Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Multidimensional Languages

A grammar specification for the definition of multidimensional languages

Relatore

Ch.mo Prof. Gennaro Costagliola

Candidato

Alfonso Piscitelli

Matr. 0522500392

Anno Accademico 2018/2019

Acknowlegments

I would first like to thank my thesis advisor, Prof. Gennaro Costagliola, for the support received in this year of work. I have had the opportunity to learn a lot and now, at the end of this journey, I can say that I grew up professionally and academically. I would also thank Dott. Mattia De Rosa for the technical support during the implementation of the IDE.

When I started this "journey" I had a lot of doubt about the success of it, and without the support of my family and my friends, I don't think that I would have success. For these reasons I would thank my family, my parents Giovanni and Enza and my brother and sisters Chiara, Alessio and Benedetta. During the difficulties of this path, I could receive support from my friends. Don Carmine, Luisa and Cesare supported me from the beginning of this experience. They knew first my decision to start the master degree. I would also thank Umberto for his support in final phases, Domenica and Paola for their closeness.

Finally, I would thank all persons happy for my goal: Vittorio, Domenico and Antonella, my ICT and More colleagues, my church friends including anyone that during the trip told me an encouragement.

Abstract

After many years of research, visual languages are gaining again the interest of the scientific community and of software companies that have started to release proprietary visual environments to allow software development through graphical tools. One of the factors of diffusion is certainly the ease of use of visual languages. One of the open questions is still the efficacy of formal specifications for graphical environments. To this aim, many formalisms have been defined but they are usually demanding to understand and to use. In this thesis, a new grammar specification for visual languages is introduced and a smart editor for it is defined and implemented. The resulting *integrated development environment* statically detects errors and provides warnings and suggestions while a language developer writes a grammar in the new format. Moreover, the environment translates the correctly completed grammar in the format of a positional grammar for which parser generation tools already exist.

Contents

1 Introduction

The work in this thesis focuses on the study of *Visual Languages* and grammar formalisms that can define them, with the first being an increasingly hot topic both in the scientific community and among companies.

The scientific community has started to become interested in Visual Languages in the 70s. In those years, the scientific community was focused on *picture processing* [1]. There are two possible definitions given by Chang and reported in [2]:

- Languages for processing visual information
- Languages for programming with visual expressions

The availability of Visual Languages, nowadays, is very high and a lot of these are used both in the scientific community and in working environments such as UML, CASE [3, 4], statecharts, flowcharts, automata and many other. The ease with which they can be learned and used is certainly one of the key factors of their diffusion. This aspect makes them interesting even in the educational world, to bring an increasingly wider audience of people closer to programming. *Scratch*[5] and *App Inventor*[6], from Massachusetts Institute of Technology, are the most famous but not the only ones.

In the last few years also companies have started to have interest in Visual Languages, in particular about *low-code* and *no-code* environments. These kinds of instruments are used to reduce the usage of the code and also to improve productivity. Sometimes these instruments are included in more complex tools in order to provide the customer with an easy way to extend their product's functionalities.

In this direction, there are two important companies: *SAP*, that allows customers to define custom Workflow in their software (Figure 2) with *SAP Workflow Builder*; and *Open Text*, that allows to define custom extensions that will be executed on their Open Text instance. In Figure 1 there is an example of an extension developed with *Open Text Process Designer*

This research is also focused on grammars formalisms that allow the definition of *Visual Languages*. Among many formalisms defined so far for the specification of diagrammatic languages, in this thesis, we will refer to *Positional Grammars* (PG) [7] and its extension *Extended Positional Grammar* (XPG) [8]. For both of them, there are tools for the generation of parsers.

All of these formalisms are very powerful but are also difficult to understand and to write. This problem introduces the main goals of this thesis: extending the XPG formalism, creating a new grammar formalism easy to understand and, finally, help the developer in its definition work. To achieve these goals this thesis proposes a solution organized in the following steps:

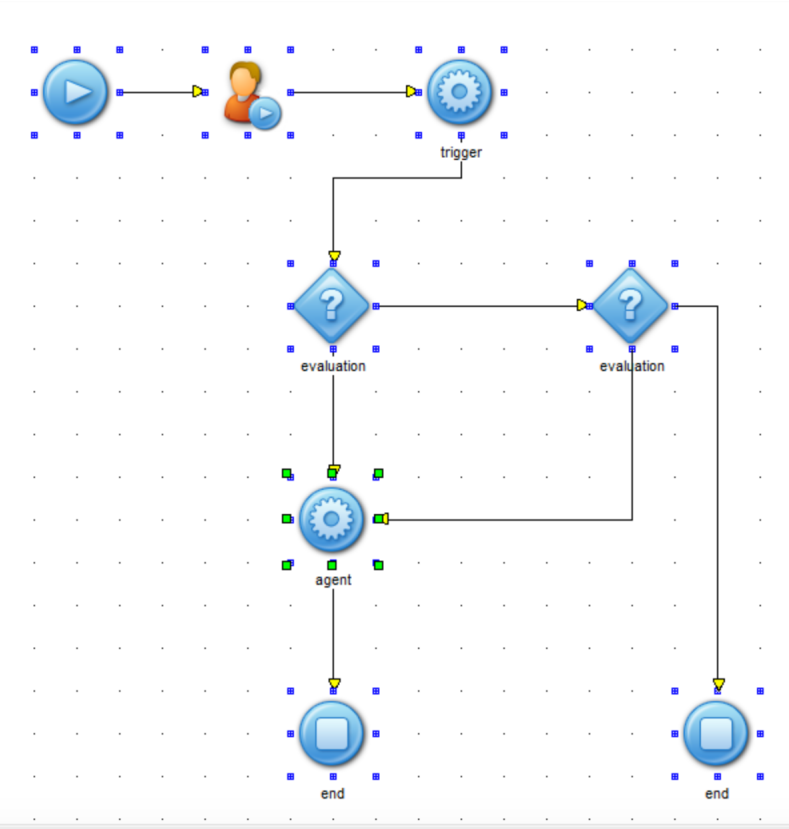

Figure 1: OpenText Process Designer

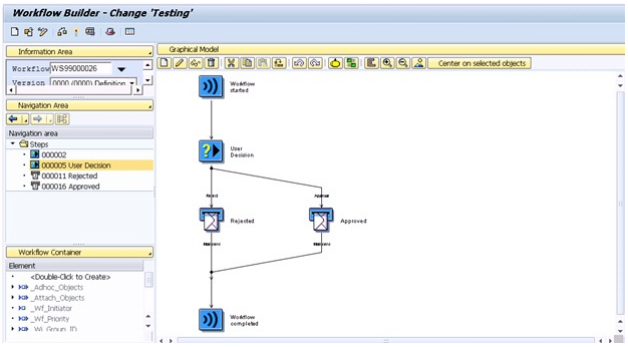

Figure 2: SAP Workflow builder

- Definition of a new grammar specification. The new grammar specifications, called *Multidimensional Grammar* (MG) has a simpler syntax than XPG and is based on logic programming predicates. MG can describe the same languages that are described by XPG.
- In order to use existing tools for parser generation, *MG* is translated into *XPG*.
- In order to help a developer to write MG grammars, an *Integrated Development Environment* (IDE) detecting errors and providing warning and suggestions is developed.

This thesis is structured as follows. Chapter 2 describes the architecture of the proposed solution. Chapter 3 introduces the formal definition of a *Multidimensional Grammar* and the description of its main elements. The chapter ends with the definitions of some visual language examples written with this formalism. Chapter 4 describes the development of the *Integrated Development Environment* and the implementation of *Multidimensional Grammars* through the Xtext programming language. In this chapter, the focus is both on the grammar and the algorithms that transform an MG in an intermediate form. Chapter 5 describes the algorithms that translate the intermediate form of an MG into an equivalent XPG. Before describing the algorithms, the formal definitions of *PG* and *XPG* formalisms are resumed. The chapter ends showing the translation into XPG of the visual language examples presented in chapter 3.

2 System Architecture

In the introduction, we mentioned two important grammar formalisms for Visual Languages: *PG* and *XPG*. These formalisms (described exhaustively in Section 5) are very powerful, can represent significative classes of languages but are difficult to use and to understand. It is also true that for these formalisms a parser generator has already been built. However, the non existence of a development environment makes these grammars very hard to write. For these reasons, the first step of the proposed solution is the definition of a new, more intuitive grammar formalism called *Multidimensional Grammar*. An exhaustive treatment of this formalism is given in Section 3.2. The second step is to create an *Integrated Development Environment* based on an Eclipse Instance to help developers to write an *MG* and translate it into *XPG* in order to benefit of its implementation. The architecture of the IDE is shown in Figure 3.

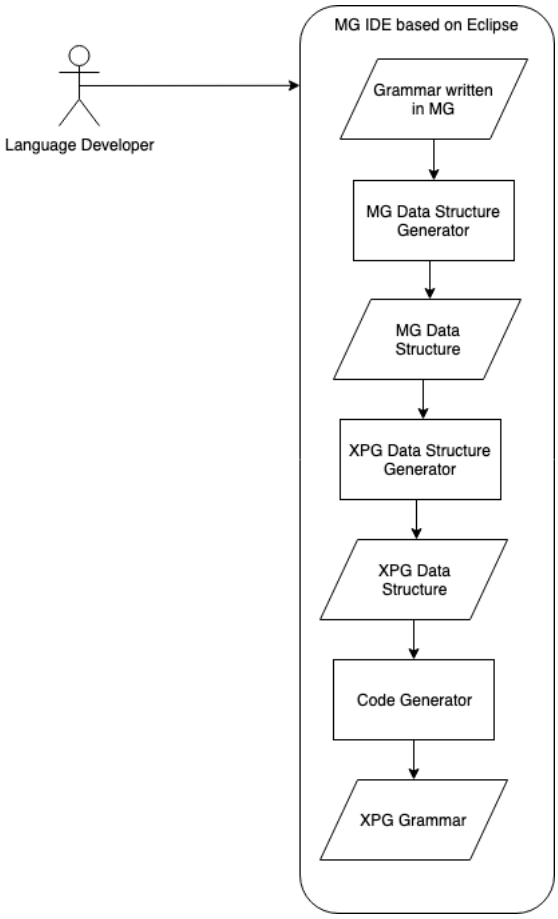

Figure 3: Architecture of the IDE proposed

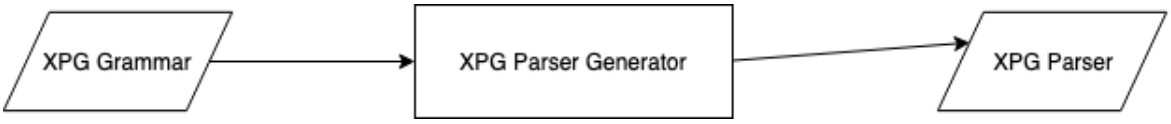

Figure 4: Workflow after the generation of XPG

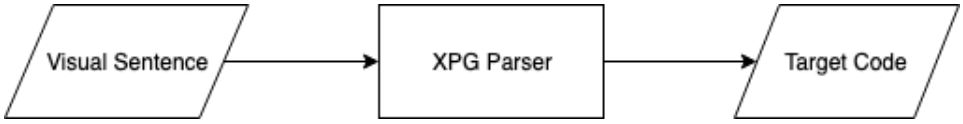

Figure 5: Workflow from an input visual sentence

The IDE has been developed by using two Eclipse Foundation's frameworks: *Xtext* and *Xtend*.

Xtext is an open-source software framework for developing domain-specific languages (DSLs) [9]. Chapter 4 describes the *Xtext* implementation of the grammar specification for MG and the *MG data structure* as the first intermediate representation. This representation is the output of the *MG Data Structure Generator* that has been written with the Xtend language presented in detail at the end of section 4. The *XPG Data Structure Generator* takes as input the *MG Data Structure* and produces an equivalent *XPG data structure* as described in chapter 5. Finally, the Code Generator produces an XPG equivalent to the input MG as described at the end of chapter 5.

Once the XPG specification has been obtained, the XPG Parser generator produces the corresponding XPG parser (see Figure 4) that can be used to parse/compile the visual sentences of the implemented visual language (see Figure 5).

3 Multidimensional Languages

In this chapter, we describe what a *Multidimensional Language* is and give a formal definition and some examples. A *multidimensional sentence* is given by a set of *composite elements* which are related through their *components*. An example of multidimensional sentence is:

$$\textbf{snode}(a, b)\ \textbf{node}(c, a)\ \textbf{fnode}(b, c) \tag{1}$$

where **snode**, **node** and **fnode** are the names of composite elements with two components each, identified by their position. Each component has a type and a semantic. As an example, to represent a graph, for each composite element, position 1 is set to have type *attach point* indicating an outgoing attach point, semantically, while position 2 is also set to have type *attach point* indicating an incoming attach point, semantically. In order to define the relationships among components, each position is associated to a *joint identifier* whose type specifies the specific relation involved. The relationship type and the component types must be compatible. Moreover, the relation direction is from the leftmost element in the string to the rigthmost ones. In the example, the joint identifiers are a, b and c and their type is given by the relation *connection* which is compatible with the type *attach point* of the components. In particular, a connects the *out* (first) component of **snode** to the *in* (second) component of **node**, b connects the *in* (second) component of **snode** to the *out* (first) component of **fnode** and c connects the *out* (first) component of **node** to the *in* (second) component of **fnode**.

Each component maybe related with others through more than one relation. As an example, sentence (1) can be modified in

$$\textbf{snode}(a, b)\ \textbf{node}(c, \{d, a\})\ \textbf{fnode}(\{d, b\}, c) \tag{2}$$

where both the second component of **node** and the first component of **fnode** present two joint identifiers.

Figure 6 shows a possible visual interpretation of sentence (2). Here, the single component of each element is depicted as a circle, the relation *connection* is depicted as an outgoing or an incoming arrow depending on the components to where they are attached.

As another example, let us consider the sentence

$$\textbf{circle1}(\{a, b\})\ \textbf{circle2}(\{c, a\})\ \textbf{circle3}(\{b, c\}) \tag{3}$$

In this case we set the type of the component of each element to be a *Cartesian plane position* and a is defined with type *"left_ of"*, b with type *"up_ left_ of"* and c with type *"up_ right_ of"*. A possible visual interpretation of sentence (3) is then given in Figure 7. Note that the types of the joints (the relations) are again compatible with the types of the element components to

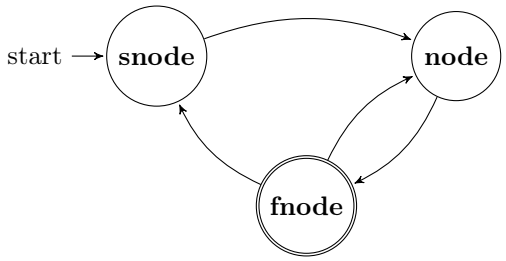

Figure 6: Visual interpretation of sentence (2)

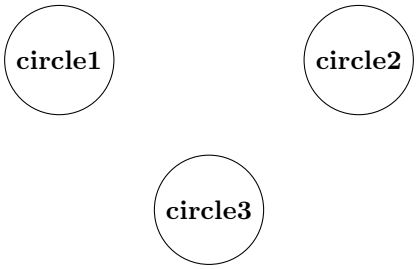

Figure 7: Visual interpretation of sentence (3)

which they are applied. Furthermore, in general, a joint identifier a is of a type given by an $n-ary$ relation, with $n \geq 1$: the first argument of the relation is the first element from the left in the sentence marked by a while the remaining arguments are given by the $n-1$ components, following the first, marked by the same joint identifier a in the string.

3.1 Formal definition for Multidimensional Grammar

A *Multidimensional Language* is defined as a set of multidimensional sentences built on the alphabets of the composite elements and of the relations as described above. Examples of *Multidimensional Languages* include a set of groups of people with the same age and the same salary, or a set of automata, a set of structured and/or non structured flowcharts, a set of paths that a robot should follow.

Multidimensional Grammars are a formalism for the definition and implementation of *Multidimensional Languages*.

In the following, the formal definition of the *Multidimensional Grammar*, the rules and some examples of *Multidimensional Languages* are given.

More formally, a *Multidimensional Grammar* can be represented by a seven-tuple, $[V_T, V_N, S, R, J, T, P]$, where:

- V_T, is a finite non-empty set of *terminals*.

- V_N, is a finite non-empty set of *non-terminals*
  where $V_T \cap V_N \neq \emptyset$.
- $S_N \in V_N$, is a start *non-terminal* of the language.
- R, is a finite non-empty set of relationships.
- J, is a finite set of symbols called Joint identifiers.
- T, is a finite set of symbols called Tie-Point identifiers.
- P, is a finite non-empty set of productions.

Both *terminals* in V_T and *non-terminals* in V_N are composite elements with n components, otherwise said *predicates* with n arguments each one with a type. Each *non-terminal* A is described by a production rule in P with format: $A(a_1, a_2, \ldots, a_n)\ \alpha \rightarrow \beta$ where β is a list of *terminal* and/or *non-terminal* instances and α is a subset possibly empty of *terminals* instances in β. The arguments $a_1, a_2, \ldots, a_n$ of the *non-terminal* predicate A in the rule head are tie-points identifiers in T while the arguments on the right-hand side of the production are (set of) joint or tie-points identifiers in $J \cup T$. The names *joint* and *tie-point* are equivalent to those defined in [10].

As an example, the following productions describe the *non-terminal* Statements as a list of two instances of the *non-terminal* Statement and as the terminal INSTRUCTION. Both the terminal and *non-terminal* argument type is set to *attach_point*. Furthermore, $a, c \in T$ and $b \in J$:

$$Statements(a, c) \quad \rightarrow \quad Statement(a, b)\ Statement(b, c) \ \mid \ INSTRUCTION(a, c)$$

R contains the types of relations that can be used in the grammar and they are used to give a type to the joint identifiers. In our example, if $R = \{connection\}$ then it is possible to declare
connection.attach_point b;
Finally, $S_N \in V_N$ is the starting *non-terminal*.

3.2 MG file Specification

This section describes the format of an MG file representing a *Multidimensional Grammar*.

The format of the MG file is expressed in Backus-Naur Form and will be the base for writing the XTex MG grammar specification.

An MG file consists of four sections as shown in the following rule.

mgrammar ⟶ CSCODE
RelationSect
TerminalSect
RuleSect

In particular,

- CSCODE: defines instructions written in C# language. These instructions will not be processed by the ***translator***.
- RelationSect: defines the section of the grammar that contains the relationships definitions.
- TerminalSect: defines the section of the grammar that contains the *terminal* definitions.
- RuleSect: defines the section of the grammar that contains the productions and other important declarations of the language (such as Joints and Tie-points).

3.2.1 The RelationSect section

The *Multidimensional Grammars* have the characteristic that can define many types of relationships between two predicates. These relationships must be declared in the **Relation Section** in a format that is defined with the following BNF rule. This section starts with the **relations** keyword.

RelationSect ⟶ 'relations' '{'
Relations
'}'

The **Relations** *non-terminal* models a list of single **Relation** elements that are defined in the **RelationSect** section.

Relations ⟶ Relation
Relations
|
ε

Relation ⟶ RelationName
‘(’ IdList ‘)’
‘[’ Threshold ‘]’
|
RelationName
‘(’ IdList ‘)’
RelationName ⟶ ID

Because a single Relation element can have a list of attributes (that define the type of the syntactic attributes for the relation), it has the **IdList** *non-terminal* that models these types of attributes.

IdList ⟶ ID
OtherIdList
OtherIdList ⟶ ID
OtherIdList
|
ε

At the end of the production there is the Threshold *non-terminal*, that models the threshold of the Relation and it is defined with the variable **Thresholds** that is described in the following rule.

Threshold ⟶ ‘threshold’
‘=’
FLOAT

The relationships defined in the **RelationSect** section are used in the **Rule** section with the definition of Joints and Tie-Points. These elements will be used in the production definitions to specify how the predicates are related.

3.2.2 TerminalSect section

As with other grammars formalism, in the *Multidimensional Grammar* there are **Terminal** elements. *Terminals* represent the alphabet of the language defined by the grammar. In our formalism the declarations of the *Terminals* are set in the **terminals** section, modeled by the *non-terminal* **TerminalSect**.

TerminalSect ⟶ ‘terminals’ ‘{’
Terminals
‘}’

Terminals ⟶ Terminal
Terminals
|
Terminal

A single *Terminal* is defined by these properties:

- Semantic type, that can be a string, int or other custom type defined by C# language in the CSCODE section;
- The optional character @ that defines if the *terminal* is a start *terminal* or not;
- The name of the *terminal* predicate;
- An integer, that defines how many times the *terminal* can be visited by the parser;
- A double value, that defines the threshold of the *terminal*;
- A semantic value, that can be a C# code, string, integer or double value;

Terminal ⟶ TerminalType
IsStartTerminal
Predicate
TerminalOptions
SemanticValue
|
TerminalType
IsStartTerminal
Predicate
SemanticValue

TerminalType, IsStartTerminal and SemanticValue are variables and their definitions are the following:

- **TerminalType** ⟶ TypeId | ε;
  This rule models the semantic type of the *terminal*;
- **IsStartTerminal** ⟶ '@' | ε;
  This rule models the character [@] that, if present, sets the *terminal* as start-terminal;
- **SemanticValue**⟶ '=' Value | ε;
  This rule models the semantic value of the *terminal*;

A *Terminal* element can have two options. One of these is **NVisit** and establishes how many times the parser can visit the *terminal*. The other option is **Threshold** and is related to the threshold of the *terminal*.

TerminalOptions ⟶ '[' Nvisit ',' Threshold ']'
| '[' Threshold ',' Nvisit ']'
| '[' Threshold ']'
'[' Nvisit ']'

NVisit ⟶ '#visit=' INT
| '#visit=' NOLIMIT

Threshold ⟶ 'threshold'
'='
FLOAT

3.2.3 Rule section

The Rule section is another important section of our implementation of *Multidimensional Grammars*. In the rule section it is possible to define the following elements:

- **Joints**: the elements that link two or more predicates;
- **Tie-Points**: the elements that specify the syntactic attributes of a *non-terminal*
- **Non-Terminals**: the variables of the grammar with the related production rules.

The rule that models the Rule section is **RuleSect** and is defined as follows:

RuleSect ⟶
'rules' '{' Decls Productions '}'
'rules' '{' Productions '}'

Productions ⟶
Production Productions
Production

The variable *Production* models the definition of an MG *non-terminal* and the related production rule.

Production $\longrightarrow$

NonTerminalType
ID
LeftHandSideParam
LPredicate
ARROW
RightHandSideProduction

Each Production is characterized by a semantic type that specifies the type of the variable (for example a string, an integer etc) and by a name.

NonTerminalType $\longrightarrow$

TypeId
|
ε

In the definition of the *non-terminal* it is important to specify if the element has attach points and the type of these points. The attach points specify how the elements, when they are used as a predicate, are linked to the other predicates. This functionality is managed by the variable *LeftHandSideParam*.

LeftHandSideParam $\longrightarrow$

‘(’ Points ‘)’
|
‘(’ ‘)’
|
ε

The following productions describe how the points (joint or tie-points) are modeled in the MG formalism:

Points $\longrightarrow$ Point OtherPoints
| Point OtherPoints ‘@’ Point OtherPoints
| ‘@’ Point OtherPoints
| ‘@’ Point
| Point
ε

```
OtherPoints  ⟶  ',' Point
                    OtherPoints
             |      ',' Point;
Point        ⟶  ',' ID '('
                Idlist ')'
             |      ID;
```

In our specification of *Multidimensional Grammar* it is possible to send to the parser the instructions about the management of the input, in particular the addition of other elements to the input. Substantially it is possible to specify to the parser that a specific rule is reducible not only by one single variable (the *non-terminal*) but also by more elements.

```
LPredicate  ⟶
                    ID
                    '(' Arglist ')'
            |
                    ID
```

On the right side of the production it is possible to define the body of the rule with a list of predicates (where each predicate can have attributes to specify what kind of relationship there is between them). Furthermore it is possible to define other interesting properties:

- *Threshold*: the threshold of the *non-terminal*;
- *CSCode*: some instructions written in C# language that the parser will execute;

These properties aren't mandatory, it is only possible to define one or none at all.

```
RightHandSide  ⟶
                        RPredicates
                        TauRulesOptional
                        ThresholdOptional
                        CSCodeOptional
RPredicate     ⟶
                    ID '(' Arglist ')'
               |
                    ID
```

Another important element of the language is the *TauRule* element. This element, modeled by the *TauRulesOptional* rule, contains many semantic instructions that will be sent to the parser. For this reason, the instructions must be written in C# language.

TauRules ⟶ ‘([’ TauRule
TauRulesCommaSeparated
‘])’

TauRulesCommaSeparated ⟶ ‘,’ TauRule
TauRulesCommaSeparated
|
‘,’ TauRule

The list of TauRules starts with the token **([** and it ends with the token **])** . The instructions that are between these tokens are considered as TauRule. Each TauRule has an identifier name, a list of points, some instructions written in C# language and a Value that can be of four types:

- Integer value
- String value
- Float value
- C# code value

TauRule ⟶ ‘(’ ID
‘(’ OptionalPoints ‘)’ ‘,’
CSCODE
OptionalTauRuleValue ‘)’

The following rules show the definitions for *OptionalPoints* and for *OptionalTauRuleValue*.

OptionalPoints ⟶ Points
|
ε

OptionalTauRuleValue ⟶ ‘,’ INT
| ‘,’ STRING
| ‘,’ FLOAT
| ‘,’ CSCODE
| ε

3.3 Language examples and related grammars

In this section some examples of languages are presented together with their related grammar written in the format described above. The first language that is described is the language of palindrome strings. This language is *Context free* and is first specified through a string CFG and then through a Multidimensional Grammar. Another language that will be presented is the Flow chart language. Finally the language $A^nB^nC^n$ is shown. This language, that isn't *Context free*, is described with the MG formalism.

3.3.1 A simple CFG language: palindrome strings

A palindrome is a string that reads the same forward and backward. For example, the following strings are palindromes: *pop, otto, madamimadam*. It is possible to define a simple *Context Free Grammar* for this language. For the sake of simplicity, for this example, an alphabet with two symbols is considered: A and B.

$$
\begin{array}{lcl}
S & \longrightarrow & A\ S\ A \\
 & | & B\ S\ B \\
 & | & A \\
 & | & B \\
 & | & \varepsilon
\end{array}
$$

This language can also be described in the MG formalism, and the result is reported in the following rows:

```
({ })
relations{ }

terminals {
        string A(attachPoint);
        string B(attachPoint, attachPoint);
}

rules{
        S -> A S A | B S B
             | A | B | ;

}
```

It can be noted that the **Multidimensional Grammar** reduces to a traditional string grammar when dealing with string languages.

3.3.2 A language for flow chart

In this section a simple version of the *Flowchart diagram* is presented. Generally a flow chart has different types of elements: instructions, predicates, deciders and other types of elements that can define simple and complex instruction flow. In our example the language has only these types of *terminals*:

- **START** element: this element represents the begining of the flow;
- **INSTRUCTION** element: this element represents the instruction, a generic operation that will be executed;
- **PREDICATE** element: this element represents a predicate that, generally, has a condition and two ways to continue the execution (for example the if predicate);
- **END** element: this element represents the final block of the flow.

Generally these elements are represented by geometric shapes linked between them. The line represents a *link* relationship between two *attach points*.

In Figure 8 it is possible to see a simple flow chart with a start element, an instruction and a predicate with a boolean condition. In the following the links between the element components in the chart are listed:

- A link between the **START** element and the **INSTRUCTION** element;
- A link between the first **INSTRUCTION** and the **PREDICATE**;
- A left link between the **PREDICATE** and the **INSTRUCTION**;
- A right link between the **PREDICATE** and the **INSTRUCTION**;
- Two links between the **END** element and the relative instructions elements;

At this point of the discussion we can make these observations:

- the **START** and **END** *terminals* have 1 attach point;
- **INSTRUCTION** has 2 attach points;

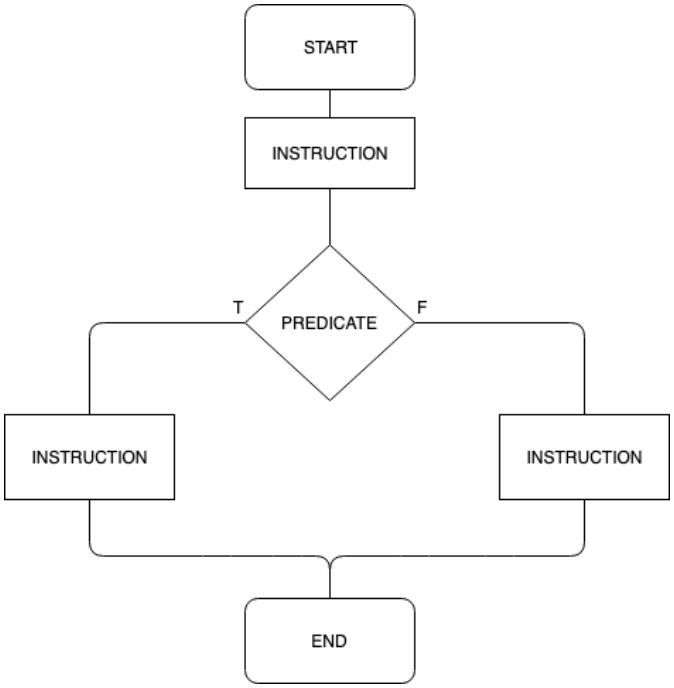

Figure 8: Example of Flowchart

- **PREDICATE** is linked with an element on the top and with two other elements on the left (in case of a true condition) and on the right (in case of the a condition). So, **PREDICATE** has 3 attach points.

It is very important to distinguish the various attach points in a single element. A simple way to distinguish the attach points is to enumerate them. The result is represented in Figure 9 .

With this image we can write some sentences that describe the chart. The following links apply:

- The 1st point of **START** to the 1st point of the topmost **INSTRUCTION**
- The 2nd point of the topmost **INSTRUCTION** to the 1st of **PREDICATE**
- The 2nd point of **PREDICATE** to the 1st of the leftmost **INSTRUCTION**
- The 3rd point of **PREDICATE** to the 1st of the rightmost **INSTRUCTION**
- The 2nd point of the leftmost **INSTRUCTION** to the 1st of **END**
- The 2nd point of the rightmost **INSTRUCTION** to the 1st of **END**

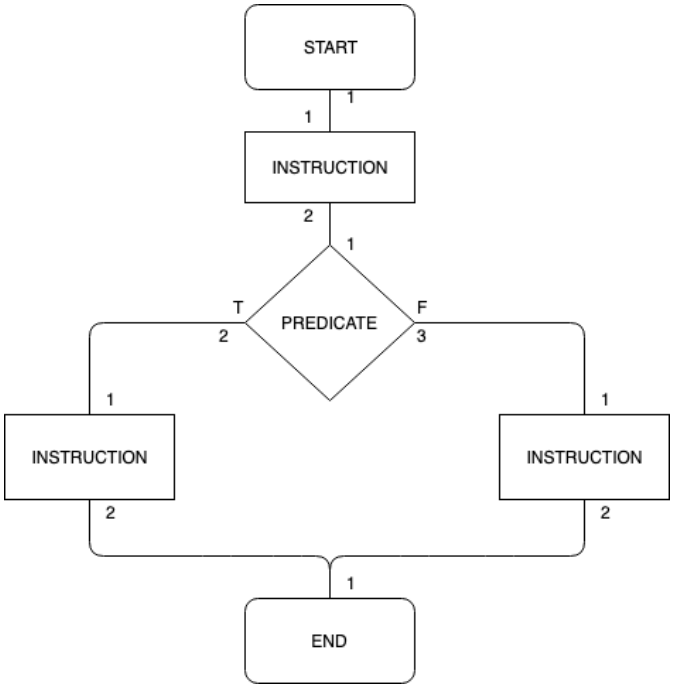

Figure 9: Flowchart with the evidence of attach points

We start presenting the MG grammar for the Flowchart language with the presentation of the *terminals* and the relationships.

The only relationship that we have in this language is the *Link* between two attach points.

```
relations {
  link(attachPoint,attachPoint);
}
```

In this declaration a link is a connection between two *attachPoint*. The *terminals* are the four elements described below.

```
terminals {
  string @START(attachPoint);
  string INSTRUCTION(attachPoint, attachPoint);
  string PREDICATE(attachPoint, attachPoint, attachPoint);
  string END(attachPoint);
}
```

The symbol '@' before the declaration of **START** means that **START** is the first *terminal* that will be read by the parser during the execution. For each *terminal* it is important to define

the number (and the type) of the attach points. So, now we can say that START has 1 attach point, INSTRUCTION has 2 attach points, PREDICATE has 3 attach points and END has 1 attach point.

After the declaration of *Terminals* and *Relations*, it is necessary to write the productions of the language. In our flow-chart example there are these elements:

- **Statement**, i.e., a simple instruction or a predicate instruction leading to an *if-then*, an *if-then-else* or a *while* instruction;
- **Statements**, i.e., a sequence of **Statement**;
- **Program**, i.e., the final flowchart composed by a **START** element, **Statements** and an **END** element.

In the common string CFG formalism the previous rules could be modelled by the *terminals* for the main elements (START, END, INSTRUCTION, PREDICATE) and two rules, one to model the Statement (a Statement can be a single instruction or a predicate) and the other to model the Statements (a list of Statement).

This representation reports the basic structure of a flowchart but does not take into account the relationships among the language elements. In our case, nor the links nor their attach points are specified. Finally another limitation of the last representation is the impossibility to specify custom actions to execute depending on the link and, related about this topic, the impossibility to specify custom relations between the elements.

These limits can be overcome by describing this language as a *Multidimensional Language*. With a *Multidimensional Grammar* it is possible to consider the *link* as a relationship between two predicates. With the declaration of the Joints identifiers it is possible to specify the specific relationship between the predicates, while with the declarations of tie-point identifiers it is possible to specify how groups of elements can be related to others. These declarations are at the beginning of the *Rule section*.

```
rules {
        link.attachPoint a, b, c, d;
}
```

Here, a, b, c, and d are declared both as joints of type link and tie-point of type attachPoint. This means they can be used in a production in both ways.

After the joint and tie-point identifier declarations, it is possible to define the productions of the grammar. For example, the sentence:

```
Program() -> START(a) Statements(a,b) END (b)
```

means that a flowchart Program is composed of the *terminal START*, the *non-terminal* Statements and the *terminal END*. *START* is linked to the predicate Statements through the joint *a*, since *a* is the first argument in both the START and Statements predicates, and Statements is linked to *END*, since *b* is the second argument in Statements and the first argument in END. So, in particular, it is possible to say that between START and Statements there is a link from the 1st attach point of START to the 1st attach point of Statements. In the XPG grammar formalism this would be represented as:

```
Program -> START 1_1 Statements 2_1 END
```

Just as it is possible to use a joint identifier to specify the existence of the relations, it is also possible the usage of identifiers to specify the non-existence of relationships between two predicates. This is possible by adding the symbol '!' before the identifier. For example, the following sentence

```
S() -> A(a) B(a,!b) C(b)
```

means that the there must be no link relation between the 2nd attach point of B and the 1st attach point of C. In the XPG formalism this would be denoted as:

```
S() -> A link(1,1) B nolink(2,1) C
```

In the following block there is the complete grammar that defines our flow chart language.

```
Program()   ->  START(a) Statements(a,b) END(b) ;

Statements(a,c)  -> Statements(a, b) Statements(b,c)
    |
        Statement (a,c)     ;

Statement(a,c) ->  INSTRUCTION(a,c)
    |
       PREDICATE(a,b,c) Statements(b,a)
    |
       PREDICATE(a,b,c) Statements(b,c)
    |
       PREDICATE(a,b,d) Statements(b,c:!a) Statements(d,c);
```

3.3.3 A Multidimensional Grammar for the $A^nB^nC^n$ language

In this section we present a grammar that generates strings in the format $A^nB^nC^n$. This language is *Context Sensitive*, i.e., it can be defined by a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of *terminal* and *non-terminal* symbols.

The language $A^nB^nC^n$ is composed by the strings that satisfy the following properties:

- The string is composed by only A following by only B following by only C;
- The number of As, Bs and Cs are the same.

The strings *AAABBBCCC*, *AABBCC*, *AAAAABBBBBCCCCC* are in the set of strings in $L(A^nB^nC^n)$; *BBBCCCAAA*, *AABBCCCC* aren't in $L(A^nB^nC^n)$.

We start the presentation of the language with the description of the *terminals*. Because we have strings in the format $A^nB^nC^n$, the *terminals* of this language are:

- A
- B
- C

For this language, the parser we want to build through an MG grammar first analyzes A^nB^n then analyzes B^nC^n. To do this two relationships will be used:

- LEFT
- RIGHT

The main idea to create this grammar and accept the language $A^nB^nC^n$ is as follows: during the analysis of the first part of the string (A^nB^n), the parser, while reducing an AB instance, deletes A from the input but keeps B. This behaviour is important because it allows the analysis of the second part (B^nC^n) to continue. Here, the parser will start the analysis right after the last B and will proceed from left to right to recognize the Cs and from right to left to re-parse the Bs. In this case, at each reduction of BC it will delete both from the input.

In the following block there are the declarations of the *Joint* and *Tie-point* identifiers. In particular, a and b are tie-points of type Coordinate, while r, r1 and l are joint or tie-points. These elements will be used to relate the elements of the grammar.

```
Coordinate a, b;
RIGHT.Coordinate r, r1;
LEFT.Coordinate l;
```

The *non-terminals* are:

- The start *non-terminal*: S
- The *non-terminal* BlockAB that models the part A^nB^n of the string;
- The *non-terminal* BlockC that models the part B^nC^n of the string;

The start *non-terminal* (S) is simple and it is described in the following block.

```
S() -> BlockAB(_, r) BlockC(r,_);
```

This rule considers the input sentence as composed by a block of A^nB^n letters and a block of B^nC^n letters (with Bs overlapping). Let us consider now the following rule:

```
BlockAB(a, b) B(b) ->  A(r:a) BlockAB(r,r1) B(r1:b)
        |
            A(r:a) B(r:b) ;
```

The production described above is particular. Looking carefully at its left side it is possible to see that it contains two elements:

- The *non-terminal BlockAB*
- The *terminal B*.

When the parser reduces the production it will:

- substitute in the stack the matched elements on the right side with the first element in the head of the production (BlockAB)
- delete from the input the matched *terminals* on the right side of the production (A and B)
- add to the input the *terminals* on the left side of the production after the first elements (B).

One constraint is that the *terminal* on the left side must be equal to a *terminal* on the right side (B(b), is indeed equal to B(r1:b)).

The following block presents the complete grammar for this language.

```
relations {
    LEFT(Coordinate,Coordinate);
    RIGHT(Coordinate,Coordinate);
}

terminals {
      A(Coordinate);
      B(Coordinate);
      C(Coordinate);
}

rules {

    Coordinate a, b;
    RIGHT.Coordinate r, r1;
    LEFT.Coordinate l;

    S() -> BlockAB(_, r) BlockC(r,_);

    BlockAB(a, b) B(b) ->  A(r:a) BlockAB(r,r1) B(r1:b)
        | A(r:a) B(r:b) ;

    BlockC(a, b) -> BlockC(r, l) C(r:a) B(l:b)
        | C(l:a) B(l:b) ;
}
```

In the last production the joint identifier 'l', defined as a left relation, allows the re-inserted Bs backwards to visit.

4 Multidimensional Grammar Implementation

The previous chapter describes the grammar specification of the *Multidimensional Grammars*. In this chapter, we describe the first part of the implementation of the IDE which has been created for the MG formalism in this thesis. In particular, we will describe the Xtext specification, the internal representation of an MG, the main algorithms to generate it, and the frameworks used for their implementation.

4.1 The Development Environment for Multidimensional languages

The *development environment* created is an Eclipse Instance: Figure 10 shows a running instance of this editor. This editor helps the developer during the language specification, through the error highlighting, warning, and some suggestions. The technologies used to implement this editor are the following:

- Java Standard Edition;
- Xtext Framework;
- Xtend Framework;

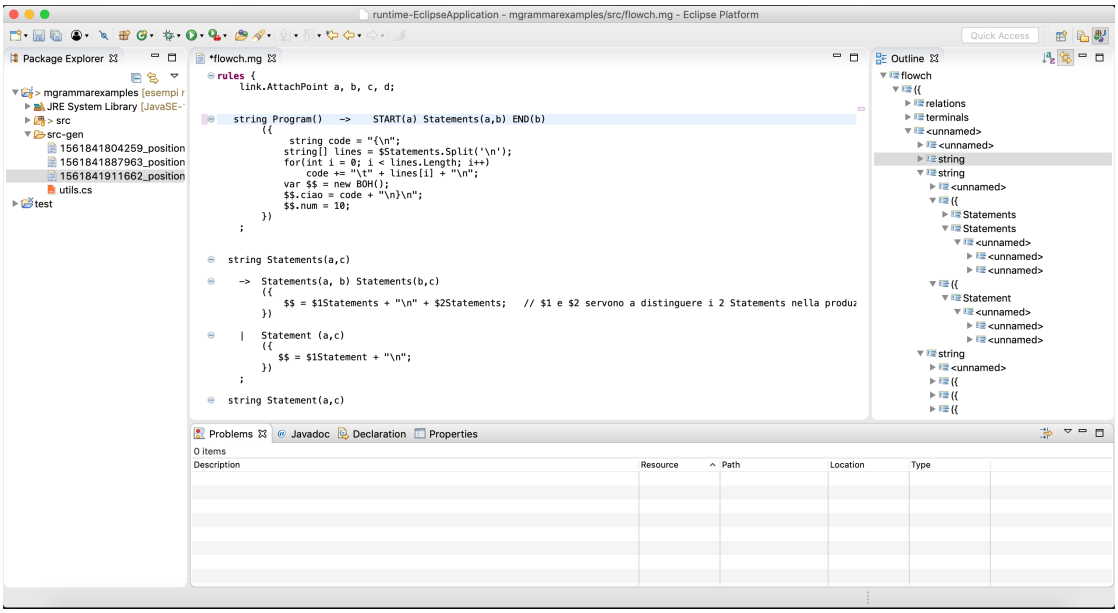

Figure 10: A screen of the Visual Editor developed

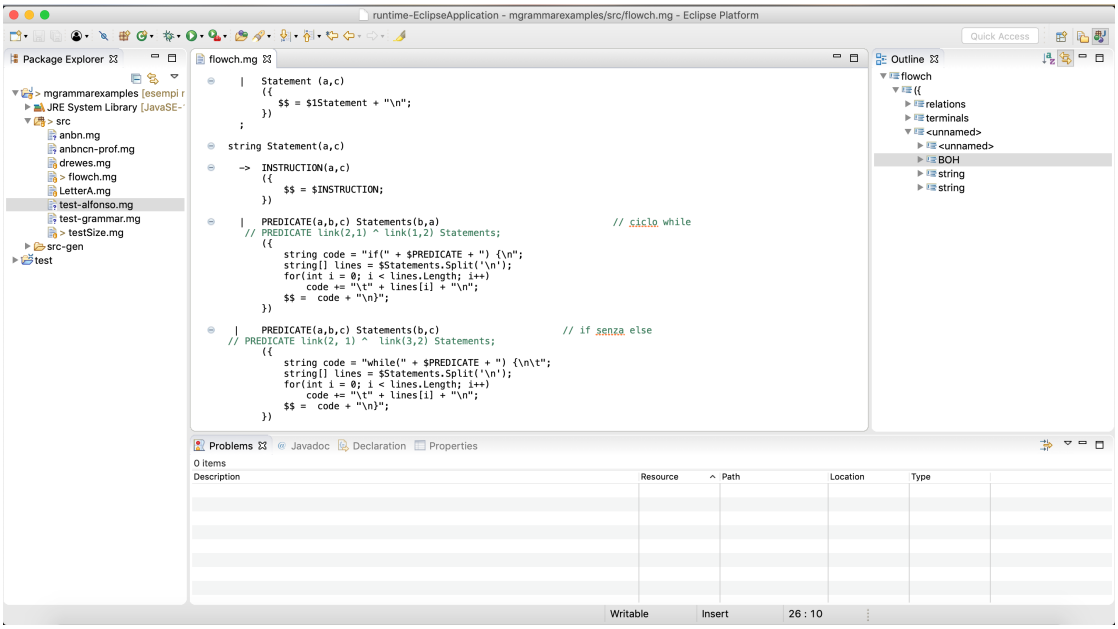

Figure 11: An image of the IDE developed

Xtext is a framework developed by the *Ecplise foundation* in order to define custom programming languages or *domain specific languages*, [11]. With this framework it is possible to specify the grammar of the language with the description of the single elements that compose the languages.

The necessary steps to implement MG with Xtext are described below:

- Define some configuration rules (such as the file extension, the encoding) and which plugins are used (for example the code generator);
- Specify the grammar of the language in the Xtext format;
- Define *validation* methods: they verify if the rules defined by the developer are valid;
- Define the rules to transform the parse tree representation (generated automatically by Xtext framework) in the following intermediate form: *MG data structure*.

The values of the configuration rules for the MG are reported in the following list:

- File extension: **.mg**
- Encoding: **UTF-8**

- Line Delimeter: **\n**
- Plugins: *codegenerator*, *validator*

In the following block, it is possible to see the configuration file in *mwe2* format, specific for Xtext.

```
Workflow {
    component = XtextGenerator {
        configuration = {
                project = StandardProjectConfig {
                        baseName = "it.unisa.di.cluelab.mglr"
                        rootPath = rootPath
                        runtimeTest = {
                                enabled = true
                        }
                        eclipsePlugin = {
                                enabled = true
                        }
                        eclipsePluginTest = {
                                enabled = true
                        }
                        web = {
                                enabled = true
                        }
                        createEclipseMetaData = true
                }
                code = {
                        encoding = "UTF-8"
                        lineDelimiter = "\n"
                        fileHeader = ""

                }
        }
        language = StandardLanguage {
                name = "it.unisa.di.cluelab.mglr.MGrammar"
                fileExtensions = "mg"
                generator = {
                        generateXtendMain = true
                }
                serializer = {
                        generateStub = false
                }
                validator = {
                        generateDeprecationValidation = true
                }
                junitSupport = {
                        junitVersion = "5"
                }
        }
    }
}
```

4.1.1 MG implementation through Xtext

Writing a grammar with Xtext isn't hard because the framework is based on the *EBNF* format and then has many simplifications that help the developer in his work, [12]. For example, in the following block, there is a grammar that generates all the strings that *can have* an exclamation point before the word. With the *CFG* formalism, for instance, there are two necessary

productions: one that generates a string with the exclamation point before the words, the other without the exclamation point.

S ⟶ "!" Words | Words

With Xtext it is possible to write only one rule because the framework allows the developer to specify that a part of the production is optional.

S ⟶ "!"? Words

Before presenting the implementation of *Multidimensional Grammar* in detail, the following list defines the *terminals* of the grammar.

```
terminal ARROW:
        '->';
terminal CSCODE:
        '({'->'})';
terminal FLOAT:
        INT '.' INT;
```

The first production describing MG in Xtext format is related to the *non-terminal* **Model**. The *non-terminal* Model is the start *non-terminal* for each grammar written in the Xtext format. In our implementation, Model coincides with the *non-terminal mGrammar*, as it is possible to see in (4):

$$Model : mGrammar; \tag{4}$$

It is very easy to understand this rule. Model is *non-terminal* and represents the *head* of the production. *mGrammar* represents the body of the production. Xtext framework automatically creates the syntactic tree by adding the variables before the production predicates. For this reason, it is possible to write production (4) in the following way:

$$Model : grammar = mGrammar; \tag{5}$$

The variable *grammar* doesn't have a role in the grammar specification, but it's used to visit the parse tree of the language during the parsing phase. The next block shows the body of the production *mGrammar*.

```
mGrammar:
        cscode=CSCODE
        relations=RelationSect
        terminals =TerminalSect
        rules=RuleSect;
```

RelationSect, *TerminalSect* and *RuleSect* are already known elements. They represent the section where the developer can declare respectively *Relations*, *Terminals*, *Joints*, *Tie-Points* and *Productions*. The following block shows the declaration of the *RelationSect* variable. In this declaration it is possible to see another simplification in the production writing:

```
RelationSect:
        name = 'relations' '{'
            list+=Relation*
        '}';
```

The first observable thing is *Relation**. This notation recalls the *regex*. Exactly as we would expect, the rule *Relation** indicates that there are zero or more instances of Relation. Because *Relation** generates a list of relations, the variable that contains the Relations should add the elements. For this reason, there is *list+=*:

```
Relation:
        name = ID
        '(' syntatticAttrib=Idlist ')'
        ('[' threshold=Threshold ']')? ';'
        ;

Idlist:
        list+=ID
        (',' list+=ID)*;
```

More similar than *RelationSect* is the section where the *terminals* are declared. The *TerminalSect* is specified below:

```
TerminalSect:
         name = 'terminals' '{'
         list+=Terminal*
         '}';
```

Each *terminal* has many properties that are already described in Sec. 3.2. The following block describes the rules written in Xtext for the *Terminal* element and related variables: *TerminalOptions*, *Predicate*, *Value*, *NVisit*, *Targ*.

```
Terminal:
        semanticType = TypeId?
        start='@'?
        predicate = Predicate
        options=TerminalOptions?
        ('=' semanticValue=Value)? ';';

TerminalOptions:
        '['
            ((nvisit=Nvisit (',' threshold=Threshold)?)
            |
            (threshold=Threshold (',' nvisit=Nvisit)?))?
        ']';

Value:
        (       valueInt=INT
            |   valueString=STRING
            |   valueFloat=FLOAT
            |   valueCs=CSCODE
        );

Predicate:
        name = ID
            '(' (syntatticAttributeTypes+=Targ
            (',' syntatticAttributeTypes+=Targ)*)?
        ')';

Targ:
        nameTarg=ID;

Nvisit:
        '#visit=' (value=INT | isNoLimit='NOLIMIT');
```

The last part of the language described is *RuleSect*. In this rule, as we described in section 3.2, there are the declarations *Joints*, *Tie-points* and *Non-Terminals* with the related *Production* rules.

```
RuleSect:
                'rules' '{'
                        decs +=Decl*
                        (prods+=Production+
                            |
                        prods+=Production*
                        prods+=InitProduction
                        prods+=Production*
                        )
                '}';
```

The variable *Decl* is related to the definition of *Joints* and *Tie-Points*. *Production* and *InitProduction*, instead, are related to the definition of the Production rules. The following blocks describe the rule in Xtext format related to the Joints and Tie-Points declarations.

```
Decl:
        (
        listJoint+=JointPointDecl
        |
        listTiePoints+=TiePointDecl)
;
```

The following block, in particular, is related to the Joints definitions.

```
JointPointDecl:
        jType=JointPointType
        op=Op?
        jointName+=JointName* ';';

JointName:
        jointName=ID |
        {JointName} (',' jointNameOp=Op? jointName=ID)
;
JointPointType:
        relationName=ID '.' attributeName=ID;
```

The following block, in particular, is related to the Tie-Points definitions.

```
TiePointDecl:
        tType=TypeId
        (                decl +=TiePointSingleDecl        )
        ( ','    decl +=TiePointSingleDecl )*
                 ';';

TiePointSingleDecl:
                tieName=ID       expr=ExpressionDefinition? ;
```

Each Tie-Point can have a function (a set expression or another type of expression written in C# language). The following rules describe in Xtext format the definition of these expressions.

```
ExpressionDefinition:
        '(' attribute=Idlist ')'
        '='
        expression=SetExpr?
        csCode=CSCODE?
;

SetExpr:
        expr1=BaseSetExpr
        otherExpr+=OtherPart* ;

OtherPart:
        op=('union' | 'intersect' | '-')
        expr=BaseSetExpr
;

BaseSetExpr:
        paramName=ID
        | '(' SetExpr ')' ;
```

For the Production rules are necessary two variables to distinguish the start *non-terminal* (*InitialProduction*) from the other productions (*Production*).

```
InitProduction:
        initProductiontype=TypeId?
        isStart='@'
        productionName=ID
        '(' attachPoints=Points? ')'
        rulesPredicate+=LPredicate*
        ARROW
        rightHandExpression+=RightHandSide
        ('|' rightHandExpression+=RightHandSide)* ';';

Production:
        productionType=TypeId?
        productionName=ID
        ('(' attachPoints=Points? ')')?
        rulesPredicate+=LPredicate*
        ARROW
        rightHandExpression+=RightHandSide
        ('|' rightHandExpression+=RightHandSide)* ';';
```

The body of the productions represented by the *non-terminal Production* and *InitProduction* are very similar: the only difference is the presence of the symbol, with the characteristic to set the rule as *start non-terminal* of the language. Also in this notation is possible to see that productions have two parts separated by the *ARROW terminal*. On the left side, there is the semantic type (specified by the *TypeId* variable), the name of the *non-terminal* (ID), the list of points and the predicates already described in the previous section.

```
TypeId:
        ID;
Points:
        listPoints+=Point (',' listPoints+=Point)*
        |
            (listPoints+=Point ',')*
            '@' listPoints+=Point
            (',' listPoints+=Point)*;

Point:
        pointName=ID ('(' attrList+=Idlist ')')?;

LPredicate:
        predicateName=ID ('(' predicateArg=Arglist ')')?;
```

The right side of the production, instead, is very similar to the classic grammar formalism. The rule *RightHandSide* describes the body of the production with a list of predicates that could have some parameters as joints or tie-points to specify the relationships between them.

```
RightHandSide:
         predicate+=RPredicate*
         tauRules=TauRules?
         ('[' threshold=Threshold ']')?
         cscode=CSCODE?;


RPredicate:
        predicateName=ID
        ('(' predicateArg=Arglist ')')?;
```

About the argument list, modelled by the *ArgList* rule, the following block describes the productions involved in their definition.

```
4  package it.unisa.di.cluelab.mglr.validation
5
6  import it.unisa.di.cluelab.mglr.mGrammar.MGrammarPackage
5
6  /**
7   * This class contains custom validation rules.
8   *
9   * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
0   */
1  class MGrammarValidator extends AbstractMGrammarValidator {
2
3      public static val INVALID_NAME = 'invalidName'
4
```

Figure 12: Class for custom validation methods

```
Arglist:
        args+=Arg
        (',' args+=Arg)*;

Arg:
        args+=ArgDefinition
        (':' args+=ArgDefinition)*;

ArgDefinition:
        (
                op=Op?
                argName=ArgName | any='!'
        )       ;

ArgName:
        ID | ID '.' ID;
```

4.1.2 Validation methods

With Xtext is possible to specify some validation functions. These functions are executed while the developers write their MG, to help them by showing errors, warnings and suggestions. A validation function must be written in the *mglr.validation* package, where there is a class with a name that finishes with *Validator*. For example, the class name of our implementation is *MGrammarValidator* (see Figure 12).

The main validation methods defined for our *MG Editor* are the following:

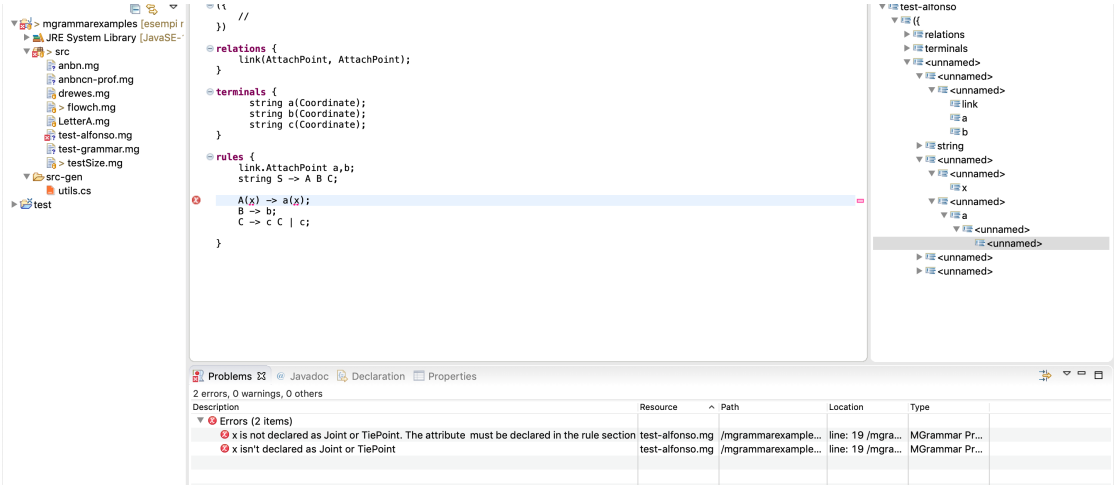

Figure 13: The editor shows error position and details

- A method that checks the presence of the cyclic links;
- Methods that check if the elements used are declared;
- A method that verifies if the predicates added on the left side are of size one;

To write a validation function it is necessary to add a method in the class already described. In order to describe to the framework that the method added is a validation method, it is necessary to add the *@Check* annotation before the method signature.

```
@Check
def checkPointsDeclaration(ArgDefinition arg) {
...
}
```

In the body of the method, after the check of the property, it is possible to send messages to the language developer. This messages can be of three types:

- Error
- Warning
- Info

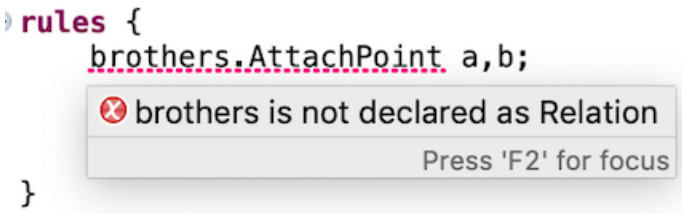

Figure 14: Example of validation in action

The parameters of these functions can set the position of the error. For example, in the following block, there is a validation method that checks if the relation used is declared in the respective section or not.

```
1     @Check
2   def checkRelationDeclaration(JointPointDecl decl) {
3         var relations = retrieveRelationsDeclaration(decl);
4         if(relations.contains(decl.JType.relationName))
5                   return;
6         error(decl.JType.relationName +
        " is not declared as Relation ",
            MGrammarPackage.Literals.JOINT_POINT_DECL__JTYPE,
          0);
7 }
```

The method *checkRelationDeclaration* has one parameter and its type is *JointPointDecl*. The type coincides with the name of the rule written in Xtext (and described a few pages ago). In this way, the framework executes this method for each Joint declaration. At row 4 it is possible to read the check. From the variable *decl* it is possible to navigate the parsing tree without other implementation. *jType* is the name of the variable associated to the rule *JointPointType* that represents the type of the relationship.

At line 6 is possible to see the *error* function that shows the error message if the relation used isn't declared in the right section. Figure 14 shows an error of this type.

Another interesting validation method is the function that checks if the predicates used in the left side of the production are or not of size one.

```
@Check
def checkIfTerminalIsSizeOne (Production p) {

    val\textit{terminals} = retrieveTerminalsDeclaration(p);
    var nonTerminals = retrieveNonTerminalsDeclaration(p);
    var nonTerminalsObject = retrieveNonTerminalsDeclared(p);
    var size = 0;

    for(var i = 0; i < p.rulesPredicate.size(); i++) {
        val predicateName =
            p.rulesPredicate.get(i).predicateName;
        size = 0;
        if(!terminals.contains(predicateName)){
            if(nonTerminals.contains(predicateName)) {
                var ntt = nonTerminalsObject.findFirst[nt | {
                if(nt instanceof Production) {
                  return nt.productionName.equals(predicateName);
                } else if (nt instanceof InitProduction) {
                  return nt.productionName.equals(predicateName);
                }
                  return false;
                }] as Production;
            var max = 0;
            for(var j = 0; j < ntt.rightHandExpression.size; j++) {
                var s =
                    computeSizeProduction(ntt.rightHandExpression.get(j),
                    new ArrayList<String>());
                if ( s > max) {
                max = s;
            }
        }
            size = max;
            if(size != 1) {
                error(predicateName +
                " isn't a size 1 non-terminal, but it has size " + size,
                MGrammarPackage.Literals.PRODUCTION__RULES_PREDICATE, i);
            }
        } else {
            error(predicateName + " isn't defined as Terminal or NonTerminal",
            MGrammarPackage.Literals.PRODUCTION__RULES_PREDICATE, i);
        }
    }
    }
}
```

The heart of this function is the method *computeSizeProduction* that computes, for the predicate in input, the max sentence size generated by its. From the rule in analysis, the function tries to navigate the syntactic tree with the adding of one unit every time the production generates a *terminal*. Obviously, if after the generation of the *terminal*, the function visit a *non-terminal* already read, there is a cycle and from this information, it is possible to deduce that the *non-terminal* has *infinite* size. The following block shows the source of the *computeSizeProduction* method wrote in the Xtend [13] language.

```
def computeSizeProduction(RightHandSide p, List<String> visited) {
    var size = 0;
    val\textit{terminals} = retrieveTerminalsDeclaration(p);
    var nonTerminalsObject = retrieveNonTerminalsDeclared(p);
    for(var i = 0; i < p.predicate.size; i++) {
        val predicateName = p.predicate.get(i).predicateName;
        if (terminals.contains(predicateName)){
            size++;
        } else {
            if(!visited.contains(predicateName)) {
                visited.add(predicateName);
                var ntt = nonTerminalsObject.findFirst[nt | {
                if(nt instanceof Production) {
                    return nt.productionName.equals(predicateName);
                } else if (nt instanceof InitProduction) {
                    return nt.productionName.equals(predicateName);
                }
                    return false;
                }] as Production;
                var max = 0;
                for(var j = 0; j < ntt.rightHandExpression.size; j++) {
                    var s =
                    computeSizeProduction(ntt.rightHandExpression.get(j), visited)
                                        as Integer;
                    if(s > max) {
                        max = s;
                    }
                }
                if(max == Integer.MAX_VALUE) {
                    size = max;
                } else {
                    size += max;
                }
            }
            else {
                return Integer.MAX_VALUE;
            }
        }
    }
    return size;
}
```

Type	Name	Description
String	cscode	A snippet of C# code to specify struct, elements and everything necessary to evaluate semantic rules.
List<Terminal>	terminals	A list of*terminals* that are declared in the *terminals* section.
List<Relation>	relations	A list of Relations that are declared in the relations section.
List<Joint>	joints	A list of Joints that are declared in the rules section.
List<TiePoint>	tiepoints	A list of Tie-Points that are declared in the rules section.
List<NonTerminal>	nonTerminals	A list of *non-terminal*s that are declared in the rules section as productions.
int	initialNonTerminal	The index of *non-terminal* that is declared as start *non-terminal*.

Table 1: Properties of the **NewGrammar** element

4.2 Multidimensional Grammar internal representation

In this section there is the description of the data structure defined for the implementation of *Multidimensional Grammar*. Figure 4.2 shows the UML class diagram with the description of entities and the relationships between each element defined for the language.

All the structure of the language is represented by the *NewGrammar* object. In compliance with the language definition (described in the last few pages), this object contains the definitions of *Terminals* and *Non-Terminals*; the definition of the Relationships of the language and the their usage as *Joint* during the production writing. The *NewGrammar* object contains also the definition of the *Tie-Points*, and a string that contains some instructions that will send to the *XPG parser*. Finally the NewGrammar object contains a field to define the start *non-terminal*. In the Tab. 1 it is possible to read the description in details of the *NewGrammar* class.

The list of the *terminals* declared in the language are in the *terminals* field, that represents a list of *terminal* object. The *terminal* has a name, a specific semantic type, a boolean field that specifies if the *terminal* is a *startterminal* of the language. There are other two params: *nvisit* and *threshold* that specifies information for the XPG parser. Finally it is possible to

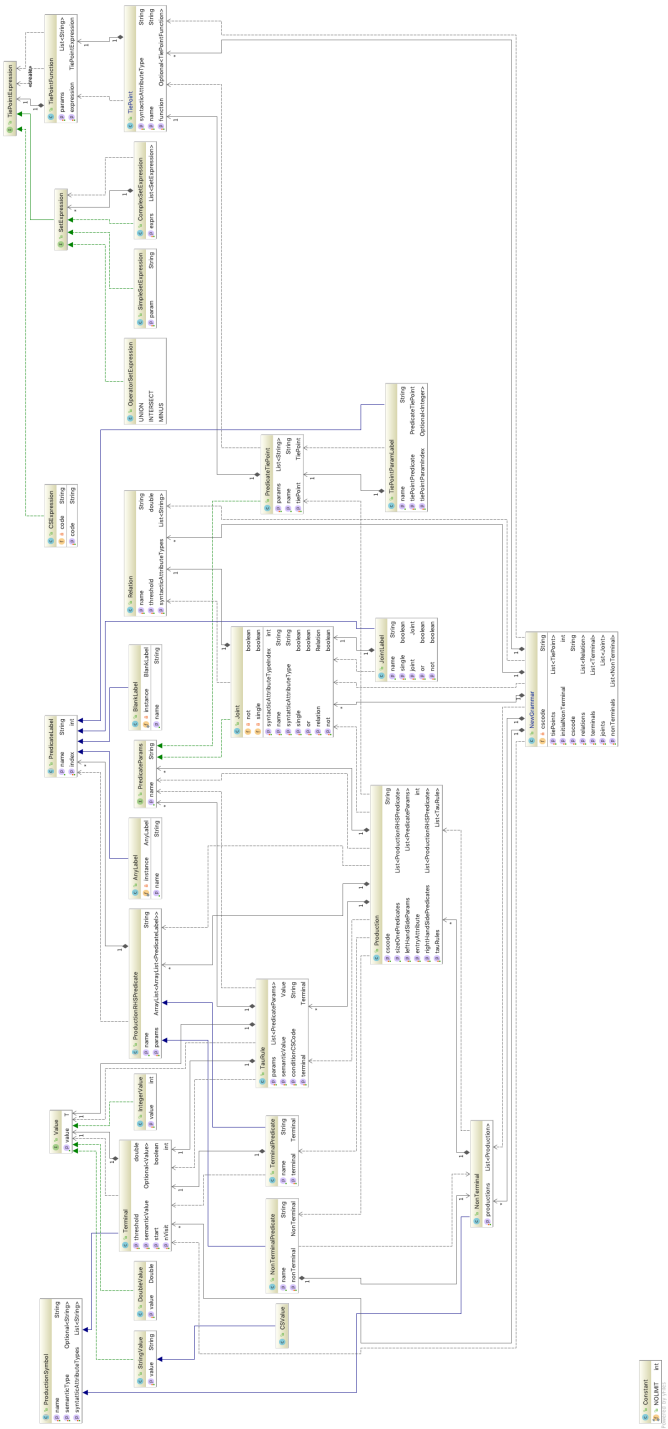

Figure 15: UML Diagram of MG Data Structure

Type	Name	Description
boolean	start	Specifies if the *terminal* is the start *terminal*.
List<String>	syntacticAttributeTypes	A list of syntactic types of the relations, such as AttachPoint or AttachLine.
int	nVisit	Specifies how many times the *terminal* could be visited by parser.
double	threshold	The threshold of the relation with a value in the range of 0 to 1.
Optional<Value>	semanticValue	Specifies the semantic type of the *terminal* (such as string, int, or other custom types that the developer has defined in cscode).
String	name	The name of the *terminal*.
Optional<String>	semanticType	The semantic type of the *terminal*.

Table 2: Properties of the **Terminal** element

Type	Name	Description
String	name	The name of the relation being defined.
List<String>	syntacticAttributeTypes	A list of syntactic types of the relations, such as AttachPoint or AttachLine.
double	threshold	The threshold of the relation with a value in the range of 0 to 1.

Table 3: Properties of the **Relation** element

define, for each *terminal*, a list of attributes that are used during the rule writing: the *syntactic attributes* (for example *AttachPoint*, *AttachLine* ...). The Tab. 2 specifies in detail the properties of *terminal* object.

The list of relationships declared in the language are in the *relations* field, that represents a list of Relation objects. The Relation element has a name and a list of syntactic attributes that will be used during the rule writing. There is another parameter, *threshold* that is used by XPG parser during its execution. The Tab. 3 describes in detail the properties of Relation element.

The Relation element is strictly related to the Joint concept. A Joint represents a link between two predicates and the link type is defined by the Relation. The Joint definitions are in the *joints* field, that represents a list of Joint objects.

A single Joint represents a variable available to use to link two or more predicates. Because

Type	Name	Description
Relation	relation	The Relation represented by the Joint.
int	syntacticAttributeTypeIndex	The index of the attribute type that specifies the Joint.
String	name	The name of the declared Joint.
boolean	not	Specifies if the declaration of the Joint is in negative form.
boolean	or	Specifies if the declaration of the Joint is in 'or' form.
boolean	single	Specifies if the declaration of the Joint is in 'single' form.

Semantic Duty: The properties not and single cannot both be true simultaneously.

Table 4: Properties of the **Joint** element

both Joint and Tie-Point are used as parameters of predicates, Joint implements an *Interface*: *PredicateParams*. In Tab. 4 the properties of the Joint element are defined.

Another type of *PredicateParams* is the *TiePoint* element. With this element is possible to specify the link between the head of the production and the related body, is possible to create expressions (a set expression, for example) passed as argument in the production writing. In the Tab. 5 the properties of the Tie-Point element are defined. The class definition of the *Tie-point expression* is described in the Tab. 6

The list of *Non-Terminal* declared in the language are in the *nonterminals* field, and represents the list of *NonTerminal* objects. A single *non-terminal* is characterized by the Name, the semantic type and the list of Production rules that specify how the string should be generated. In Tab. 7 the properties of the *non-terminal* element are defined.

For each *non-terminal* element is possible to define a list of rules that are called *Productions*. The production style in our implementation of *Multidimensional Grammar* is similar to the *context sensitive* grammar [14], so in the representation of the Production element there are two main elements:

- The left side of the arrow, where there are the *non-terminal* (with attribute if it is necessary) name and a list of left predicates;
- The right side of the arrow, where there is the classic rule definition (with some custom elements such as *TauRule*).

Type	Name	Description
String	syntacticAttributeType	The semantic type of the Tie-Point (such as String, int or other custom types that the developer has defined).
String	name	The name of the Tie-Point.
Optional <TiePointFunction>	function	A function related to the Tie-Point that will be interpreted by the parser.

Tie-Point element is related to another element in the data-structure:

- TiePointFunction

Table 5: Properties of the **Tie-Point** element

Type	Name	Description
List<String>	params	The params of the Function.
TiePointExpression	expression	The expression that will be evaluate by parser.

TiePointExpression is modelled by a Java Interface. This interface is implemented by two classes that represent the two type of expression:

- **CSExpression**: that has a string as a parameter, and it represents an expression wrote in C#. This expression isn't interpreted by the parser but only copied in the intermediate representation of the language.
- **SetExpression**: that represents a sequence of Set instructions (with set operation as union, intersect, difference). This kind of expressions are interpreted by the parser that converts the set expression in C# expression for the intermediate representation.

Table 6: Properties of the **TiePointFunction** element

Type	Name	Description
List<Production>	productions	The productions of the *non-terminal*.
String	name	The name of the *non-terminal*.
Optional<String>	semanticType	The semantic type of the *non-terminal*.

Non-Terminal element is related to another element in the data-structure:

- Production

Table 7: Properties of the **Non-Terminal** element

Type	Name	Description
List <PredicateParam>	leftHandSideParams	The arguments in the left-side of the production.
List <ProductionRHSPredicate>	rightHandSidePredicates	The List of predicates that compose the rule of the production.
List <ProductionPredicate>	leftHandSidePredicates	The List of predicates that compose the left side of the production.
List <TauRule>	tauRules	The List of TauRule.
String	csCode	A snippet of C# code to specify instruction to the positional parser.

Table 8: Properties of the **Production** element

Another important field of the Production element is the *leftHandSidePredicates*, that represents the predicates on the left side of the productions. In Tab. 8 the properties of the Production element are defined.

A Production rule is defined as a sequence of *terminal* and *non-terminal*, connected with the Joints. To implement this definition we have defined a single production rule as a list of *Production Predicate*. In our implementation, the *ProductionRHSPredicate* is realized as an **abstract class** implemented by two specific classes:

- *terminal* Predicate
- *non-terminal* Predicate

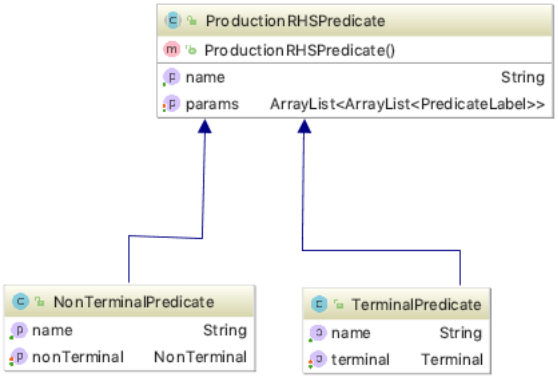

Figure 16: UML Diagram of ProductionRHSPredicate

Type	Name	Description
NonTerminal	nonTerminal	The *non-terminal* used in the production rule.
List<List<PredicateLabel»	params	The params of the predicate.

Table 9: Properties of the **NonTerminalPredicate** element

In Figure 16 is possible to see the diagram of the class *ProductionRHSPredicate* and its specializations. The tables Tab. 9 and Tab. 10 describe the properties for the specialization classes: *NonTerminalPredicate* and *TerminalPredicate*

Type	Name	Description
Terminal	terminal	The *terminal* used in the production rule.
List<List<PredicateLabel»	params	The params of the predicate.

Table 10: Properties of the **TerminalPredicate** element

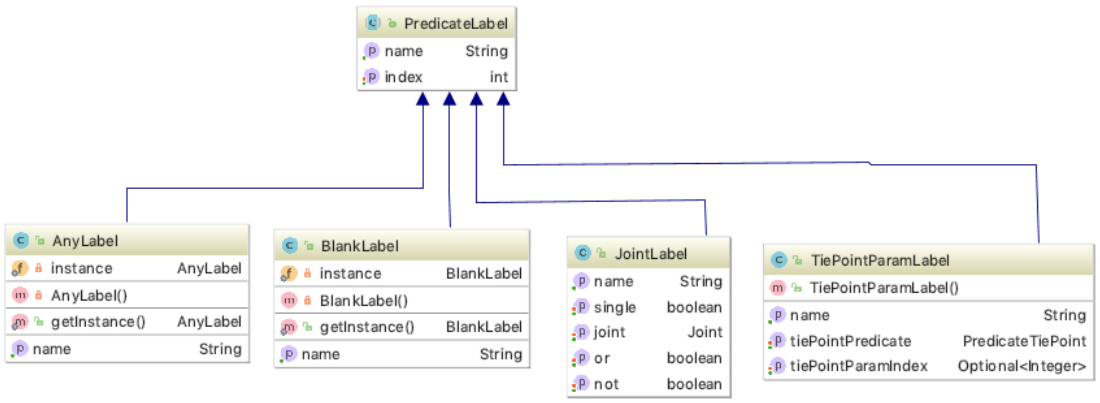

Figure 17: UML Diagram of PredicateLabel

Type	Name	Description
Joint	joint	The Joint linked to the label.
boolean	not	This value is true if the usage of the label is in negative way.
boolean	or	This value is true if the usage of the label is in or way.
boolean	single	This value is true if the usage of the label is in single way.

Table 11: Properties of the **JointLabel** element

The links between two or more predicates are managed by the Joint or Tie-Point elements. After the declaration of this elements is possible to use the variables instanced as parameters of the predicate. If two predicates have the same attribute label, they have a link. These labels can be of four types:

- **JointLabel**: a label which refers to a Joint.
- **TiePointLabel**: a label which refers to a TiePoint.
- **BlankLabel**: a label that doesnâĂŹt have a name.
- **AnyLabel**: a label for refers to other predicates without attributes with the same name.

In Figure 17 is possible to see the UML diagram of the *PredicateLabel* **abstract class** and their specialization classes.

A single Production element can have one or more TauRules. A single TauRule is a set of instructions for the positional parser. These instructions, that are written in C#, can mapping

Type	Name	Description
TiePoint	tiePoint	The TiePoint linked to the label.
int	tiePointParamIndex	The index of the param in tiePoint.

Table 12: Properties of the **TiePointLabel** element

Type	Name	Description
Terminal	terminal	The *terminal* linked to the TauRule.
List<PredicateParams>	params	The params of the TauRule.
String	conditionCsCode	The instructions wrote in C# language.
Value	semanticValue	The semantic value of the TauRule.

Where Value is an interface that are implemented by these for value types:

- IntegerValue: that represents a numeric integer value.
- StringValue: that represents a string value.
- DoubleValue: that represents a numeric decimal value.
- CSValue: that represents a C# code value.

Table 13: Properties of the **TauRule** element

instructions between attach points, can create other element or other label to add to input. In Tab. 4.2 the properties of the TauRule element are defined.

4.3 Parsing algorithms

This section describes the algorithms and methods written in order to generate and populate the data structure described in the previous section 4.2. In the section 4.1.2 was presented the framework Xtext. This framework generates the syntactic tree automatically and the other compilation phases.

Xtext framework uses *Xtend* language to write the methods that will be executed by the *Eclipse instance* when the file will be saved. "*Xtend* is a flexible and expressive dialect of Java, which compiles into readable Java 8 compatible source code"[13]. The first step necessary, in order to parse the source file written by the developer, is create a *generator* class that extends the *AbstractGenerator*. In this class there is a method that is necessary to override:

```
void doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext context)
```

The variable *resource* contains the syntactic tree of the language written. To access to this tree:

```
resource.contents.get(0)
```

Because it isn't sufficient the only syntactic tree, there are many methods written to create and populate the MG data structure. The main method is the *parseGrammar* method, and it is presented in the following block.

```
def parseGrammar(ModelImpl model) {
    var grammar = model.newGrammar;
    g.cscode = grammar.cscode
    g.terminals = retrieveTerminalsFromGrammar(grammar.terminals);
    g.relations = retrieveRelationsFromGrammar(grammar.relations);
    retrieveRulesFromGrammar(grammar.rules as RuleSectImpl);
    checkNonTerminalPredicateDeclaration();
    return g;
}
```

After the creation of the grammar object, the parser tries to find the declaration of *terminal* and relations. The methods are unsophisticated and presented below.

```
def retrieveRelationsFromGrammar(RelationSect rs) {
    var relations = new ArrayList<it.unisa.di.cluelab.datastructure.Relation>();
    for (var i = 0; i < rs.list.size; i++) {
        var r = rs.list.get(i) as Relation;
        var rImpl = new it.unisa.di.cluelab.datastructure.Relation();
        rImpl.name = r.name;
        rImpl.syntacticAttributeTypes = r.syntatticAttrib.list;
        relations.add(rImpl);
    }
    return relations;
}
```

The function *retrieveTerminalsFromGrammar* navigates the syntactic tree and creates the *Terminal* object with the value read from the parsing tree.

```
def retrieveTerminalsFromGrammar(TerminalSect ts) {
    var terms = new ArrayList<Terminal>();
    for(var i = 0; i < ts.list.size; i++) {
        var t = ts.list.get(i) as Terminal;
        var tImpl = new Terminal();
        tImpl.syntatticAttributeTypes = new ArrayList<String>();
        tImpl.name = t.predicate.name;
        tImpl.start = t.start !== null && t.start.equals("@");
        if(!tImpl.semanticType.isPresent && t.semanticType!==null) {
            tImpl.semanticType = Optional.of(t.semanticType);
        }
        if(t.semanticValue === null) {
            tImpl.semanticValue = Optional.empty();
        } else {
        if(t.semanticValue.valueString !== null) {
            tImpl.semanticValue = Optional.of(new
            StringValue(t.semanticValue.valueString));
        } else if (t.semanticValue.valueFloat !== null) {
            tImpl.semanticValue = Optional.of(new
            DoubleValue(Double.parseDouble(t.semanticValue.valueFloat)));
        } else if (t.semanticValue.valueCs !== null) {
            tImpl.semanticValue = Optional.of(new
            CSValue(t.semanticValue.valueCs));
        } else  {
            tImpl.semanticValue = Optional.of(new
            IntegerValue(t.semanticValue.valueInt));
        }
        }
        if(t.options !== null) {

        if(t.options.nvisit !== null &&
            (t.options.nvisit.isNoLimit === null ||
            t.options.nvisit.isNoLimit.isEmpty))
        {
            tImpl.setnVisit = t.options.nvisit.value;
        } else {
            tImpl.setnVisit = Constant.NOLIMIT;
        }
        if(t.options.threshold !== null &&
            t.options.threshold.value !== null) {
        tImpl.threshold =
            Float.parseFloat(t.options.threshold.value);
    }
}


var listAttribute = t.predicate.syntatticAttributeTypes as List<Targ>;
for(var j = 0; j < listAttribute.size; j++) {
tImpl.syntatticAttributeTypes.add(listAttribute.get(j).nameTarg);
}

terms.add(tImpl);
}
return terms;
}
```

The methods described above are very simple, and similarly, it is possible to parse the tree in order to obtain the Joint and Tie-points declarations. For these reasons their treatment will be omitted. Related to the concept of Tie-Point there is the concept of Expression. The Tie-Point can have a C# expression or a set expression. The parsing of these expressions is described in the following block with the recursive method *parseExpression*.

```
def parseExpression(SetExprImpl expression, ComplexSetExpression list) {

    var right = expression.expr1;
    if (right.paramName !== null) {
        var simple = new SimpleSetExpression();
        simple.param = right.paramName;
        list.exprs.add(simple);
    } else {
        var complex = new ComplexSetExpression();
        parseExpression(right as SetExprImpl,complex);
        list.exprs.add(complex);
    }
    for(var i = 0 ; i < expression.otherExpr.size; i++){
        var left = expression.otherExpr.get(i) as OtherPartImpl;
        if(left.op == "-")
            list.exprs.add(OperatorSetExpression.MINUS);
        if(left.op == "union")
            list.exprs.add(OperatorSetExpression.UNION);
        if(left.op == "intersect")
            list.exprs.add(OperatorSetExpression.INTERSECT);
        if(left.expr instanceof SetExprImpl) {
            var complex = new ComplexSetExpression();
            parseExpression(left.expr as SetExprImpl,complex);
            list.exprs.add(complex);
        } else {
            var simple = new SimpleSetExpression();
            simple.param = left.expr.paramName;
            list.exprs.add(simple);
        }
    }
}
```

After the parsing of *Terminals*, *Relations*, *Joints* and *TiePoints* elements, with particular attention about the expressions related to the Tie-Point, the generator implemented parses the production rules. For each production rule it is necessary to retrieve the following elements:

- The semantic type
- The name of the variable
- A list of left predicates
- The body rule with a list of right predicates
- The Tau-Rule definition

The first part of the method is dedicated to the left part analysis. In this phases, there are some semantic checks about the usage of Joint and Tie-Point or *terminal* without the relative declaration, and the presence of cycle in the linking between predicates.

```
var p = r.prods.get(i) as Production;

var hasCycle = checkCyclicLinkInAProduction(p);
var mappingCycleVars = new HashMap<String, VariableCounter>();

for(var j = 0; j < hasCycle.size(); j++)
{
        mappingCycleVars.put(hasCycle.get(j), new VariableCounter(hasCycle.get(j)));
}
nt.name = p.productionName;
for(NonTerminal nnt : nonTerminals) {
        if(nnt.name.equals(nt.name)) {
                nt = nnt;
                toInsert = false;
        }
}

if(this.g.findNonTerminal(nt.name)!=null) {
        nt = this.g.findNonTerminal(nt.name);
}
if(p.productionType!=null){
        nt.semanticType = Optional.of(p.productionType);
} else {
        nt.semanticType = Optional.empty();
}
```

Before parsing the production rule, the method parses the left predicates added before the arrow. The parsing checks two fundamental requirements:

- The predicate must be of size one
- The predicate must be declared on the right side. For this check, there is already a validation method that prevents the compiler phase if this issue is noticed.

```
if(p.rulesPredicate !== null  && p.rulesPredicate.length > 0 ) {
    for(var prs = 0 ; prs < p.rulesPredicate.size; prs++) {
        var k = p.rulesPredicate.get(prs) as LPredicate;
        val params = new ArrayList<ArrayList<PredicateLabel>>();
        if(k.predicateArg !== null)
        k.predicateArg.args
        .map[ pred | pred.args.get(0).argName]
        .forEach[pred | {
        if(g.findJoint(pred) !== null) {
                var jl =  new JointLabel();
                jl.joint = g.findJoint(pred);
                jl.index = 1;
                var al = new ArrayList<PredicateLabel>();
                al.add(jl);
                params.add(al);
        } else if(g.findTerminal(pred) !== null) {
                var tppl = new TiePointParamLabel();
                tppl.tiePointPredicate = new PredicateTiePoint();
                tppl.tiePointPredicate.tiePoint = g.findTiePoint(pred);
                tppl.index = 1;
                var al = new ArrayList<PredicateLabel>();
                al.add(tppl);
                params.add(al);
        }
        }];

        if(g.findTerminal(k.predicateName) !== null){
            var tpp = new TerminalPredicate();
            tpp.terminal = g.findTerminal(k.predicateName);
            tpp.params = params;
            pNew.sizeOnePredicates.add(tpp);
        } else {
            var ntpp = new NonTerminalPredicate();
            if(g.findNonTerminal(k.predicateName) !== null)      {
                ntpp.nonTerminal = g.findNonTerminal(k.predicateName);
            }
            else {
                ntpp.nonTerminal = new NonTerminal();
                ntpp.nonTerminal.name = k.predicateName;
            }
            ntpp.params = params;
            pNew.sizeOnePredicates.add(ntpp);
        }
    }
}
```

After the parsing of the left predicate, the method continues the execution with the parsing of the point used inside the name of the *non-terminal*. Each point used in the production is checked in the grammar for the declaration and linked to the rule.

```
if(p.attachPoints !== null) {
var points = p.attachPoints.listPoints;
for(var k = 0; k < points.size(); k++) {
        var ptp = new PredicateTiePoint();
        var tp = new TiePoint();
        tp.name = points.get(k).pointName;
        if(g.findTiePoint(tp.name)!== null)
        {
                tp.syntacticAttributeType =
                    g.findTiePoint(tp.name).syntacticAttributeType;
                nt.syntatticAttributeTypes.add(tp.syntacticAttributeType);
                if(g.findTiePoint(tp.name).function.isPresent) {
                        tp.function =
                            Optional.of(g.findTiePoint(tp.name).function.get.clone);
                } else {
                        tp.function = Optional.empty();
                }
        } else if (g.findJoint(tp.name)!== null) {
                tp.syntacticAttributeType = g.findJoint(tp.name).relation.name;
                nt.syntatticAttributeTypes
                    .add(g.findJoint(tp.name).syntatticAttributeType);
        } else {
                throw new IllegalArgumentException
                    ("TiePoint or Joint not declared " + tp.name);
        }
        /**
         * Replace name internal with name declared in left
         */
        if(points.get(k).attrList.size > 0) {
                var newParams = new ArrayList<String>();
                for(String attrName : points.get(k).attrList.get(0).list ) {
                        newParams.add(attrName);
                }
                tp.function.get.params =  newParams;
        }
        ptp.tiePoint =  tp;
        pNew.leftHandSideParams.add(ptp);
        //Params
        //Check se il tiepoint Ãl dichiarato
}
}
```

The parsing of the production body starts with the check of the *terminal* and *non-terminal* declarations. At the moment of the parsing there are all declarations of the *terminal* but not for the *non-terminal*, because their declaration can exist after the rule analyzed. In a second moment, the algorithms will check the *non-terminal* declarations.

```
if(g.findTerminal(predicateName) !== null){
        var tpp = new TerminalPredicate();
        tpp.terminal = g.findTerminal(predicateName);
        tpp.params = params;
        pNew.rightHandSidePredicates.add(tpp);
} else {
        var ntpp = new NonTerminalPredicat  e();
        ntpp.nonTerminal = new NonTerminal();
        ntpp.nonTerminal.name = predicateName;
        ntpp.params = params;
        pNew.rightHandSidePredicates.add(ntpp);
        this.nonTerminalDoCheked.add(ntpp);
}
```

For each predicate, the algorithm generates the link between them. In this phase there is another check that the algorithm can do: in order to generate the link between the predicates

the parser doesn't consider cycle the link generates with the same variable. So, for example:

```
S -> PredicateA(a) PredicateB(a) PredicateC(a)
```

The body rule of the S variable can generate a cycle because the predicates are linked between them with the same relationships and the same variable. So, the parser doesn't consider this cycle and transform this production in the equivalent following production:

```
S -> PredicateA(a1) PredicateB(a1:a2) PredicateC(a2)
```

This behaviour is possible with the property *index* of the PredicateLabel as it is possible to see in Figure 17.

```
if (g.findTiePoint(predicateArg.args.get(m).argName) !== null) {
        var pointName = predicateArg.args.get(m).argName;
        if(mappingCycleVars.get(pointName) !== null) {
                var cycleCounter = mappingCycleVars.get(pointName);
                cycleCounter.incrementUsage;
                if(cycleCounter.getnUsages == 1) {
                        var tppl = new TiePointParamLabel();
                        tppl.tiePointPredicate = new PredicateTiePoint();
                        tppl.tiePointPredicate.tiePoint =
                            g.findTiePoint(predicateArg.args.get(m).argName);
                        tppl.index = cycleCounter.getnUsages;
                        listLabel.add(tppl);
                } else if(cycleCounter.getnUsages == 2)     {
                        var tppl = new TiePointParamLabel();
                        tppl.tiePointPredicate = new PredicateTiePoint();
                        tppl.tiePointPredicate.tiePoint =
                            g.findTiePoint(predicateArg.args.get(m).argName);
                        tppl.index = cycleCounter.getnUsages -1;
                        listLabel.add(tppl);

                        var tpplCompt = new TiePointParamLabel();
                        tpplCompt.tiePointPredicate = new PredicateTiePoint();
                        tpplCompt.tiePointPredicate.tiePoint =
                            g.findTiePoint(predicateArg.args.get(m).argName);
                        tpplCompt.index = cycleCounter.getnUsages;
                        listLabel.add(tppl);

                } else if (cycleCounter.getnUsages > 2) {
                        var tppl = new TiePointParamLabel();
                        tppl.tiePointPredicate = new PredicateTiePoint();
                        tppl.tiePointPredicate.tiePoint =
                            g.findTiePoint(predicateArg.args.get(m).argName);
                        tppl.index = cycleCounter.getnUsages -1;
                        listLabel.add(tppl);

                        var tpplCompt = new TiePointParamLabel();
                        tpplCompt.tiePointPredicate = new PredicateTiePoint();
                        tpplCompt.tiePointPredicate.tiePoint =
                            g.findTiePoint(predicateArg.args.get(m).argName);
                        tpplCompt.index = cycleCounter.getnUsages;
                        listLabel.add(tppl);
                }
        } else {
                var tppl = new TiePointParamLabel();
                tppl.tiePointPredicate = new PredicateTiePoint();
                tppl.tiePointPredicate.tiePoint =
                g.findTiePoint(predicateArg.args.get(m).argName);
                listLabel.add(tppl);
        }
```

```
} else if (g.findJoint(predicateArg.args.get(m).argName) !== null) {
        var pointName = predicateArg.args.get(m).argName;

        if(mappingCycleVars.get(pointName) !== null) {
                var cycleCounter = mappingCycleVars.get(pointName);
                cycleCounter.incrementUsage;
                if(cycleCounter.getnUsages == 1) {
                        var jdl = new JointLabel();
                        jdl.joint = g.findJoint(predicateArg.args.get(m).argName);
                        if(predicateArg.args.get(m).op !== null) {
                                jdl.not = predicateArg.args.get(m).op.equals("!");
                        }
                        jdl.index = cycleCounter.getnUsages;
                        listLabel.add(jdl);
                } else if(cycleCounter.getnUsages == 2)     {
                        var jdl = new JointLabel();
                        jdl.joint = g.findJoint(predicateArg.args.get(m).argName);
                        jdl.index = cycleCounter.getnUsages -1;
                        if(predicateArg.args.get(m).op !== null) {
                                jdl.not = predicateArg.args.get(m).op.equals("!");
                        }
                        listLabel.add(jdl);

                        var jdlCmpt = new JointLabel();
                        jdlCmpt.joint = g.findJoint(predicateArg.args.get(m).argName);
                        jdlCmpt.index = cycleCounter.getnUsages
                        if(predicateArg.args.get(m).op !== null) {
                                jdlCmpt.not = predicateArg.args.get(m).op.equals("!");
                        }
                        listLabel.add(jdlCmpt);

                } else if (cycleCounter.getnUsages > 2) {

                        var jdl = new JointLabel();
                        jdl.index = cycleCounter.getnUsages -1
                        jdl.joint = g.findJoint(predicateArg.args.get(m).argName);
                        if(predicateArg.args.get(m).op !== null) {
                                jdl.not = predicateArg.args.get(m).op.equals("!");
                        }
                        listLabel.add(jdl);

                        var jdlCmpt = new JointLabel();
                        jdlCmpt.joint = g.findJoint(predicateArg.args.get(m).argName);
                        jdlCmpt.index = cycleCounter.getnUsages;
                        if(predicateArg.args.get(m).op !== null) {
                                jdlCmpt.not = predicateArg.args.get(m).op.equals("!");
                        }
                        listLabel.add(jdlCmpt);
        } else {
                var jdl = new JointLabel();
                jdl.joint = g.findJoint(predicateArg.args.get(m).argName);
                if(predicateArg.args.get(m).op !== null) {
                        jdl.not = predicateArg.args.get(m).op.equals("!");
                }
                jdl.index = 1;
                listLabel.add(jdl);

        }
} else {
        throw new IllegalArgumentException("TiePoint or Joint not declared " +
                predicateArg.args.get(m).argName);
}
```

The last element that is parsed by the algorithm is the *Taurule* element. The Tau-rule object is composed by a*terminal*, a semantic value, a list of params and some instruction wrote in C# language. The semantic value can be of four types:

- String
- Integer
- Double
- CSCode

The following block describes the Xtend code wrote in order to parser the TauRule of the production.

```
if(p.rightHandExpression.get(j).tauRules!==null) {
        for(var k = 0; k < p.rightHandExpression.get(j).tauRules.list.size; k++) {
                var newRule = new TauRule();
                var tRule = p.rightHandExpression.get(j).tauRules.list.get(k);
                newRule.terminal = g.findTerminal(tRule.tauRuleName);
                if(newRule.terminal === null) {
                throw new IllegalArgumentException
                ("The following Terminal it isn't declared " + tRule.tauRuleName);
                }
                newRule.conditionCSCode = tRule.cscode;
                if(tRule.tauRuleValueString !== null) {
                        newRule.semanticValue =
                            new StringValue(tRule.tauRuleValueString);
                } else if (tRule.tauRuleValueCsCode !== null) {
                        newRule.semanticValue =
                            new StringValue(tRule.tauRuleValueCsCode);
                } else if (tRule.tauRuleValueInt !== 0) {
                        newRule.semanticValue =
                            new IntegerValue(tRule.tauRuleValueInt);
                } else if (tRule.tauRuleValueFloat !== null) {
                        newRule.semanticValue =
                            new DoubleValue(tRule.tauRuleValueFloat);
                }
                newRule.params = new ArrayList<PredicateParams>();
                for(var l = 0; l < tRule.tauRuleAttr.listPoints.size; l++) {
                        var predicateArgName =
                            tRule.tauRuleAttr.listPoints.get(l).pointName;
                        if(g.findTiePoint(predicateArgName)!== null) {
                                var tppl = new PredicateTiePoint();
                                tppl.tiePoint = g.findTiePoint(predicateArgName);
                                newRule.params.add(tppl);
                        } else if (g.findJoint(predicateArgName)!== null) {
                                var jdl = g.findJoint(predicateArgName);
                                newRule.params.add(jdl);
                        } else {
                                throw new IllegalArgumentException
                                ("TiePoint or Joint not declared "+predicateArgName);
                        }
                }
                pNew.tauRules.add(newRule);
        }
}
```

5 From Multidimensional Grammars To eXtended Positional Grammars

The previous chapter describes the first part of the *IDE* developed. Following the flowchart shown in Figure 3, in this chapter, we describe the second part of the editor, composed by another *intermediate form generator* that populates the *XPG data structure*. The chapter starts with the description of the formalism used as an *intermediate form*, the *Extended Positional Grammar*. Next, the chapter continues with the description of the XPG data structure. Finally, there is a description of the algorithms executed by the *intermediate form generator* to populate the *XPG data structure*

5.1 PG and XPG Formal Definitions

This section describes two formalism related between them. The first one is *Positional Grammar*. The *Positional Grammar* is defined as a direct extension of Context Free Grammar where it is possible to not only have a string concatenations relationships but also more general relationship. Finally, there is the definition of the *Extended Positional Grammar* as an extension of *Positional Grammar*.

Formally, the definition of *Positional Grammar* is described as follows:

$$PG = (N, T, S, P, POS, PE) \tag{6}$$

where

- N is a finite nonempty set of nonterminals
- T is a finite nonempty set of *terminals*, with N > T = O
- S denotes the starting nonterminal
- P is a finite nonempty set of productions
- POS is a finite set of binary relation identifiers
- PE is a pictorial evaluator

In the *Positional Grammars* each *terminal* and *non-terminal* are graphical objects. The production writing, in *PG*, must consider the relationship between the *non-terminal*s and the related index. Each production has a form described below:

$$A \rightarrow x_1 \mathbf{R}_1 x_2 ... \mathbf{R}_{\text{m-1}} x_m, \tag{7}$$

Where A is the *non-terminal*, x_1 ... x_m are the grammar objects and, finally, R_1 ... $R_{\text{m-1}}$ describes the relationships between grammar objects.

An Extended Positional Grammar (XPG) is defined as a pair (G, PE), where PE is a positional evaluator, and G is a particular type of context-free string attributed grammar (N, T $\cup$ POS, S, P) where:

- N is a finite nonempty set of nonterminal vsymbols;
- T is a finite nonempty set of *terminal* vsymbols, with $N \cap T = \emptyset$;
- POS is a finite set of binary relation identifiers, with $POS \cap N = \emptyset$ and $POS \cap T = \emptyset$;
- $S \in N$ denotes the starting vsymbol;
- P is a finite nonempty set of productions having the format shown in (8) .

$$A \rightarrow x_1 \mathbf{R}_1 x_2 ... \mathbf{R}_{\text{m-1}} x_m, \tag{8}$$

Type	Name	Description
int	initialNonTerminal	The index of *non-terminal* that is declared as start *non-terminal*.
String	cscode	A snippet of C# code
List<Terminal>	terminals	A list of *terminals*
List<Relation>	relations	A list of Relations
List<NonTerminal>	nonTerminals	A list of *non-terminal*s
HashMap<TiePoint,String>	functions	A list of functions related to the TiePoint

Table 14: Properties of the **PositionalGrammar** element

Type	Name	Description
String	name	The name of the *terminal* or *non-terminal*.
Optional<String>	semanticType	The semantic type of the *terminal* or *non-terminal*.

Table 15: Properties of the **ProductionSymbol** element

5.2 Equivalent Data Structure and model for XPG

In this section, there is the description of the data structure defined for the implementation of *Extended Positional Grammar*. Figure 18 shows the UML class diagram with the description of entities and the relationships between each element defined for the language.

All the structure of the language is represented by the *PositionalGrammar* object. In respect of the language definition (described in the last few pages), this object contains the definitions of *terminals* and NonTerminals; the reference to the start *non-terminal*, and the definition of the relationships. Tab. 14 describes in detail the properties of the *PositionalGrammar* element.

Terminal and *non-terminal* are specifications of the abstract class *ProductionSymbol*. Figure 19 reports the UML schema of these classes.

The *terminal* element of the XPG is similar to the *terminal* Object in MG. It has an integer that specifies the main syntactic attribute of the *terminal*; a boolean value that specifies if the *terminal* is a start *terminal*; a name; the values for the options *nVisit*, *thresholds* and the syntactic attributes. Tab. 16 describes extensively the properties of the *Terminal* class.

The list of relationships declared in the language are in the *relations* field. The Relation object is similar to the Relation object in MG, but the XPG object can define the property *negateRelation*. The *negateRelation* field contains the inverse relation of that modelled. Tab. 17 shows the properties of the *Relation* object.

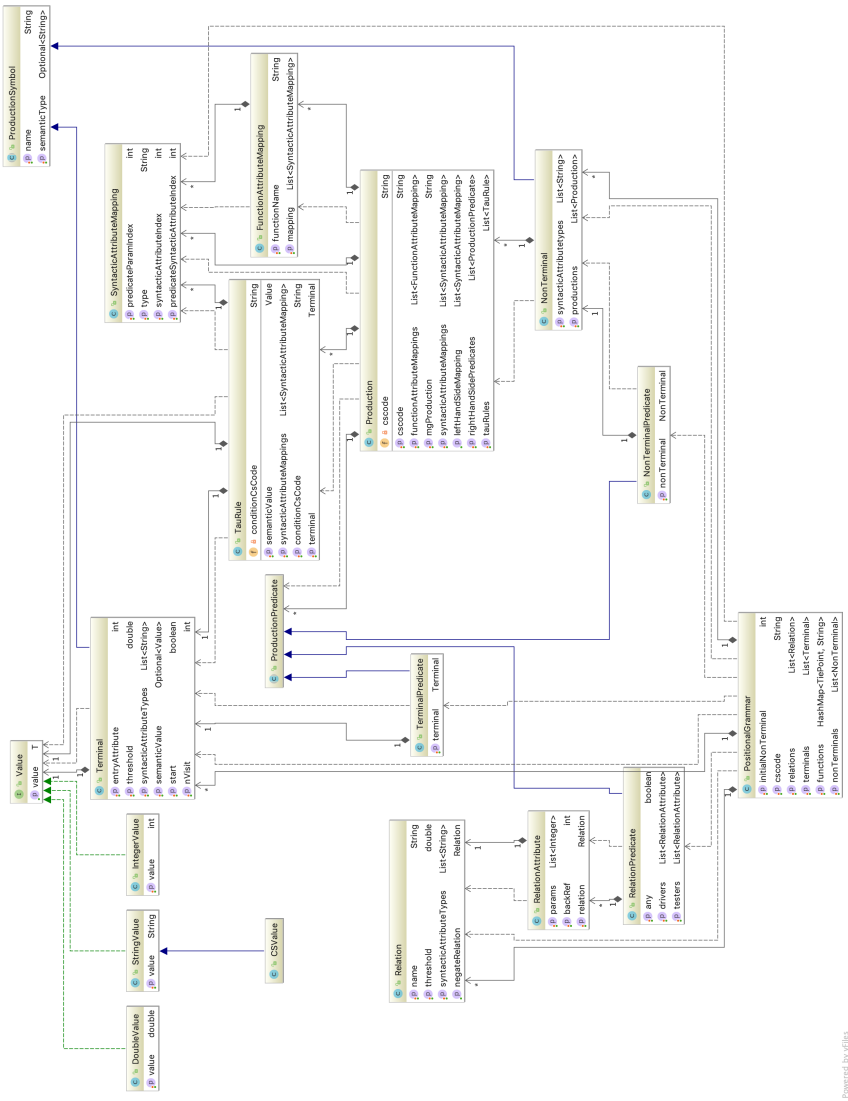

Figure 18: UML Diagram of XPG Data Structure

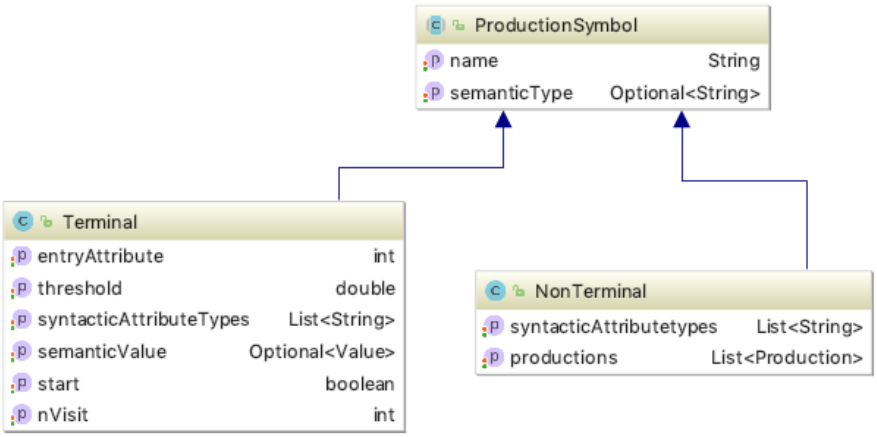

Figure 19: UML Diagram of Production Symbol

Type	**Name**	**Description**
int	entryAttribute	An integer that specify the first syntactic attribute.
List<String>	syntacticAttributeTypes	The list of syntactic attributes.
Optional<Value>	semanticValue	The semantic value of the *terminal*.
boolean	start	Specifies if the *terminal* is a start *terminal*.
double	threshold	The threshold of the *terminal*.
int	nVisit	Specifies how many timesthe *terminal* could be visited by XPG parser.

Table 16: Properties of the **Terminal** element

Type	**Name**	**Description**
String	name	The name of the Relation
double	threshold	The threshold of the Relation.
List<String>	syntacticAttributeTypes	The list of syntactic attribute types.
Relation	negateRelation	The respective negate relation.

Table 17: Properties of the **Relation** element

Type	Name	Description
List<String>	syntacticAttributeTypes	The list of syntactic attributes.
List<Production>	productions	The list of productions.

Table 18: Properties of the **Non-Terminal** element

The *non-terminal* object of the XPG formalism adds two other fields to its parent class, syntactic attributes and the list of productions. Tab. 18 shows the properties of this object.

A single *non-terminal* has a list of the Productions. The object that models the Production rule is described in Tab. 19. The main characteristics of this object are related to the syntactic mapping for the function, the attribute and the mapping between left and right side. Finally, there is the body of the production with a list of production predicates.

An important object related to the concept of Production is the *SyntacticAttributeMapping* object. This object models the mapping between the points used on the left side of production with the points (Joints or Tie-Points) used on the right side. Tab. 5.2 shows the properties of the object.

On the left side of the production, the *non-terminal* can have, as a parameter, a function associated with the Tie-Point. The object *FunctionAttributeMapping* manages the mapping between the parameter of the function and the points in the predicates. Tab. 5.2 shows the class details.

On the right side of the production there is the production body and the TauRules. Unlike what happens in MG, where the ProductionPredicates are only of two types (TerminalPredicate and NonTerminalPredicate), in XPG there is a third type of ProductionPredicate: the *Relation-Predicate*. The *List<ProductionPredicate>* is an ordered list and the order is used to determine the order of the predicates and the relationship between two or more predicates. The class *Pro-ductionPredicate* is an *abstract class* without attributes. The diagram shown in Figure 20 shows the ProductionPredicate entity and its specialization.

The *RelationPredicate* models the Relationships between predicates. The Relations can be of three types:

- driver
- tester
- any

Tab. 22 shows in detail the properties of the object.

Type	Name	Description
String	cscode	The instruction for XPG parser wrote in C#.
List<FunctionAttributeMapping>	functionAttributeMappings	The mapping between function and predicates.
String	mgProduction	A string that contains the production rule wrote in MG.
List<SyntacticAttributeMapping>	syntacticAttributeMappings	A list of the mappings between the point declared on the left side and the point used in the right side.
List<SyntacticAttributeMapping>	leftHandSideMapping	A list of the mappings between the predicates in the left side with the predicate in the right side.
List<ProductionPredicate>	rightHandSidePredicates	The list of predicates that compose the predicate rule.
List<TauRule>	tauRules	The list of Taurule.

Table 19: Properties of the **Production** element

Type	Name	Description
String	type	The type of the point in the mapping.
int	predicateParamIndex	The position of the param in the left side.
int	syntacticAttributeIndex	The position of the attribute in the predicate.
int	predicateSyntacticAttributeIndex	The position of the predicate in the right side.

Table 20: Properties of the **SyntacticAttributeMapping** element

Type	Name	Description
String	functionName	The name of the function.
List<SyntacticAttributeMapping)	syntacticAttributeMappings	The syntactic mapping of the function.

Table 21: Properties of the **FunctionAttributeMapping** element

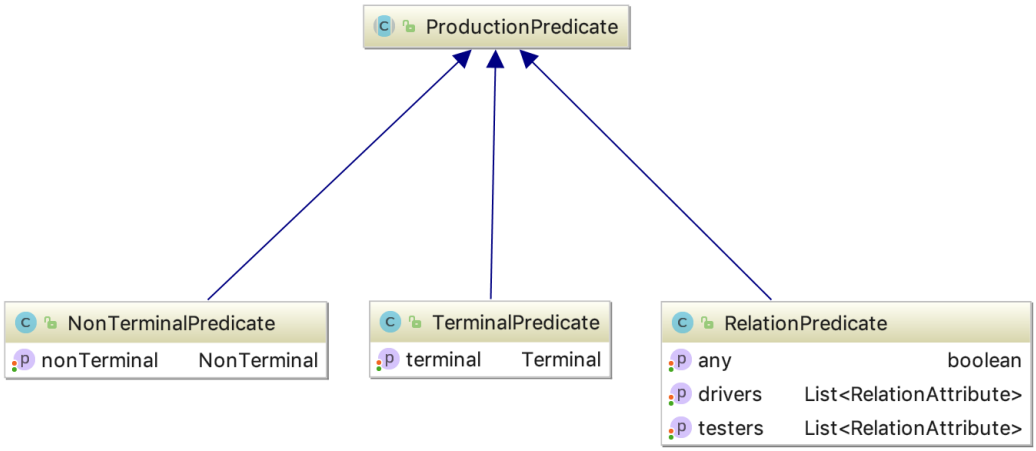

Figure 20: UML Diagram of Production Predicate

Type	Name	Description
boolean	any	Specifies if the relation is an any relations.
List<RelationAttribute>	drivers	The list of *driver* relations.
List<RelationAttribute>	testers	The list of *tester* relations.

Table 22: Properties of the **RelationPredicate** element

Type	Name	Description
List<Integer>	params	The index of params involved in the relationship.
int	backref	The number of back step that the parser should to link the predicate
Relation	relation	The Relation

Table 23: Properties of the **RelationAttribute** element

Type	Name	Description
Value	semanticValue	The semantic value of the TauRule.
List<SyntacticAttributeMapping)	syntacticAttributeMappings	The syntactic mapping of the TauRule.
String	conditionalCsCode	The instruction written in C# code for the TauRule
Terminal	terminal	The *terminal* related to the TauRule.

Table 24: Properties of the **TauRule** element

The object *RelationAttribute* models the mapping between the Relations used, the parameters that create the relationships. Finally, the parameter *backref* specifies the backstep necessary to link the two predicates. In Tab. 23 there are the properties of the object *RelationAttribute*.

Another object already defined in MG specification is *TauRule*. In Tab. 5.2 there is a description of the properties of the object.

5.3 Translation specification from MG to XPG

This section shows the rules and the algorithms that transform the MG data structure to the XPG data structure, presented in the previous few pages. At the end of the execution of these functions, an object *PositionalGrammar* are populated with the relative *terminals*, *non-terminals*,*relations* and *productions*.

The transformer, as the first task, begins to transform an instance of MG *terminal* to an instance of XPG *terminal*. The transformation is reported in the following block.

```
public Terminal convertInPositionalTerminal
(it.unisa.di.cluelab.datastructure.productionsymbol.impl.Terminal t) {
    terminal toReturn = new Terminal();
    toReturn.setEntryAttribute(0);
    toReturn.setName(t.getName());
    toReturn.setnVisit(t.getnVisit());
    toReturn.setSemanticType(t.getSemanticType());
    if (t.getSemanticValue().isPresent()) {
        Value v = convertInPositionalValue(t.getSemanticValue().get());
        toReturn.setSemanticValue(Optional.of(v));
    }
    toReturn.setSyntacticAttributeTypes(t.getSyntatticAttributeTypes());
    toReturn.setThreshold(t.getThreshold());
    return toReturn;
}
```

The transformation of the *Relation* object is similar to the *terminal* transformation. During the production parsing, the set of the negate relationships may compute.

```
public Relation convertInPositionalRelation
        (it.unisa.di.cluelab.datastructure.Relation r) {
    Relation rel = new Relation();
    rel.setName(r.getName());
    rel.setSyntacticAttributeTypes
        (r.getSyntacticAttributeTypes());
    rel.setThreshold(r.getThreshold());
    return rel;
}
```

After the parsing of these two simple elements, the algorithms find all the *non-terminal* declarations. In this phase, the algorithm doesn't parse the production, but only finds the *non-terminal* declaration.

```
public NonTerminal convertInPositionalNonTerminal(
    it.unisa.di.cluelab.datastructure.productionsymbol.impl.NonTerminal n) {
        NonTerminal nt = new NonTerminal();
        nt.setName(n.getName());
        nt.setSemanticType(n.getSemanticType());
        nt.setSyntacticAttributetypes(n.getSyntatticAttributeTypes());
        return nt;
}

nonTerminals = n.getNonTerminals()
                .stream()
                .map(nt -> convertInPositionalNonTerminal(nt))
                .collect(Collectors.toList());
```

After the research of the *non-terminal* definition, the transformer executes a function in order

to parse the production. The methods that compute this parsing operation is *convertInPositionalProduction*. This method is composed of four parts:

- The parsing of the predicates on the right side of the production;
- The parsing of the TauRule
- The computation of the mapping between the predicates in the left side of the production and the predicates on the right;
- The calculation of the mapping between the params on the left side of the production and the predicates on the right;

The parsing of the predicates on the right side starts with the transformation of the MG predicates to the XPG predicates.

```
prod.setRightHandSidePredicates(p.getRightHandSidePredicates().stream().map(rhs -> {
    ProductionPredicate rhsPred;
    if (rhs instanceof TerminalPredicate) {
        TerminalPredicate _rhsPred = new TerminalPredicatePredicate();
            _rhsPred.setTerminal(
                terminals.stream()
                .filter(t -> t.getName().equals(((TerminalPredicate) rhs)
                .getTerminal()
                .getName()))
            .findFirst().get());
            rhsPred = _rhsPred;
    } else {
        NonTerminalPredicate _rhsPred = new NonTerminalPredicate();
        _rhsPred.setNonTerminal(
            nonTerminals.stream()
            .filter(t -> t.getName().equals(
                ((NonTerminalPredicate) rhs)
                .getNonTerminal()
                .getName()))
            .findFirst().get());
        rhsPred = _rhsPred;
    }
    return rhsPred;
}).collect(Collectors.toList()));
```

After the transformation of the MG predicates in an instance of XPG predicates, the algorithm generates the RelationPredicates between the *terminal* or NonTerminal predicate. From the last predicate of the production, the algorithm searches the same point (Joint or Tie-Point) to create the RelationPredicate respecting the cardinality of the point. This is important because the number of relationships generated by the point cannot more than the number of times that a variable is used minus one. So, for example:

S –> A(a) B(a) C(a)

The variable a has cardinality 3, and it cannot generate more than 2 relationships. The block of the algorithm with this goal is reported in the next block.

```
for (int n = numberOfLeftPredicates; n >= 0; n--) {
 ArrayList <ArrayList<PredicateLabel>> params_before =
  p.getRightHandSidePredicates().get(n)
  .getParams();
 for (int q = 0; q < params_before.size(); q++) {
  ArrayList<PredicateLabel>params_before_entry=params_before.get(q);
  if (
    params_before_entry
    .stream()
    .anyMatch(pbe -> (pbe.getName()
                          .equals(paramLastEntryName) &&
                      pbe.getIndex() == index)))
  {
   if (cardinalityVars.get(paramLastEntryName) > 0) {
    cardinalityVars.replace(paramLastEntryName,
     cardinalityVars.get(paramLastEntryName) - 1);
    RelationAttribute relAttr = new RelationAttribute();
    relAttr.setBackRef(m - n - 1);
    relAttr.getParams().add(q);
    relAttr.getParams().add(kk);
    if (isNot) {
     Relation norel = findRelation(paramLastEntryName)
      .getNegateRelation();
     relAttr.setRelation(norel);
     relations.add(norel);
    } else {
     relAttr.setRelation(findRelation(paramLastEntryName));
    }
   }
  }
 }
}
```

Finally, the relationships are added as RelationPredicate.

```
mapping.forEach((k, v) ->
    prod.getRightHandSidePredicates().add(v, k));
```

The last check that this algorithm made is to verify that between each predicate there is a relation. If there isn't a Relation, the algorithm adds an *any* relationship between these predicates.

```
for (int i = 0; i < prod.getRightHandSidePredicates().size(); i++) {
 if (i == 0) {
  prev = prod.getRightHandSidePredicates().get(i);
  continue;
 }
 if (prod.getRightHandSidePredicates().get(i) instanceof TerminalPredicate ||
  prod.getRightHandSidePredicates().get(i) instanceof NonTerminalPredicate) {
  if (prev instanceof TerminalPredicate || prev instanceof NonTerminalPredicate) {
   RelationPredicate rel = new RelationPredicate();
   rel.setAny(true);
   prod.getRightHandSidePredicates().add(i, rel);
   i++;
  }
 }
 prev = prod.getRightHandSidePredicates().get(i);

}]
for (int i = 0; i < prod.getRightHandSidePredicates().size(); i++) {
 if (i == 0) {
  prev = prod.getRightHandSidePredicates().get(i);
  continue;
 }
 if (prod.getRightHandSidePredicates().get(i) instanceof TerminalPredicate ||
  prod.getRightHandSidePredicates().get(i) instanceof NonTerminalPredicate) {
  if (prev instanceof TerminalPredicate || prev instanceof NonTerminalPredicate) {
   RelationPredicate rel = new RelationPredicate();
   rel.setAny(true);
   prod.getRightHandSidePredicates().add(i, rel);
   i++;
  }
 }
 prev = prod.getRightHandSidePredicates().get(i);

}
```

In XPG the Relations specify the position of the point linked of the predicates. This mapping, that in MG is guaranteed by the same name of the point, in XPG is computed by the algorithm. The function computes the mapping and populate the *SyntacticAttributeMapping* object.

```
for (int j = 0;
    (j < p.getRightHandSidePredicates().size() && !mappingFound && !functionMapping);
    j++)
    {
    ProductionRHSPredicate right = p.getRightHandSidePredicates().get(j);

    for (int k = 0; (k < right.getParams().size() && !mappingFound); k++) {
        ArrayList<PredicateLabel> params = right.getParams().get(k);
        boolean res = params.stream().filter(param -> {
            if (param instanceof TiePointParamLabel) {
                return ((TiePointParamLabel)param)
                .getTiePointPredicate()
                .getTiePoint()
                .getName()
                .equals(paramName);
            } else if (param instanceof JointLabel) {
                return ((JointLabel) param)
                .getJoint()
                .getName()
                .equals(paramName);
            } else {
                return false;
            }

            }).findAny().isPresent();
            if (res) {
                SyntacticAttributeMapping mapping = new SyntacticAttributeMapping();
                mapping.setSyntacticAttributeIndex(i);
                mapping.setPredicateSyntacticAttributeIndex(j);
                mapping.setPredicateParamIndex(k);
                mapping.setType(getTypeFromJointOrTie(paramName));
                prod.getSyntacticAttributeMappings().add(mapping);
                mappingFound = true;
            }
    }
}
```

Lastly, the algorithm parses the TauRule elements with the transformation of the MG TauRule in XPG TauRule. The code that executes this transformation is very simple and is described in the following block.

```
prod.setTauRules(p.getTauRules().stream().map(tr -> {
    TauRule ptr = new TauRule();
    ptr.setConditionCsCode(tr.getConditionCSCode());
    ptr.setSemanticValue(convertInPositionalValue(tr.getSemanticValue()));
    ptr.setTerminal(
                terminals.stream()
                .filter(t -> t.getName()
                        .equals(tr.getTerminal()
                                .getName()))
                                .findFirst()
                                .get());
    return ptr;
    }).collect(Collectors.toList()));
```

5.4 Example of Language translation between these formats

In this section are presented some examples of languages, their related MG grammar and the final XPG grammar result with the transformation of the XPG data structure in the XPG code. The languages are the same described in the previous section (See 4.2 for more details). In the following section, there is the final XPG result: for the MG grammar see the previous sections.

5.4.1 A simple CFG language: palindrome strings

```
StartSymbol: S;
StartMovement: Sp;
namespace ProgramFlowChart_Test;
nonTerminals(
 S;
)
terminals(
 string A(AttachPoint: 1); string B(AttachPoint: 2);
)
movements(
 Sp;
)
nonterm S(()
 -> A any(0, 0) S any(0, 0) A; {} {}
 -> B any(0, 0) S any(0, 0) B; {} {}
 -> B;
 {} {}
 -> A;
 {} {}
 ->
 ; {} {}
)
```

5.4.2 A language for flow chart

```
StartSymbol: Program;
StartMovement: Sp;
nonTerminals(
 BOH Program; string Statements; string Statement;
)
terminals(
 string START(AttachPoint: 1); string INSTRUCTION(AttachPoint: 2);
 string PREDICATE(AttachPoint: 3); string END(AttachPoint: 1);
)
movements(
 Sp; link(AttachPoint, AttachPoint); nolink(AttachPoint, AttachPoint);
)
nonterm Program(()
 -> START link(1, 1) Statements link(2, 1) END; {} {
  string code = "{\n";
  string[] lines = $2.semanticValue.Split('\n');
  for (int i = 0; i < lines.Length; i++)
   code += "\t" + lines[i] + "\n";
  var $$ = new BOH();
  $$.ciao = code + "\n}\n";
  $$.num = 10;
 }
)
nonterm Statements((AttachPoint: 2)
 -> Statements link(2, 1) Statements; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $2.attachpoint[2];
 } {
  $$ = $1.semanticValue + "\n" + $2.semanticValue;
 }
 -> Statement; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $1.attachpoint[2];
 } {
```

```
  $$ = $1.semanticValue + "\n";
 }
)
nonterm Statement((AttachPoint: 2)
 -> INSTRUCTION; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $1.attachpoint[2];
 } {
  $$ = $1.semanticValue;
 }
 -> PREDICATE link(2, 1) ^ link(1, 2) Statements; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $1.attachpoint[3];
 } {
  string code = "if(" + $1.semanticValue + ") {\n";
  string[] lines = $2.semanticValue.Split('\n');
  for (int i = 0; i < lines.Length; i++)
   code += "\t" + lines[i] + "\n";
  $$ = code + "\n}";
 }
 -> PREDICATE link(2, 1) ^ link(3, 2) Statements; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $1.attachpoint[3];
 } {
  string code = "while(" + $1.semanticValue + ") {\n\t";
  string[] lines = $2.semanticValue.Split('\n');
  for (int i = 0; i < lines.Length; i++)
   code += "\t" + lines[i] + "\n";
  $$ = code + "\n}";
 }
 -> Statements(b, c: a) Statements(d, c)
    PREDICATE link(2, 1) ^ nolink(1, 2) Statements
    link(3, 1)(-1) ^ link(2, 2) Statements; {
  $$.attachpoint[1] = $1.attachpoint[1];
  $$.attachpoint[2] = $2.attachpoint[2];
```

```
} {
 string code = "if(" + $1.semanticValue + ") {\n";
 string[] lines = $2.semanticValue.Split('\n');
 for (int i = 0; i < lines.Length; i++)
  code += "\t" + lines[i] + "\n";
 code += "\n}\n else { \n\t";
 lines = $3.semanticValue.Split('\n');
 for (int i = 0; i < lines.Length; i++)
  code += "\t" + lines[i] + "\n";
 $$ = code + "\n}";
})
```

5.4.3 A language for $A^nB^nC^n$ strings

```
StartSymbol: S; StartMovement: Sp;
nonTerminals(
 string S; string BlockAB; string BlockC; )
terminals(
 string A(Coordinate: 1);
 string B(Coordinate: 1);
 string C(Coordinate: 1);
)
movements(
 Sp; LEFT(Coordinate, Coordinate); RIGHT(Coordinate, Coordinate);
)
nonterm S(()
 -> BlockAB RIGHT(2, 1) BlockC; {} {}
)
nonterm BlockAB((Coordinate: 2)
 -> A RIGHT(1, 1) BlockAB RIGHT(2, 1) B; {
  $$.coordinate[1] = $1.coordinate[1];
  $$.coordinate[2] = $3.coordinate[1];
 } { keepToken(1, $3, 1); }
 -> A RIGHT(1, 1) B; {
  $$.coordinate[1] = $1.coordinate[1];
  $$.coordinate[2] = $2.coordinate[1];
 } { keepToken(1, $2, 1); }
)
nonterm BlockC((Coordinate: 2)
 -> BlockC RIGHT(1, 1) C LEFT(2, 1)(-1) B; {
  $$.coordinate[1] = $2.coordinate[1];
  $$.coordinate[2] = $3.coordinate[1];
 } {}
 -> C LEFT(1, 1) B; {
  $$.coordinate[1] = $1.coordinate[1];
  $$.coordinate[2] = $2.coordinate[1];
 } {} )
```

6 Conclusions and Further Work

In this thesis, a new grammar formalism to define *Visual Languages* called *Multidimensional Grammar* was described. From MG it was developed an *Integrated Development Environment* that helps language developer in their definition work. MG is a grammar formalism that allows the definition of Visual Languages through an easier and more readable syntax than other formalisms described below. Through the IDE, the definition of Visual Languages is easy: the editor helps developers through syntax highlighting, errors and warnings reporting. Results obtained with this thesis are in line with what was expected at the beginning of this research. With the new grammar, formalism is possible to define a big set of languages. Finally, the IDE developed has been supporting language developers in particular through errors supporting.

The next steps are specifically concerned with the Multidimensional Grammar definition, and with the improvement of some limitations. In parallel to these improvements, it is important to study this formalism in order to have a precise idea of the *class languages* collocation.

Moreover in the first part are shown some language examples written with MG and these examples affirm that the class of languages represented by MG are bigger than *CFG* and *CSG*, and are at least bigger as the *XPG* class languages. Further research will focus on this theme.

Regarding the developed *IDE*, the next steps concern the integration with an *XPG* parser generator, in order to collect all the necessary functions in a unique tool. Furthermore, another aspect which shows potential interest for future investigation is the development of a *language server*[15][16], in order to integrate the IDE with other development tools. Finally, it is also important to involve other developers in order to retrieve their feedback about the IDE and the MG formalism.

References

[1] S. K. C. et. other, "Picture processing grammar and its applications," *Information Science*, 1971.

[2] S.-K. Chang, *Visual Programming Systems*. Prentice Hall, 1990.

[3] A. S. Fisher, *CASE: Using Software Development Tools*. New York: Wiley and Sons, second ed., 1992.

[4] C. McClure, *CASE is Software Automation*. Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

[5] Massachusetts Institute of Technology, "Scratch - imagine program share."

[6] Massachusetts Institute of Technology, "Mit app inventor."

[7] G. C. et al, "A parsing methodology for the implementation of visual systems," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 23, no. 12, 1997.

[8] G. P. Gennaro Costagliola, Vincenzo Deufemia, "A framework for modeling and implementing visual notations with applications to software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 13, 2004.

[9] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2016.

[10] J. Feder, "Plex languages," *Information Science*, vol. 3, pp. 225–241, 1971.

[11] E. Foundation, "Xtext framework."

[12] E. Foundation, "Xtext - wiki."

[13] Eclipse Foundation, "Xtend framework."

[14] K. Z. Zhang, Da-Qian and J. Cao, "A context-sensitive graph grammar formalism for the specification of visual languages," *The Computer Journal*, vol. 44, no. 3, 2001.

[15] Krill Paul, "Xtext framework."

[16] Microsoft, "Language server protocol implementation."