Università degli Studi di Salerno
Dipartimento di Informatica
Corso di Laurea Magistrale in Informatica

Tesi di Laurea Magistrale
in
Informatica

A formal grammar definition to describe Multidimensional Languages

Advisors

Ch.mo Prof. Gennaro Costagliola

Candidate

Luca Reccia
Mat. 0522500653

anno accademico 2019/20

Abstract

Multidimensional languages have always been integrating part of the whole software development process, programmers and software designers often use them to better represent and model difficult concepts. Decision tables, flow charts, logic diagrams, etc. are all examples of multidimensional languages.

It is useful, as in the monodimensional case, to define some formal grammar model to represent and work easily with these languages. Many grammar formalisms to define multidimensional languages already exist in the literature, but they are either too much domain-specific to describe all kinds of languages (RGG, Hyperedge Grammars) or too complex to be used in practice (XPG).

The goal of this thesis is to address the problem of defining a generic multidimensional language in such a way that it is both efficient to parse and programmer-friendly to write. We will redefine a grammar formalism called Multidimensional Grammar (MG) and we will show how it is possible to generate a parser.

Contents

1 Introduction **3**

2 Multidimensional Languages **4**
2.1 An Example Language 5

3 Positional Grammars **8**
3.1 Grammar Definition 8
3.2 The parser 9
3.2.1 The parser input syntax 9
3.2.2 How does the parser work? 10
3.2.3 The parser's constraints 11
3.3 An Example of Positional Grammar 15

4 Multidimensional Grammars **19**
4.1 Grammar definition 19
4.1.1 Grammar syntax 20
4.2 An Example of Multidimensional Grammar 21
4.3 Notes on the comparison with other formalisms 23
4.3.1 Reserved Graph Grammar 23
4.3.2 Contextual Hyperedge Replacement Grammars 24

5 Converting Multidimensional to Positional Grammars **29**
5.1 The algorithm 29
5.1.1 Non-terminal analysis 30
5.1.2 Grammar transformation 34
5.2 Correctness 36
5.3 Implementation 38
5.3.1 Programs structure 38
5.3.2 LSP validations and suggestions 39
5.3.3 Algorithm implementation 39
5.3.4 Testing 46

6 Conclusion and future developments **47**

Chapter 1

Introduction

During the last years there has been an increasing interest on Visual languages (or Multidimensional Languages) both in the academic and business world. The ease of use and great flexibility of visual languages has led to the creation and release of an increasing number of applications that use them. Being able to parse a multidimensional language is useful for many reasons. There exist a lot of CASE tools and software to model UML diagrams, flowcharts, etc., and being able to define them using a Multidimensional Grammar could be easier and would also add a syntax validation step. Multidimensional languages' parsers would also make a lot easier the development of visual programming languages (VPL) which due to their structure are becoming increasingly popular, for examples in learning environments (Scratch[1], App Inventor[2]) but also in low-code or no-code environments (SAP Workflow Builder[3], OpenText[4]) where VPLs allow the existence of easier to use and less prone to bugs applications.

The content of this thesis is organized as follows. In the first chapter, we present the multidimensional languages, explain their importance along with some examples. Next, we will present and describe the Positional Grammar formalism, for which different kinds of parsers already exist. We will show its power and flexibility, but also some downside, such as its difficulty to use and its complex operational syntax. In the fourth chapter, we will introduce the multidimensional grammar formalism along with many examples of use and comparisons with already existing formalisms. In the fifth chapter, we will show how it is possible to convert any given Multidimensional Grammar into a Positional Grammar that generates the same language. This transformation is useful because it allows us to reuse the same parsers generation algorithms originally built for Positional Grammars while working with Multidimensional Grammars. In the last chapter, we end this thesis with a discussion on what we have understood after this work and on what future researches will need to focus on.

Chapter 2

Multidimensional Languages

During the last hundred years, context-free languages have been largely studied. A language can be formally defined as a set of strings of finite length build upon an alphabet. As an example, saying

$$L = \{a^n b^n | n \geq 1\} \tag{2.1}$$

means that L is a set containing all the possible strings having n occurrence of the letter a and then n occurrence of the letter b, where n is a natural number. Each instance of a letter in any word belonging to L can be seen as concatenated to the previous symbol. In other words, there is an implicit concatenation relation between each symbol and his subsequent as shown in figure 2.1a.

What would happen if instead of multiple concatenations we want to be able to define other kinds of relations between symbols? For example, let us say we want to define a new L' such that each instance of the letter a should be on the ***right*** of the previous one (besides the first one) and should have an instance of the letter b on his ***above***, as is shown in figure 2.1b. This is a first, really basic, example of a multidimensional language (or visual language).

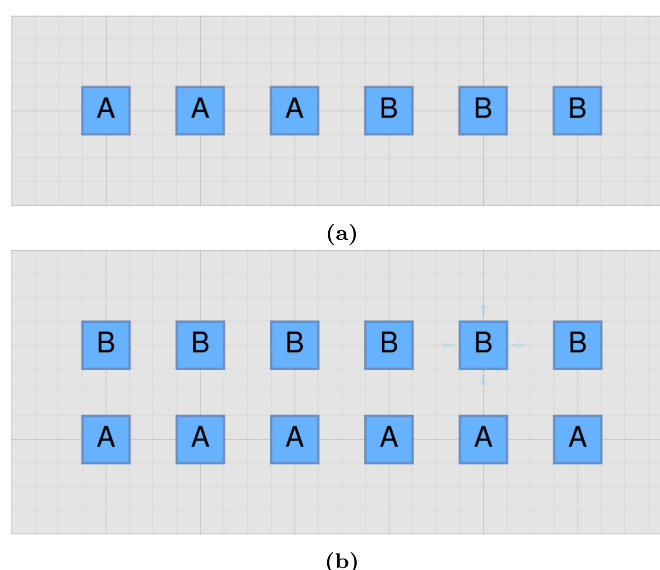


Figure 2.1: (a) Example of a word that can be generated using a monodimensional language. (b) Example of a word that can be generated using a multidimensional language.

More formally, we use symbols and relations to describe multidimensional languages. A symbol can be either terminal or non-terminal and has a name and it might have a list

of syntactic attributes. Relations are predicates defined on two syntactic attributes of two different symbol instances. We define a multidimensional language as a set of **visual words**, where each word is a set of **terminal instances** that shares relations.

Thinking back to the example of the multidimensional language L', in order to define the relations ***right*** and ***above*** a solution might be to use as syntactic attribute of each symbol its coordinates. We could define that a symbol b with coordinate (x, y) is on ***above*** of a with coordinate (x', y') iff $x = x'$ and $y = y' + 1$ as shown in figure 2.2a. Using this approach many kind of different relations can be defined.

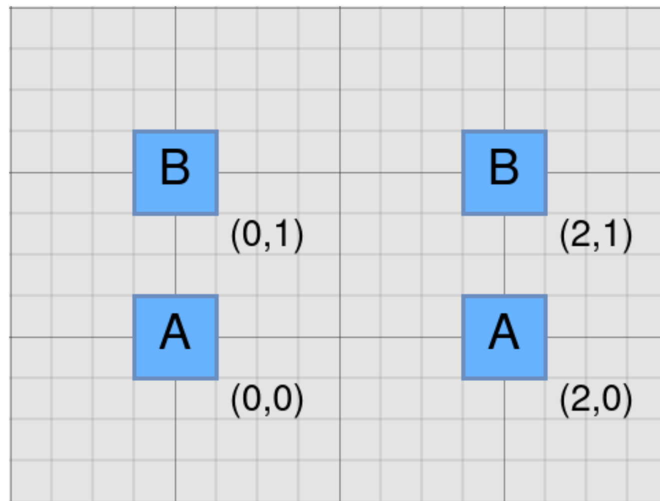
One relation that will be used a lot trough this dissertation is the ***link*** relation, where each symbol can have many attaching points as attributes. There is a link relation between two attaching points of two symbols iff a line connects them as shown in figure 2.2b.
Another interesting relation is the ***contains*** relation. If, for example, we define each symbol to be a square and we use the top-left and the lower-right coordinate as attributes, we can define the contains relation as follows: A symbol A with top-left coordinate (a, b) and lower-right coordinate (c, d) ***contains*** a symbol B with top-left coordinate (a', b') and lower-right coordinate (c', d') iff $a \leq a'$, $b \geq b'$, $c \geq c'$ and $d \leq d'$ as in figure 2.2c.

Context-free languages can be defined using context-free grammars. For example, the grammar $G = (V, \Sigma, R, S)$ such that $L(G) = L$ defining $V = S$, $\Sigma = \{a, b\}$ and $R = \{S \rightarrow aSb|ab\}$, generates the language $L = \{a^n b^n | n \geq 1\}$.
Currently, there are many formal grammar definitions for multidimensional languages, but before presenting them it is first worth informally describe how an example visual language might look like.

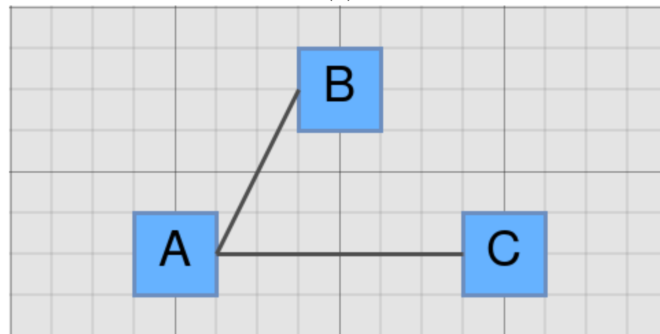
2.1 An Example Language

Let us suppose that we want to write a **program** using some visual syntax. In our example, a **program** might have many **functions**, a **function** is characterized by a **function declaration** along with some **statements** *below*. Each **statement** is executed sequentially after the one on *above* and before the one *below*, except for a **condition**, which after its evaluation, depending on the result, the next executed **statement** might be the one connected to the first or the second attaching point. Both separate flows need to rejoin at a certain point, a rejoin is characterized by the connection of an attaching point of both the last separate statements to the attaching point of the next statement. Lastly, another custom construct is the **function call** block, which is similar to the statement block, but it needs to have a connection with a function declaration. An example of what we can build using all these symbols and their connection is shown in Figure 2.3.

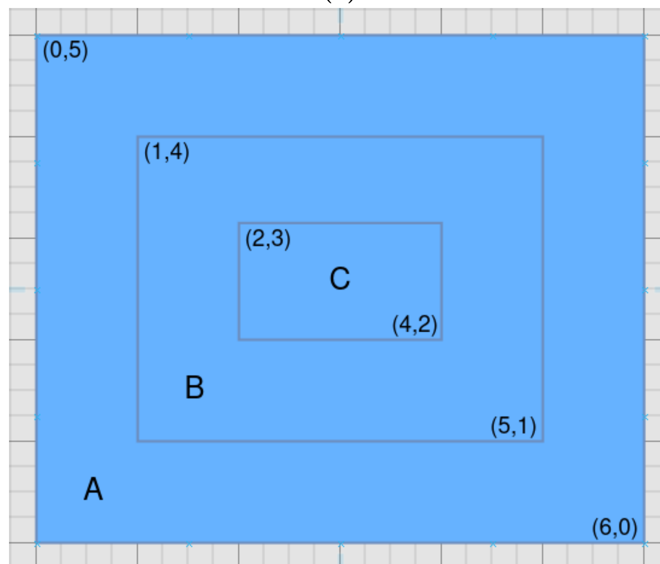
The simple multidimensional language presented above is a subset of what might be a real visual programming language. It is small enough to be discussed in depth in this dissertation, but big enough to contains some nontrivial components. For example, the function_call block might require the target grammar to be context-sensitive and the function blocks, which do not share any relation between each other, might require the grammar parser to be either generalized or to be capable of handling run-time conflicts in some way.



(a)



(b)



(c)

Figure 2.2: (a) Example of a visual sentence using center coordinates as attributes. (b) Example of a visual sentence using attaching points as attributes. (c) Example of a visual sentence using two pairs of coordinates as attributes.

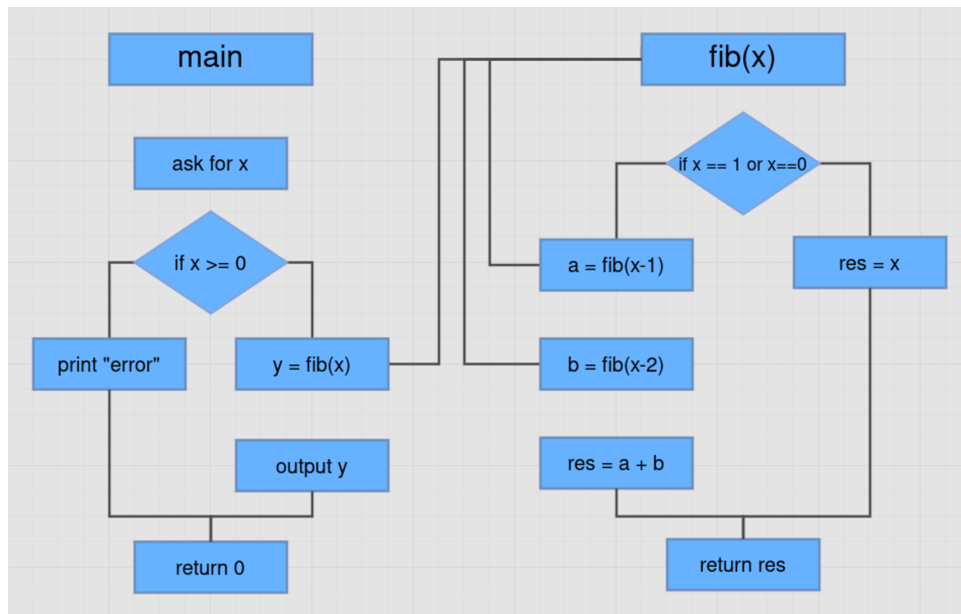


Figure 2.3: How recursive Fibonacci might look like in our sample visual language.

Chapter 3

Positional Grammars

We will use **Positional Grammars** (PG)[5] as a target specification for our transformation algorithm. More specifically, we will use the **eXtendend Positional Grammar** (XPG)[6] formalism since it allows users to define a wider class of multidimensional languages adding context sensitiveness.

3.1 Grammar Definition

An XPG grammar, described in [6] but listed here for quick reference, is a pair $XPG = (G, PE)$, where PE is a positional evaluator, and G is a tuple $G = (N, T \cup POS, S, P)$, where:

- N is a finite nonempty set of non-terminal symbols;
- T is a finite nonempty set of terminal symbols such that $N \cap T = \emptyset$;
- POS is a finite set of binary relations such that $POS \cap N = \emptyset$ and $POS \cap T = \emptyset$;
- $S \in N$ is the starting symbol;
- P is a finite nonempty set of *productions*.

Symbols and relations are as described in chapter 2. Each *production* is expressed as $A \rightarrow x_1 R_1 x_2 R_2 \cdots x_{m-1} R_{m-1} x_m, \Delta, \Gamma$ where:

- A is a non-terminal symbol;
- each x_i is either a terminal or a non-terminal symbol;
- each R_j is a pair $\left(\left\langle REL_1^{h1}, \cdots, REL_i^{hi}\right\rangle, \left\langle REL_{i+1}^{hi+1}, \cdots, REL_n^{hn}\right\rangle\right)$, where $REL_i \in POS$ and, $\forall i \in [1, n]$, the first sub-sequence of relations is called *driver set* and the second subsequence of relations is called *tester set*;
- Δ is a set of rules used to synthesize the values of the syntactic attributes of A;
- Γ is a *set of triples*.

Each *triple* in Γ is expressed as $(T_j, Cond_j, \Delta_j)$ with $j \geq 0$ where:

- T_j is a symbol to be be inserted in the input;
- $Cond_j$ is a precondition to be verified to insert T_j;

- Δ_j is the rule used to compute the values of the syntactic attributes of T_j from those of $x_1 \cdots x_m$.

At last, "for each production, the number of tuples in Γ whose conditions can simultaneously evaluate to true must be less than $m - 1$". This means that no more than $m - 2$ symbols can be inserted in the input during the application of a production[7], thus granting the convergence of the parser.

3.2 The parser

The XPG formalism is very powerful and, most importantly, given an XPG grammar, its syntax makes it easier to create a parser. More specifically, all the visual languages' parsers we will cite or discuss in this thesis are always generalized, unless otherwise specified.

There exist parser generators that automatically create different kinds of parser for each input grammar. In this thesis we have used and worked on a **bottom-up** generalized parser generator[6], therefore we will need to discuss its structure and its problem. Another **top-down** generalized parser generator is also being studied and developed at the time of writing and will be the focus of future researches.

3.2.1 The parser input syntax

During this thesis, we will be using the syntax described here while writing XPG examples. Non-terminals, terminals, relations (called movements) are pretty straight forward to understand from the example in Listing 3.1.

Listing 3.1: Example of XPG language to illustrate its syntax

```
1  StartSymbol: S;
2
3  StartMovement: Sp;
4
5  nonTerminals(
6     S;
7  )
8
9  terminals(
10    a(AttachPoint:3);
11    b(AttachPoint:3);
12    c(AttachPoint:2);
13 )
14
15 movements(
16    Sp;
17    link(AttachPoint,AttachPoint);
18 )
19
20 nonterm S( ()
21   ->  a link1_2 & link2_3 ^ link3_1 b link1_1(-1) c any a'
22     {
23         $$.attachPoint[1] =$1.attachPoint[1];
24         $$.attachPoint[2] =$3.attachPoint[2];
25     }
26     {
```

```
27          preserve(4);
28      }
29  )
```

Productions instead have a slightly different syntax. In our example we have that: The **left part** of the production is `S`, while the **right part** goes from `a` to `a'`. We will distinguish different instances of the same symbol using either some marks or their index. For example, `a` is the 1st symbol in the right side of the production, while `a'` is yet another instance of the same symbol but it refers to the 4th symbol in the production.

The **relation** `link1_2` means that the 1st attribute of the first symbol to the left of this relation is *linked* to the 2nd attribute of the first symbol to the right of the relation. If there is a negative number `x` in round brackets alongside the relations, that means that the fist symbol to use in the relationship is the $x-th$ to the left. For example `link1_1 (-1)` means that there is a *link* relation between the first attribute of the symbol `a` and the first attribute of the symbol `c`.

Relations can be of two kinds, **explicit** and **implicit**, depending on whether they are visually depicted in the input. An example of explicit relations is the **link** relation, represented in the input using a line connecting two symbols. Implicit relations are, for example, **above** relation, **contains** relation, etc.

A special relation called **any** is used to specify that the following symbol doesn't need to have any specific relationship with any of the previous ones, in our example this would mean that the parser should try to accept the input using any possible instance of the terminal `a` found in the input.

Between each symbol in the right part of the production, there might be many relations, either drivers and testers. First are listed the **driver relations**, separated by the & character, eventually there might be **tester relations**, separated by the ^ character.

Each symbol has an **entrance point** set consisting of all the attaching points used in the previous driver relations insisting on it. For example, in the single production in Listing 3.1 the symbol `b` has {2, 3} as entrance point set. The entrance point of the first symbol of the right-hand side of a production with left-hand side non-terminal A depends on the driver relations insisting on all the instances of A in the other productions.

Lastly, we have two sections, each delimited by braces:

- The first one represents the **delta** rules, used to synthesize the attributes of the left hand side non terminal. In the example, the non-terminal `S` synthesizes its first attaching point from the first attaching point of `a` (the first symbol) and the second attaching point from the second attaching point of `c` (the third symbol);

- The second one represents the **gamma** rules which are expressed through the **preserve** function. This allows us to avoid to remove a specific terminal from the input. In the example above, we are preserving the 4th symbol in the production.

Both these rules can and will be omitted in the examples if not specifically required.

3.2.2 How does the parser work?

A detailed explanation of the underlying algorithm, that builds an XPG parser from its formalism, is provided in [6]. Here we only focus on a short explanation to be used as a reference and to motivate some requirements listed in next sections.

The XPG parser derives its structure from LR parsers, widely used for monodimensional grammars. It can be schematized using the typical action table, where along with **actions** and **goto** columns, there is also a **next** column. This last column contains the relations to follow to find the next input symbol to parse. This is required because, while in the monodimensional case each symbol is always on the left of the previous, in multidimensional grammars, without this information, we would not know which terminal to parse next. After the execution, the input is accepted iff the parser reaches an accepting state, all the symbols have been seen and all the explicit relations (such as the *link* relations) have been parsed.

Conflicts

Having followed the structure of bottom-up monodimensional parsers, the parser table might contain well-known **shift-reduce** and **reduce-reduce** conflicts. Unluckily, these are not the only kind of conflicts that may make a the parser non deterministic, as a matter of fact, among others, the new column **next**, used to identify subsequent input elements to parse, might also cause a different type of conflict. They are called **run-time conflicts** [8] and they occur at run-time whenever the parser, by applying a relation, finds 2 or more symbols in the input to be considered as the next one.

For example, let us consider the input in Figure 3.1 and the grammar having a single production `S -> A link1_1 B link1_1(-1) B`. The parser execution will eventually reach the point where, after having seen A, it searches for the next input symbol. Knowing the production, the parser will search for an instance of a B symbol by following the `link1_1` relation and it will find two of them. In this specific situation, whatever the choice and depending on the rest of the grammar, the parser might eventually accept the input (since the Bs have the same context), but this might not be the case with more complex examples. This type of conflict depends on how the positional grammar productions have been designed.

To handle conflicts, the parser needs to be generalized[9]. This means that each time a conflict is found, regardless of its type, the parser execution is simply split and each new parser continues following each one of the alternatives that created the conflict. This approach of course has a clear downside, some specific input can make the execution time of the parser explode significantly.

Latest researches, [8], show how it is possible to statically detect the possible occurrences of these run-time conflicts. However, further studies are needed to define new approaches and the set of constraints to respect to easily build run-time conflicts free grammars.

3.2.3 The parser's constraints

Due to the operational nature of the XPG syntax and to some operations performed by the parser explained above there are some constraints that need to be respected in order to use an XPG grammar for creating a parser from it. In other words, not all the positional grammars can be used to create a parser by this specific algorithm.[1]

[1] This does not mean that we are reducing the class of generable languages. A visual language generated by any generic XPG grammar can also be generated by another XPG grammar that respects these constraints.

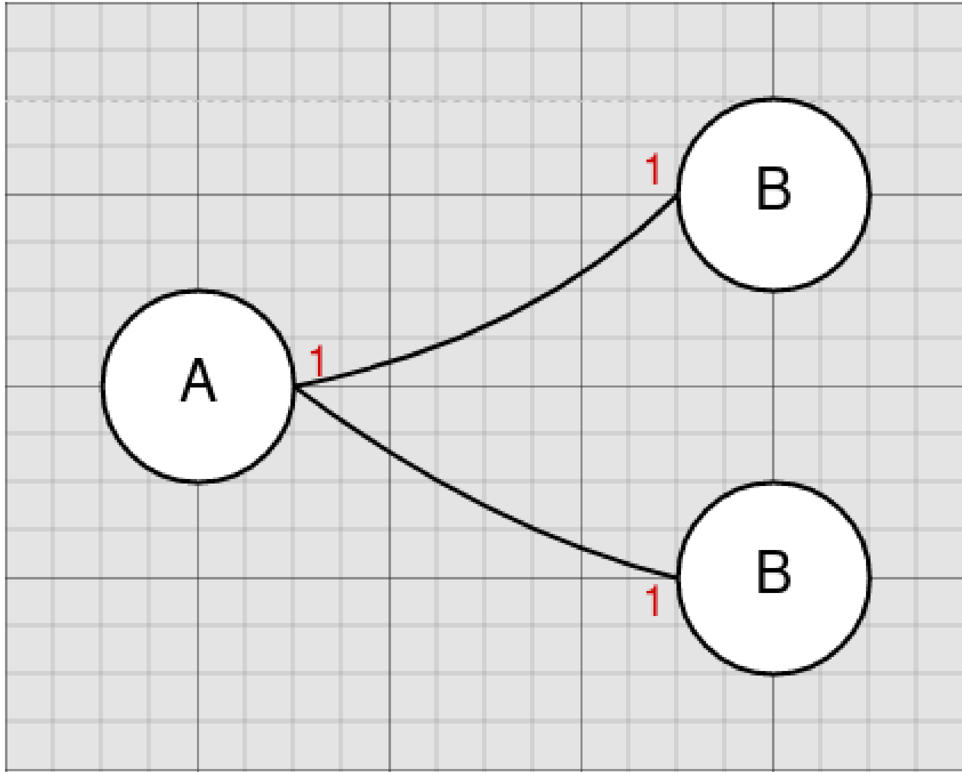


Figure 3.1: An input that might result in a run-time conflict.

More specifically, there are two kinds of constraints: hard and soft constraints. **Hard constraints** always need to be satisfied in order to be sure that the generated parser accepts all and only the sentences of the intended language. **Soft constraints**, on the other hand, are just suggestions that, if respected, may improve the efficiency of the parser.

Productions with valid entrance point

We say that a production `p` is *valid* if each entrance point of the left non terminal symbol of `p` is synthesized through the syntactic rules Δ of `p` from the first symbol of the right side of `p`. In this case, we also say that this first symbol is *valid* in `p`.

We are now ready to define what a *valid* positional grammar is. This sets a **hard constraint** on the use of positional grammars for pLR parsing. In fact, only valid positional grammars can be considered for pLR parsing due to the construction methodology of pLR parsers.

Definition 1. *A positional grammar is said to be valid if all of its productions are valid, i.e., if each entrance point of the left non terminal symbol of* `p` *is synthesized through the syntactic rules* Δ *of* `p` *from the first symbol of the right side of* `p`*.*

As an example let us consider the productions in Listing 3.2 and assume that the non-terminal `C` has valid productions. The entrance point set of the non-terminal `B` is {1, 2}, and, in order to guarantee the validity property, we should check each production with `B` as its left side, on each element of this set.

Attribute 1 of `B` in the only production `p` of `B` starting at line 4 is synthesized from the first symbol therefore the validity requirement so far is satisfied. On the other hand,

attribute 2 of `B` is synthesized from the second symbol, therefore the production is not valid and the whole grammar definition is invalid. The easiest way to make this grammar valid is to change the relations in the first production from `link1_1 & link2_2` to `link1_1 ^ link2_2`. This also reduces the driver set to $\{1\}$.

Listing 3.2: XPG with invalid entrance point due to big driver set

```
1 S
2     ->  a link1_1 & link2_2 B
3
4 B
5     -> C link3_3 C
6     {
7         $$.attachPoint[1] = $1.attachPoint[1]
8         $$.attachPoint[2] = $2.attachPoint[2]
9     }
```

This is not the only approach to reach validity. Let us consider Listing 3.3 where the entrance point set of `B` is only $\{1\}$ and again notice that in this case the attribute 1 is synthesized from the second symbol, breaking the rule. In this case, the proper way to solve the problem is to rewrite the second production switching the symbol's positions and changing the relations accordingly. Listing 3.4 shows the correct productions, note that the **right** relation has been substituted by the inverse **left** relation.

Listing 3.3: XPG with invalid entrance point due to bad written production

```
1 S
2     ->  a left1_1 B
3
4 B
5     -> a right2_2 C
6     {
7         $$.attachPoint[1] = $2.attachPoint[1]
8     }
```

Listing 3.4: XPG with valid entrance point and rewritten production

```
1 S
2     ->  a left1_1 B
3
4 B
5     -> C left2_2 a
6     {
7         $$.attachPoint[1] = $1.attachPoint[1]
8     }
```

Unfortunately, it's not always that simple. In the next example, Listing 3.5, we have a case where both the previous approaches fail to solve the issue: the entrance point set $\{1, 2\}$ for `B` cannot be changed by modifying the drivers and switching the symbols order still recreates the same issue. In these cases, the only way to solve the problem without having to redefine the grammar is to substitute the `B` non-terminal with two different non-terminals `B_0` and `B_1`. Listing 3.6 shows the edited grammar with a valid syntax.

Listing 3.5: XPG with invalid entrance point

```
1 S
2     ->  a link1_1 B
3     -> a link1_2 B
```

```
4
5   B
6       -> C link3_3 D
7       {
8           $$.attachPoint[1] = $1.attachPoint[1]
9           $$.attachPoint[2] = $2.attachPoint[2]
10      }
```

Listing 3.6: XPG with valid entrance point

```
1   S
2       ->  a link1_1 B_1
3       -> a link1_2 B_2
4
5   B_1
6       -> C link3_3 D
7       {
8           $$.attachPoint[1] = $1.attachPoint[1]
9           $$.attachPoint[1] = $2.attachPoint[2]
10      }
11
12  B_2
13      -> D link3_3 C
14      {
15          $$.attachPoint[1] = $2.attachPoint[1]
16          $$.attachPoint[1] = $1.attachPoint[2]
17      }
```

Production's drivers

Driver sets are a crucial aspect of positional grammars. Generally speaking, the more relations a driver set has, the less chances there are to find a run-time conflict, the more the parser is likely to execute faster. *Choosing the right driver set* is a **soft constraint** and it depends mostly on the semantic meaning of the symbol we are modeling, therefore there is no algorithm to "fix" bad grammars, it is completely up to the programmer.

Luckily enough, it is not so difficult to come up with a good grammar, the best approach is to simply try to put into the driver set the relations most likely to discriminate a specific couple in the most common inputs. For example, let us consider the production `S -> a link1_1 ^ link2_2 b` and a possible correct input that looks like Figure 3.2, the generalized parser is forced to split its execution multiple times in order to find the correct path. While instead, just editing the production to be `S -> a link1_1 & link2_2 b` or `S -> a link2_2 ^ link1_1 b` is enough to avoid, at least in this example, any run-time conflict at all. As of last note, of course between the two latest productions, it's obviously better to pick the first one, since it has a driver with more relations and then with more constraints on the next symbol to parse, the latter however has been proposed as an alternative since, as explained in the previous constraint, the first might not always be available. It must be noted, however, that a driver with many relations takes more time to be executed with respect to a driver with fewer relations. So a tradeoff must be found depending on the type of correct language sentences and on the possible editable incorrect ones.

Relations and symbols order

The driver size is not the only factor to consider to write efficient to parse productions. Another **soft constraint** is to *correctly place the relations and the symbols inside of a production.*

The same production often can be represented in multiple ways. For example the production `S -> a link1_1 b link1_1 c` is semantically equivalent to the production `S -> a any c link1_1(-1) & link1_1 b`. Substituting one with the other in a grammar does not change the generated visual language. However, the latter places two directly unrelated symbols close in the production, thus requiring the **any** production. As already mentioned, the less specific a driver is the more likely the parser is to have performance issues. In this case, rearranging the symbols may allow faster parser execution.

3.3 An Example of Positional Grammar

Having presented this formalism we can finally define a positional grammar for the sample visual language introduced in the previous chapter.
The terminal set T is composed of all the visual elements that the user sees and uses while constructing his input, so after re-reading the requirements it is obvious to come up with the set:

Listing 3.7: XPG terminals of our example language

```
1 terminals(
2    function_name(Coordinate:1, AttachPoint:1);
3    statement(Coordinate:1, AttachPoint:2);
4    condition(Coordinate:1, AttachPoint:3);
5    function_call(Coordinate:1, AttachPoint:3);
6 )
```

To obtain the non-terminal set N we can apply the same techniques used while writing monodimensional grammars. Thinking at the reductions that we might use to define the grammar we note that all the related statement blocks will form a **Statements** symbol, meanwhile a function_name with its statements might form a **Function** symbol and many functions will form the **Program** symbol (which will also be our starting non-terminal symbol). Hence we have:

Listing 3.8: XPG non-terminals of our example language

```
1 StartSymbol: Program;
2
3 nonTerminals(
4     Program;
5     Function;
6     Statements(Coordinate:2, AttachPoint:2);
7 )
```

The relation set, also called movements in the parser, is straightforward, we only need to think to what kind of relations there might exist between two symbols. We have a **below** relation between *function_name* and *statements* as well as between *statements* and other *statements*; A **link** relation is used when there is a *condition* and between a *function_call* block and its *function* definition. Hence the relation set is:

Listing 3.9: XPG relations of our example language

```
1  movements(
2     link(AttachPoint,AttachPoint);
3     below(Coordinate,Coordinate);
4  )
```

Lastly, the productions needed to generate our language are written in listing 3.10. The syntax is pretty intuitive, relations are placed between symbols in each right side of the productions. Among all of them, we note a special relation **any** which is required whenever there are no specific relations between a symbol and any of its antecedent in the production.

Listing 3.10: XPG productions of our example language

```
1  Program
2    ->  Program any(0,0) Function;
3
4    -> Function;
5
6  Function
7    ->  function_name below(1,1) Statements;
8
9  Statements
10   -> statement;
11   {
12     $$.coordinate[1] = $1.coordinate[1];
13     $$.coordinate[2] = $1.coordinate[1];
14     $$.attachPoints[3] = $1.attachPoints[2];
15     $$.attachPoints[4] = $1.attachPoints[3];
16   }
17
18   -> Statements below(2,1) Statements;
19   {
20     $$.coordinate[1] = $1.coordinate[1];
21     $$.coordinate[2] = $2.coordinate[2];
22     $$.attachPoints[3] = $1.attachPoints[3];
23     $$.attachPoints[4] = $2.attachPoints[4];
24   }
25
26   -> condition link(3,3) Statements link(4,3)(-1) Statements link(4,3)(-1) & link(4,3)
        Statements;
27   {
28     $$.coordinate[1] = $1.coordinate[1];
29     $$.coordinate[2] = $4.coordinate[2];
30     $$.attachPoints[3] = $1.attachPoints[2];
31     $$.attachPoints[4] = $4.attachPoints[4];
32   }
33
34   -> function_call link(4,2) function_name;
35   {
36     $$.coordinate[1] = $1.coordinate[1];
37     $$.coordinate[2] = $1.coordinate[2];
38     $$.attachPoints[3] = $1.attachPoints[2];
39     $$.attachPoints[4] = $1.attachPoints[3];
40   }
41   {
42     preserve(2);
43   }
```

Coming up with this grammar from the language specification is not trivial. The programmer, while writing the productions, needs to keep track of all the constraints, the order of the symbols, and needs to deal with a syntax which might be too verbose. If there are many relations between two symbols (as per the condition production) the programmer should choose how to define the driver and the tester sequences on his own, keeping in mind all the previously listed constraints. Moreover, while writing the productions, hard constraints should be checked.

In the next chapter, we show how a new grammar formalism can relieve the user from these tasks and make him/her focus only on the language itself.

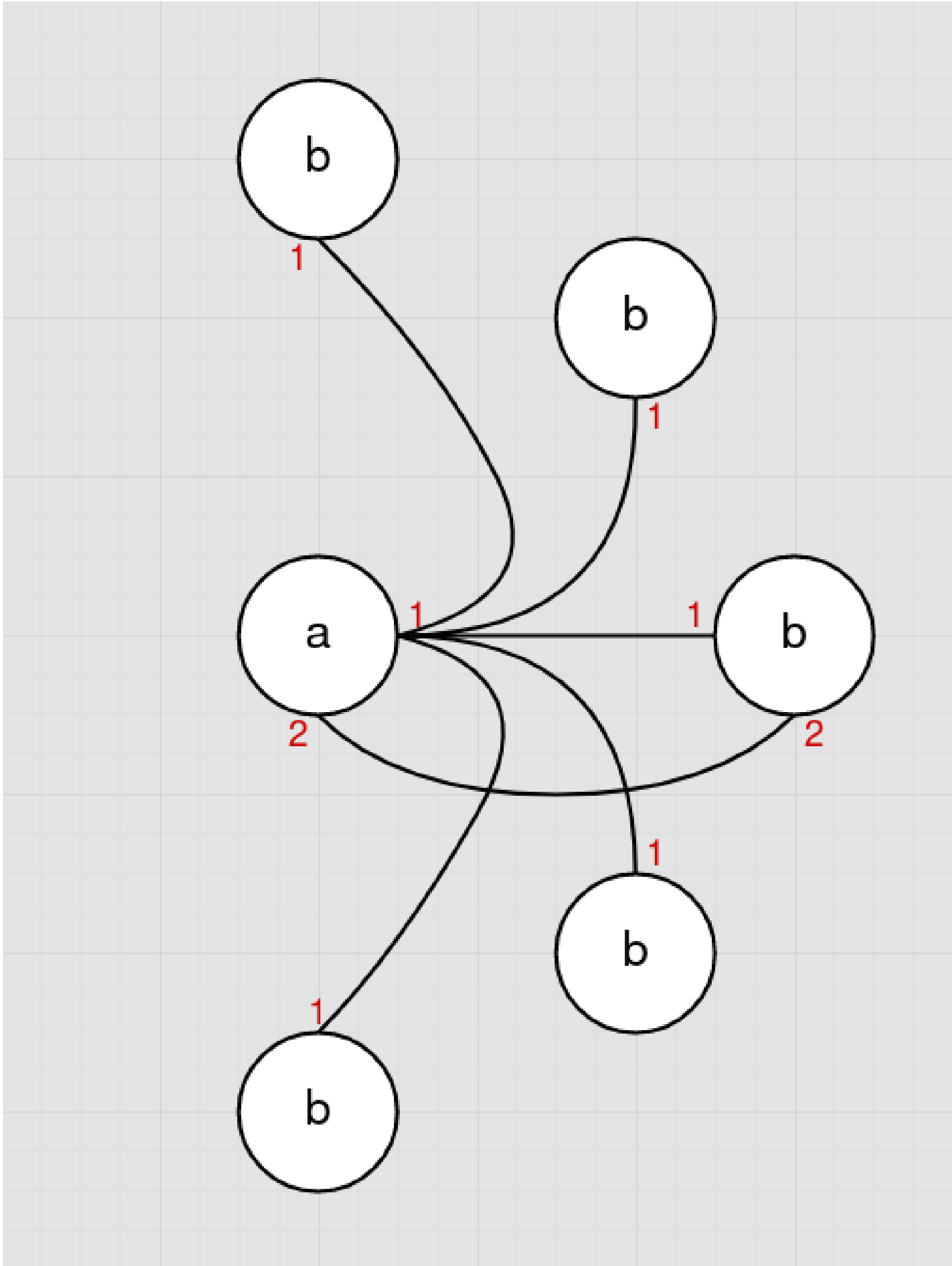


Figure 3.2: An example of input that with wrong productions might cause performance issues.

Chapter 4

Multidimensional Grammars

In this chapter, we redefine the Multidimensional Grammar formalism (MG) first proposed in Piscitelli's master thesis [10]. Multidimensional Grammars are a formalism to easily model custom visual programming languages. This formalism is proposed as an easy to learn and to use alternative to other VPL formalisms.

Its declarative syntax makes it possible to hide all the forced requirements of different kinds of parsers, creating an abstract layer that enables the user to only focus on the key aspects of the grammar to be created. It has been found easy to work with both while creating new grammars and while transforming it into other formalisms.

We will first present the new grammar definition and then present the new file structure used to define an MG grammar and lastly we show some examples and comparisons with the state of the art.

4.1 Grammar definition

A **Multidimensional Grammar** G is a tuple (T, NT, S, R, J, TP, P), where:

- T is a finite non-empty set of *symbols* called **terminals**;
- NT is a finite non-empty set of *symbols* called **non-terminals** where $T \cap NT = \emptyset$;
- S is the **starting non-terminal**;
- R is a finite non-empty set of **relations**;
- J is a finite set of **joints**;
- TP is a finite set of **tie-points**;
- P is a finite non-empty set of **productions**.

Symbols and relations are as defined in chapter 2 with the only difference that each relation is either symmetric, as for example *link*, or has an inverse, as for example, *left* is the inverse of *right*[1].

Both *tie-points* and *joints* are used to connect and relate different *symbols attributes* in each production. **Tie-points** define where each production left hand side symbols

[1] The necessity of the relation to either be symmetric or to have an inverse comes from the conversion algorithm, which needs to be able to change symbols' order in the production (more details in chapter 5).

synthesize their *syntactic attributes* from. **Joints** instead are used to define a relation between two symbols. Both of them are defined as a pair (N, A), where:

- N is a *label* representing its unique name;
- A is its *attribute type*.

Each **production** is expressed as

$$A(l_0)\ x_1(l_1)\ \cdots\ x_n(l_n)\ \rightarrow\ y_1(l_{n+1})\ \cdots\ y_m(l_{n+m})$$

where:

- A is a non-terminal symbol;
- x_1 to x_n is a possibly empty list of terminal symbols;
- y_1 to y_m is a nonempty list of terminal or non-terminal symbols;
- l_0 to l_{n+m} are lists of labels in $J \cup TP$ and each l_i is named **symbol argument list**;
- $A(l_0)\ x_1(l_1)\ \cdots\ x_n(l_n)$ is called the **left part** of the production;
- $y_1(l_{n+1})\ \cdots\ y_m(l_{n+m})$ is called the **right part** of the production.

Each **symbol** also has a semantic type and each **production** might have semantic instructions. This last aspect is mostly dependant on the language semantics and doesn't influence the grammar itself, so they have been left untouched as defined in [10].

4.1.1 Grammar syntax

Multidimensional Grammars will be defined using the already existing file syntax slightly modified to work with the latest edits. Listing 4.1 shows an example of grammar with all the needed syntax. First, lines 1-2 show an empty section used to write parser dependent code for semantic actions, something we will not use in this dissertation.

The block of code starting at line 4 defines the **relations** section. In this case, we have two relations: a **link** which is symmetric and relates two syntactic attributes of type **AttachPoint**; and a **left** relation, with its inverse **right**, separated using a colon.

Line 9 starts the **terminals** section. Here we have two terminals, both with semantic type **string**. Multiple syntactic attribute types might be added, as in the case of relations, using a comma, or, if they are equals, using a colon along with an integer representing the number of attributes, as in the case of the **node** terminal. At line 14 we have the declaration of **non-terminals**, similar to the terminals one. The first non-terminal is used as the starting one.

The **rules** section is composed of three main parts. The first two are used to declare tie-points and joints. In the listing, the labels x and y are declared as tie-points, $l1$ as a joint of type **link** and $r1$ and $r2$ as joints of type **right**.

Lastly, we have the actual **productions**. Multiple productions with the same left part can be separated using a pipe. Each tie-point used in a production must appear once in the left part and once in the right part, while instead joints may appear either in the right part or in the left part. Multiple tie-points or joints can be used as a single symbol argument using a colon as shown in line 26 and if an argument doesn't need to be used it can be left blank using an underscore.

Listing 4.1: Example of MG language to illustrate its syntax

```
({
})

relations {
    link(AttachPoint, AttachPoint);
    left(Coordiantes, Coordinates) : right(Coordinates, Coordinates);
}

terminals {
    box(Coordinates);
    node(AttachPoint:2);
}

non-terminals {
    S;
    Segment(AttachPoint:2);
}

rules {
    tiepoint x, y;

    link l1;
    right r1, r2;

    S ->
        box(r1) box(r1:r2) box(r2) ;

    Segment(x, y) ->
        node(x, l1) node(l1, y) |
        node(x, l1) Segment(l1, y) ;
}
```

4.2 An Example of Multidimensional Grammar

Recalling the example of Visual Language from section 2.1, we can describe it through a Multidimensional Grammar in an easy way by just concentrating on the productions, without having to focus on any of the complex constraints, which are used to be both difficult to understand and to comply. Listing 4.2 shows a Multidimensional Grammar to represent our language whose structure we discuss in the following.

The first thing that we may notice is the presence of a new relation. As a matter of fact, beside the **link** and **below** relation, also defined in the XPG counterpart, we have an **above** relation defined as the inverse of **below**. The need for this relation comes from the possible necessity by the parser to scan symbols in the right parts of the productions in a different order. This aspect is better described in chapter 5 where we show how to transform MG productions into XPG ones. Please note that the **link** relation is symmetric due to its structure, therefore it is the inverse of itself.

Both the terminals and the non-terminals used have already been described in Section 3.3 and should be now self-explanatory. But it's worth to motivate some of the choices made while defining the attributes of some symbols. **Statement** needs to have two **AttachPoint** since it could mark either the start or the end of a condition, same logic applies to **function_call**, which also has an attaching point to *link* to the referred

function, and to **condition**, which might be needed in case of nested *ifs*. **function_name** terminal also has a single **AttachPoint** used to connect to any **function_call**. **Statements** non-terminal has a starting and an ending **Coordinate** used to store the position of the first and the last statement.

Lastly the productions. It's obvious that the following syntax is much shorter and easier to read than the XPG one, but the benefits are not just these. There is no need to explicitly define the role of each relation, this also allows the symbols in the production to be shifted and changed in order while only making sure to use the correct inverse relation when needed. If for example we would rewrite the production `Function(x) -> function_name(b,x) Statement(b)` we could define a new **above joint** called **a** and use the production `Function(x) -> Statement(a) function_name(a,x)`. These kinds of transformations come handy when it comes to transforming this grammar in XPG. Note that, for the rule at line 34 to be applied, a `function_name` terminal must have already been created. To generate the diagram in Figure 2.3 the grammar will then create the function fib(x) before the function main. This implies that the language does not describe mutual recursive inputs. On the other hand, the positional parser built on this grammar will need to be warned that, at the end of the parsing of the input, there might be reintroduced `function_name` tokens still marked as non visited. There are other discussions to be made about the use of this grammar for parsing, also with respect to mutual recursive input, but they are out of the scope of this thesis.

Listing 4.2: MG example from the language defined in chapter 2

```
1  ({
2  })
3
4  relations {
5    link(AttachPoint, AttachPoint);
6    below(Coordinate, Coordinate) : above(Coordinate, Coordinate);
7  }
8
9  terminals {
10   function_name(Coordinate, AttachPoint);
11   statement(Coordinate, AttachPoint:2);
12   condition(Coordinate, AttachPoint:3);
13   function_call(Coordinate, AttachPoint:3);
14 }
15
16 non-terminals {
17   Program;
18   Function(AttachPoint);
19   Statements(Coordinate:2, AttachPoint:2);
20 }
21
22 rules {
23   tiepoint x, y, k, w, j, z;
24
25   below b;
26   link l, l1, l2, l3;
27
28   Program ->
29     Program Function(_) |
30     Function(_) ;
31
32   Function(x) -> function_name(b,x) Statements(b,_, _, _, _) ;
```

```
33
34   Statements(x,y,k,w) function_name(j,z) ->
35     function_call(x:y,k,w,l) function_name(j,z:l) ;
36
37   Statements(x,y,k,w) ->
38     condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3) Statements(_,y,l3
         ,w) |
39     statement(x:y,k,w) |
40     Statements(x,b,k,_) Statements(b,y,_,w) ;
41 }
```

4.3 Notes on the comparison with other formalisms

Our multidimensional grammar formalism is not the first one created to address visual language representations and definitions. Many other formalisms exist in literature. Here we briefly present some of the most common alternatives and informally compare MG with them.

The focus of this comparison is on the usability and the class of generable visual languages.

4.3.1 Reserved Graph Grammar

Reserved Graph Grammar or RGG is a formalism for visual programming languages [11]. Its strength derives from the ability to define context-sensitive languages and to be the basis for building parsers that most of the time have a polynomial complexity. The main limitation of this formalism is that it can only represent a restricted class of visual programming languages, the diagrammatic one. In other words, the only relation it can represent is the *link* one.

Translating RGG grammars to MG requires three main steps.

Firstly, the non-terminals and terminals of the RGG grammar should be copied to MG. Each MG symbol should have an *AttachPoint* as a syntactic attribute mapped to each of the *vertices* of its RGG counterpart. Eventually, if a symbol uses its *super vertex* in any of the replacement rules then it should also be mapped to another *AttachPoint* of the corresponding MG symbol.

Secondly, after having mapped each vertex of each RGG symbol to each AttachPoint of each MG symbol, a *starting non-terminal* `S` needs to be added.

Lastly, all the replacement rules need to be transformed into their equivalent MG syntax. Each RGG rule has a left and a right graph, which intuitively correspond to the left and the right part of its equivalent MG production. The non-terminal in the left graph will be used as *left non-terminal*, all the other terminals will follow it without any specific order in the left part of the MG production. Each symbol in the right graph should be added to the right part of the production without any specific order. Each added symbol should have joints representing all the link relations between vertices. Finally, tie-points should be added according to the numerical labels written inside each vertex.

Figure 4.1 presents an example grammar for process flow diagrams expressed as RGG and Listing 4.3 represents the same grammar using the MG formalism.

Listing 4.3: MG translation of RGG example

```
1 ({
```

```
2  })
3
4  relations {
5    link(AttachPoint, AttachPoint);
6  }
7
8  terminals {
9    start(AttachPoint);
10   end(AttachPoint);
11   assign(AttachPoint:2);
12   if(AttachPoint:3);
13   endif(AttachPoint:2);
14   fork(AttachPoint:2);
15   join(AttachPoint:2);
16   send(AttachPoint:3);
17   receive(AttachPoint:3);
18 }
19
20 non-terminals {
21   Program;
22   Statement(AttachPoint:2);
23 }
24
25 rules {
26   tiepoint j, k, x, y, w, z;
27   link l1, l2, l3;
28
29   Program -> start(l1) Statement(l1,l2) end(l2) ;
30
31   Statement(x, y) ->
32     assign(x, y) |
33     if(x, l1, l2) Statement(l1, l3) Statement(l2, l3) endif(l3, y) |
34     fork(x, l1) Statement(l1, l2) join(l2, y) |
35     receive(x, y, _) |
36     Statement(x, l1) Statement(l1, y) ;
37
38   Statement(x, y) fork(j, k) join(w, z) ->
39     fork(j, k:l1) Statement(x:l1, y:l2) Statement(l1, l2) join(w:l2, z) ;
40
41   Statement(x, y) receive(j, w, z) -> send(x, y, l1) receive(j, w, z:l1) ;
42 }
```

4.3.2 Contextual Hyperedge Replacement Grammars

The Contextual Hyperedge Replacement Grammar or Contextual Grammar (CHG) formalism is presented in [12]. It has been proposed to specifically represent the structure of object-oriented programs as an alternative to diagrams such as UML and similar. Unlike RGG, it is used to define languages on hypergraphs, which are graphs where each edge may connect more than two nodes. Parsing Contextual Grammars is an NP-hard problem but using some techniques the authors claim to have reasonable execution times.

Of course, as for RGG, they are limited to generate graphs, but they allow some more flexibility. The edge of the graph can be labeled, allowing the definition of multiple explicit relations as for example shown in Figure 4.2. These new explicit relations can be converted into MG grammars using one of two approaches: the first one is trivial, for

each labeled edge used in the rules define a new relation and use all of them to create the productions; the second instead only requires the **link** and **overlap** and for each different labeled edge, a new terminal should be added. Both are valid approaches, sometimes it might be easier to apply the first one, such as in the case of grammar to generate input such as the one in Figure 4.2, in other cases however the grammar obtained using the second approach might result more simple and intuitive, the example grammar in Figure 4.3a translated later in this sections uses this last approach.

To translate CHG to MG as a first step we should create a set of non-terminals containing an element for each visual label in CHG. The terminals set is obtained subtracting the visual labels from the set of symbols used in the rules. Unlike RGG, the relations set can contain more than one element, all the relations between symbols used in the rules should be added. Relations are defined on syntactic attributes, therefore each symbol requires them. To compute them it is possible to follow the same approach used for RGG. The MG productions directly derive from the CHG rules which are also split into a left and a right part. For each rule, a production needs to be created. To construct the left part we first select the only visual label in the left part of the CHG rules and make it the production's left non-terminal, then the following contextual symbols are added in the left part of the production without any specific order. The right part of the production is simply computed linearizing the graph in the right part of the rule. At last, tie-points should be added to correctly derive syntactic attributes.

Figure 4.3a shows the productions of an example of CHG grammar and Figure 4.3b shows an example of derivation taken from [12]. As shown in the example, CHG grammars are expressed using a visual syntax where symbols are drawn along with their relations. To convert this language in MG we should first define each used symbol and relation. From the definition in Figure 4.3a we identify:

- two triangles, one pointing down and the other pointing up, representing respectively the **start** terminal and the **end** terminal of a program. Both of them have one attaching point;
- a square representing a **statement** terminal, with two attaching points;
- a rhombus representing a **condition** terminal, with three attaching points;
- an arrow used as **connector** terminal, with two attaching points;
- and finally a non-terminal **D** with only one attaching point.

These symbols are related by using **link**s, that are represented using a line, and **overlap**s, represented overlapping two attaching points.

In Listing 4.4 we have defined a MG grammar that describes the same visual language defined by the CHG grammar in Figure 4.3a. Our formalism allows programmers to work more easily with grammars both by making it easier to recognize their elements, such as symbols and relations, and by allowing explicit writing of all the relationships involved in each production.

Listing 4.4: MG Grammar derived by the CHG grammar in Figure 4.3a

```
1  ({
2  })
3
4  relations {
```

```
5    link(AttachPoint, AttachPoint);
6    overlap(AttachPoint, AttachPoint);
7  }
8
9  terminals {
10   start(AttachPoint:1);
11   end(AttachPoint:1);
12   statement(AttachPoint:2);
13   condition(AttachPoint:3);
14   connector(AttachPoint:2);
15 }
16
17 non-terminals {
18   S;
19   D(AttachPoint:1);
20 }
21
22 rules {
23   tiepoint x, y, z;
24
25   link l1, l2;
26   overlap o, o1, o2;
27
28   S -> start(o) D(o) ;
29
30   D(x) ->
31     connector(x,o) end(o) |
32     connector(x,o) statement(o,l2) D(l2) |
33     connector(x,o) condition(o,l1,l2) D(l1) D(l2) ;
34
35   D(x) connector(y,z) ->
36     connector(x,o) connector(o:y,z) ;
37
38 }
```

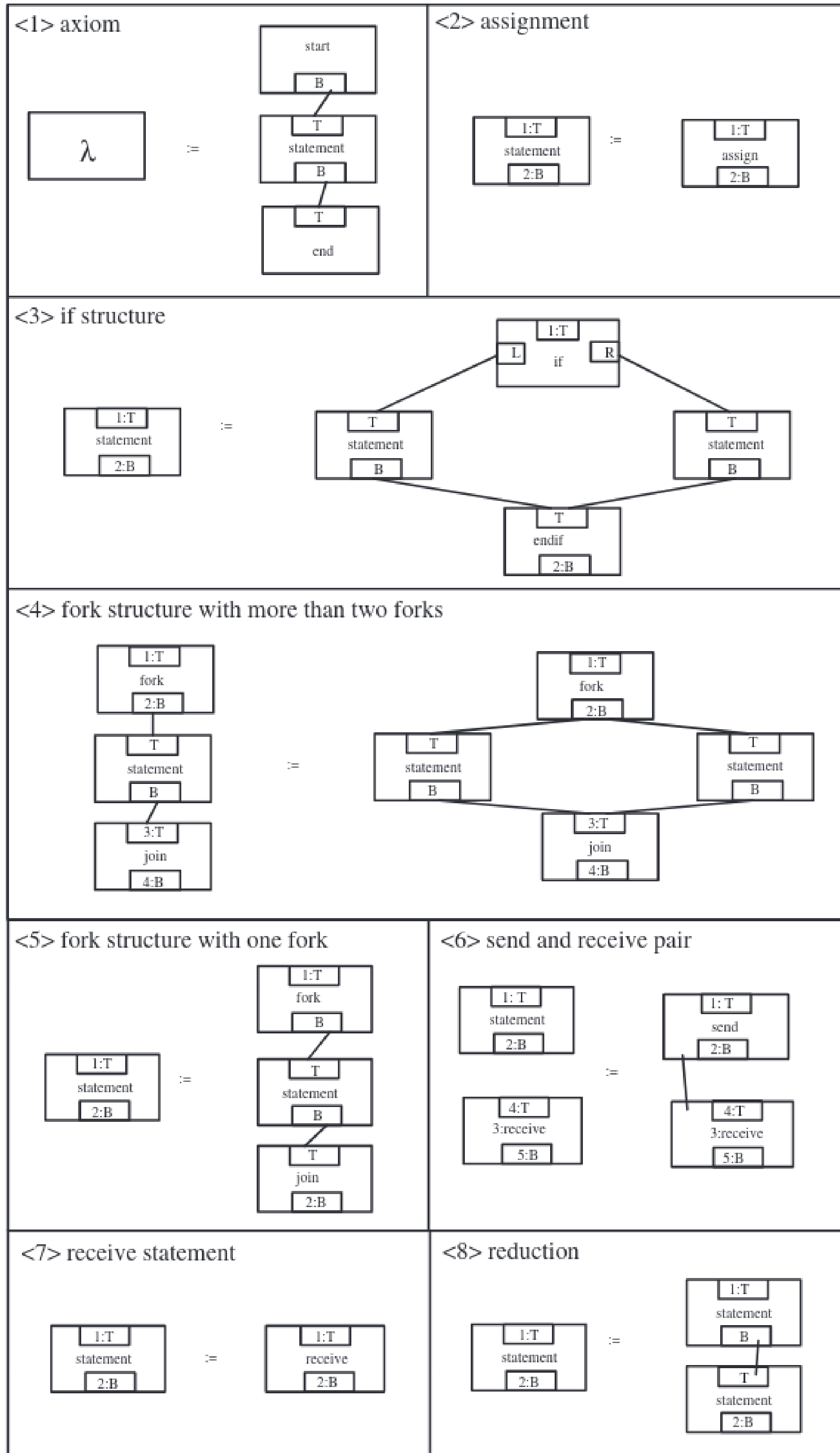


Figure 4.1: A reserved graph grammar specifying process flow diagrams [11].

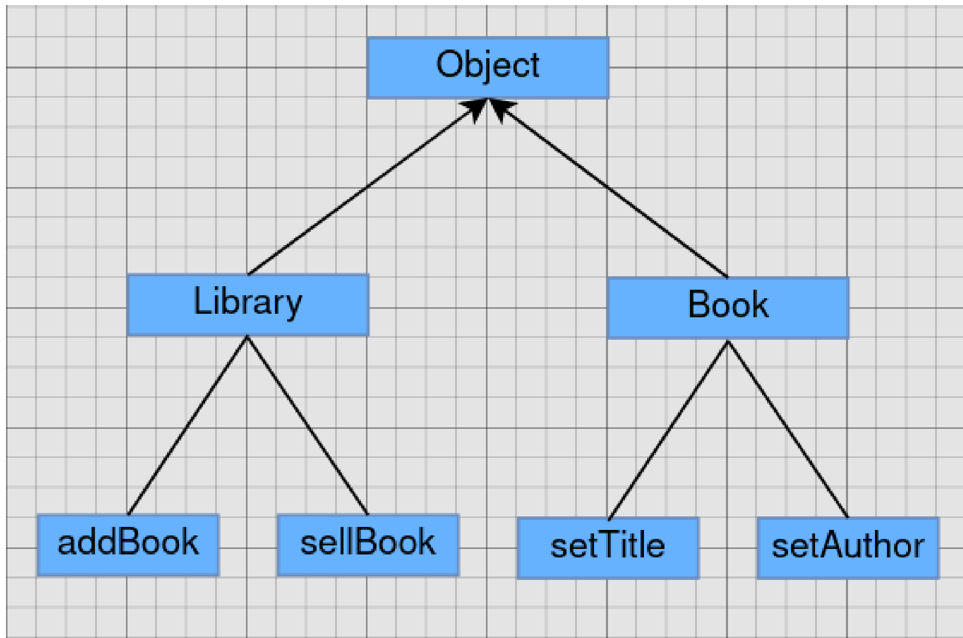


Figure 4.2: Reduced example of an object-oriented program representation. Pointed arrows represent the *extends* relation.

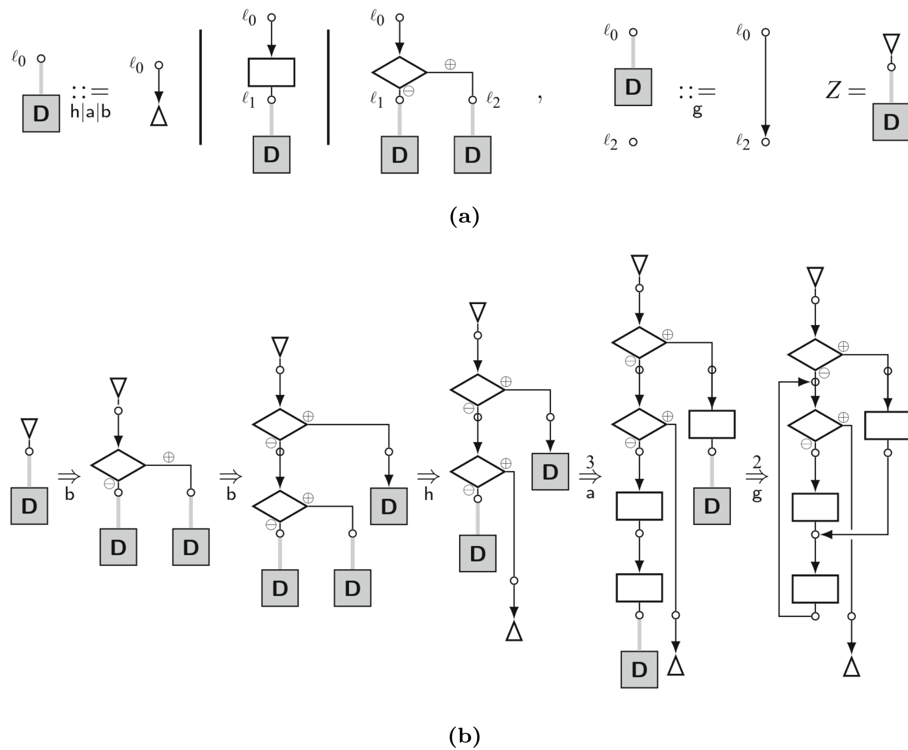


Figure 4.3: (a) Productions generating unrestricted control flow diagrams [12]. (b) A derivation of an unstructured control flow diagram [12].

Chapter 5

Converting Multidimensional to Positional Grammars

To allow programmers to define multidimensional grammars, in [10] it has been used the Eclipse DSL (Domain Specific Language) IDE, along with XText, a DSL framework, and XTend, a Java's dialect, to implement a plugin that allows Eclipse to understand the MG syntax.

Starting from this original project, we first converted it to create an LSP (Language Server Protocol) executable program. The LSP defines the protocol used between an editor or IDE and a language server that provides language features such as auto complete, go to definition, find all references, etc.[13]. By coding only one language server, this allows us to edit MG grammars in all the most common IDEs and code editors (Sublime Text, Visual Studio Code, Vim, Intellij, Eclipse, etc.).

After that, we modified the language server itself to work with our redefinition of MG. In [10], the translation from an MG grammar to PG did not take into account the constraints listed in Section subsection 3.2.3 and did not apply any transformation, often leading to non efficient or not usable positional grammars.

In this chapter, we present the algorithm used to correctly transform each Multidimensional Grammar in its equivalent Positional form complying to all the required constraints aforementioned. We will use the same class structure used and defined in [10] to both represent MG grammars and PGs.

5.1 The algorithm

Since our target specification is the positional syntax working with our bottom-up parser, our grammar conversion algorithm should make our output comply with the rules defined in Section subsection 3.2.3. This version of the algorithm focuses his computation on creating productions with valid entrance points, which is a hard constraint and needs to be respected, and on creating the biggest driver sets possible. Future improvements will be able to also reduce the occurrence of the **any** relation by allowing further rearrangement of the right parts of the productions. We have already seen that the constraints are mostly bound to the productions. In fact, our algorithm will leave unchanged both terminals and relations, create and edit new XPG productions from the MG ones, and eventually add some non-terminals. The algorithm can be divided in two stages. The first one is used to analyze the grammar and retrieve information for each non-terminal on its possible

entrance points. The second stage uses these information to add productions to an XPG grammar.

5.1.1 Non-terminal analysis

In this stage, we need to define three structures, the first one, **outputTab**, will be used by the whole algorithm and the last two, **dependecyTab** and **currentImpossibleSet**, will only be used in this stage.

The **outputTab** is a 3-dimensional table that associates a non-terminal **nt**, a set **entranceSet** and a production **prod** to a list of new productions equivalent to **prod**. In the case the couple (nt, entranceSet) is not possible, for each production the list will be set to empty. In the pseudocode, this property is set through the line `mark outputTab[ nt, entranceSet] as impossible`.

The following production p:`A(x, y) -> B(_, x, y)` can use the set $\{0,1\}$ as entrance set iff the non-terminal B can use the set $\{1,2\}$ as entrance set. Since most of the productions added in all the lists of the previous table might have similar dependencies, we use the **dependencyTab** table to keep track of all of the dependencies. It associates a couple (non-terminal, entrance set) to a list of new productions and their relative non-terminal, entrance set and initial production. Continuing the previous example, if p were a newly added production, we would add to the `dependencyTab[B, {1, 2}]` list a pointer to the non-terminal `A`, the entrance set $\{0,1\}$, the original production and the newly added production p.

The **currentImpossibleSet** is a set used to store, through each iteration of the algorithm, couples of non-terminals and entrance sets that are found to be impossible, so that at last the algorithm can check impossible combinations, find the productions that depend by them using the **dependencyTab** and delete them.

Listing 5.1: 1st stage algorithm

```
1  Input: A multidimensional grammar
2  Output: outputTab
3
4  let outputTab be a table
5  let dependencies be a table
6
7  let x be the max number of attributes that a non-terminal has
8  for i in range [1,x]
9      for each non-terminal nt with at least i attributes
10
11         let currentImpossibleSet be an empty set
12
13         for each possible set s of attributes of nt of cardinality i
14
15             if i is not 1
16                 if it exists a subset s' of s such that outputTab[nt, s'] is impossible
17                     mark outputTab[nt,s] as impossible
18                     break
19
20             for each production p with left-part nt
21                 let ss be a set of all symbols in the right part tied to the attributes
                       in s of nt
22                 if ss is empty
23                     mark outputTab[nt, s] as impossible
24                     add (nt, s) to currentImpossibleSet
```

```
25                      break
26                  else if there is a terminal t in ss
27                      let p' be a production equivalent to p but with t as first element
28                      add p' to the list outputTab[nt, s, p]
29                  else
30                      for each element nt' of ss
31                          let p' be a production equivalent to p but with nt' as first
                                element
32                          add p' to the list outputTab[nt, s, p]
33                          let s' be the set of attributes used by nt' to connect to s of nt
34                          add to the list dependecies[nt', s'] the tuple (nt, s, p, p')
35
36          if i is not 1
37              while currentImpossibleSet is not empty
38                  push an element (nt', set') from currentImpossibleSet
39                  for each tuple (nt, s, p, p') in dependencies[nt',s]
40                      remove p' from list outputTab[nt, s, p]
41                      if list tab[nt, s, p] is empty
42                          mark outputTab[nt, s] as impossible
43                          add (nt, s) to currentImpossibleSet
44
45  return outputTab
```

In Table 5.1 and Table 5.2 we have schematized the output of this first phase using the grammar in Listing 4.2 as input. We will discuss this example to give a clear idea on what our algorithm does.

First, we notice that the non-terminal **Program** has not been evaluated. This happens when a non-terminal does not have any syntactic attribute. Any symbol without syntactic attributes can be only related to other symbols by using the **any** relation and an empty set as entrance points, which could in turn lead to performance issues. In our case, however, **Program** is only used as starting non-terminal and its use cannot be avoided.

Non-terminals with only one attribute still need to be processed. Even though we are not interested in finding the bigger entrance point set since there is only one attribute, it might still be necessary to correctly convert the productions to use that only entrance point as driver. In our example we have been lucky and the single production related to **Function** was already in a correct form.

The last thing to remember is that each cell of **equivalent production** is a list. In most of the cases, this list always contains one element. However, if the input grammar has many link relations, it is more likely that equivalence operations would create different equivalent productions. For example, the production `S(x) -> A(x:l) B(l)`, given that S, A and B are non-terminals with only one attaching point attribute, that x is a tie-point and that l is a link joint, might both be left as it is or converted to `S(x) -> B(x:l) A(l)`.

Table 5.1: Productions of the grammar in Listing 4.2

Name	Production
Prod 1	Program ->Program Function(_)
Prod 2	Program ->Function(_)
Prod 3	Function(x) ->function_name(b,x) Statements(b,_)
Prod 4	Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)
Prod 5	Statements(x,y,k,w) ->statement(x:y,k,w)
Prod 6	Statements(x,y,k,w) ->Statements(x,b,k,) Statements(b,y,_,w)
Prod 7	Statements(x,y,k,w) ->condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3) Statements(_,y,l3,w)

Table 5.2: outputTab of grammar in Listing 4.2

non-terminal	entrance points	production	equivalent productions
Program	No attaching points		
Function	{0}	Prod 3	[Function(x) ->function_name(b,x) Statements(b)]
Statements	{0}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(x,b,k,_) Statements(b,y,_,w)]
		Prod 7	[Statements(x,y,k,w) -> condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3) Statements(_,y,l3,w)]
	{1}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(b,y,_,w) Statements(x,b,k,_)]
		Prod 7	[Statements(x,y,k,w) -> Statements(_,y,l3,w) condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3)]
	{2}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(b,y,_,w) Statements(x,b,k,_)]
		Prod 7	[Statements(x,y,k,w) -> condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3) Statements(_,y,l3,w)]
	{3}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(b,y,_,w) Statements(x,b,k,_)]
		Prod 7	[Statements(x,y,k,w) -> Statements(_,y,l3,w) condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3)]
	{0,2}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(x,b,k,_) Statements(b,y,_,w)]
		Prod 7	[Statements(x,y,k,w) -> condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3) Statements(_,y,l3,w)]
	{1,3}	Prod 4	[Statements(x,y,k,w) function_name(j,z) ->function_call(x:y,k,w,l) function_name(j,z:l)]
		Prod 5	[Statements(x,y,k,w) ->statement(x:y,k,w)]
		Prod 6	[Statements(x,y,k,w) ->Statements(b,y,_,w) Statements(x,b,k,_)]
		Prod 7	[Statements(x,y,k,w) -> Statements(_,y,l3,w) condition(x,k,l1,l2) Statements(_,_,l1,l3) Statements(_,_,l2,l3)]

5.1.2 Grammar transformation

The second phase is easier and straight forward. The desired output of this final step is an XPG valid grammar that generates the same visual language as the input Multidimensional Grammar. Of course, we will also use the **outputTab** generated previously.

The algorithms starts off picking the first non-terminals and recursively build up the final grammar. For each non-terminal it runs through all the productions and one at the time translates them in their XPG version. After the translation, each remaining non-terminal needs to be converted. Each of them is recursively used into the same algorithm, which first checks if the same instance has already been executed using some simple memoization and eventually repeats the whole process.

There is a case in which the **secondStage** algorithm returns null. If the non-terminal `nt` has some attributes and $|ep| > 1$, the **outputTab** might return that the set is impossible. In this case, the algorithm reacts by reducing its driver size. There are different ways to reduce the driver set, the solution proposed in this example (Line 17-21) is the easiest one, randomly choose an element from the set and remove it. Here we presented this approach trying to simplify this algorithm, which at first sight seems already quite intricate. More advanced techniques can be defined, for example, using the added table to check if any subset has already been parsed might reduce the number of new non-terminal and productions added to the final grammar.

Listing 5.2: 2nd stage algorithm

```
1  function secondStage(nt, ep)
2      if added(nt, ep) is not null
3          return added(nt, ep)
4
5      if nt have attributes
6          find (nt, s) such that outputTab(nt,s) is non marked impossible and ep is a
               subset of s
7          if nt is null
8              return null
9          let newNt be a new non-terminal called nt_s
10     else
11         newNt = nt
12
13     for each production p of nt
14         translate p to xpg
15         replace nt with newNt in the left part of p
16         for each non-terminal nt' and its entrance point ep'
17             do
18                 let ntToChange = secondStage(nt', ep')
19                 if(ntToChange is null)
20                     remove an element from ep'
21             while ntToChange is null
22
23             replace nt' in p with ntToChange
24             replace original entrance point with ep' in p
25         add the p to the xpg grammar
26
27         add newNt to the xpg grammar
28         added(nt, ep) = newNt
29     return newNt
30
31
```

```
32 secondStage(Program, {})
```

Finishing our example, following this algorithm, the initial grammar written in Listing 4.2 becomes the XPG grammar in Listing 5.3.

Listing 5.3: Output XPG grammar

```
1  StartSymbol: Program;
2
3  StartMovement: Sp;
4
5  nonTerminals(
6    Program;
7    Function__0;
8      Statements__1_3;
9  )
10
11 terminals(
12   function_name(AttachPoint:1,Coordinate:1);
13   statement(AttachPoint:2,Coordinate:1);
14   condition(AttachPoint:3,Coordinate:1);
15   function_call(AttachPoint:3,Coordinate:1);
16 )
17
18 movements(
19   Sp;
20   link(AttachPoint,AttachPoint);
21   below(Coordinate,Coordinate);
22   above(Coordinate,Coordinate);
23 )
24
25 nonterm Program( ()
26   ->  Function__0;
27   {  }
28   { }
29
30   -> Program any(0,0) Function__0;
31   {  }
32   { }
33 )
34
35 nonterm Function__0( (AttachPoint:1)
36   ->  function_name below(1,1) Statements__1_3;
37   {
38     $$.attachPoints[1] = $1.attachPoints[2];
39   }
40   { }
41 )
42
43 nonterm Statements__1_3( (AttachPoint:2,Coordinate:2)
44   ->  function_call link(4,2) function_name;
45   {
46     $$.coordinate[1] = $1.coordinate[1];
47     $$.coordinate[2] = $1.coordinate[1];
48     $$.attachPoints[3] = $1.attachPoints[2];
49     $$.attachPoints[4] = $1.attachPoints[3];
50   }
51   {
52     preserve(2);
```

```
53    }
54
55    -> Statements__1_3 any(0,0) condition link(3,3) Statements__1_3 link(4,4) & link
          (3,4)(-2)^link(4,3)(-1) Statements__1_3;
56    {
57      $$.coordinate[1] = $2.coordinate[1];
58      $$.coordinate[2] = $1.coordinate[2];
59      $$.attachPoints[3] = $2.attachPoints[2];
60      $$.attachPoints[4] = $1.attachPoints[4];
61    }
62    { }
63
64    -> Statements__1_3 below(1,2) Statements__1_3;
65    {
66      $$.coordinate[1] = $2.coordinate[1];
67      $$.coordinate[2] = $1.coordinate[2];
68      $$.attachPoints[3] = $2.attachPoints[3];
69      $$.attachPoints[4] = $1.attachPoints[4];
70    }
71    { }
72
73    -> statement;
74    {
75      $$.coordinate[1] = $1.coordinate[1];
76      $$.coordinate[2] = $1.coordinate[1];
77      $$.attachPoints[3] = $1.attachPoints[2];
78      $$.attachPoints[4] = $1.attachPoints[3];
79    }
80    { }
81  )
```

5.2 Correctness

Definition 2. *Two* ***productions*** *are equivalent if the use of one or the other in any multidimensional grammar does not change the generated visual language.*

First, we note that given a production, its right part is always the result of a linearization operation on a graph. If two productions are equals except for their right part and the two graph represented by the right part are isomorphic, then the two productions are said to be equivalent. As an example, if considering the two productions in Listing 5.4 both their right part represent the same graph shown in Figure 5.1, therefore they are equivalent.

Listing 5.4: Two equivalent productions

```
1  link l1, l2, l3;
2
3  S ->
4      a(l1,l2) b(l2,l3) c(l3,l1) |
5      b(l2,l3) a(l1,l2) c(l3,l1) ;
```

Definition 3. *Let $T(i,j)$ be a transformation on a generic production p*

$$p: \; S \rightarrow X_0(A_0) \cdots X_{n-1}(A_{n-1})$$

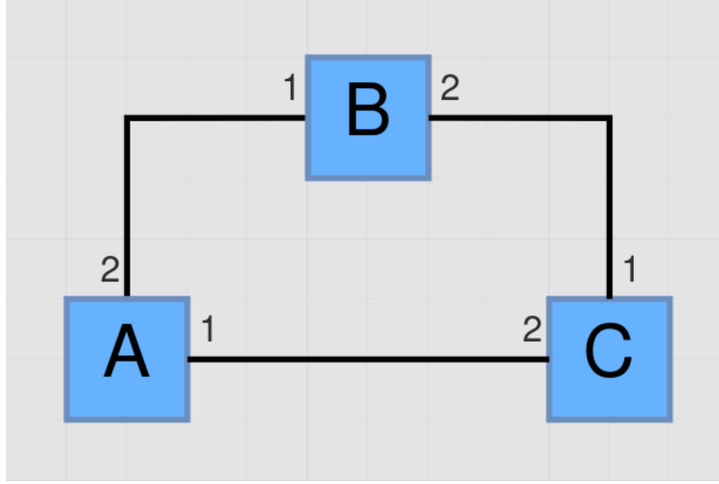


Figure 5.1: The graph represented by the right part of the productions in Listing 5.4

where S represents a generic left part of the production, each X_k represents a symbol, each A_k represents the list of tie-points and joints of an instance symbol, $i \geq 0$, $i < j$ and $j < n$. The result of the transformation is the production p'

$$p' : S \rightarrow X_0(A_0) \cdots X_{i-1}(A_{i-1})\, X_{i+1}(A'_{i+1}) \cdots X_{j-1}(A'_{j-1})\, X_i(A'_i)\, X_j(A_j) \cdots X_{n-1}(A_{n-1})$$

where each A'_k is the same as A_k except for all the joints connected with A_i which are replaced with joints of the inverse relations, for each $i <= k <= j - 1$.

Intuitively the transformation $T(i, j)$ on a production p shifts the $i - th$ symbol of the right part into the position j and changes all the joints between the shifted symbol and each joint of all the symbols with index between $i + 1$ and $j - 1$ with an inverse relation joint. For example, in Listing 5.4 the production at line 5 is the result of the transformation $T(1, 2)$ of the production at line 4.

Lemma 1. *Applying any transformation $T(i, j)$ to any generic production will always result in an equivalent production.*

Proof. The proof is straightforward. Both the production before and after the transformation have the same symbols and the same relations. The right part changes, but the resulting graph is still isomorphic to the original one, therefore the productions are equivalent. □

When using the term **productions of a non-terminal nt** we will reefer to all the productions of a grammar having nt as left non-terminal.

Definition 4. *Two **non-terminals** are equivalent if the use of one or the other in any multidimensional grammar does not change the generated visual language.*

Definition 5. *Given two non-terminals `nt` and `nt'`, if for each production of `nt` there exists another equivalent production of `nt'` and vice versa, then `nt` and `nt'` are equivalent.*

We recall, as defined in Section 3.2.3, that in this thesis a positional grammar is said to be valid if its productions are valid. It is then required that the entrance set of each non terminal in the left side of a production is synthesized using the first symbol of the right part of that production.

Theorem 1. *Applying the algorithm described in Section 5.1 on a multidimensional grammar will always result in a valid positional grammar and both the grammars generate the same visual language.*

Sketch of Proof. This proof is split in two parts: first, we need to show that the output positional grammar generates the same visual language as the input multidimensional grammar; then we demonstrate that the output grammar is valid.

The algorithm always starts from the starting symbol and recursively adds non-terminals as it founds them in the productions. Since the basic assumption here is that the considered grammars are always well formed, i.e., all of their non-terminals are reachable and always derive a (sub)sentence, the above statement grants us that all the grammar non-terminals are considered.

For each non-terminal `nt`, if it has no attributes, we leave it as it is, otherwise we replace it with a new non-terminal `nt'`. Due to the construction of what has been previously referred to as the **outputTab** we know that for each production of `nt` there exists another equivalent production `p'` of `nt'` constructed applying the transformation T multiple times, therefore `p` and `p'` are equivalent and `nt` and `nt'` are equivalent. Therefore, after translating the grammar in positional with drivers sets computed using **outputTab**, the two grammars generate the same visual language.

Lastly, having shown how to obtain an equivalent positional grammar, we need to prove that the entrance set hard constraint is always verified. This is trivial since while searching for a non-terminal to be replaced we used **outputTab** just to check if there exists a non-terminal satisfying these requirements, if this is not the case, the algorithm simply reduces the driver size until it finds a *valid* non-terminal. Eventually, the driver may become of size one, but in this case, due to the construction of **outputTab** we are guaranteed to always have at least one equivalent non-terminal that might be used with at least every single attribute. Therefore we will always be able to create a valid positional grammar. □

5.3 Implementation

The base implementation created in [10], from which we started working, was an Eclipse DSL Plugin that implemented some IDE features, such as contextual suggestion, inline errors display and many others. It is mostly written in Java and XTend, a Java's dialect. The DSL was written using the XText framework. It was created to do only a direct translation from MG to XPG and only in the Eclipse environment.

5.3.1 Programs structure

Since IDEs and code editors continuously change and each programmer has his favorite one, an effort has been made to make the language, along with its syntax helper available on all the most popular IDEs. This was possible thanks to the Language Server Protocol (LSP)[13], which defines a protocol to communicate to many editors using only a single language server. Thanks to XText supporting LSP as output, in order to make this change in the original project, it was only necessary to edit most of the internal settings, the actual codebase remained with the same original structure. As shown in Figure 5.2 the source project is used to generate two programs.

The first is the language server, it is used to provide error highlighting, contextual suggestions, and other functionalities while writing the grammar. It can be used by any editor that supports LSP either using the integrated editor's settings (such as in Intellij Rider[14]) or writing a small client plugin that only needs to establish a socket connection with the server (such as VSCode[15]).

The second program is the actual tool for converting MG grammars. It offers a CLI interface that makes it possible to use it as standalone or to configure it with various IDE settings such as adding it to External Tools and binding it to a keystroke or adding it to the File Watcher to launch the conversion each time an MG file is saved.

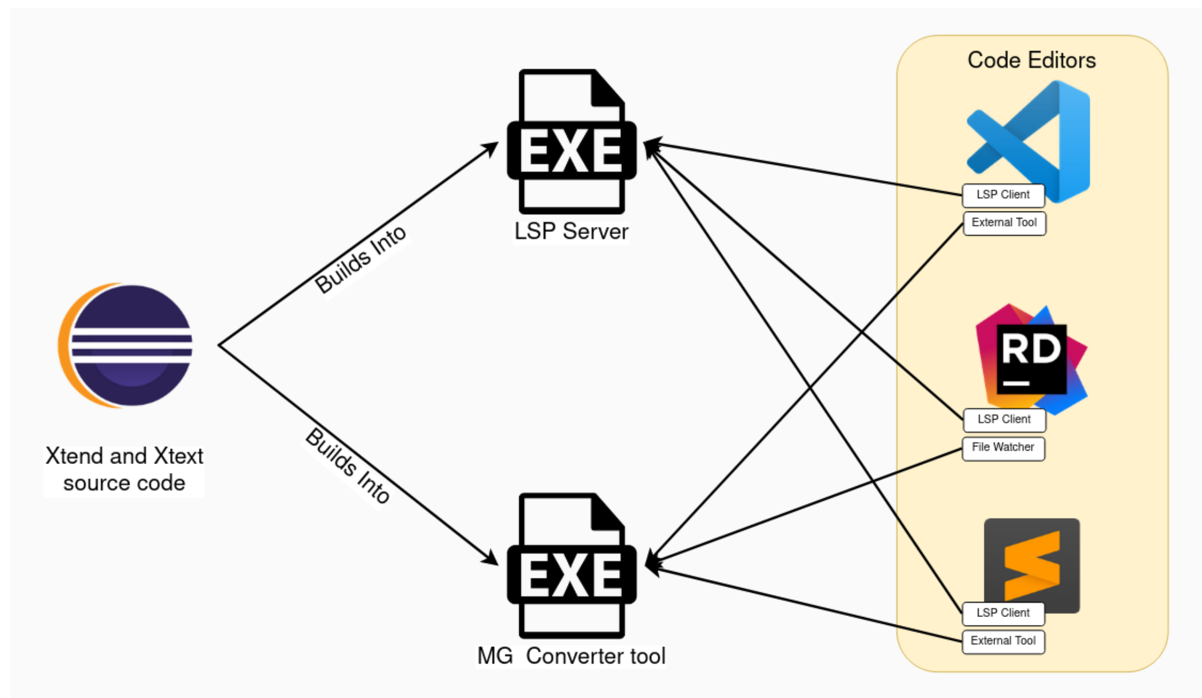


Figure 5.2: Output executable structure.

5.3.2 LSP validations and suggestions

Having made the DSL work with other code editors without major changes we managed to preserve all the validations already implemented in [10]. Old validations have been updated or removed to reflect the differences in the new MG formalism and many new validations have been added. All this work allows programmers to have contextual feedback while still writing the grammar as shown in Figure 5.3a.

Another improvement made possible by LSP was to create contextual suggestions. As shown in Figure 5.3b it is now possible to define templates to eventually select in any specific code section. These templates contain sample code that can be easily selected and modified using the tab key.

5.3.3 Algorithm implementation

The biggest issue with the old version, however, is that its translations often generate invalid grammars. Moreover, the declarative MG syntax made it even more difficult for

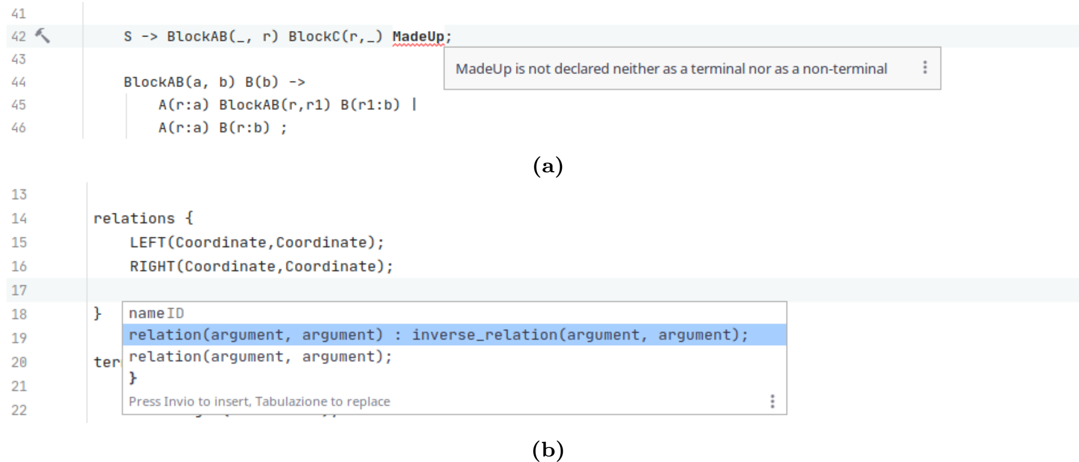


Figure 5.3: (a) Error highlighting while writing a MG grammar. (b) Contextual suggestions while writing a MG grammar.

the programmer to think to a valid grammar. In our work, we extended the code base to also include the converting algorithm above presented.

The MG and the XPG data structures are the same used in [10] and will not be re-explained. What we need to present, however, is the conversion algorithm implementation and his structure.

Multidimensional Transformer is the main class that manages the translation, its public **transform** method deals with the two phases of the algorithm. The first one is performed by the method **availableEntryPoint** which is presented in Listing 5.5 and the second one by the method **editProductionsToUseNewNonTerminal** which is presented in Listing 5.6. This section of code uses **dependency** and **currentImpossibleSets** just as explained above.

Listing 5.5: availableEntryPoint method

```
private void availableEntryPoint() {
    // Each non terminal is mapped to a set of index of arguments usable as driver
        mapped to a list of production to use
    Dependencies dependencies = new Dependencies();

    // Mainly used to store couple of NonTerminals and index sets of arguments that
        are not possible
    ImpossibleSets currentImpossibleSets = new ImpossibleSets();

    int maxAttachPointsNum = initialNonTerminals.stream().max(Comparator.comparing(nt
        -> nt.getNumAttachPoints())).get().getNumAttachPoints();

    IntStream.range(1, maxAttachPointsNum+1).forEachOrdered(setSize -> {

      currentImpossibleSets.initialize();

      List<NonTerminalInstance> ntWithEnoughAttachPoints = initialNonTerminals.stream
          ().filter(nt -> nt.getNumAttachPoints() >= setSize).collect(Collectors.toList
          ());
      for(NonTerminalInstance nti : ntWithEnoughAttachPoints) {

        List<Set<Integer>> possibleSetsOfSize = Combinator.getCombinations(nti.
            getNumAttachPoints(), setSize);
        for(Set<Integer> set : possibleSetsOfSize) {
```

```
19
20          //If the current set has some impossible subset then it should be simply
                marked as impossible
21          if(setSize != 1 && nti.hasImpossibleSubsetOf(set)){
22            nti.addImpossibleSet(set);
23            currentImpossibleSets.addIfNotAlredyParsed(nti, set);
24            continue;
25          }
26
27          EquivalentNonTerminal equivalentNonTerminal = nti.
                getEquivalentNonTerminalOfSet(set);
28          //line 19 - 29
29          for(Production prod : nti.getNonTerminal().getProductions()) {
30
31            List<PredicateParams> leftParams = prod.getLeftHandSideParams();
32            Set<String> productionTiePoints = set.stream().map(index -> leftParams.get(
                  index).getName()).collect(Collectors.toSet());
33
34            List<ProductionRHSPredicate> rightPredicates = prod.
                  getRightHandSidePredicates();
35
36            for(int predicateIndex = 0; predicateIndex < rightPredicates.size();
                  predicateIndex++) {
37              ProductionRHSPredicate currentRightP = rightPredicates.get(predicateIndex
                    );
38
39              List<ArrayList<PredicateLabel>> currentRightPParams = currentRightP.
                    getParams();
40              Map<String, Integer> currentJoints = new HashMap<String, Integer>();
41              for(int indexOfPredicate = 0; indexOfPredicate<currentRightPParams.size();
                     indexOfPredicate++) {
42                List<PredicateLabel> labels = currentRightPParams.get(indexOfPredicate)
                      ;
43                for(PredicateLabel jl : labels) {
44                  currentJoints.put(jl.getName(), indexOfPredicate);
45                }
46              }
47
48              if(currentJoints.keySet().containsAll(productionTiePoints)) {
49                try {
50                  Production newProd = (Production) prod.clone();
51                  List<ProductionRHSPredicate> rightPredsToEdit = newProd.
                        getRightHandSidePredicates();
52                  ProductionRHSPredicate toMove = rightPredsToEdit.get(predicateIndex);
53                  rightPredsToEdit.remove(predicateIndex);
54                  rightPredsToEdit.add(0, toMove);
55
56                  checkRelationsInProduction(newProd, toMove, rightPredsToEdit);
57
58                  if( currentRightP instanceof TerminalPredicate) {
59                    equivalentNonTerminal.addEquivalentProduction(prod, newProd, true);
60                    break;
61                  } else {
62                    EquivalentProduction eqProd = equivalentNonTerminal.
                          addEquivalentProduction(prod, newProd, false);
63                    Set<Integer> targetProdSet = productionTiePoints.stream().map(itm ->
                           currentJoints.get(itm)).collect(Collectors.toSet());
64                    dependencies.addDependency(((NonTerminalPredicate) toMove).
```

```
                  getNonTerminal(), targetProdSet, eqProd);
65              }
66
67            } catch (CloneNotSupportedException e) {
68              System.err.println("Unable␣to␣clone␣a␣Production!");
69              e.printStackTrace();
70            } catch (Exception e) {
71              e.printStackTrace();
72            }
73          }
74        }
75        if(equivalentNonTerminal.setImpossibleIfProductionEmpty(prod)) {
76          currentImpossibleSets.addIfNotAlredyParsed(nti, set);
77          break;
78        }
79      }
80    }
81  }
82
83  if(setSize != 1) {
84    NonTerminalInstance nti;
85    while((nti = currentImpossibleSets.getNonTerminalInst()) != null) {
86      Set<Integer> set;
87      while(( set = currentImpossibleSets.getSetOfNT(nti)) != null){
88        currentImpossibleSets.setAlreadyUsed(nti, set);
89        List<EquivalentProduction> eqProdsToRemove = dependencies.
              getAllDependentProductions(nti.getNonTerminal(), set);
90        if(eqProdsToRemove != null) {
91          for(EquivalentProduction eqProdToRemove : eqProdsToRemove) {
92            EquivalentNonTerminal eqProds = eqProdToRemove.getEqProductions();
93            if(eqProds.removeAndCheckIfImpossible(eqProdToRemove)) {
94              currentImpossibleSets.addIfNotAlredyParsed(eqProds.
                    getNonTerminalInstance(), eqProds.getJointSet());
95            }
96          }
97        }
98        currentImpossibleSets.removeDep(nti, set);
99      }
100   }
101  }
102 });
103 }
```

Listing 5.6: editProductionsToUseNewNonTerminal method

```
1  private void editProductionsToUseNewNonTerminal() {
2      initialGrammar.getNonTerminals().forEach(nt -> {
3        nt.getProductions().forEach(prod -> {
4          final Set<String> usedParams = new HashSet<String>();
5          prod.getLeftHandSideParams().stream().filter(x -> x instanceof
               PredicateTiePoint).forEach(pred -> {
6            usedParams.add(((PredicateTiePoint) pred).getTiePoint().getName());
7          });
8          prod.getRightHandSidePredicates().forEach(pred -> {
9            if(pred instanceof TerminalPredicate) {
10             pred.getParams().forEach(args -> {
11               args.forEach(arg -> {
12                 usedParams.add(arg.getName());
```

```
13          });
14        });
15      } else {
16        final Set<Integer> wantedDriver = new HashSet<Integer>();
17
18        //here for each predicate we retrieve all the possible index of argument
              that could be used as driver
19        pred.getParams().forEach(args -> {
20          args.forEach(arg -> {
21            if(usedParams.contains(arg)) {
22              wantedDriver.add(pred.getParams().indexOf(args));
23            } else {
24              usedParams.add(arg.getName());
25            }
26          });
27        });
28
29        if(!usedParams.isEmpty()) {
30          String nonTerminalName = pred.getSymbol().getName();
31          NonTerminalInstance nti = initialNonTerminals.stream().filter(x -> x.
                getNonTerminal().getName().equals(nonTerminalName)).findFirst().get()
                ;
32
33          Map<Set<Integer>, EquivalentNonTerminal> equivalentNonTerminals = nti.
                getList();
34          Set<Integer> bestSet = null;
35          int bestSizeDriver = 0;
36          for(Set<Integer> currentSet : equivalentNonTerminals.keySet()) {
37            if(bestSizeDriver >= currentSet.size()) continue;
38            Set<Integer> copyWantedDriver = new HashSet<Integer>(wantedDriver);
39            copyWantedDriver.retainAll(currentSet);
40            if(bestSizeDriver <= wantedDriver.size() - copyWantedDriver.size() && !
                  equivalentNonTerminals.get(currentSet).isImpossible()) {
41              bestSet = currentSet;
42              bestSizeDriver = wantedDriver.size() - copyWantedDriver.size();
43            }
44          }
45          if(bestSet != null) {
46            EquivalentNonTerminal bestNonTerminal = equivalentNonTerminals.get(
                  bestSet);
47            ((NonTerminalPredicate)pred).setNonTerminal(bestNonTerminal.
                  getNewNonTerminal());
48
49            //In this case there might be left side in the production that should
                  be changed too
50            if(nti.getNonTerminal().getSyntatticAttributeTypes().size() == 1) {
51              prod.getSizeOnePredicates().forEach(leftPred -> {
52                if(leftPred instanceof NonTerminalPredicate) {
53                  NonTerminalPredicate leftNonTerm = (NonTerminalPredicate) leftPred
                        ;
54                  if(leftNonTerm.getNonTerminal().equals(nti.getNonTerminal())) {
55                    final Set<String> leftParams = new HashSet<String>();
56                    leftNonTerm.getParams().forEach(params -> {
57                      params.forEach(arg -> {
58                        leftParams.add(arg.getName());
59                      });
60                    });
61
```

```
62                              if(leftParams.size() == 1 && usedParams.containsAll(leftParams))
                                    {
63                                ((NonTerminalPredicate) leftPred).setNonTerminal(
                                      bestNonTerminal.getNewNonTerminal());
64                              }
65                            }
66                          }
67                        });
68                      }
69                    }
70                  }
71                }
72              });
73            });
74          });
75        }
```

During the transformation the algorithm uses a list of **NonTerminalInstance**s. Each non-terminal with at least one attribute has its **NonTerminalInstance** object. These objects map each possible subset of the terminal attributes to an **Equivalent-NonTerminal** or to a special object if the subset is impossible, as shown in Table 5.3. The **EquivalentNonTerminal** objects represent the new equivalent non-terminals to use for each different type of entrance sets. Each new non-terminal has a reference to all of its related productions that eventually might be added in the grammar, as shown in Table 5.4. The **EquivalentProduction** simply stores some back reference to retrieve the context in which it should be used and the new target production, shown in Table 5.5. This structure in whole is used to store all the information previously summarized in the **outputTab** table.

Table 5.3: Properties of **NonTerminalInstance**.

Name	Type	Description
nonTerminal	NonTerminal	A reference to the original non-terminal.
numAttachPoints	int	Number of attaching point of the non-terminal used for quick reference.
list	Map<Set<Integer>, EquivalentNonTerminal>	Each set represent an Entrance Set and is related to an EquivalentNonTerminal.

Table 5.4: Properties of **EquivalentNonTerminal**.

Name	Type	Description
nti	NonTerminalInstace	Back reference to the NonTerminalInstance using this object.
newNonTerminal	NonTerminal	Reference to the new non-terminal to use with the new equivalent productions.
jointSet	Set<Integer>	Back reference to the Entrance Set used along with nti.
impossible	boolean	Special value to represent impossible Entrace Sets
equivalences	Map<Production, List<EquivalentProduction>>	Each Production has a list of EquivalentProduction

Table 5.5: Properties of **EquivalentProduction**.

Name	Type	Description
nti	NonTerminalInstace	Back reference to the NonTerminalInstance using this object.
sourceSet	Set<Integer>	Back reference to the Entrance Set used along with nti.
eqProds	EquivalentNonTerminal	Back reference to the EquivalentNonTerminal using this object.
sourceProduction	Production	Old production to be replaced.
finalProduction	Production	New production to use.

5.3.4 Testing

What we mainly want to show in this thesis is that Multidimensional Grammar formalism is handy to work with when it comes to generating visual languages. Our algorithm only serves for translating a multidimensional grammar to a specific kind of positional grammars complying with specific rules that will most likely be prone to changes once new parsers will be created. Therefore, our main goal is to try and detect as many software regressions while developing new features.

This is the reason that leads us to focus mostly on snapshot testing. The testing sub-project contains many examples of different multidimensional languages that cover all the formalism's functionality in different contexts.

Snapshot tests work by comparing the output of the algorithm before and after an edit. If the outputs differ then the test is said to be failing, else the test is passing.

Ideally, after each change in the algorithm or the formalism, all the tests should be executed. For each failing test, the programmer should check if the difference in the output is intended or if it is caused by an edge case he did not take into account. After fixing the code, when all the failing tests are caused by legit changes the snapshot should be updated, the code merged and a new release can be created.

Chapter 6

Conclusion and future developments

The goal of this thesis was to provide programmers with some tools to easily develop applications that use multidimensional languages. We have initially presented visual languages and have shown some of their applications both in the academic and business world.

We first presented the Positional Grammars as a first alternative to generate visual languages. We discovered that PGs can define a wide class of languages and that there exist many algorithms to generate different parsers based on grammar specifications. However, despite these advantages, we have underlined its complexity and difficulty of use caused by its constraints and the operational nature of its syntax.

We have then presented an improvement of an already existing formalism, Multidimensional Grammars. We defined its structure, and, using some examples and comparisons with other formalisms found in literature, we have shown both its handiness and expressive power.

At last, we covered an important hole that was still missing, improving the existing algorithm to convert MG in PG. The algorithm now allows us to always come up with valid grammars, leveraging the user for most of the work and allowing him to exploit the full potential of the renewed declarative syntax making it possible to represent a wide class of visual languages. We have explained the transformation algorithm and then proved its correctness and while doing so, we also have shown two important things: first, it is clear how MG grammars are intuitive to model and work with, also while writing algorithms; second, the same presented approach could be used as a model and reused for many other cases. Better and better visual language parsers will continue to be researched and built; new parsers, new constraints, and even new target syntax will be used and we can now abstract all their features behind this front-end. Besides that, we have also changed the project structure in such a way that it is now possible to build both the conversion tool and an LSP server, which allows us to work with MG having all the typical IDE features (suggestions, errors highlighting, etc.) in all of the most popular IDEs.

Further researches will be now focused on reducing the parsing time as much as possible. Conflicts, especially run-time ones, will be important subjects for future improvements. The goal is to find some new constraints to successfully create run-time conflicts free grammars, which are one of the biggest issues behind parsers' inefficiency. Additionally, different heuristics can be tested to select the most efficient equivalent production if there are more than one alternatives, and applying a rearrangement of the right parts of the equivalent productions could lead to occurrences of the *any* relation. Further developments on the LSP Server should be done to provide additional functionalities such

as on hover definitions, diagnostics suggestions, and, once the protocol will support it, syntax highlighting.

Bibliography

[1] "Scratch - imagine program share." Massachusetts Institute of Technology. [Online]. Available: https://scratch.mit.edu/

[2] "Mit app inventor." Massachusetts Institute of Technology. [Online]. Available: https://appinventor.mit.edu/

[3] "Sap." [Online]. Available: https://www.sap.com/index.html

[4] "Open text." Open Text Corporation. [Online]. Available: https://www.opentext.com/

[5] G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora, "A parsing methodology for the implementation of visual systems," *IEEE Trans. Software Eng.*, vol. 23, no. 12, pp. 777–799, 1997. [Online]. Available: https://doi.org/10.1109/32.637392

[6] G. Costagliola and G. Polese, "Extended positional grammars," in *2000 IEEE International Symposium on Visual Languages, VL 2000, Seattle, Washington, USA, September 10-13, 2000, Proceedings.* IEEE Computer Society, 2000, pp. 103–110. [Online]. Available: https://doi.org/10.1109/VL.2000.874373

[7] G. Costagliola, V. Deufemia, F. Ferrucci, and C. Gravino, "Exploiting XPG for visual languages: Definition, analysis and development," *Electron. Notes Theor. Comput. Sci.*, vol. 82, no. 3, pp. 612–627, 2003. [Online]. Available: https://doi.org/10.1016/S1571-0661(05)82631-3

[8] ——, "Run-time conflict detection in visual language parsing," *J. Comput. Lang.*, vol. 57, p. 100943, 2020. [Online]. Available: https://doi.org/10.1016/j.cola.2020.100943

[9] G. R. Economopoulos, *Generalised LR parsing algorithms.* Royal Holloway, University of London, 2006.

[10] A. Piscitelli, "Multidimensional languages," Master's thesis, Università di Salerno, 2019.

[11] D. Zhang, K. Zhang, and J. Cao, "A context-sensitive graph grammar formalism for the specification of visual languages," *The Computer Journal*, vol. 44, no. 3, pp. 186–200, 2001.

[12] F. Drewes and B. Hoffmann, "Contextual hyperedge replacement," *Acta Informatica*, vol. 52, no. 6, pp. 497–524, 2015. [Online]. Available: https://doi.org/10.1007/s00236-015-0223-4

[13] "Language server protocol," Microsoft. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[14] "Intellij rider." JetBrains. [Online]. Available: https://www.jetbrains.com/rider/

[15] "Visual studio code," Microsoft. [Online]. Available: https://code.visualstudio.com/