and Rosa, played the most important role in my academic and personal life and for that I am eternal grateful. I also would like to thank all of my friends for the support provided and the good relaxing times.

I am also grateful for the opportunity of being part of AMADEUS project: Aspects and Compiler Optimizations for MatLab System Development funded by Fundação para a Ciência e Tecnologia (FCT). For the providing suggestions and brainstorming meetings, I am thankful to João Cardoso, Miguel Monteiro and João Fernandes.

Finally, working with TOM framework was a good challenge, thus I would like to express my appreciation to the support team of TOM for their help in some implementation issues, particularly to Pierre-Etienne Moreau and Emillie Balland.

Resumo

MatLab é uma linguagem de programação amplamente utilizada em computação científica, sistemas de controle, processamento de sinais, processamento de imagem, engenharia de sistemas, simulação, etc. É uma linguagem interpretativa e imperativa, que no entanto é desprovida de mecanismos eficientes de modularidade. Com efeito, durante o ciclo de desenvolvimento os programadores podem ter que manter várias versões de seus programas. Isto é particularmente problemático no contexto da computação embebida, onde diferentes hardwares/configurações têm de ser consideradas. *Programação Orientada aos Aspectos* (POA) é uma metodologia de programação recente, fortemente centrada na separação de preocupações, que tem o propósito de facultar um poderoso sistema de modelação para linguagens de programação.

Esta dissertação tem por objectivo definir uma metodologia para construir um Weaver conforme os princípios da programação estratégica e de forma a enriquecer o MatLab com características da POA. O processamento de aspectos é a componente do sistema que, dado um programa (padrão) MatLab e uma especificação de vários aspectos, gera um programa MatLab que incorpora a acção desses aspectos no seu código. Tal componente do programa é chamado de Weaver. A abordagem adoptada para construir este interpretador foi fundamentada nos princípios de *Programação Estratégica*, que oferece um controle genérico e completo para travessias de árvores abstractas. No contexto desta tese, apresentamos a implementação e desenvolvimento deste Weaver utilizando o paradigma de programação estratégica.

Abstract

MatLab is a programming language widely used in scientific computing, control systems, signal processing, image processing, system engineering, simulation, etc. It is an interpretative and imperative language, that suffers from the lack of modularity. Indeed, during the development life cycle the programmers may have to maintain multiple versions of their programs. This is particularly problematic in the context of embedded computing, where different hardware/configurations have to be considered. *Aspect-Oriented Programming* (AOP) is a recent programming methodology that heavily focused on the separation of concerns, and thus providing a powerful module system for programming languages.

This dissertation aims to define a methodology to create a *Strategic-Based Weaver* for an AOP extension to MatLab. The aspect weaving process is the system component that given a (standard) MatLab program and the specification of several aspects, builds a MatLab program that embeds such aspects on its code. Such program component is called weaver. The adopted approach to build this interpreter was based on *Strategic Programming* principals, which offers a generic and full traversal control for crossing AST. In this thesis context, we present the implementation of this weaver developed using the strategic programming paradigm.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Currently, there are a diversity of programming languages that in their own specific way offers some implementation features for systems conception. Systems productivity is an increasing notion on systems development. The lacking of modular techniques in programming languages leads to an improper identification of the software functionalities. Thereby, the evolution of some modern programming languages, such object-oriented (Java), already embrace the decomposition of system into its functionalities (e.g. classes).

MatLab [15] is a high-level, interpreted, *Domain-Specific Language* (DSL), mainly based on matrix data-types and operations on them. The MatLab environment, the richness of the language, the existence of domain-oriented packages and the associated software tools make the language one of the preferred choices to model and simulate complex systems [15].

However, this language suffers from the absence of modularity. Enforcing logic boundaries into similar components in MatLab projects, makes the code more efficient and less error-prone. Another major problem dealing with the separation of concerns is that changes in the original code must be done. This manual code refactoring is a time consuming practice and the occurrence of code errors increases drastically. Along with this, the developer must maintain different versions of their program to ensure that none functionality is lost. All these drawbacks of standard MatLab programs present, a motivation to introduce *Aspect-Oriented* features into MatLab.

In the *Aspect-Oriented Programming* (AOP) paradigm, the transformation program that processes the aspect-oriented extension is called *Weaver*. The challenge for this dissertation is to study how to efficiently implement this Weaver to support aspect features (Aspect-

MatLab). The automatic composition of aspects with the source code is one of the biggest advantages of this process. In general, given a MatLab standard module and a specification of several aspects, the Weaver analyze them and weave the aspects to incorporate a new MatLab module as a result (see Figure 1).

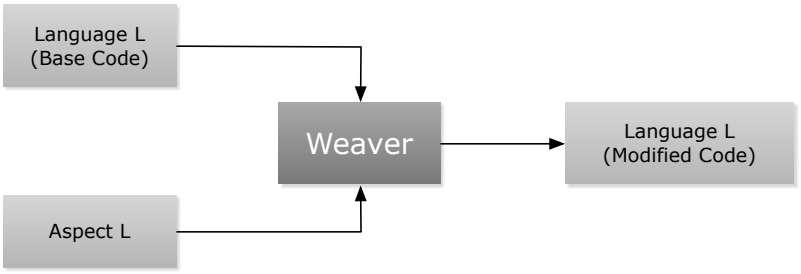

Figure 1.1: General weaving process.

In this project, the approach to conceive this Weaver heavily relies on *Strategic Programming* (SP). SP is a generic programming technique that defends the use of strategies for traversal heterogeneous data structures, considering the type-specific behavior of the program. For this project, it will be used the *TOM* [8] framework which is a strategy language extension in Java that provides powerful control features over data tree structures.

1.1 Motivation

MatLab is not unheard of among the computer engineering community. Many systems use or incorporate its functionalities in order to efficiently improve it. However, this language is not perfect, and tasks such as including handlers to watch specific behaviors, among others, are extremely cumbersome and tedious.

In order to improve the MatLab language, the AMADEUS project proposes an introduction of aspect language to solve many problems associated to its lack of modularity. The efficiency that such task requires in processing both aspects and MatLab programs, urges the need of implementing a Weaver. It will be responsibility of this Weaver to incorporate a set of aspects, as normal program instructions, in the original source code.

Furthermore, this study will reflect efficient techniques to build aspect-oriented compilers and its use in the context of Aspect-MatLab language. In section 2.1, we discuss in great detail why the Strategic Programming is a competent solution for the Weaver's

construction. The generic and concise implementation that this programming paradigm provides [20, 21, 25, 26], makes it possible to update and maintain the Weaver's features.

This dissertation presents an approach to build a Strategic-Based Weaver and shows a possible path that might motivate implementing a similar Weaver for other aspect languages.

1.2 MatLab

MatLab is an interpretative and imperative language (similar to C) that handles primary double precision matrix as basic data-types [10]. It is a competent tool in areas such system engineering, signal an image processing, control systems, etc.

MatLab offers a set of useful features and packages that guarantee a high level system's implementation. This high expressiveness is characterized by features like functions polymorphism, operator overloading and dynamic type specialization. Function polymorphism enables the function executions with distinct types and numbers of arguments. Operator overloading means the possibility of an operation in MatLab be used with different types, like the operator '*' can have both integer or arrays as arguments. Dynamic type specialization enables, during runtime, the possibility of the variables representing distinct data-types.

```
function printNumbs(startval, inc, endval)
  val = startval;

  while (val <= endval)
    fprintf(1, '%.2f\n', val);
    val = val + inc;
  end
```

Listing 1.1: MatLab printing function example.

In order to offer some insight over MatLab programming language, Listing 1.1 present a very elucidative example. The *printNumbs* function codifies the printing action of all numbers within the *startval* and *endval* variables, according to a specific increment (*inc*). For instance, executing this function with this arguments: *startval=1, endval=8* and *inc=3*, the result will be priting *{1,4,7}*.

1.3 Systems Modularization

A large scale of software systems are complex to develop, maintain and update. Thus, such systems should be decomposed into several smaller and simpler components (solving specific problems) that are easier to build, sustain and improve. The overall software systems is then a combination of these components. This approach to software development is an ancient goal of programmers and researchers [23]. As a consequence, modern programming languages provide mechanisms to structure our programs in a modular way. However, more recently and powerful techniques to modularize software have been provided, like aspect-oriented programming.

1.4 Aspect-Oriented Programming Paradigm

Currently, in the software industry the systems modularization is still under studying, but yet with some weak approaches. Therefore, this brings numerous problems related to the fact that software implementations are imprecise. Code scattering and code tangling are some of these issues, which result from the unstructured implementation of the system functionalities causing its maintenance extremely difficult [13].

Separation of concerns is a concept increasingly in vogue in software engineering. *Aspect-Oriented Programming* (AOP) is the paradigm that attempts to deal with this notion. This paradigm advocates a careful organization of the source code, to guarantee the maintenance of its functionalities and at the same time to set up a competent way to add new features. There are some important concepts related to AOP and their definitions are vital to understand this paradigm. The following definitions deserves our attention [4, 14]:

1. **Concerns:** are characteristics referring to high-level requirements (mostly remote to non-functional requirements, like the interaction with the users) and to low-level implementation issues (requirements more related to how the system should be implemented).

2. **Crosscutting concerns:** most of the concerns related to a program are due to various system modules. More precisely, this concept is related to when two (or more) properties of the program behavior must be coordinated, although they are composed differently. For instance, let us considered the following example:

```
public class Test {
  public static void main(String[] args) {
    Test t = new Test();

    t.HelloWorld();
  }
  void HelloWorld() {
    System.out.println("Hello␣world!");
  }
}
```

Listing 1.2: Class test for crosscutting concerns.

Listing 1.2 contains a very small Java class but we can infer some crosscutting concerns over its implementation like: test if the variable *t* is initialize or tracing when a method is invoked.

3. **Join points:** are precise points in the control flow of a program where some specific behavior can be attached, working as a reference to a structure of aspects. The list of the join points present in a program may differ, since the introduction of new aspects causes the definition of more join points. Considering the Listing 1.2 example we can apply a join point capture over the class implementation.

```
pointcut methodCall() : call(void Test.HelloWord());
```

Listing 1.3: Join point for the class Test (AspectJ syntax).

The verification of when the method *HelloWorld()* of the class *Test* is called, is the main objective of this join point.

4. **Advice:** is the procedure that represents the behavior to be applied to a given join point. Advices can be called and executed according to their function. Therefore, they can run before, after, instead of, or around specific join points. For the join point specification in Listing 1.3 we can attach the action that must be trigged, when it occurs .

```
before() : methodCall() {
    System.out.println("Method␣HelloWorld␣is␣about␣to␣be␣called!");
}
```

Listing 1.4: Join point advice for the class Test (AspectJ syntax).

By this definition, when the method *HelloWorld()* is called the warning message, on the definition above, must be added before the join point matching instruction.

5. **Aspects:** are the components that define how to implement a concern and where, when and how to invoke it. Like a concern, an aspect is connected to many parts of the system and can be distinguished based on the function that it has in the program (e.g. control, memory). To implement a concern where tracing the calls of the *HelloWorld()* method, presented in Listing 1.2, is the goal then it is only necessary to structure the join point and advice previously defined. Next, we present such aspect:

```
aspect Tracing {
  pointcut methodCall() : call(void Foo.HelloWord());

  before() : methodCall() {
    System.out.println("Method␣HelloWorld␣is␣about␣to␣be␣called!");
  }
}
```

Listing 1.5: Aspect for the class Test (AspectJ syntax).

This paradigm brings new concepts to justify a separation of concerns more efficient than paradigms such as *Object-Oriented* (OO), which already have some weak characteristics for modularization (e.g. methods, classes) [13]. Having all the core functionality of the system and the aspects definition, it is important to incorporate a process to organize these two modules.

The general process of computing the aspects of a particular language, that aims to provide a more efficient code, can be described by the scheme present in Figure 1 that shows the general execution of aspects. The central theme of this dissertation is the discussion around the construction of a Weaver for our Aspect-MatLab language.

1.5 Languages

Programming languages area are primordial in software industry because it represents the only way to define logical computations to be executed. However, a set of syntactic and semantic rules must be followed, according to the language that is being used.

1.5.1 Grammars

In this Weaver conception the grammar notion appear as the way to define both MatLab and Aspect-MatLab syntax languages. Grammars offer a very simple and readable form to specify languages, contrasting to the definition of using formal languages that cannot always be specified [1]. Notions like language generator, recognizer and parsing appears attached to this concept. A grammar is defined by a set of logic rules (recursive and nested structures) that specify the syntax language. Using systematic processes on these rules it can be proved if a string is grammatically correct. Some of the methodologies allow the syntax validation by producing programs for that effect, like tools that generate parsers [1].

In the Aspect-MatLab context the parsing is accomplished by the ANTLR [24] tool. However, the abstract signature for this language is mandatory and its formal specification and detailed explanation can be found in section 4.3.2.

1.5.2 Abstract Data-Type

In the context of *abstract data-type* there is an abstraction of some characteristics that do not define the core structure of a language. An abstract data-type is an algebraic specification that only encodes the logical operations that sustains the desired structure without considering concrete information, like syntactic sugar.

This formal specification is very clean and effective because it allows a simple definition for a data structure that recognizes some particular language without the constraints related to reserved words or another auxiliary information. Additionally, the manipulation for this type of structures is less complex mainly because the number of conditions to verify is also lower.

The result of all parsers developed in this project are *Abstract Syntax Trees* (AST's). Such trees are usually modeled by abstract data-types. For the Aspect-MatLab specification one of the goals is to make this formalism an embedded *Domain-Specific Aspect Language* (DSAL) in Java. A more detailed exposure of this characteristic is discussed in chapter 4.

1.5.3 Exposure in the project context

In context of this project, the Weaver is a program that receives as argument a core/standard MatLab program the aspect definition and produces a different MatLab program. Thus, the

Weaver can be seen as a program that analysis and transforms MatLab programs. In order to be able to reason about MatLab programs we need to define a front-end to the aspect language (ANTLR/TOM).

AMADEUS project is divided into several development teams. Thus, because we have a front-end for MatLab already constructed in Java environment, justifies the use of TOM as the strategic framework to conceive this Weaver.

TOM is a framework that is used to define the abstract behavior of Aspect-MatLab language. TOM works with abstract rule-based systems, therefore no need to consider the lexical context that a language encompasses. However, in order to check the aspect semantics urges the need to build a parser. Such parsing will be implemented in ANTLR that offers a vast range of options to automatically generate a parser based on the defined grammar rules.

1.6 Dissertation Content

This document aims to present a detailed overview around the conception of a Weaver that processes an aspect language extension for MatLab. In order to guide and help the understandable idea of this project, each section of the dissertation approaches fundamental concepts to build up this type of interpreters.

Since there is no aspect language recognized by *MathWorks Corporation*[1] we will attempt to introduce one capable of enriching this language with the aspect paradigm characteristics. For that propose, a brief explanation of the basis theoretical concepts, like Aspect and Strategic (see section 2.1) Paradigms, clarifies the main concerns related to the Weaver functionalities.

In this context, an explanation of the strengths of Strategic Programming is mandatory (see chapter 2). An ample source of examples (all developed in AMADEUS project) and a detailed description of the strategic systems (see chapter 3) will appear as a complementary source to increase the comprehension of the key concepts that support the Weaver implementation.

The architecture and design models (see section 4.3) presented are capable of processing a standard MatLab program and embed the correspondent aspect specification (see

[1] company that reserves all copyrights of MatLab

section 4.5). However they have limitations, serving the purpose to enforce the researching in this area, serving this Weaver as the starting point.

Additionally, we suggest some improvements (see chapter 5) that this interpreter should suffer to improve the processing of aspects definitions and the usability in the user point of view.

Chapter 2

Strategic Programming Paradigm

In this project concept, the construction of a Weaver has to ensure a mechanism to process MatLab programs and aspect features. Such approach will follow the *Strategic Programming* fundamentals. The main advantage of strategic paradigm resides on the many techniques that define traversal schemes.

Strategic Programming is a methodology that aims to provide a full traverse control, i.e., in this programming paradigm programmers get for free functions that traverse abstract syntax trees using a pre-defined recursion pattern (the strategy). Two key advantages are implied in this paradigm:

- Firstly, when traversing (possibly) large trees, only a small set of nodes have to be consider - the ones where work have to be done.
- Secondly, if the tree changes, due to the introduction of new language constructors, etc, a strategic function may not be affected by such language evolution.

The following sections provide a detailed description of the strategic programming concepts and its main contributions for strategic systems development, which allows the implementation of rule based systems and transformations on them.

2.1 Theoretical Concepts

Strategic Programming is a generic programming technique to processing heterogeneous data-type (e.g. terms, objects). This strategic approach provides many techniques that

are very useful, particularly to define traversal schemes. By introducing new forms of abstraction and modularization, strategic programming provides high level of conciseness, composability, structure-shyness, and traversal control [26].

Strategic programming is a methodology that aims to provide a full traverse control. Traverse control is the process that performs logic basic actions to the right data-type in the right order, providing an understanding and control on which (order) data is visited and under which conditions. In this way, strategic programming allows an apprehension of reusable generic traversal schemes [19].

One of the issues about the best traversal scheme to use is related with what type of data we want to collect, especially in heterogeneous data structures. Therefore, sometimes it has to be applied more than one traversal scheme in order to obtain the desired result. This represents one of the capabilities of strategic programming that is the possibility to define new traversals through the composition of strategic schemes that can be reused within applications.

The methodology around the strategic concept is based on the identification of the problem specific ingredients, with the recognition of the respective reusable traversal schemes, and synthesis of the traversal by parameter passing. These ingredients represent the data processing according to type-specific (or generic) actions with specific branches [19]. The traversal scheme associated can range from a type-specific problem to a more typical generic traverse (reusable).

There are many characteristics around the foundations of strategies, which contribute to clarify and motivate its implementation. These characteristics are the following [26]:

- **Genericity:** is one of the most important aspects around the concept of strategy, because its application covers a large heterogeneous data-type.

- **Specificity:** although most of the strategies are generic, they can be applied to a specific problem by defining correct operations.

- **Composability:** refers the possibility to express the conditional, compound and iterated strategy application.

- **One-layer traversal:** represent the capacity of enabling generic traversals to the datum of heterogeneous data-structures, and its direct subcomponents.

- **Partiality:** it is an important characteristic when the failure of a strategy in a datum occurs, because it is possible to recover from this situation.
- **First-class:** this aspect symbolizes the fact that a strategy can be named, passed as arguments, etc.

Strategic programming can efficiently incarnate into any programming paradigm or language based on all of this characteristics. At the same time, these references provide useful information on reviewing other generic approaches.

In strategic programming environments there are many traversal schemes that can be applied depending on which result and problem that is being consider. Thus, we have schemes that make a full traverse of the tree to apply some strategy, while some only have the concern to find the first datum where the argument strategy succeeds.

In the next sections we will emphasize the range of traversal schemes that this paradigm covers, more precisely the none-traversal control (section 2.1.2) and the traversal control (sections 2.1.3, 2.1.4, 2.1.5 and 2.1.6).

2.1.1 Basic Combinators

All the traverse schemes has its definition based on some primitive strategy combinators. Its main goal is allowing the specification of patterns to more complex tree traverse over the incoming data. In strategic paradigm such basic combinators are:

Strategic basic combinators	
id	Identity strategy
fail	Failure strategy
seq(s, s)	Sequential composition
choice(s, s)	Left-biased choice
all(s)	All immediate components
one(s)	One immediate component
adhoc(s, a)	Type-based dispatch

Table 2.1: Strategic basic combinators definition.

The strategy *id* always succeeds in any datum, thus the returned result is the unchanged input term. On the opposite side, the *fail* combinator never succeeds on any datum, reacting to any input term with failure. The sequential composition (*seq*) applies its two argument in

a successively form, the first argument first than the second over the outcome result of the first . With the same number of arguments, strategy *choice* attempts to execute firstly its first argument, and only if it is not succeed is executed the second argument. The combinator *all* applies its argument to all input's immediate sub-terms. Combinator *one* has a similar behavior to *all*, but *once* only applies the argument to a single immediate sub-term of the input. The type-based dispatch (*adhoc*) apply the second argument if matches the type of the input term, otherwise apply its first argument.

In the following sections we present how the cooperation between these primary combinators can define the generic traverse schemes to process heterogeneous data-type.

2.1.2 Non-Traversal Control

The strategy combinators presented in this group do not provide traversal control. The traversal control is the concept that defines how basic actions are applied in which order, and what specific conditions must be respected [26]. Thus, the combinators that this group covers are *try* and *repeat*. These combinators are useful in the characterization of traversal schemes. They define the application of actions, which can be repeatedly (*repeat(s)*) applied or invoked only once (*try(s)*). Their semantic definitions are:

$$try(s) = choice(s,id)$$
$$repeat = try(seq(s,repeat(s)))$$

These definitions suggest the way in which these two combinators work. The *try* combinator tries to apply its argument strategy *s*, but if it fails returns the *id*. In the *repeat* case, apply continuously the argument *s* until it fails.

2.1.3 Full Top-Down and Bottom-Up

The combinators *full_td* and *full_bu* represent the traverse scheme of full top-down and full bottom-up, respectively. These two models apply the argument strategy to the datum, and to all immediate or non-immediate components, of a specific data structure. The only difference between them resides on the first datum that they approach. While top-down starts applying the strategy at the root of an *Abstract Syntax Tree* (AST) and its subnodes, bottom-up approaches firstly the terminals of this tree until reaching the root. For better

understanding of the functionality of these two models, it we present the definition in terms of the semantics of the strategy combinators:

$$full\_td(s) = seq(s,all(full\_td(s)))$$
$$full\_bu(s) = seq(all(full\_bu(s)),s)$$

In these two definitions, it is possible to verify that the argument strategy s is applied to a received datum and it is continuously applied to its sub-components. These models are very useful when a full application of the argument strategy to all nodes (to a data structure) is required. Figure 2.1 shows the behavior that the *full_td* combinator has in the tree traverse.

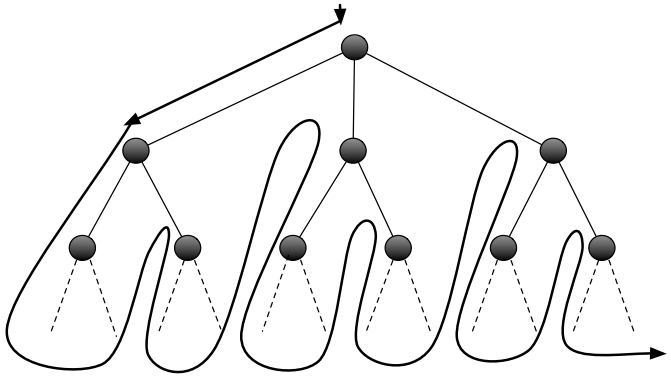

Figure 2.1: Full top-down combinator approach.

2.1.4 Once Top-Down and Once Bottom-Up

Any of these modules apply the argument strategy until it finds a component at which it succeeds. In the same way as full *top-down* and *bottom-up*, the *once_td* and *once_bu* have similar behavior. The only difference resides in the fact that both, *once top-down* and *bottom-up*, stop when it's found the first parameter that respects this argument, contrasting with the full traversal schemes. The definition of these modules in strategy combinators semantic is the following:

$$once\_tp(s) = choice(s,one(once\_tp(s)))$$
$$once\_bu(s) = choice(one(once\_bu(s)),s)$$

In these definitions the only highlight property is the restriction of the *one* combinator that marks the acceptance of the argument strategy only for the first successful parameter. This is the main difference to the traversal schemes described above. They represent useful models when the traverse of a tree only requires the identification of the first component that respects the argument strategy *s*. Figure 2.2 illustrates the path that these two combinators follow in the traverse of a tree.

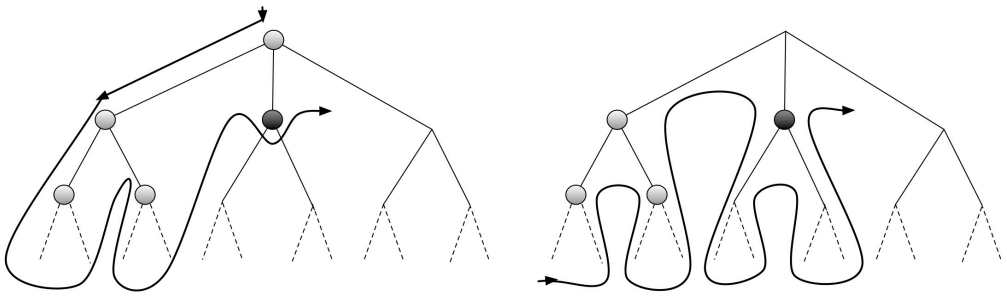

Figure 2.2: once_tp(left) and once_bu(right) combinators.

2.1.5 Stop Top-Down

The *stop_td* combinator has a different philosophy than *full_td*. The *stop_td* does not perform a total traverse of the tree, instead it only applies the argument strategy until one branch fulfils the restrictions presented in this application. However, like *full_td* this application starts in the root of the tree and proceeds applying the argument to the immediate components. The only reason that distinguishes these two models is that the *stop_td* can finish the traversal before crossing all components. The *stop_td* can be also expressed in semantics of the strategy combinators:

$$stop\_td = choice(s, all(stop\_td(s)))$$

This definition is very similar to the *full_td*, but the semantic *choice* sets the traverse stopping when the argument strategy is fulfilled by any branch of the tree. This combinator provides a powerful tool when a full traverse is unwanted, in contrast of reaching the branch where an argument strategy has a successful application. In order to show the behavior of this combinator, Figure 2.3 presents the path executed by *stop_td*.

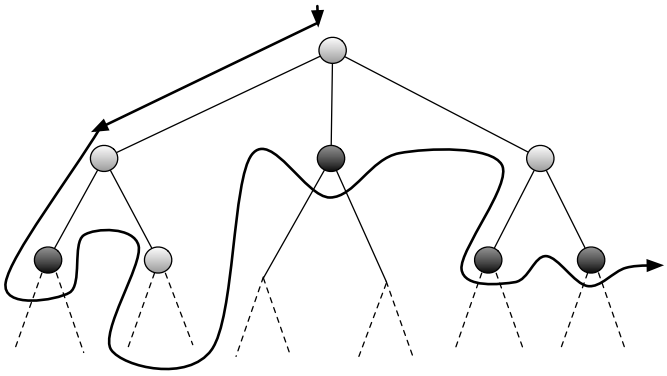

Figure 2.3: Stop top-down approach.

2.1.6 Innermost and Naive-Innermost

These combinators represent the *leftmost innermost* evaluation strategy. In any of these strategies, the argument strategy is evaluated from left to right in the traversed tree. Although they implement the same evaluation strategy, the innermost combinator is more efficient. They can be defined as semantics of the strategy combinators:

$$innermost(s) = seq(all(innermost(s)),try(seq(s,innermost(s))))$$
$$naive\_innermost = repeat(once\_bu(s))$$

2.2 The Strategic Paradigm in TOM

ToOne Matching (TOM) [8] is an embed *Domain-Specific Language* (DSL) in Java, inspired in rule based systems like ELAN [9], Stratego [20] and JJTraveler [12]. TOM represents the incarnation of strategic programming in the object-oriented paradigm.

Like the methodology that generic strategic paradigm defends, TOM provides mechanisms to ensure a full traversal control, by introducing a strategic library capable of processing traverse in any data-type. The way that any strategy is applied has the following generic procedure:

- Chose the correct node (subject) to visit.
- To each visit select the default behavior of the strategy.
- Define the argument strategy action to be performed.

There are two ways to apply a strategy to term: *visitLight* and *visit*. These two methods are specified in the *Visitable* interface. Although the *visit* and *visitLight* options have similar behaviors, *visitLight* is more efficient because it do not take into account the idea of environment. Environment is the notion that maintains the current position and the next subnode where the strategy is applied. In cases where these characteristics are needed *visit* appear as the efficient method, updating at each step those environment variables.

(Identity)[t]	⇒	t
(Fail)[t]	⇒	failure
(Sequence(s1,s2))[t]	⇒	failure if (s1)[t] fails (s2)[t'] if (s1)[t] ⇒ t'
(Choice(s1,s2))[t]	⇒	t' if (s1)[t] ⇒ t' (s2)[t] if (s1)[t] fails
(All(s))[f(t1,...,tn)]	⇒	f(t1',...,tn') if (s)[t1] ⇒ t1', ..., (s)[tn] ⇒ tn' failure if there exists i such that (s)[ti] fails
(All(s))[cst]	⇒	cst
(One(s))[f(t1,...,tn)]	⇒	f(t1,...,ti',...,tn) if (s)[ti] ⇒ ti' failure (s)[t1] fails, ..., (s)[tn] fails
(One(s))[cst]	⇒	failure

Table 2.2: TOM Basic Combinators Definition, adapted from [6]

TOM also covers the basic combinators that do not provide any control over the heterogeneous data-type. In TOM they are defined in the *sl* library and they represent the essential components to define more complex strategies. The basic combinators that TOM supports are presented in Table 2.2.

In the previous section we presented a detailed definition of all the traverse schemes and with these combinators implemented in TOM it is possible to describe the same definition in the Java environment. Although TOM is embedded in an object-oriented paradigm, the modularization of this set of traversal schemes is based on a recursive implementation. The incarnation in TOM resembles the theoretical definition, however in TOM it was defined two sets of traversals: one set for *Identity* default behavior and other for *Failure*. The

approach to apply the argument strategy is very similar but the internal definition in this framework differs in each case. The formal definitions for schemes that are extended to *Identity* are:

Try(s)	=	Choice(s,Identity)
Repeat(s)	=	μ x.Choice(Sequence(s,x),Identity())
OnceBottomUp(s)	=	μ x.Choice(One(x),s)
BottomUp(s)	=	μ x.Sequence(All(x),s)
TopDown(s)	=	μ x.Sequence(s,All(x))
Innermost(s)	=	μ x.Sequence(All(x),Try(Sequence(s,x))

Table 2.3: TOM Strategies Definition (Identity), adapted from [6]

With the same pattern and with the goal to implement more efficient strategies, especially to perform leftmost-innermost normalization, and with identity considered as failure. The definition of these traversal schemes on this core framework are:

(SequenceId(s1,s2))[t]	$\Rightarrow$	(s2)[t'] if (s1)[t] $\Rightarrow$ t' with t$\neq$t' t otherwise
(ChoiceId(s1,s2))[t]	$\Rightarrow$	t' if (s1)[t] $\Rightarrow$ t' with t$\neq$t' (s2)[t] otherwise
(OneId(s))[f(t1,...,tn)]	$\Rightarrow$	f(t1,...,ti',...,tn) if (s)[ti] $\Rightarrow$ ti' with ti$\neq$ti' f(t1,...,tn) otherwise
(OneId(s))[cst]	$\Rightarrow$	cst
TryId(s)	=	s
RepeatId(s)	=	μ x.SequenceId(s,x))
OnceBottomUpId(s)	=	μ x.ChoiceId(OneId(x),s))
OnceTopDownId(s)	=	μ x.ChoiceId(s,OneId(x)))
InnermostId(s)	=	μ x.Sequence(All(x),SequenceId(s,x)))
OutermostId(s)	=	μ x.Sequence(SequenceId(s,x),All(x)))

Table 2.4: TOM Strategies Definition (Failure), adapted from [6]

TOM covers all possible scenarios for crossing AST's. The modularization of full or partial traverse is provided and the user merely raises the most accurate to apply some strategy. The specification of a strategy in TOM is quite simple and effective. The sample bellows represents a fairly succinct description of a strategy implementation.

```
%strategy TransformTerm() extends Identity() {
  visit Term {
    t() -> {`newT();}
  }
}
```

Listing 2.1: Generic TOM strategy.

This constructor is seen as an embedded TOM method in Java. The only restrictions fall within the definition of the term to visit (*Term*), which the strategy will be applied (subsequent productions, eg. *t()*), and the argument strategy definition described after the '->' symbol (to each production). This strategy is applied when a match to an element of type *t* occurs and this component is converted into another element (*newT()*) that belongs to the same visit type (*Term*) and according to a previously defined abstract signature.

In order to provide a more understandable idea of how this constructor works in TOM, Table 2.5 presents the grammar that defines its nomenclature.

StrategyConstruct	::=	'%strategy' StrategyName '(' [StrategyArguments] ')' extends' ["'] Term '{' StrategyVisitList '}'
StrategyName	::=	Identifier
StrategyArguments	::=	SubjectName ':' AlgebraicType (',' SubjectName ':' AlgebraicType)*
	|	AlgebraicType SubjectName (',' AlgebraicType SubjectName)*
StrategyVisitList	::=	(StrategyVisit)*
StrategyVisit	::=	'visit' AlgebraicType '{' (VisitAction)* '}'
VisitAction	::=	[LabelName':'] PatternList '->' ('{' BlockList '}' | Term)

Table 2.5: Strategies Grammar, adapted from [6]

According to the specification presented above it becomes clear a few constraints of

implementing strategies. The general pattern has the following properties:

- Name of the strategy;
- The strategy can have arguments or not (e.g. collections);
- Set the default behavior of the strategy (*Identity()* or *Fail()*);
- Define the set of terms to apply the strategy (*StrategyVisitList*);
- To each term define the execution action (*VisitAction*);

The incarnation in Java environment of all strategic fundamentals is complete. TOM provides the generic traverse schemes needed to process any data-type (known as GOM signature) and tools to delineate the strategy argument to perform any type of action under the selected terms. In section 3.3 we will discuss with more detail the process of building up an interpreter.

Chapter 3

Strategic Systems

Software implementation calls for a set of requirements that defines its functional behavior. Enabling such concepts, like strategic programing, in a practical environment is crucial to effectively process program transformations.

Among strategic community there are few systems that offer the demanded strategic techniques. Some of these systems are embedded in typical programming languages, like Haskell or Java, allowing express and construct rule-based systems in distinct environments. The state of art of strategic systems are vast, but some of the existent frameworks include suitable features for traverse control over AST's. The widely known and subject of countless studies are: Strafunski, Stratego/XT and TOM.

Under the conditions that this project is based, the Weaver was developed using TOM framework. More details about its conception are discussed in chapter 4.

3.1 Stratego/XT

The *Stratego/XT* framework is designed to develop and support programs transformation systems [20]. This framework is a *Domain-Specific Language* (DSL) for strategic programming and presents the interaction between the Stratego language and the XT transformation tools.

The foundations under the conception of the Stratego language are based on the paradigm of the rewriting terms under the control of strategies, process know as strategic term rewriting. The programmable rewriting strategies are extremely useful to control the applications

of basic rules that express basic transformations. To define the concrete syntax that characterizes any language that is intended to implement, is vital the identification of dynamic rewriting rules that expresses the context-sensitive transformations.

The XT toolset is a well establish infrastructure that provides an efficient parser for grammars and pretty-printing generators. The reusable attributes for transformations tools is an asset, mainly because it allows an abstraction over the implementation of traversal schemes for AST's, proved by the fact that they are generic and therefore capable of crossing any data-type structure.

Like any other strategic system, Stratego/XT ensures the term traversal by applying the strategy combinators to the rot term and correspondent sub-terms. These combinators, as referenced before, are highly generic and they define what and which order rules are applied. The Stratego/XT offers the following strategy combinators [20, 26]:

- Sequential composition (s1 ; s2)
- Deterministic choice (s1 <+ s2 ; first try s1 , only if that fails s2)
- Non-deterministic choice (s1 + s2; same as <=, but the order of trying is not defined)
- Guarded choice (s1 < s2 + s3; if s1 succeeds then commit to s2 else s3)
- Testing (where(s); ignores the transformation achieved)
- Negation (not(s); succeeds if s fails)
- Recursion (rec x(s)).

However, these combinators are only usefull to apply to the roots of each term, but if the desired result implies that one strategy must consider the directed sub-terms, Stratego/XT also provides combinators for term traversal. For example, *all(s)* applies the argument *s* to all direct sub-terms of an term [26].

Stratego/XT can be applied to a wide type of transformations, such as compilation, generation, analysis, and migration [20]. Besides the fact that Stratego is a very powerul language, is not a common programming language being completely independent of languages such Haskell or C. Problems may arise related to its self-learning about the Stratego methodology, contrasting with systems such as Strafunski where the language is pure Haskell [16]. However, the result of Stratego/XT is presented in C syntax which can mitigate this problem.

In summary, Stratego/XT is an expressive framework that supports an infrastructure for program transformation to express type-specific languages, where the strategic paradigm is strongly represented.

3.2 Strafunski

Strafunski is a Haskell-centered software bundle that aims to provide generic programming and language processing [21]. Strafunski efficiently support programming for systems analyses and transformations over language components. Like Stratego/XT, Strafunki tools can be applied in various specific areas of software engineering, like program optimization and software metrics [21, 26].

Strafunski is an *Embedded Domain-Specific Language* (EDSL) because the programming language is pure Haskell, therefore there is no need to understand new programming concepts being only necessary follow the functional paradigm characteristics that this programming language stands for.

The generic programming that Strafunski attempts to incarnate is the strategic paradigm, thus the *StrategyLib* library was augmented to support generic traversal schemes over any specific data-type based on the notion of functional strategy. Let us considered the following example [21]:

full_tdTU s x = (s x) ‘mappend‘ (allTU mappend mempty (full_tdTU s) x)

This is a small example but explanatory enough to understand the way that Strafunski handles the application of generic strategies. In this case, the strategic argument *s* is applied to the *x* term and recursively to all the immediate sub-terms, using *mempty* as initial value and considering all the intermediate results (*mappend*).

Another major tool that Strafunski offers is the DrIFT: a generative tool for supporting the use of libraries in a vast range of data-types. For better understanding the architecture behind Strafunski, Figure 3.1 is very clarifying [26].

It is possible to verify that algebraic data-types can have two different sources: by *Syntax Definition Formalism* (SDF) or by XML schemas. The dependence of libraries to build a functional language processor as a quite good illustration in the Figure 3.1. The *StrategicLib* to apply functional strategies to parse trees, the *ATermLib* for data interchange (like SDF data-type to algebraic data-types in Haskell), the *DrIFT* as the mediator of interchange

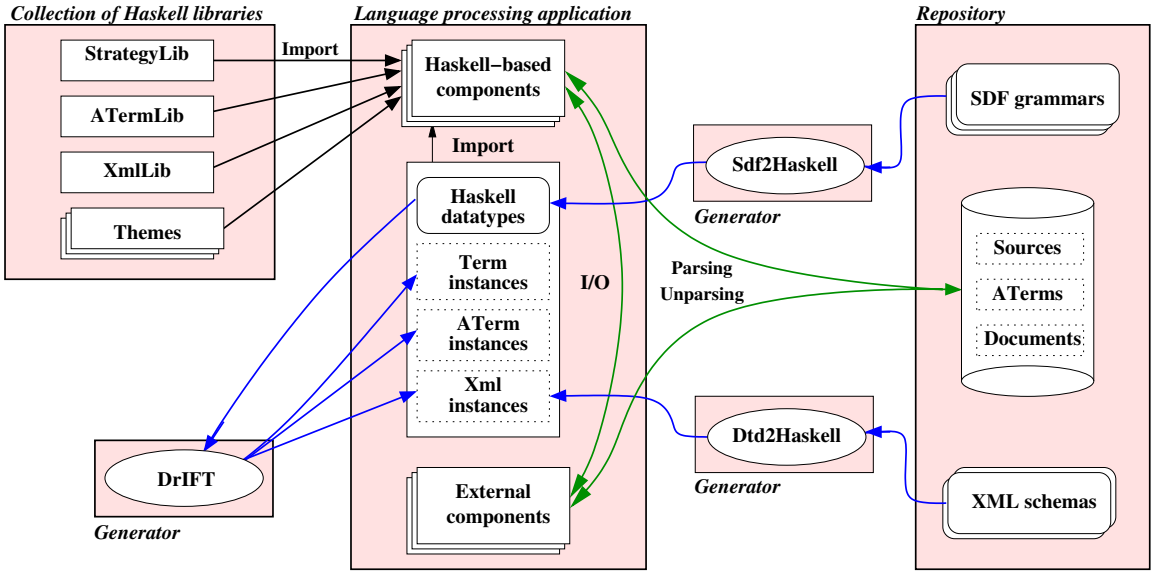

Figure 3.1: Strafunski architecture, adapted from [21].

formats process and many others powerful tools, establish Strafunski as an efficient bundle for language processing [21].

The parser is an external component for SDF that needs to be integrated in Strafunski. The SDF corresponds to the grammar specification for the language that is intended to be implemented. This parsing has a Left to right (LR) character for analyses of the SDF generic grammar [21].

Strafunski is supplied by powerful packages that enrich the language processing method. All the generators and augmented libraries around this Haskell-centered bundle maintain an efficient balance between integration with external components and generic traversals to incorporate any application to language processors.

3.3 ToOne Matching (TOM)

TOM is a framework for programming rule-based systems and along with a very capable toolset allows an efficient manipulation of tree structures and XML documents. TOM incarnates the strategic paradigm through an extension of Java language. Along with this extension TOM also provides a pattern matching compiler to look over a specific datatype [7].

One major advantage of TOM is that the underlying host language is Java, benefiting

of all extensions and libraries that Java offers [8]. All of this reinforces the notion that any Java program can become a TOM program [7].

Like others rule based systems TOM provides a very similar form, related to functional languages to implement this type of programming. *Gom* [7] is the signature that TOM offers to describe type-specific structures. The general form of a Gom signature is:

```
module Rule
abstract syntax

Term = token1
     | (...)
     | tokenN
     | Conc(a:token1,..., n:tokenN)
```

Listing 3.1: Structure of Gom signature.

In this case *Rule* is the name of the module, *Term* is the type of the sort and the tokens represent the constructors that this sort might need. For better understanding this methodology, let us considered the following example:

```
module BinTree
imports int String
abstract syntax

BinT = Node(n:int, left:BinT, right:BinT)
     | Leaf(l:int)
     | Empty()
```

Listing 3.2: Binary tree Gom signature.

This Gom signature represents a binary tree. It is possible to verify that *BinT* sort must have the Node (and correspondent left and right sub-trees), Leaf and Empty constructs to specify all the elements that characterize this type of trees. The representation is quite simple because TOM does not process any syntactic sugar.

The pattern matching in TOM is handled by the *%match* constructor. Although this constructor is executed in an imperative context, the semantics and methodology behind its implementation was based on the functional paradigm [8].

```
public static String prettyTree(BinT t){
  %match(t){
    Node(n, left, right) -> {
```

```
      return "Tree:" + ‘n + "," + prettyTree(‘left) +
      prettyTree(‘right);
    }
    Leaf(l) -> {return "␣"+‘l;}
    Empty() -> {return "";}
  }
}
```

Listing 3.3: Pretty printer using the *%match* constructor.

The left-hand side of the rules are Gom productions and the right-hand side can be either a Java statement or a grammar term. In case that is the term that has to be returned, a formal anchor (‘) in TOM has to be used, for transpose its definition to an executable statement in Java. When a match occurs it is returned the instruction present in the right-hand side, having a similar behavior to the switch/case Java operation [8]. Listings 3.3 expresses the pretty printer of a binary tree. It is possible to verify the need to transform a term to a Java statement (e.g. *‘left*) in order to present all the tree nodes.

TOM strategy language has three distinct types: *elementary strategies, recursive and parameterized strategies* and *exploration strategies*. The *elementary strategies* correspond to the ones having minimal transformation, like *Identity* that does nothing, *Fail* that always fails or a set of rewrite rules that only are applied to a root of a term. The *recursive* and *parameterized strategies* aim to gain more control by combining elementary strategies with the following basic combinators [8]:

Sequence(s1,s2)[t]	⇒	s2[t’] if s1[t] ⇒ t’ failure if s1[t] fails
Choice(s1,s2)[t]	⇒	t’ if s1[t] ⇒ t’ s2[t’] if s1[t] fails
All(s)[f(t1,...,tn)]	⇒	f(t’,...,tn’) if s[t1] ⇒ t1’,..., s[tn] ⇒ tn’ failure if there exists i such that s[ti] fails
One(s)[f(t1,...,tn)]	⇒	f(t1,...,ti’,...,tn) if s[ti] ⇒ ti’ failure if for all i, s[ti] fails

Table 3.1: TOM basic combinators.

The definition of strategies for this framework, like top-down or bottom-up, requires recursive declarations. For instance, the top-down definition in TOM is *TopDown(s)* =

μx.Sequence(s,All(x)), where the tree traversal starts at the root of an AST and is recursively applied to the immediate sub-terms. The exploration strategies are needed when the parameterization of some strategy is necessary, like collect some useful information when traversal the AST. The following example presents a situation where that is required:

```
%strategy TreeHeight(Collection c) extends Identity() {
  visit BinT{
    c@Node(n,_,_)->{c.add(‘n);}
    c@Leaf(l)->{c.add(‘l);}
  }
}

public static void SumBinTree(BinT bt, HashSet nr){
  try{
    ‘TopDown(Try(TreeHeight(nr))).visit(bt);
  }
  catch(VisitFailure e){
    System.out.println("Strategy␣TreeHeight␣failed!");
  }
}
```

Listing 3.4: Strategy tree height.

In this example the goal is to infer the nodes present in the binary tree to get as a result the sum of that tree. The traverse scheme that was used in this case was the top-down, which attempts to execute the argument strategy *TreeNode* to the *bt* tree. So, for this case it is only necessary to know the *node* and *leaf* heights present in any sample for this data type structure.

All the recursive definitions of generic traversal schemes are present in the *sl* library, allowing an abstraction of how it is implemented, invoking only the most appropriate one for AST traverse. After defining the data structure, it is only necessary the application of strategies to their components. The strategies that *sl* library supports are: *Try(s); Repeat(s); OnceBottomUp(s); BottomUp(s); TopDown(s); Innermost(s)* [8].

Another tool in TOM that deserves to be highlighted is the *Gom Antlr Adaptor*. Since TOM only support abstract grammar definition, ANTLR [24] appears as the only connection technology to TOM that provides the semantic analyzer and parser. However, the AST that ANTLR produces is not legible in the TOM environment. Thus, we have to apply an adaptor that converts this AST produced in ANTLR into a Gom tree. This is achieved by

the Gom Antlr Adaptor. Although this tool takes a Gom signature as input, so additionally is necessary the definition of the correspondent abstract signature for the concrete grammar in ANTLR.

In summary, TOM extension language offers pattern matching and rewriting to Java. This framework covers a vast application filed, providing implementation of rule-based systems, transformation on XML documents and offers a simple way to describe algebraic transformations.

3.4 Parsing Tools

In this project context a parser is vital to extend Aspect Oriented features into MatLab projects. A parser is one of the most important components of the Weaver. Its value relies on these characteristics:

- Validation of the input to its grammatical structure.
- Identifying the meaning of the input.
- Providing, with precision, the execution of the described action on the input over the data structure.

To cover the need to insert syntactic analysis over the input of aspect tokens is indispensable the implementation of such parser. The lack of resources in TOM framework in dealing with this kind of information must be aided with some external parsing generators. This constraint is where tools like JavaCC and ANTLR are strikingly vital. In next sections we present some features of these two parsing technologies.

3.4.1 Java Compiler Compiler

Java Compiler Compiler (JavaCC) is a parser generator used worldwide. The *modus operandi* of this tool is based on converting a standard grammar specification into a recognizable Java program. This tool provides some useful features to define formal rules of a specific language.

However, to describe such grammatical rules in JavaCC requires some implementation conditions. JavaCC has four major steps of building up a syntactic analyzer [18]:

1. Define the grammar in an *Extended Backus-Naur Form* (EBNF).
2. Convert the EBNF formalism into JavaCC notation (four specifications blocks).
   (a) **Options Block**: defines the control over the type of input.
   (b) **Skeletal Class Declaration Block**: defines the structure of a valid Java class within two JavaCC "flags", *PARSER_BEGIN* and *PARSER_END*.
   (c) **Lexical Specifications Block**: specifies the grammar terminals into JavaCC nomenclature to parser these input's symbols.
   (d) **Productions Block**: Transposing of each grammar production to JavaCC instructions.
3. Generate standard Java code over the JavaCC specification.
4. Incorporate the features present in the generated Java code into a Java project.

JavaCC generates LL(1), with some exceptions, parsers through top-down traverse. Nevertheless, in the definition of a grammar the left recursion (direct or indirect) is not allowed. There are some algorithms that help to remove this problem, but despite this minor draw back, this tool offers some useful functions: data tree building processor, supported by the JJTree features and the creation of documents about the grammar specification (like Java API).

For all that was exposed JavaCC appears as a very competent tool to implement parsers in Java context.

3.4.2 ANother Tool for Language Recognition

ANother Tool for Language Recognition (ANTLR) is a framework that offers a wide range of tools for the manipulation of rule-based systems. It is continuously being updated, increasing the number of useful features for language processing.

Like JavaCC, ANTLR generates parsers by applying a top-down analysis through the grammar definition. The difference lies in the LL parsing since ANTLR has a LL(*) (no restricted number of tokens to lookahead) behavior. Still, ANTLR does not allow the specification of left recursion rules but offers some useful information to rebuild a grammar within ANTLR parameters [24].

A formal structure must be followed in order to implement DSL's in ANTLR. Such structure is composed by:

```
grammarType grammar name;
≪optionsSpec≫
≪tokensSpec≫
≪attributeScopes≫
≪actions≫

start rule :...|...|...;
rule1 :...|...|...;
rule2 :...|...|...;
...
```

Table 3.2: ANTLR Grammar Structure, adapted from [24]

For example, having the complete specification for a *G* type grammar, the ANTLR has some embedded tools that split its structure into the specific functionalities (in Java context) [24]:

- GLexer.java encapsulating the lexical rules;
- GParser.java containing the parsing specifications;
- G.java containing the tree grammar;
- G.tokens containing all the grammar vocabulary;

The generic parsing process that ANTLR provides has some particular aspects, like the path present in Figure 3.2. The tree walker feature offers some interesting information, like actions in the AST . Particularly, it supplies the grammar implementation with good debugging messages that might appear in some rule definition.

Additionally, for the specification of the grammar rules, ANTLR offers a very capable GUI editor (ANTLRWorks) that allows the implementation and validation of the grammar. In order to embed such specification in any Java project there are some pre-build methods that allows the recognition of that data structure.

In summary, ANTLR supports an extensive range of features for DSL's implementation, therefore yielding an effective parser generator tool. But the lack of strategic approach to build up an interpreter must be filled with some external framework. The next section, we present a small example describing this relationship with TOM framework.

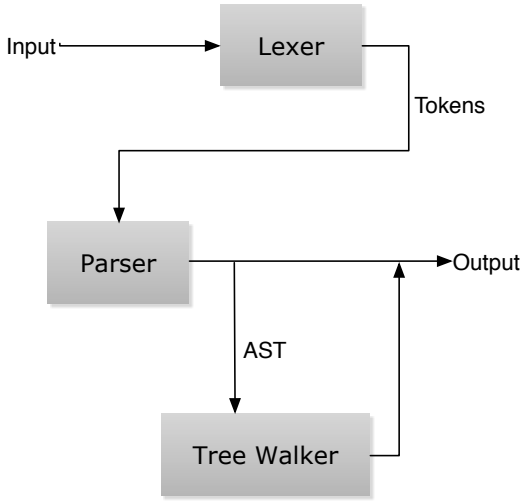

Figure 3.2: ANTLR data flow diagram, adapted from [24]

3.5 Regular Expressions Case Study

This section will attempt to show the phases that the conception of this Strategic-Based Weaver for the Aspect-MatLab must pass. After giving a brief overview about the TOM framework and its capabilities, it is clear the needing of implementing two main aspects: concrete and abstract signature for a specific language; and the implementation of the correct strategies for processing this language. For this case study it will be considered the implementation of an interpreter that is cable of processing and normalize regular expressions.

Regular Expressions are widely used in the recognition of character patterns and its most important application lies in the description of basic symbols of a programming language. First it is required a deep analyses of what are the main characteristics present in this language. Elementary regular expressions are composed by simple literals (e.g. a). But in another level these expressions can be composed by:

- Sequence of expressions: ab, a(cd);
- Alternative combinator (boolean or), allows the separation between two expressions: a|b, (ab)|c;
- Optional combinator, which defines the occurrence of the expression between zero or one time: a?, a(bc)?;

- Star combinator defining that there are zero or more occurrences of an expression: a*, a|b*, a(bc)*;
- Plus combinator defining that there are one or more occurrences of an expression: a+, ab?(cd)+;
- Terminal symbols that this language covers are: '(', ')', '*', '+', '?', '|' and alphabetical small letters from 'a' to 'z';

Having the language analyses careful done the outlines for a grammar that describes this language are getting down to a full specification. In order to set out the closure between data types in TOM and data types in Haskell this grammar will be presented at these two levels, but still with more focus on the TOM environment.

3.5.1 Regular Expressions in Haskell

Haskell [16] allows the specification of user-defined types using the *data* constructor. This structure do not contain much implementation restrictions being only necessary the declaration of the main types that represents the regular expression language. After the inspection of these language characteristics it becomes clear that some of the sorts, like alternative, demand that this definition must be recursive [2]. Such data type has the following implementation in Haskell:

```
data RegExp = Epsilon
            | Literal Char
            | Opt RegExp
            | Alt RegExp RegExp
            | Seq RegExp RegExp
            | Star RegExp
            | Plus RegExp
```

Listing 3.5: Regular expression data type in Haskell.

This algebraic data type is purely abstract, since there is no reference to symbols like '?' or '|'. The data constructors such *Alt* or *Opt* are representative of the recursion. The declaration above is simply to define the types for *Alt :: RegExp -> RegExp -> RegExp* and *Opt :: RegExp -> RegExp*, and to all similar cases like these. But, using these data constructors we are able to write regular expressions in Haskell and checking the sentences syntax. For instance, the regular expression *e = a|(bc*)* can be written as:

```
e = Alt a (Seq b (Star c))
    where a = Literal 'a'
          b = Literal 'b'
          c = Literal 'c'
```

Listing 3.6: Regular expression example in Haskell.

Building an interpreter for this type of language in pure Haskell without the concrete notation is impractical in the user point of view. Therefore, the representation of the concrete aspects for the Haskell environment demands a parser construction. However, this task is very time consuming and the errors associated to standard recursion manipulation do not sustain the efficiency of assembling an interpreter. As presented in section 3.2 , Strafunski provides very capable tools to language processing and can model the concrete signature for regular expressions, therefore allowing the implementation of a strategic interpreter to this language.

3.5.2 Regular Expressions in TOM

Transposing the same concept ideas to the TOM environment the abstract signature that describe the regular expression language can be specified. Since this type of grammar does not consider any syntactic sugar it is only necessary to specify the main productions. The following Gom signature encompasses the specification for this language:

```
RegExp = Empty()
       | Literal(c:String)
       | Star(s:RegExp)
       | Plus(p:RegExp)
       | Opt(o:RegExp)
       | Seq(s1:RegExp, s2:RegExp)
       | Alt(a1:RegExp, a2:RegExp)
```

Listing 3.7: Regular expression data type in Tom.

This specification closely resembles the Haskell representation. Still the concrete notions to implement this interpreter are missing. This problematic was already referred in section 3.3, so additionally it is necessary the definition of this grammar in ANTLR. This

is extremely useful because ANTLR provides the specification of all the lexical rules. This action mitigates the problem of dealing with concrete inputs in TOM, and at the user point of view the input task is simplified. For instance, a simple concrete input like *ab** should be typed has *Seq(Literal(a),Star(b))* in the abstract form. However, the transportation of the grammar above to ANTLR cannot be done directly because ANTLR does not provide left-recursion parsing. The constructor *Seq* for example should be able to have its normal specification, like *regExp: regExp SEQ regExp*, but such scenario does not exist because of that ANTLR restriction. Therefore, the concrete grammar in ANTLR has to suffer some adjustments in order to implement the regular expression language.

```
regExp : exp  alt;

alt    :
       |  '|'  exp  alt;

exp    :
       | sglExp exp;

sglExp : LITERAL simb
       | '(' regExp ')' simb;

simb   :
       |'?'
       |'*'
       |'+';

LITERAL : ('a'..'z');
```

Listing 3.8: Grammar fo regular expressions in ANTLR.

In this stage another problem arises: the connection between ANTLR and TOM is responsibility of the Gom Antlr Adaptor. This tool requires that every constructor in ANTLR must have the correspondent constructor in TOM. However, the concrete and abstract grammars are very distinct, so the solution for this problem passes by the specification of an auxiliar abstract grammar that literally corresponds to the exact concrete grammar. This association can be executed with the following auxiliar grammar in TOM:

```
RegExpAux = RegE(e:Exp, a:Alt)

Alt       = AltAntlr(e:Exp, a:Alt)
          | NoAlt()
```

```
Exp       = ExpAntlr(s:SingleExp, e:Exp)
          | NoExp()

SingleExp = SglExp(l:String, s:Symbol)
          | Bracket(re:RegExpAux, s:Symbol)

Symbol    = OptAntlr()
          | StarAntlr()
          | PlusAntlr()
          | NoSymbol()
```

Listing 3.9: Regular expression auxiliary data type in TOM.

Finally, it is necessary to rewrite the ANTLR rules to the auxiliar grammar described above. The way to process that action is presented in the following instructions:

```
regExp : exp  alt -> ^(RegE exp alt) ;

alt    :  -> ^(NoAlt)
       |  '|'  exp  alt -> ^(AltAntlr exp alt);

exp    :  -> ^(NoExp)
       | sglExp exp -> ^(ExpAntlr sglExp exp);

sglExp : LITERAL simb -> ^(SglExp LITERAL simb)
       | '(' regExp ')' simb -> ^(Bracket regExp simb);

simb   :  -> ^(NoSymbol)
       |'?' -> ^(OptAntlr)
       |'*' -> ^(StarAntlr)
       |'+' -> ^(PlusAntlr);

LITERAL : ('a'..'z');
```

Listing 3.10: Regular expression association between TOM an ANTLR.

Despite the connection between the TOM and ANTLR is establish, the resulting abstract grammar does not correspond to the correct one. Thus, it is required to convert the auxiliar signature into the normal and more accurate grammar. So, the definition of a method in the TOM environment that treats this rewriting approach is necessary. Such method has the following implementation:

```
public static RegExp evalRegExpAux(RegExpAux reg){
  %match(reg) {
    RegE(ExpAntlr(s,e1), AltAntlr(e2,a)) -> {
      return `Alt(Seq(evalSingleExp(s), evalExp(e1)),Alt(evalExp(e2),
                    evalAlt(a)));
    }
    RegE(NoExp(), AltAntlr(e,a)) -> {
      return `Alt(evalExp(e),evalAlt(a));
    }
    RegE(ExpAntlr(s,e), NoAlt()) -> {
      return `Seq(evalSingleExp(s), evalExp(e));
    }
    RegE(NoExp(),NoAlt()) -> {
      return `Empty();
    }
  }
  throw new RuntimeException("RegExpAux␣RunTime␣Error");
}
```

Listing 3.11: Converting method for the standard TOM grammar.

This conversion is extremely important to provide the application of strategies to a more competent grammar by decreasing the number of operations necessary to build up this interpreter, mainly because the number of constructors is minor as well. After implementing all of these steps, any regular expression can now be normalized. Using strategies for this effect, the solution in TOM is quite simple:

```
%strategy Norm() extends Identity(){
  visit RegExp{
    Plus(p) -> {
      return `Seq(p,Star(p));
    }
    Opt(o) -> {
      return `Alt(Empty(),o);
    }
  }
}
`BottomUp(Norm()).visit(rg);
```

Listing 3.12: Strategy that normalizes a regular expression.

Bearing in mind that the *Plus* and *Opt* are productions of the regular expressions grammar, they might appear at the begging, middle or end of a regular expression definition, thereby a full traverse approach is recommended. The traverse scheme used in Listing 3.12 was the *BottomUp* but it could also be applied the *TopDown* scheme, because they both provide a full traverse over regular expressions specification.

The case study presented is a very elucidative example about implementing interpreters in TOM. A detailed description of potential problems, and correspondent solutions, helps to understand the importance of all stages. The Weaver will pass by some of these stages, but since this implementation is more complex than regular expressions language more problems are expected, but the core philosophy remains intact.

Chapter 4

Aspect-MatLab Weaving

MatLab is a powerful tool, well establish in the computer engineering environment. The richness and variety of its associated functionalities coupled with its great level of documentation makes MatLab widely known and used.

The extension of an aspect language to MatLab aims to introduce valuable means to control certain behaviors as monitoring variables or handler rules. Tasks like these are very time consuming and hard to manage because changes in the original source code are demanded as well as the implementation of new one required. In the context of the AMADEUS project, this Aspect-MatLab language is being developed and the completion of the correspondent Weaver are the most important phases to extend MatLab to aspect features.

4.1 Aspect Module

The aspect module for this language is shaped by a set of primitives that define its structure. Requirements like naming the aspect, defining the join point capture (*select*) or even specify the action for that join point (*apply*) are mandatory.

The *select* primitive is restricted to three types of MatLab: variables, arrays and functions. For the same aspect definition it is possible to compose join points, using the logical operators: and (&&), or (||) and negation (!). To call up some join point was established some capture functions presented in Table 4.1.

In order to trigger some action when a join point succeeds, the definition of the *apply*

primitive is required. The action is described by a string that is directly inserted on the original code, according to the execution type: *before*, *around* or *after*. A join point might involve more than one action, for cases like handling the value before and after of a variable assignment. For that purpose, this language allows the definition of multiple apply actions.

```
aspect aspect_name
  select: Join Point Capture
  apply: Action Description_1 :: execute before|after|around
  apply: Action Description_2 :: execute before|after|around
end

aspect aspect_name2
  ...
end
```

Listing 4.1: Aspect module structure.

The aspect definition modeled by the structure in Listing 4.1 is restricted within the *aspect* and *end* keywords. The next primitive sections are organized in a common sense way, because it is contradictory defining the action for a non-defined join point.

There are some similarities with the AspectJ [17] approach. The explanation for that fact is that we are trying to implement a very simple and easy to mange language. Since the AspectJ's semantics is widely known in aspect programming, the philosophy to extend a similar language to MatLab falls on the same criteria. Considering all these factors, the construction of the Strategic-Based Weaver will rely on the same principals. Regarding the study of efficient techniques for implementing aspects, this development will be based on some characteristics present in *abc* compiler: an efficient, open-source and easily to reuse compiler for AspectJ [5].

Explaining with more details of this aspect module is responsibility of another research group of AMADEUS project. A complete specification, and updates, can be found and study in the *"An Domain Specific Aspect Language for MatLab"* [22] dissertation.

4.2 Join Points Functions

The *select* primitive is responsible for declaring the join points. However, to define what function should be invoked to capture some MatLab instruction an explanation about the objective of each join point primitives is imperative.

Arrays	Variables/Constants	Functions
add()	read()	call()
get()	write()	function()
sizeOf()		

Table 4.1: Primitives for join point capture.

It is possible to capture arrays instructions in three ways: when an array is assigned to some value (*add()*), when an array is called (*get()*) or to extract the array size (*sizeOf()*).

At the same level it is possible to capture variables (or constants) when an assignemt (*write()*) occurs or when the value of that varaible is being read (*read()*).

Finally, if we are looking to capture a function specification, the primitive *function()* should be called. But, if we desire to catch when a function is called we must use the *c*all() primitve.

As mentioned in the previous section, more joint points might be defined and can be found in [22].The next sections will demonstrate the design and implementation of the Aspect-MatLab Weaver.

4.3 Weaver: The Design and Architecture

> *"Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system"* in [14].

In this project concept, the construction of a Weaver has to ensure a mechanism to process MatLab programs and the Aspect-MatLab [11] features. The approach to build it will be based on Strategic Programming. The main focus is to efficiently introduce aspects in MatLab projects without changing the main functionalities, but producing a final base code better organized and competent.

The AMADEUS project has some complementary tools for weaving, with the purpose to split these aspect extensions to its implementation needs. The enrichment of MatLab with the aspects requires the development of the following features:

- A parser for the MatLab language, due to restrictive copyrights that compilers to MatLab have.

- An Aspect-MatLab language definition along with a competent parser.
- An unparsing tool that translates a weaved program to MatLab instructions.

For the parsing needs two tools were used: JavaCC and ANTLR. TOM appears as the core implementation framework to process the weaving functionalities. The architecture that supports such implementation is based on the following data flow diagram:

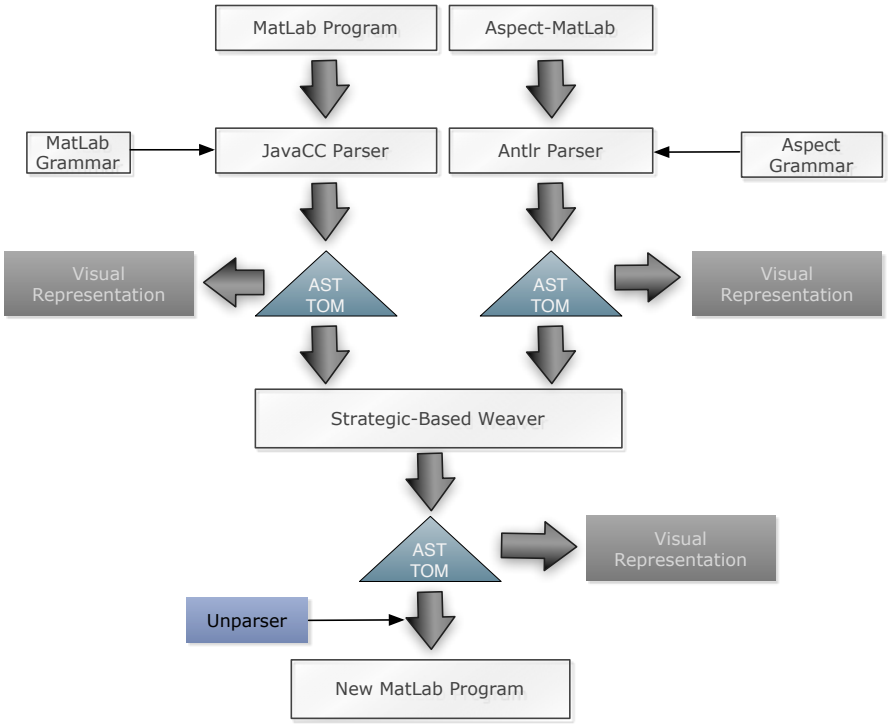

Figure 4.1: Aspect-MatLab weaving process.

This scenario covers all the necessary steps to build a Strategic-Based Weaver to process the Aspect-MatLab features. The final result of this procedure generates a new MatLab source code more competent, less error-prone and better organized. In the next subsections it will be presented a detailed description of all the stages that the weaving process encompasses.

4.3.1 MatLab Language Parsing

At this level it is used the JavaCC parser generator to process MatLab instructions. The main concern was to efficiently translate a MatLab program to a TOM data structure. How-

ever, this task is not easy because of the MatLab polymorphism. MatLab does not require the variable type definition, and thus, until the execution time this type remains unknown. For the parsing definition this is a major drawback because it is not clear the instruction category. For instance, in the Listing 4.2 example the *sumvals* can be applied to both scalar or array types of values.

```
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];
```

Listing 4.2: Function *sumvals* in MatLab.

Additionally, MatLab is continuously being updated and so this parser might not be capable of processing all instructions of this language.

Regarding the fact that JavaCC does not provide strategic features, the major effort is to provide a legible data-type in the TOM environment. The aspect execution will be applied to this type and will be returned according to the TOM signature.

An additional functionality was added to perform analyses over both input and output MatLab programs. Such feature generates a *Graphviz* visual representation file, which later is converted to a pdf format. A concrete example of this is presented at appendix D according to example in Listing 4.2.

The complete specification of the TOM abstract data-type for MatLab is presented in appendix A, because of its extension (over 78 different type of nodes) and complexity.

4.3.2 Aspect-MatLab Parsing

The syntactic analyzer used to describe the Aspect-MatLab language is ANTLR. Since this implementation was developed side by side with its definition, the resulting parser do not cover all the features that should process, yet.

The main focus was to establish the most appropriate structure for this language. An aspect file can have the following specification:

- One ore more aspect definition;
- An aspect can have more than one join point;
- An aspect can have more than one execution type;

As mentioned before, the definitions of both concrete and abstract grammar are indispensable. Considering the formal structure for aspects, the first step is to identify the main abstract constructors that might need to be implemented. Currently, the definition of Aspect-MatLab language only requires the following TOM data-type:

```
LstAspectML   = NoAspectML()
              | ConsAspectML(a:AspectML, l:LstAspectML)

AspectML      = Aspect(name:String, args:LstAspectArg, body:ActionsBody)

LstAspectArg  = NoAspectArg()
              | ConsAspectArg(arg:AspectArg, l:LstAspectArg)

AspectArg     = Arg(a:String)

ActionsBody = Actions(input:Input, sel:LstSelect, ap:LstApply, wh:Where)

LstSelect     = NoSelect()
              | ConsSelect(s:Select, l:LstSelect)

Select        = ReadVar(n:String)
              | WriteVar(n:String)
              | CallFun(n:String)
              | Fun(n:String)
              | AddArray(n:String)
              | GetArray(n:String)
              | SizeArray(n:String)

LstApply      = NoApply()
              | ConsApply(a:Apply, l:LstApply)

Apply         = ExecuteBefore(insert:String)
              | ExecuteAfter(insert:String)
```

```
| ExecuteAround(insert:String)
| HeaderExecuteBefore(insert:String)
| HeaderExecuteAfter(insert:String)
| HeaderExecuteAround(insert:String)
| BodyExecuteBefore(insert:String)
| BodyExecuteAfter(insert:String)
| BodyExecuteAround(insert:String)
| HeaderArgsExecuteBefore(insert:String, ag:LstAspectArg)
| HeaderArgsExecuteAfter(insert:String, ag:LstAspectArg)
| HeaderArgsExecuteAround(insert:String, ag:LstAspectArg)
| BodyArgsExecuteBefore(insert:String, ag:LstAspectArg)
| BodyArgsExecuteAfter(insert:String, ag:LstAspectArg)
| BodyArgsExecuteAround(insert:String, ag:LstAspectArg)
```

Listing 4.3: Aspect abstract grammar.

This Gom signature takes the sort *LstAspectML* as the first instance of the aspect tree. This constructor can be either empty (*NoAspectML*) or contain one or more aspects definition (*ConsAspectML*). However, the concrete starting point on aspect definition relies in the production *AspectML*. Naming the aspect, defining if there will be arguments, specifying the join points and execution instructions are some characteristics that this constructor requires (by direct or nested rules).

Additionally, the concrete signature requires another level of detail for the lexical and parser rules in ANTLR, which has to ensure the similarities to its abstract signature. For lexical definition we attend to specify all the reserved words (e.g. *"aspect", "select", "apply"*, etc.) and terminal symbols. Since ANTRL disallows the left recursion, new parser rules were defined with some distinct behavior than the structure in TOM. Along with this constraint, the concrete semantics increases the number of possibilities for term rewriting. For instance, let us reflect on the following code sample extracted from ANTLR parser:

```
select
  : (...)
  | WRITE '('NAME')'
  | WRITE '('NAME')' '&&'
  | WRITE '('NAME')' '||'
  (...)
```

Listing 4.4: Aspect concrete grammar sample

This sample models the join point that captures a variable assign. Such action can be trigged in two distinct ways: a simple join point definition (e.g. *write(a)*); or composition (logical and/or) between join points (e.g. *write(a)* || *write(b)*, *write(a)* && *write(b)*). Since a new data-type in TOM is required for ANTLR, this issue is resolved by that approach.

In addiction to all of these, a *Graphviz* visual representation feature was also implement to increase the understandability of the idea about this abstract data-type. For example, considering the following concrete aspect definition:

```
aspect test
  select: read(c)
  apply: "disp('The␣value␣of␣this␣variable␣is'␣c)"::execute after
end
```

Listing 4.5: Aspect example

Regarding this specification a dot file is created containing the abstract representation of the aspect. Such representation has the shape as in Figure 4.2.

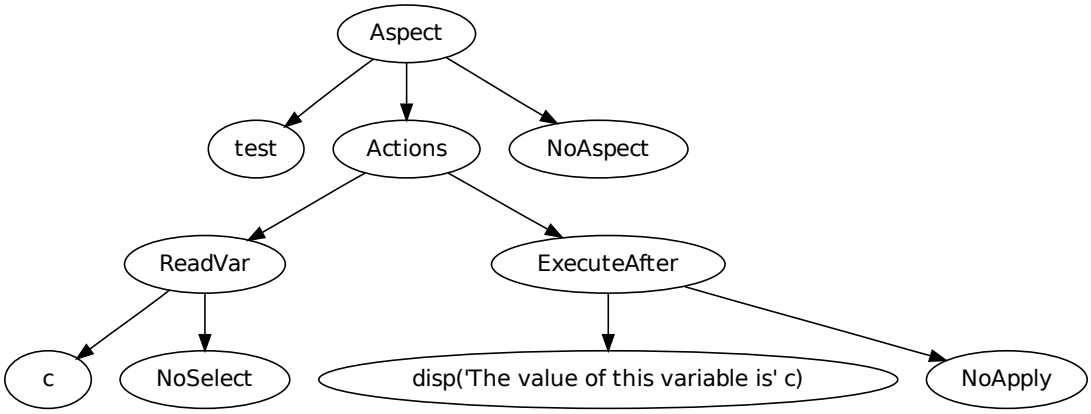

Figure 4.2: Aspect graphical view.

The differences between abstract and concrete grammar can be analyzed in appendix B, which contains the ANTLR specification. During the assembly of this parser there were no issues relevant enough to mitigate its development.

4.3.3 Embedded DSAL in Java

In the course of developing the data-type in TOM, one goal was to make it as closer as possible to the syntax keywords of the aspect language specification.

Aspect Syntax	TOM data-types
read()	ReadVar()
write()	WriteVar()
call()	CallFun()
function()	Fun()
add()	AddArray()
get()	GetArray()
sizeOf()	SizeArray()

Table 4.2: Correspondence between concrete and abstract syntax

In the previous section 4.3.2 we presented the aspect TOM signature, and comparing with the join point capture present in Table 4.2 it is possible to verify that the algebraic types in TOM resembles the concrete syntax. For instance, considering the execution of the following aspect over the *sumvals* (Listing 4.2) function:

```
aspect variable_tracing()
  select: write(s)
  apply: "display('Tracing␣value␣of:'␣+␣s)" :: execute after
end
```

Listing 4.6: Aspect for *sumvals* function.

The goal of this aspect is to introduce the *apply* action *after* each assign instruction for variable *s*. This aspect can easily be expressed with TOM constructors.

```
ConsAspectML(
  Aspect("variable_tracing",NoAspectArg(),
    Actions(ConsSelect(WriteVar("s"),NoSelect()),
      ConsApply(ExecuteAfter("disp('Tracing␣value␣of:'␣+␣s)"),NoApply()),
      )
  ),
  NoAspectML()
)
```

Listing 4.7: Aspect for *sumvals* function in TOM.

Besides some additional and needed internal rules in TOM (e.g. *NoAspectArg()*), this example is quite representative of the potential of the abstract aspect definition. Although this embedded DSAL allows this aspect specification, for more complex aspects the tendency of occurring syntax errors would drastically increase.

Since the goal is to define simple aspects without considering the concrete components, this DSAL emerges as an efficient alternative specification method.

4.3.4 Unparsing TOM to MatLab

This tool transforms a weaved MatLab program in TOM into standard MatLab instructions. The methodology behind its implementation is nothing more than a pretty printer according to MatLab nomenclature.

This feature is extremely useful because it produces a program that can be recognized and processed by the MatLab toolbox. The final stage of weaving is reached when aspects were embedded in the original MatLab program.

4.4 Strategic-Based Weaver

This section addresses the weaving tasks and expresses the strategic approach to embed aspects into MatLab projects.

Covering all the parsing stages, the implementation of the weaver functionalities can have its first design. The first approach to start the Weaver delineation is based on the definition of these primordial concepts:

- What kind of inputs that Weaver should receive.
- What type of traversal scheme to use.
- What kind of result that Weaver must return.

The inputs that such function should receive is the abstract representation of a MatLab program and Aspect-MatLab specification. The MatLab program works has the target to execute some described action of the aspect definition. This covers all the needed information to perform program transformation on MatLab files.

The correct traversal scheme for this function relies on the specification of the aspect join point. For example, if we want to change the assignment to every appearances of the variable *'x'*, a full traversal is recommended, but if we intended to display the result of the first execution of a specified function it is advisable to approach in a once top-down manner. The second scenario has no possible behavior on this language proposal. So, since the traverse of a full MatLab program is required to capture all possible join points, a full top-down approach appears has the most appropriate scheme.

The weaving result is a new MatLab program, maintaining the same core functionalities, but with some performed actions described in the aspect module. However, this result is not legible in the MatLab environment. Its representation still maintains the abstract character.

Having these parameters completely defined, the goal is to implement a weaver function as generic as possible, capable of dealing with all the join point issues. This generic characteristic decreases the invasive changes on its definition when this aspect language is updated. This is also vital in dealing with the restriction that the Weaver development coexisted with the language specification, so the TOM data-type was continuously suffering some adjustments. The algorithm that supports such weaving should have the following behavior:

Input: Abstract signatures of MatLab program and Aspect-MatLab
Output: New MatLab program with embedded aspects
foreach *Join Point* **do**
  while *MatLab and Aspect AST tree != Empty* **do**
    Top-DownTraversal(*strategy*);
    if *Join Point matches a MatLab instruction* **then**
      Execute(*applyinstrucion*);
    end
  end
end

Algorithm 1: Weaver function implementation.

This algorithm is extremely generic and do not reflects in any way the actual Weaver implementation. The only purpose is to give a more understandable idea of the actions that should be performed.

The Weaver implementation is shaped and focused on three major phases:

- Collect all the join points from aspect definition.
- Extract the MatLab sub-trees that match a join point.
- Perform the aspect actions on the MatLab program.

This covers all the steps that this implementation passed by. But, in order to present a detailed explanation of all these stages the next sections will discuss how this actions were implemented and how the interaction between them is processed.

4.4.1 Join Point Capture

The first action that the weaving process should perform is to gather the join points in aspect definitions. For that purpose, it is necessary to traverse the aspect AST in TOM and store the strategy result. Such implementation in TOM can be achieved as the following instructions:

```
%strategy JoinPointCatch(Collection jp) extends Identity() {
  visit Select{
    jp@ReadVar(_)->{jp.add(‘jp);}
    jp@WriteVar(_)->{jp.add(‘jp);}
    jp@CallFun(_)->{jp.add(‘jp);}
    jp@Fun(_)->{jp.add(‘jp);}
    jp@AddArray(_)->{jp.add(‘jp);}
    jp@GetArray(_)->{jp.add(‘jp);}
    jp@SizeArray(_)->{jp.add(‘jp);}
  }
}
```

Listing 4.8: Aspect join point capture strategy.

This strategy was very useful at the beginning of the implementation. However, over the time some requirements were refined and not only was necessary the join points capture, but also associate them to the *apply* instruction. This could be done with a more complex strategy, but some internal errors related to the traverse schemes definition in TOM do not return the correct result. The solution passes by using explicit recursion. Such implementation is as follows:

```
public static ArrayList joinPoints(LstAspectML ac){
  ArrayList jp = new ArrayList();
```

```
  ArrayList aux = new ArrayList();
  %match(ac){
    NoAspectML() -> {return jp;}
    ConsAspectML(Aspect(_, _, actions),l) -> {
      jp=jpActions(‘actions);
      aux=joinPoints(‘l);
      return concat(jp,aux);}
  }
  throw new RuntimeException("Aspect␣instruction␣fail␣in␣LstAspectML");
}
```

Listing 4.9: Aspect join point method.

The philosophy is basic the same if a strategy approach was adopted. This method returns a set with the combined join points and *apply* instructions (e.g. [WriteVar("s"), ExecuteAfter("display('Tracing value of:' + s)"),...]).

The generic traverse scheme that should be used is the top-down, because the aspect file can contain more than one aspect definition. Thus, it is required an analyses of the whole file until reach its end.

The information that the result set carries can be transposed and used to confirm if any instruction in a MatLab program matches any join point. Since the aspect and MatLab languages have distinct data-types in TOM, this result set serves as the communication link between them. Next, we present how this information is used to extract the MatLab sub-trees.

4.4.2 MatLab Sub-Trees Capture

The purpose for this section is how to infer if an join point match occurs and how to collect the correspondent tree from the MatLab program.

The target AST that has to be traversed must be consistent to a MatLab program. One of the arguments that such strategy should receive is the result of the previous join point capture method. This allows to establish a comparison between a statement in MatLab and an aspect join point. When a match occurs, is trigged the action related to the join point. This strategy can be implemented in TOM as follows:

```
%strategy MatLabJoinPointTree(Collection c, Collection jp, Collection se,
              Collection varName) extends Identity(){
```

```
  visit StatementSort{
    c@Statement(Assign(Expression(Id(Identifier(name,line1))),exp,line2),
              line3) -> {
      ArrayList<String> aux_se = new ArrayList<String> (se);
      String exec = aux_se.get(0);

      if((verifyWriteVar(jp,‘name) || verifyAddArray(jp,‘name)) &&
        verifyExecuteBefore(jp,exec)){
        StatementSort bf = ‘Statement(Expression(Text(exec,line1)),
                    line3);
        c.add(bf);
        c.add(‘c);
      }(...)
    }(...)
  }
}
```

Listing 4.10: MatLab sub-trees capture strategy.

This sample of the *MatLabJoinPointTree* strategy mentions some of the constructors of the MatLab grammar, like *Statement*, *Assign*, *Expression*, etc. For more details about this abstract grammar, as mention in section 4.3.1, has a complete description available in appendix A.

One of the milestones for this strategy is how to make it as generic as possible. For instance, the action to assign a variable to a type specific value can be done in several ways, like a direct assignment to an integer or by combining a result of a function with a double. In Listing 4.10 the goal is to cover the assignment to variables and arrays. For that purpose, there is no restriction about the ascribed type attached to those assignments. It is only used a generic *ExpressionSort* variable (*exp*). Since the type of a variable in MatLab is only known during runtime, the type specification for arrays and variables assignment have the same principles, thus the same type of statement.

The validation of a join point, coupled with its execution type, is a mandatory requirement. Considering that a join point specifies the name of a variable, is only needed traverse the MatLab AST and verify if any *Statement* type has the same correspondent name. At the same time, we have to ensure that the *apply* action is executed within to its execution type. In this case, is validated if the apply instruction is executed before the join point match.

The execution type of the *apply* action constrains the shape of the new MatLab sub-tree.

For example, if the *apply* instruction has to be executed before the join point, the resulting sub-tree should attach first the apply instruction and then the statement that matches the join point, on the original MatLab program.

The new MatLab sub-tree is stored in a set with the purpose of being used in other auxiliary strategies in weaving process.

4.4.3 Weaving MatLab Programs

The final requirement to process aspects over MatLab programs is to merge the result of the two previous stages and embed the aspects into the original source code. This level is where all the weaving process comes together.

Having the new sub-tree of a MatLab program, the main objective is understand how to assemble the weaved program. All MatLab instructions fall within the *ConcStatement* constructor, and the first approach for weaving relies on how to introduce the new instructions in this rule. Thus, the visit argument of such strategy must be the same as *ConcStatement*.

```
%strategy WoveConcStatStrategy(Collection tree) extends Identity(){
   visit StatementListSort{
      ConcStatement(hLst*,st,tLst*) -> {
         (...)
         ArrayList<StatementSort> aux = new ArrayList(tree);
         if(checkStatementArgs(aux,‘st)){
            StatementSort tag = ‘Statement(Expression(Text(
                                 "Execution␣Around",0)),’t’);
            if(!aux.contains(tag)){
               StatementSort fst = (StatementSort) aux.get(0);
               StatementSort snd = (StatementSort) aux.get(1);
               tree.remove(fst);
               tree.remove(snd);

               return ‘ConcStatement(hLst*,fst,snd,tLst*);
            }
            else{
               StatementSort fst = (StatementSort) aux.get(0);
               StatementSort snd = (StatementSort) aux.get(1);
               StatementSort thr = (StatementSort) aux.get(2);
               tree.remove(fst);
               tree.remove(snd);
               tree.remove(thr);
```

```
                return `ConcStatement(hLst*,fst,tLst*);
            }
        }
    }
}
```

Listing 4.11: Strategy for MatLab instructions body update.

As mentioned in the previous section, the argument for this strategy definition is the returned set from the *MatLabJoinPointTree* strategy. The join point can have a match in any instruction of the program. Thus, since *ConcStatement* invokes a list of statements we have to assume all possible scenarios:

- A join point can match the first MatLab instruction.
- A join point can have matches over the middle of the source code.
- A join point can match the last instruction.

The *WoveConcStatStrategy* has to control variables *hLst** and *tLst**, representing the set of instructions before and after a join point match, respectively. TOM is capable of inferring if the *hLst** and *tLst** variables are empty or not. So, in this way the *st* variable can cover all the possible scenarios where join points might appear. That raises the question if it is really necessary to declare these two control variables. If they were omitted the returned result would not cover all the initial instructions. For instance, if a join point occurs in the middle of the program, the weaved program does not contemplate the instructions before that join point, which disagrees with the idea of keeping the same core functionalities of the original program.

Firstly, the join point validation has to assure that the statement that is being visited matched the join point, responsibility of the *checkStatementArgs* method.

Secondly, updating the MatLab instructions can have two different forms according to the apply execution type. Considering the execution type is before or after the join point, is only necessary maintain the original statement attached to the new instruction. If the execution is before the match, then the first instruction is the *apply* action and the second the statement where the join point arises. On the other hand, if the execution is after the match

the first and second statements are the join point statement and *apply* instruction, respectively. But, using the execution type around there is no need of keeping the original join point MatLab instruction, since that is replaced by the *apply* action. For this effect, urges the need of inserting a *flag* variable that separates this notion from before and after. The only difference on the returned *ConcStatement* lies on the *apply* instruction that replaces the join point matching statement in the instructions structure.

Since, the new instructions body of the MatLab program is returned, the next step is to embed this structure into the MatLab main constructor. All the described auxiliary strategies provide the needed transformations to implement the weaving process.

```
%strategy WeaverStrategy(Collection tree) extends Identity(){
  visit StartSort{
    Start(FunctionMFile(returnVars, functionIdentifier, args,
                concSt, lineN1), lineN2) -> {
      ArrayList aux = new ArrayList(tree);

      return `Start(FunctionMFile(returnVars, functionIdentifier, args,
                    WoveConcStat(concSt, aux), lineN1), lineN2);
    }
    Start(ScriptMFile(concSt, lineN1), lineN2) -> {
      ArrayList aux = new ArrayList(tree);

      return `Start(ScriptMFile(WoveConcStat(concSt, aux), lineN1),
                    lineN2);
    }
  }
}
```

Listing 4.12: Weaving Strategy.

Taking into account that operations in MatLab can be made with a direct sequence of instructions (called *scripts*) or by including them into functions, we have to split the weaving process into these two types of files. Since these two types of files have distinct structures on MatLab grammar, specific MatLab programs should be returned according to the input file. Listing 4.12 points out this scenario, where is possible to confirm the low complexity of such strategy. The only action that is performed is crossing a MatLab program applying the strategy *WoveConcStat* in a top-down approach.

A new MatLab program, under aspect restrictions, is returned with the same core functionalities. The objective of automating the process of embedding aspects into MatLab projects, in a strategic performed way, is completed.

4.5 Weaver in Practice

After providing a complete description of the Aspect-MatLab language adopted and a prototyped Weaver implementation, we can exhibit some examples adopted from [10].

The actions that an aspect module represents can be related to some types of code handling, like logging and tracing variables. Manage code by logging requires controlling the behavior of certain variables. For example, if we want to attest that none of the variables exceeds a specific value we must insert the adequate MatLab control code. Consider the example in Listing 4.13:

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

Listing 4.13: MatLab script.

Shaping the aspect definition according to the Weaver's input must follow these specifications:

```
aspect logVar
  select: write(sum)
  apply: "if␣sum>=10000␣warning␣('sum␣too␣big!␣%f',sum);end"
          ::execute before
end
```

Listing 4.14: Aspect logging.

From this aspect definition it is possible to verify that the join point definition does not cover all the the target variables in Listing 4.13. Despite the aspect grammar accepting multi-aspect definition, the current version of the Weaver only computes one join point, because of the complexity of the method to associate several join points to several *apply* instructions. In order to implement all these control instructions, we need to define one

aspect to each variable present on the script. The execution type is always before, since these instructions appear before the match instruction. The final MatLab code based on the aspect action is as follows:

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

Listing 4.15: MatLab logging weaved script.

Tracing the value of a variable is another convenient action to watch the behavior throughout the code (very used for debugging issues). The only difference to logging action is that it does not repress the values that a variable assumes. Displaying the value in each stage of the program is the goal. Such aspect should have the following structure:

```
aspect logVar
  select: write(sum)
  apply: "dis('Tracing variable sum:',sum);"::execute before
end
```

Listing 4.16: Aspect tracing.

Like in Listing 4.14, if we intend to trace more that one variable or even trace the value before and after the join point, we must define more aspect modules. Applying this aspect to the MatLab script, the following structure of instructions is returned:

```
...
for j = 1:1:N
  dis('Tracing variable sum:',sum);
  sum = sum + A(j) * B(j+N);
end
outa(i) = sum;
...
```

Listing 4.17: MatLab tracing weaved script.

Embedding aspects into MatLab source code is a quite simple process using the Weaver. Although some of its limitations do not allow a full specification of the intended action, if

we split the target variables into distinct aspect files, the Weaver is capable of processing them. Needless to say, we have to reuse the result of an action in order to outcome a final program with all the needed instructions.

Like Figure 4.3 suggests, the process of separating the core functionalities of the program from the auxiliary aspect actions can be easily described. This methodology helps in editing and maintaining the original source code. The main objective here is keep the distance among aspects computing and the functional module of the system, accomplished by the Weaver intervention.

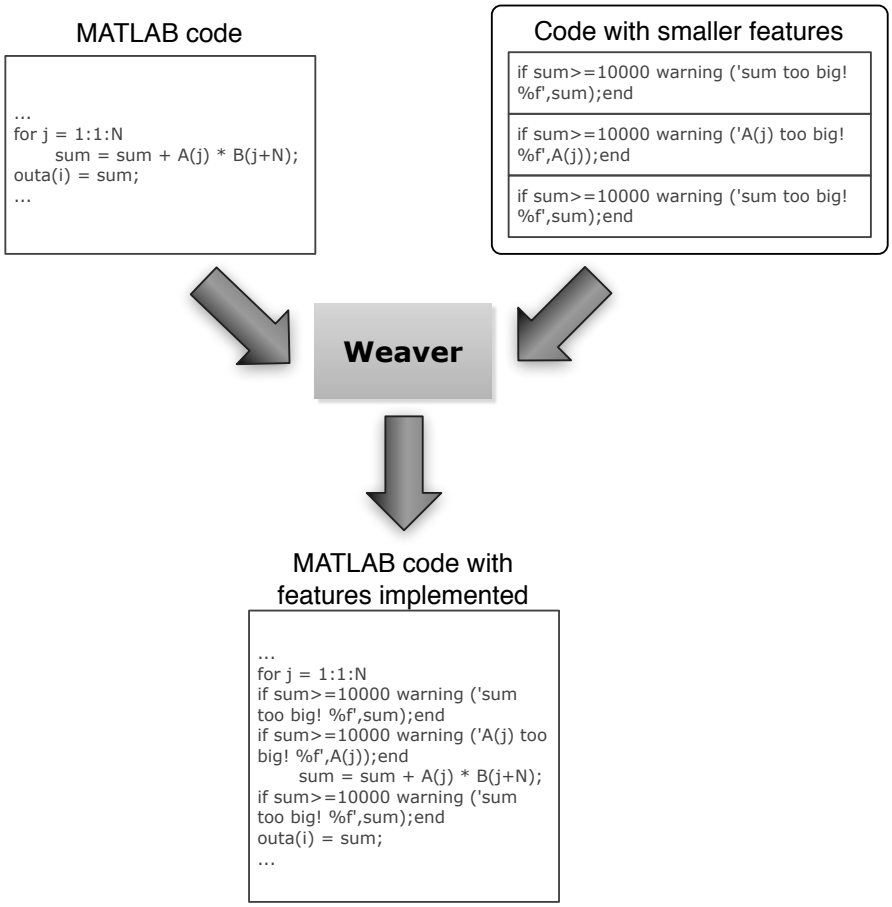

Figure 4.3: Weaving process example.

Summarizing, the automated code insertion in pre-defined join points by the Weaver features allows a fast code insertion at MatLab's input program.

4.6 Weaver in numbers

In order to present some insight about the complex implementation of the Weaver, we present some numbers related to all the developed tools: concrete and abstract aspect grammars, size of the weaver file and numerous generated files.

The concrete grammar for our Aspect-MatLab language has 126 lines of specification. Since this concrete representation demands an auxiliary abstract grammar in TOM, the complete abstract implementation, also considering the embedded DSAL presented in section 4.3.3, has 101 lines of code. The main file that imports all the TOM data-types and executes transformations on them has around 1400 lines of code. However, this complexity is minimized by the strategic approach because if this implementation was developed using explicit Java recursion the number of lines and the complexity of the methods would drastically increase.

Besides the large amount of code lines of all files, ANTLR and TOM also generates countless auxiliary files. They provide methods and features that complements the implemented weaving functionalities.

Additionally, considering all the tools developed in the AMADEUS projects, like the front-end for MatLab, emphasis even more the complexity of enriching the MatLab language with aspects features.

4.7 How to use the Weaver toolset

To use the Weaver some aspects has to be mentioned to explain how to input a standard MatLab program and an aspect specification.

To input a set of MatLab instruction (scripts or function) an external file named *MatLab.m* has to be edited. The instructions that this file contains are parsed by the MatLab front-end and only after is executed the action decribed in aspect definition.

To insert the aspect specification in Weaver the approach was similar, editing the *asp.txt* file is mandatory. This file is parsed by ANTLR and converted to TOM abstract data-type.

After and during the weaving process, some visual *Graphviz* files (*aspMatLab.dot, MatLab.dot* and *WeavedMatLab.dot*) are generated to complement the understanding about the abstract representation of a MatLab program and aspect file.

Finally, after weaved the original MatLab program a *WeavedMatLab.ir* file is generated containing the abstract representation of the new program that will be unparsed to standard MatLab instructions.

4.8 The Weaver compilation process

The Weaver is composed by many files and has several dependencies between them. The compiling of all these files is executed by specific steps, like the following:

1. Set up the TOM environment, needed for the recognition of TOM commands:

```
export TOM_HOME="tom-2.7/"
export PATH=${PATH}:${TOM_HOME}/bin
export CLASSPATH=${TOM_HOME}/lib/tom-runtime-full.jar:${CLASSPATH}
```

Listing 4.18: Commands to set up TOM environment.

2. Parsing the aspects to TOM environment using the *Gom Antlr Adaptor*:

```
gomantlradaptor --grammar AspectGram --package parser
parser/AspectRule.gom

gom --package parser parser/AspectRule.gom
tom --output parser/AspectGram.g parser/AspectGram.g.t
java org.antlr.Tool parser/AspectGram.g
```

Listing 4.19: Gom Antlr Adaptor commands.

3. Compilation Weaver main file and all the complement files, and input the aspect file:

```
tom parser/Weaver.t
javac parser/*.java
java parser/Weaver < parser/asp.txt
```

Listing 4.20: Compilation of the main file.

4. Converting the *Graphviz* files into pdf format:

```
dot -Tpdf -o aspMatLab.pdf aspMatLab.dot
dot -Tpdf -o MatLab.pdf MatLab.dot
dot -Tpdf -o WeavedMatLab.pdf WeavedMatLab.dot
```

Listing 4.21: Conversion Graphviz to pdf.

All these instruction are described in the *compile.sh* file. Thus, is not necessary call all of this instructions in future compile actions, being only necessary execute the *./compile.sh* command on the computer's terminal.

Chapter 5

Conclusions

This thesis embraces the main steps of building a *Strategic-Based Weaver*. We present the motivation that lead to an extension of aspect-oriented features to MatLab. An extended overview about the strengths of strategic programming also provides the ideas that sustains such type of Weaver.

The Aspect-Oriented paradigm introduces new concepts that advocate a stronger separation of concerns, therefore increasing modularization in MatLab based systems. Traversal programming in the strategic approach is a solid example of the purpose of AOP techniques. The independence of data structure, the support for reusing basic programmer-definable computations and the abstraction of programming language make the strategic programming an efficient idiom to build a Weaver for Aspect-MatLab features.

The entire mechanism of weaving has many associated concerns. We must ensure that the correct aspects definitions have their safe application in a standard MatLab program. For the first instance approach was studied the best solution to structure the weaving features, which was based in breaking the weaving needs into several parts, focusing on each features that had to be performed.

Splitting the traverse actions of both aspect and MatLab data-type was recommended since they are modeled by distinct structures. This is a crucial step in establishing a connection among these two TOM abstract signatures. For these languages updating, the generic traverse allows an implementation free of adjustments, since they are capable of processing any data-type.

Establishing a separation on the weaving model by its features was the most appropriate approach. Centering the attention on the action itself allows a functionality oriented imple-

mentation. This property increases the efficiency of the strategy. Accounting to the Weaver limitations, some of these strategies have an ideal and complete behavior. For instance, the strategy that collects all the join points is capable of storing one simple join point or composed join points (e.g. *write(a)&&read(b)*), covering all the possible scenarios. However, if some changes on the join point capture method arise it is only necessary to improve this strategy for the new needs, having no concern about the other auxiliary strategies.

Assembling all the strategies result into the main weaving process was relatively simple. The final MatLab program corresponds to the initial ideas of enriching this language with aspect features. The only information about the aspect execution over the input program lies on the apply instruction. In each point where a match occurs, is possible to confirm the presence of the new instruction (before, after or around) on the woven program.

For all that was presented we believe that this Weaver is extremely functional and offers some useful functions, like abolishing the manual code re-factoring on MatLab programs. This final version encloses all primary needs for embedding aspect into MatLab. For full weaving process we present a detailed overview about future interventions on this Weaver in section 5.3.

5.1 Weaver Limitations

Throughout the implementation of the Weaver, we were faced with some problems that could not be solved in time. The complexity of MatLab language and the manipulation of two distinct data-types were the biggest problems encountered.

Parsing such rich language is not an easy task, specially when dealing with its polymorphism attribute. The parser developed in the context of AMADEUS is very reliable, however it could not compute all MatLab scripts[1]. In fact, developing parsers to support the full MatLab language is a complex and time consuming task. This is the first limitation of this Weaver, but somehow understandable due to number of features that this language offers. Other limitation resides in *function* join point primitive, because the current MatLab parsing do not process more that one MatLab function.

Attending now the limitation on aspects definition, the major issues reside on handling the two data-types. For instance, the *apply* instruction should have a more complex behav-

[1] A similar limitation occurs in the MatLab parser developed by others research groups like McGill University [3]

ior if the instructions on them were analyzed in MatLab data-type environment. But such action would demand an alteration on aspect grammar, an efficient function that converts the *apply* instruction into MatLab statements and an evaluation method that verifies the instruction logic.

Regarding that aspect parsing and abstract grammar are both capable of processing any aspect definition, the Weaver is not at the same level. The Weaver is not capable of process multi aspect definitions, complex join point capture (e.g. *write(a)&&(get(B)||get(C))*) and more than one *apply* instruction. We can overcome these limitations by defining a set of aspects that execute all the intended actions, using in each aspect input the previous returned program.

Although there is some work to be done on aspects definition, the actual structure already embraces the major characteristics that an aspect language stands for.

5.2 Contributions

Model an aspect-oriented programming for MatLab evolves identifying the issues and concerns that are not well treated by its traditional programming methodologies. This project was centered in some of these constrains, widen MatLab with valuable means for such type of programming. The contributions for the AMADEUS project are:

- Improvements that the abstract grammar that translates the MatLab instructions into TOM data-type should suffer.
- Implementation of a robust Aspect-MatLab language parser in ANTLR, allowing the definition of aspect's concrete signature.
- Implementation of the abstract signature to Aspect-MatLab for further strategies implementation.
- Definition of the Weaver's architecture that sustains the aspect processing over MatLab programs.
- Development of the strategic approach for the weaving process.
  - Collect join points from aspect definition.
  - Extract all the MatLab sub-trees that matches a join point.

  - Update the MatLab original program instructions.

- Specification of the linking method between the MatLab parser/unparser and Weaver tools.

- Implementation of the visual representation to any TOM data-type.

The number of presented features are very competent for these thesis objectives. Embedding aspects into MatLab is now an automated procedure, improving the code editing and maintaining.

5.3 Future Work

Future work around this Weaver includes studying more efficient techniques (and tools) that allows the implementation of a full aspects extension to MatLab.

Processing MatLab instructions, like *ifs* or *loops*, on the *apply* instruction should improve the range of options to execute its action. Improving the association method between join point and the *apply* instruction is another to do refinement for using complex join point capture. That also implies updating the strategy that upgrades the MatLab statements set, since the information of the join points will be passed in a distinct way. At the same level, to decrease the number of aspects definitions needed to process complex join points, the Weaver should also be capable of recognizing multiple aspects definitions in the same file.

It is complex to define a complete parser for MatLab, so to enhance its parsing needs a more exhaustive research is mandatory, with more concluded examples and tests. However this effort might be unfair since new features could be added to MatLab by MathWorks and make this parser obsolete again.

The study of defining an domain specific aspect language to MatLab is continuously being update, and so the Weaver could not follow its evolution. Therefore, more restrictions and characteristics about this language were defined in [22]. Since the approach to conceive this Weaver was extremely generic, by applying the strategic principals, all the new aspect characteristics can be easily incorporated in future Weaver functionalities.

Appendix A

MatLab Abstract Grammar

```
module parser.MatLabRule
imports String int double char
abstract syntax

StartSort = Start(fileType: FileTypesSort, lineNumber: int)

FileTypesSort =
    ScriptMFile(statementList: StatementListSort, lineNumber: int)
    | FunctionMFile(returnVars: IdentifierListSort,
      functionIdentifier: IdentifierSort, args: IdentifierListSort,
      statementList: StatementListSort, lineNumber: int)
    |   EmptyFile(dummy: int)

StatementListSort = ConcStatement(StatementSort*)

StatementSort = Statement(statementType: StatementTypeSort,
                          delimiter: char)

StatementTypeSort =
    CommandForm(identifier: IdentifierSort,
                textList: ExpressionListSort, lineNumber: int)
    | Expression(expression: ExpressionSort)
    | Assign(assignee: StatementTypeSort, assigned: ExpressionSort,
             lineNumber: int)
    | AssigneeMatrix(referenceList: ExpressionListSort, lineNumber: int)
    | AssigneeCellArray(referenceList: ExpressionListSort,
                        lineNumber: int)
```

```
    | ForCommand(assignment: StatementTypeSort,
                statementList: StatementListSort, lineNumber: int)
    | ParforCommand(assignment: StatementTypeSort,
                    statementList: StatementListSort, lineNumber: int)
    | If(condition: ExpressionSort, statementList: StatementListSort,
         else: ElseSort, lineNumber: int)
    | GlobalCommand(identifierList: IdentifierListSort, lineNumber: int)
    | PersistentCommand(identifierList: IdentifierListSort,
                        lineNumber: int)
    | WhileCommand(condition: ExpressionSort,
                   statementList: StatementListSort, lineNumber: int)
    | Return(lineNumber: int)
    | Break(lineNumber: int)
    | Continue(lineNumber: int)
    | Switch(expression: ExpressionSort, caseList: CaseListSort,
             otherwise: StatementListSort, lineNumber: int)
    | TryCatchCommand(try: StatementListSort, catch: CatchSort,
                      lineNumber: int)
    | ClassDefCommand(className: IdentifierSort,
                      superClassName: IdentifierSort,
                      statementList: StatementListSort,
                      lineNumber: int)
    | Pragma(pragma: String, lineNumber: int)


ElseSort =
    Else(statementList: StatementListSort, lineNumber: int)
    | ElseIf(condition: ExpressionSort, statementList: StatementListSort,
             else: ElseSort, lineNumber: int)
    | EmptyElse(dummy: int)

CaseListSort =  ConcCase(CaseSort*)

CaseSort =
    Case(expression: ExpressionSort, statementList: StatementListSort,
         lineNumber: int)

CatchSort =
    Catch(identifier: IdentifierSort, statementList: StatementListSort,
          lineNumber: int)
```

```
ExpressionListSort = ConcExpression(ExpressionSort*)

ExpressionSort =
    ShortCircuitOr(expressionList: ExpressionListSort,
                   lineNumber: int, type: int, dims:DimSort)
    | ShortCircuitAnd(expressionList: ExpressionListSort,
                      lineNumber: int, type: int, dims:DimSort)
    | ElementWiseOr(expressionList: ExpressionListSort,
                    lineNumber: int, type: int, dims:DimSort)
    | ElementWiseAnd(expressionList: ExpressionListSort,
                     lineNumber: int, type: int, dims:DimSort)
    | LowerThan(expressionList: ExpressionListSort,
                lineNumber: int, type: int, dims:DimSort)
    | LowerThanOrEqual(expressionList: ExpressionListSort,
                       lineNumber: int, type: int, dims:DimSort)
    | GreaterThen(expressionList: ExpressionListSort,
                  lineNumber: int, type: int, dims:DimSort)
    | GreaterThenOrEqual(expressionList: ExpressionListSort,
                         lineNumber: int, type: int, dims:DimSort)
    | Equal(expressionList: ExpressionListSort, lineNumber: int,
            type: int, dims:DimSort)
    | Unequal(expressionList: ExpressionListSort, lineNumber: int,
              type: int, dims:DimSort)
    | Colon(expressionList: ExpressionListSort, lineNumber: int,
            type: int, dims:DimSort)
    | Plus(expressionList: ExpressionListSort, lineNumber: int,
           type: int, dims:DimSort)
    | Minus(expressionList: ExpressionListSort, lineNumber: int,
            type: int, dims:DimSort)
    | Times(expressionList: ExpressionListSort, lineNumber: int,
            type: int, dims:DimSort)
    | Slash(expressionList: ExpressionListSort, lineNumber: int,
            type: int, dims:DimSort)
    | Backslash(expressionList: ExpressionListSort, lineNumber: int,
                type: int, dims:DimSort)
    | ElementWiseTimes(expressionList: ExpressionListSort,
                       lineNumber: int, type: int, dims:DimSort)
    | ElementWiseDivision(expressionList: ExpressionListSort,
                          lineNumber: int, type: int, dims:DimSort)
    | ElementWiseLeftDivision(expressionList: ExpressionListSort,
                              lineNumber: int, type: int, dims:DimSort)
```

```
    | Note(expression: ExpressionSort, lineNumber: int, type: int,
          dims:DimSort)
    | UnaryPlus(expression: ExpressionSort, lineNumber: int,
                       type: int, dims:DimSort)
    | UnaryMinus(expression: ExpressionSort, lineNumber: int,
                type: int, dims:DimSort)
    | ElementWisePower(expressionList: ExpressionListSort,
                       lineNumber: int, type: int, dims:DimSort)
    | Power(expressionList: ExpressionListSort, lineNumber: int,
           type: int, dims:DimSort)
    | Transpose(expression: ExpressionSort, lineNumber: int,
                type: int, dims:DimSort)
    | ComplexTranspose(expression: ExpressionSort, lineNumber: int,
                       type: int, dims:DimSort)
    | ObjectPointerOp(identifierList: IdentifierListSort,
                      lineNumber: int)
    | FunctionCall(identifier: IdentifierSort, args: ExpressionListSort,
                   lineNumber: int)
    | CellArrayAccess(identifier: IdentifierSort,
                      args: ExpressionListSort, lineNumber: int)
    | HandlerWithDef(args: ExpressionListSort, statement: StatementSort,
                     lineNumber: int)
    | Handler(identifier: IdentifierSort, lineNumber: int)
    | Matrix(rowList: ExpressionListSort, lineNumber: int, type: int,
              dims:DimSort)
    | CellArray(rowList: ExpressionListSort, lineNumber: int)
    | Row(expressionList: ExpressionListSort, lineNumber: int,
           type: int, dims:DimSort)
    | Text(text: String, lineNumber: int)
    | End(lineNumber: int)
    | Integer(intValue: int, lineNumber: int)
    | Double(doubleValue: double, lineNumber: int)
    | Imaginary(doubleValue: double, lineNumber: int)
    | Id(identifier: IdentifierSort)

DimSort = ConcDim(int*)

TypeList = ConcType(int*)

IdentifierListSort  =   ConcIdentifier(IdentifierSort*)
```

```
IdentifierSort =
    Identifier(identifierName: String, lineNumber: int)
    | EmptyIdentifier(dummy: int)
```

Listing A.1: MatLab abstract grammar

Appendix B

ANTLR Grammar

```
grammar AspectGram;
options {
 output=AST;
 ASTLabelType=Tree;
}

tokens {
  %include { aspectrule/AspectGramAspectRuleTokenList.txt }
}

@header { package parser; }
@lexer::header { package parser; }

start : lstAspect;

lstAspect
    : -> ^(NoAspANTLR)
    | aspect lstAspect -> ^(ConsAspANTLR aspect lstAspect);

aspect
    : ASPECT NAME '(' lstArg ')' actionBody END ->
                ^(AspANTLR NAME lstArg actionBody);

lstArg
    : -> ^(NoAspArg)
    |argAsp lstArg -> ^(ConsArgANTLR argAsp lstArg);
```

```
argAsp  : NAME -> ^(ArgANTLR NAME);

actionBody
    : input SELECT':'lstSelect lstApply where ->
                ^(ActANTLR input lstSelect lstApply where);

input
    : -> ^(NoInputANTLR)
    | INPUT':' NAME -> ^(InputBodyANTLR NAME);

where
    : -> ^(NoWhereANTLR)
    | WHERE':' NAME -> ^(WhereBodyANTLR NAME);

lstSelect
    : ->^(NoSelectANTLR)
    | select lstSelect -> ^(ConsSelANTLR select lstSelect)
    | '('lstSelect')' symbolSelect lstSelect ->
                ^(BracketSelect lstSelect symbolSelect lstSelect);

select
    : ADD '('NAME')' -> ^(AddArrayANTLR NAME)
    | ADD '('NAME')' '&&' -> ^(AddArrayANTLR NAME)
    | ADD '('NAME')' '||' -> ^(AddArrayANTLR NAME)
    | GET '('NAME')' -> ^(GetArrayANTLR NAME)
    | GET '('NAME')' '&&' -> ^(GetArrayANTLR NAME)
    | GET '('NAME')' '||' -> ^(GetArrayANTLR NAME)
    | SIZE '('NAME')' -> ^(SizeArrayANTLR NAME)
    | SIZE '('NAME')' '&&' -> ^(SizeArrayANTLR NAME)
    | SIZE '('NAME')' '||' -> ^(SizeArrayANTLR NAME)
    | READ '('NAME')' -> ^(ReadVarANTLR NAME)
    | READ '('NAME')' '&&' -> ^(ReadVarANTLR NAME)
    | READ '('NAME')' '||' -> ^(ReadVarANTLR NAME)
    | WRITE '('NAME')' -> ^(WriteVarANTLR NAME)
    | WRITE '('NAME')' '&&' -> ^(WriteVarANTLR NAME)
    | WRITE '('NAME')' '||' -> ^(WriteVarANTLR NAME)
    | CALL '('NAME')' -> ^(CallFunANTLR NAME)
    | CALL '('NAME')' '&&' -> ^(CallFunANTLR NAME)
    | CALL '('NAME')' '||' -> ^(CallFunANTLR NAME)
    | FUNCTION '('NAME')' -> ^(FunANTLR NAME)
    | FUNCTION '('NAME')' '&&' -> ^(FunANTLR NAME)
```

```
    | FUNCTION '('NAME')' '||' -> ^(FunANTLR NAME);

symbolSelect
    : -> ^(NoSymbolSelect)
    | '&&' -> ^(AndSel)
    | '||' -> ^(OrSel);

lstApply
    : -> ^(NoApplyANTLR)
    | apply lstApply -> ^(ConsAppANTLR apply lstApply);

apply
    : APPLY ':' STRINGLITERAL '::' execute ->
                ^(NormApply STRINGLITERAL execute)
    | APPLY '<'HEADER'>'':'STRINGLITERAL '::' execute ->
                ^(HeaderApply STRINGLITERAL execute)
    | APPLY '<'HEADER'>''(' lstArg ')'':' STRINGLITERAL '::' execute ->
                ^(HeaderArgsApply STRINGLITERAL NAME execute)
    | APPLY '<'BODY'>'':'STRINGLITERAL '::' execute ->
                ^(BodyApply STRINGLITERAL execute)
    | APPLY '<'BODY'>''(' lstArg ')'':' STRINGLITERAL '::' execute ->
                ^(BodyArgsApply STRINGLITERAL NAME execute);

execute
    : EXECUTE BEFORE -> ^(Before)
    | EXECUTE AFTER -> ^(After)
    | EXECUTE AROUND -> ^(Around);

ASPECT  :'aspect';
END     :'end';
SELECT  :'select';
APPLY   :'apply';
INPUT   :'input';
WHERE   :'where';
ADD     :'add';
GET     :'get';
SIZE    :'sizeOf';
READ    :'read';
WRITE   :'write';
CALL    :'call';
HEADER  :'header';
```

```
BODY    :'body';
FUNCTION:'function';
EXECUTE :'execute';
BEFORE  :'before';
AFTER   :'after';
AROUND  :'around';
DISP    :'disp';
IF      :'if';
ELSE    :'else';
WHILE   :'while';
FOR     :'for';
RETURN  :'return';
STRINGLITERAL : '"' ( StringEscapeSeq | ~( '\\' | '"' | '\r' |
                    '\n' ) )* '"' ;
fragment StringEscapeSeq : '\\' ( 't' | 'n' | 'r' | '"' | '\\' | '$' |
                             ('0'..'9')) ;
NAME    : ((('a'..'z')|('A'..'Z'))('0'..'9')*('_'(('a'..'z')|('A'..'Z'))|
          '.'(('a'..'z')|('A'..'Z')))*)+;
INT     : ('0'..'9')+;
PLUS    :'+';
MINUS   :'-';
MULT    :'*';
DIV     :'/';
MOD     :'%';

WS : (' '|'\t'|'\n')+ { $channel=HIDDEN; } ;
SLCOMMENT : '//' (~('\n'|'\r'))* ('\n'|'\r'('\n')?)?
             { $channel=HIDDEN; } ;
```

Listing B.1: ANTLR aspect grammar

Appendix C

TOM Abstract Grammar for ANTLR

```
LstAspANTLR  = NoAspANTLR()
             | ConsAspANTLR(a:AspectANTLR, l:LstAspANTLR)

AspectANTLR  = AspANTLR(name:String, arg:LstArgANTLR, b:ActionsANTLR)

LstArgANTLR  = NoAspArg()
             | ConsArgANTLR(arg:AspArg, l:LstArgANTLR)

AspArg       = ArgANTLR(name:String)

ActionsANTLR = ActANTLR(input:InputANTLR, sel:LstSelANTLR,
                        ap:LstAppANTLR, wh: WhereANTLR)

InputANTLR   = InputBodyANTLR(name:String)
             | NoInputANTLR()

WhereANTLR   = WhereBodyANTLR(name: String)
             | NoWhereANTLR()

LstSelANTLR  = NoSelectANTLR()
             | ConsSelANTLR(s:SelectANTLR, l:LstSelANTLR)
             | BracketSelect(s1:LstSelANTLR, sy:SymbolSelect,
                             s2:LstSelANTLR)

SelectANTLR  = ReadVarANTLR(n:String)
             | WriteVarANTLR(n:String)
             | CallFunANTLR(n:String)
```

```
              | FunANTLR(n:String)
              | AddArrayANTLR(n:String)
              | GetArrayANTLR(n:String)
              | SizeArrayANTLR(n:String)

SymbolSelect = AndSel()
              | OrSel()
              | NoSymbolSelect()

LstAppANTLR  = NoApplyANTLR()
              | ConsAppANTLR(a:ApplyANTLR, l:LstAppANTLR)

ApplyANTLR   = NormApply(i:String, ex: Execute)
              | HeaderApply(i:String, ex: Execute)
              | HeaderArgsApply(n:String, arg:LstArgANTLR, ex: Execute)
              | BodyApply(i:String, ex: Execute)
              | BodyArgsApply(n:String, arg:LstArgANTLR, ex: Execute)

Execute      = Before()
              | After()
              | Around()
```

Listing C.1: TOM abstract grammar for ANTLR

Appendix D

Abstract Representation of Function SumVals

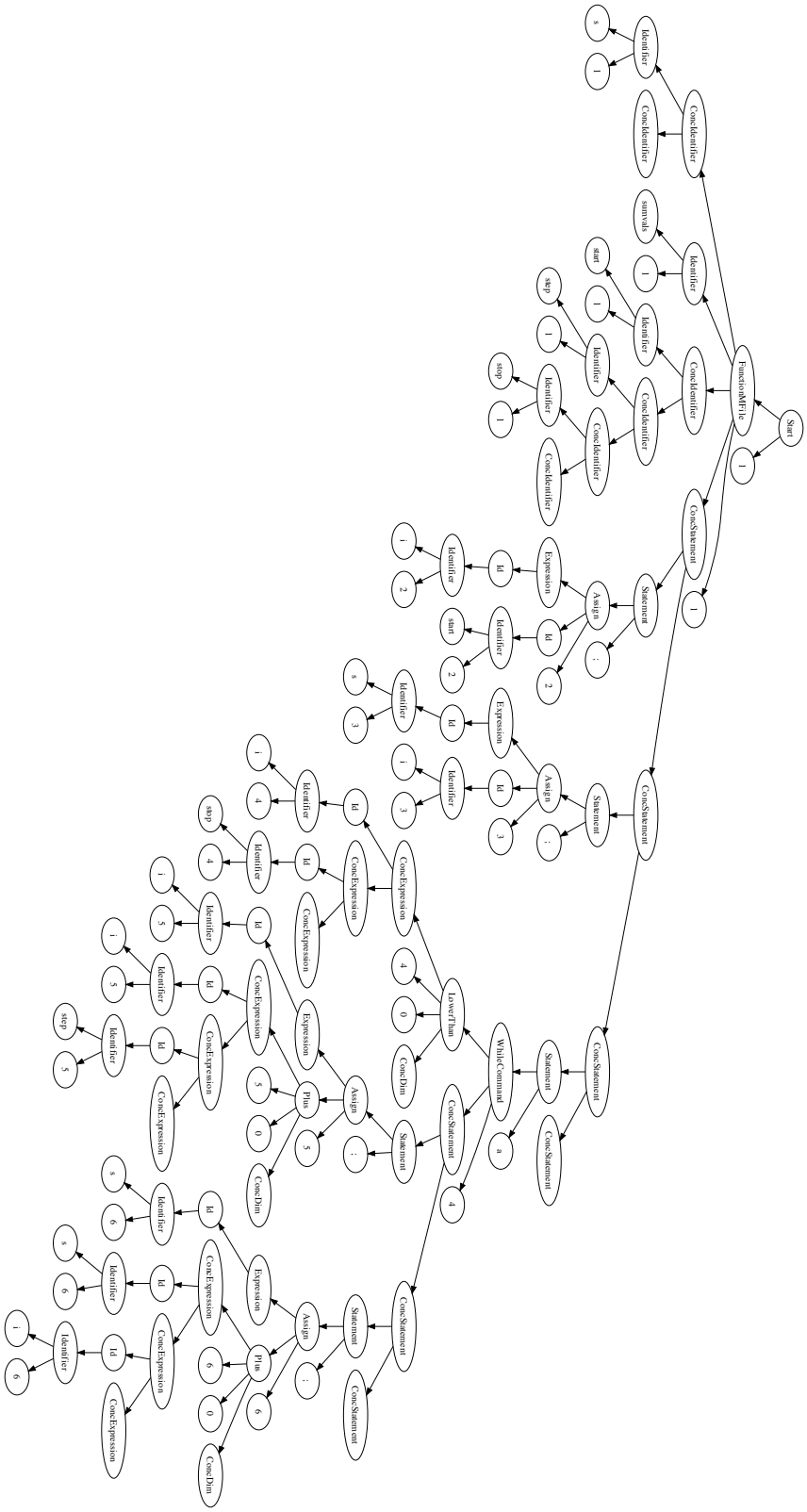

Figure D.1: Visual representation of function SumVals.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] J. ao Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In M. Hanus, S. Krishnamurthi and S. Thompson, editor, *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education*, University of Kiel Technical Report 0210, pages 133–140, September 2002.

[3] T. Aslam, J. Doherty, A. Dubrau, and L. 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.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, Jun 1997.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[6] J.-C. Bach, E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. *TOM Manual*. Institut National de Recherche en Informatique et Automatique (INRIA), May 2009.

[7] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Term Rewriting and Applications*, LNCS. Springer-Verlag, 2007.

[8] E. Balland, P.-E. Moreau, and A. Reilles. Rewriting strategies in java. *Electron. Notes Theor. Comput. Sci.*, 219:97–111, 2008.

[9] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. Elsevier Science, 1998.

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

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

[12] A. V. Deursen and J. Visser. Source model analysis using the jjtraveler visitor combinator framework. *Software: Practice and Experience*, 34, 2004.

[13] J. Fernandes and M. Monteiro. Some thoughts on refactoring objects to aspects. In *VIII Jornadas de Ingeniera de Software y Bases de Datos (JISBD)*, 2003.

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

[15] D. J. Higham and N. J. Higham. *Matlab Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 2nd edition, 2005.

[16] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. 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.

[18] V. Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.

[19] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 168–177, New York, NY, USA, 2003. ACM.

[20] R. Lämmel and J. Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in strategoxt-0.9. In L. et al., editor, *Domain-Specific Program Generation*, LNCS. Springer-Verlag, Nov 2003.

[21] R. Lämmel and J. Visser. A strafunski application letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*. Springer-Verlag, Jan 2003.

[22] P. M. R. Martins. An domain specific aspect language for matlab, 2010.

[23] D. L. Parnas. On the criteria to be used in decomposing systems into modules. pages 139–150, 1979.

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

[25] E. Visser. A survey of strategies in program transformation systems. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*. B. Gramlich and S. Lucas, May 2001.

[26] J. Visser and J. Saraiva. Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages (SBLP)*, May 2004.