View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NORA - Norwegian Open Research Archives

University of Oslo
Department of Informatics

A meta–language for UML concrete graphical syntax

Thomas Hemmingby Espe

Cand. Scient. thesis

28th April 2004

Abstract

This thesis presents the *Graphical Description Language* (GDL), a meta-language for the specification of the graphical syntax of the *Unified Modeling Language, version 2.0* (UML).

Visual languages have properties that cannot be expressed in conventional meta–languages for textual languages. To address these properties, we need to take into account that the symbols of the language most likely does not adhere to a sequential ordering. We therefore need a meta–language with some functionality that can address the fact that the symbols of most visual languages have a more or less arbitrary spatial ordering in the plane. The contribution of this thesis is such a meta–language, applicable for UML 2.0.

We give an overview of existing research in the field of visual language research and give an analysis of the graphical syntax of UML to highlight the issues a meta–language has to address. We also give a specification of a subset of UML, called T_{uml} (*Tiny UML*), to illustrate use of the meta–language.

GDL uses the *Z Notation* as its formal basis. In addition, to address the unique properties of visual languages, we use concepts from topology, geographical information systems theory and previously defined visual language formalisms.

Acknowledgements

This thesis is submitted in partial fulfilment of the Candidatus Scientiarium degree in Informatics at the Department of Informatics, University of Oslo (UiO).

I wish to thank my thesis adviser, Dr. Øystein Haugen, for his guidance, enthusiasm and patience with me.
I also wish to thank my friends and family for all their support.
Special thanks go to Arnt Inge Vistnes and Andreas Limyr for reading through and giving comments on my work.

And last, but not least, I would like to thank my wife Torunn. She has given me support and help all the way through. She has been a great motivator and has shared the highs and lows of this process with me.

Contents

1 Introduction **1**

1.1 The domain of interest 2
1.2 Goal and Requirements 3
1.3 Design criteria 3
1.4 Thesis structure 5

2 Background **7**

2.1 Grammatical approach 7
2.1.1 String Languages 7
2.1.2 String Grammars 8
2.1.3 Attributed multiset grammars 10
2.1.4 Adjacency grammars 15
2.2 Logical approaches 18
2.2.1 Definite clauses 18
2.2.2 Visual logical grammars 19
2.3 MSC and SDL 19
2.4 Diagram Interchange 20
2.4.1 The structure of Diagram Interchange 21
2.5 Aesthetics 25
2.6 Summary 26

3 Background from related disciplines **27**

3.1 Geographical Information Systems 27
3.1.1 Topological relationships 27
3.1.2 Spatial operators 30
3.2 Music notation 31

4 Analysis **35**

4.1 Introduction 35
4.2 Categorisation 35
4.3 Geometrical level analysis 37
4.3.1 Line–line 37
4.3.2 Line–text 38
4.3.3 Line–object 40
4.3.4 Text–text 41
4.3.5 Text–object 41
4.3.6 Object–object 43
4.4 Communicative level analysis 44
4.4.1 Graphical compactness 44
4.4.2 Style 46
4.5 Summary 47

5 Graphical Description Language **49**

5.1 Introduction 49
5.2 Brief introduction to Z 50
5.3 Designating graphical elements 50
5.4 Visual symbols and attributes 50
5.5 The predefined predicates 51
5.5.1 inside 52
5.5.2 centerOf 53
5.5.3 leftOf 53
5.5.4 above 54
5.5.5 overlap 55
5.5.6 disjoint 55
5.5.7 overlay 55
5.5.8 connectsTail and connectsHead 56
5.5.9 intersect 56
5.5.10 vertical and horizontal 58
5.5.11 closeTo 58
5.6 Summary of predicates 59

6 Defining a Visual Language **61**
6.1 Tiny UML 61
6.2 Naming conventions 62
6.3 The graphical syntax of T_{uml} 63
6.3.1 Class diagrams 63
6.3.2 Sequence Diagrams 66
6.3.3 State Machines 72

7 Discussion and Further Work **77**
7.1 Evaluation of Requirements 77
7.1.1 The geometrical level 78
7.1.2 The communicative level 79
7.1.3 Requirement of Precision 79
7.2 Evaluation of design 80
7.3 Further work 81
7.3.1 Applicability to other languages 81
7.3.2 Metrics 81
7.4 Concluding remarks 82

Bibliography **83**

List of Tables

2.1 Picture Layout Grammar Production Operators 14
2.2 Adjacency relations 17

4.1 Geometrical relations 36

5.1 GDL symbol attributes 52
5.2 GDL predicates 59

List of Figures

1.1 Domain model 3

2.1 PDL syntax 9
2.2 PDL concatenation operators 9
(a) $a + b$ 9
(b) $a \times b$ 9
(c) $a - b$ 9
(d) $a * b$ 9
2.3 Coordinate grammar, integral 11
2.4 Coordinate grammar, division 11
2.5 B over C 13
2.6 B over and touching C 14
2.7 Visual symbol and some of its attributes 16
2.8 A simple directed graph 18
2.9 A picture term graph 19
2.10 A picture term 19
2.11 A message sequence chart 21
2.12 Excerpt of Diagram Interchange (DI) meta–model 22
2.13 Example of a Class Diagram in DI 23
2.14 Example of a TextElement in DI 23
2.15 Example of Sequence Diagram in DI 24

3.1 The *in* relationship 30
3.2 Standard music notation 32
3.3 Example of avant–garde music notation 33

4.1 Analysis levels 36

4.2 Line crossings . . . 37
(a) Crossing lines w/o semicircular element . . . 37
(b) Crossing lines with semicircular element . . . 37
4.3 Cluttered diagrams . . . 38
(a) Cluttered diagram with semicircular elements . . . 38
(b) Cluttered diagram without semicircular elements . . . 38
4.4 Association text labels . . . 38
4.5 Placement of association text . . . 39
(a) Besides line . . . 39
(b) Overlapping line . . . 39
4.6 Complex text adjacency, class diagram . . . 40
4.7 Complex text adjacency, sequence diagram . . . 40
4.8 Horizontal rotation of text in adjacency to lines . . . 40
4.9 Horizontal rotation of text in adjacency to lines . . . 41
4.10 Example of insignificant differences in notation . . . 41
4.11 Example of significant differences in notation . . . 42
4.12 Acceptable variations in placement of text . . . 42
4.13 Erroneous placement of text . . . 43
4.14 Rewritten diagrams . . . 43
(a) Rewritten, lacks symmetry . . . 43
(b) Rewritten, more symmetrical . . . 43
4.15 Association multiplicities . . . 44
(a) Implicit . . . 44
(b) Explicit . . . 44
4.16 Class, showing all attributes and operations . . . 45
4.17 Class, all but name suppressed . . . 45
4.18 Composite structure, collaboration . . . 45
4.19 Composite structure, explicit collaboration . . . 46
4.20 Inheritance arcs and direction . . . 47
(a) Up wards joined/shared . . . 47
(b) Up wards diagonals/single . . . 47
(c) Down wards joined/shared . . . 47
4.21 Placement of pseudo states on border . . . 47

4.22 Pseudo states, alternative placement 48

5.1 Inside relationships (`inside(X, Y)`) 53
(a) Complete containment 53
(b) Boundary intersection 53
5.2 Use of `centerOf`–predicate 53
(a) Text centred in object 53
(b) Text centred on a line 53
5.3 Different left-of adjacencies 54
(a) `leftOf(X, Y)` 54
(b) Also `leftOf(X, Y)` 54
(c) Overlapping 54
5.4 Different above adjacencies 54
(a) X above Y 54
(b) X above Y and Z 54
5.5 Overlay relationship 56
5.6 Connecting predicate (`connectsTail`) 57
5.7 Intersections 57
(a) Without semicircle 57
(b) With semicircle 57
5.8 Different `closeTo` realtions 59
(a) Horizontal line and text 59
(b) Diagonal line and text 59
(c) Pseudo–state symbol and text 59

6.1 Structure of the specification 61
6.2 T_{uml} class symbol 64
6.3 Generalisation in T_{uml} 66
(a) Generalisation 66
(b) Generalisation line 66
6.4 T_{uml} sequence diagram 67
6.5 T_{uml} message symbol 72
6.6 T_{uml} state machine 75

7.1 "Weaving" of graphical symbols 80

Chapter 1

Introduction

Visual languages are all around us. We use maps to find our way in a foreign city and house plans to find out where we are in an unfamiliar building. Road signs are a form of visual language as are the sign language used to communicate with the deaf.

We have a variety of different visual languages within our field of computer science. Statecharts [30], petri nets [34], UML diagrams [48] and flowcharts are just a small sample. Programmers and program designers use different diagrams as an aid to visualise the static and dynamic aspects of program systems. Such diagrams can also be used to explain how programs are designed and work to people who are not trained in computer science.

Another application area for visual languages is geographical information systems. Here visual languages has been developed as a means to querying geographical databases, like the *Spatial-Query-By-Sketch* by Egenhofer [22] and the extended *Lvis* language developed by Bonhomme *et al.* [4]. Using diagrams and drawing to specify queries to a geographical database can enable people that are not technically trained to use create precise queries to such applications, as discussed in Favetta *et al.* [23] and Blaser *et al.* [3].

When we specify textual languages, we take advantage of a convenient feature of text, that it is a sequence of symbols. Thus we only need to specify the legal sequence of symbols. This is easily achievable with powerful meta–languages like BNF, also known as Backus–Naur Form (for an introduction to BNF, see Louden [38]). When we are specifying the syntax of visual languages, however, we need to take into account that the symbols of the language most likely does not adhere to a sequential ordering. We therefore need a meta–language with some functionality that can address the fact that the symbols of most visual languages have a more or less arbitrary spatial ordering in the plane. This problem has, as we shall see in chapter 2, been approached in a variety of ways.

This thesis will concern itself with the graphical syntax of the *Unified Modeling Language, version 2.0* (UML). UML is a language that has had considerable success in the last decade or so. It encompasses several different types of diagrams. There are diagrams for modelling various aspects of the static structure of systems in level of detail. There are also diagrams that can model the behavioural aspects, how different parts of a system interact with each other, and diagrams where we can model the legal states a system can have.

The fact that UML consists of this wide variety of diagram types, means that there are a variety of different notational forms. The meta–language needs to be able to address this.

1.1 The domain of interest

As background we will review some earlier work done in the field of specification, description and parsing of visual languages.

When we draw diagrams, at least diagrams of a certain size and complexity, there is a chance that they will become cluttered and incomprehensible. We will therefore look at the study of aesthetics and try to implement a mechanism that can be used to uphold aesthetical principles in a visual language.

Our test case is the *Unified Modeling Language, version 2.0*, and we therefore found it natural to look at the format for the exchange of UML diagrams, Diagram Interchange [46], to see how a specification of the layout of a particular diagram could be achieved. This is a format that allows UML diagrams to be interchanged between different systems and tools without loss of data or layout.

In addition, to get an idea on how other disciplines has tackled the description and specification of visual representation of data or syntax, we will review research done in the field of spatial databases and geographical information systems(GIS). We will also take a look at music notation, as it is a well known and widespread visual language.

This background information will provide us with the necessary insight to be able to design a meta–language that can specify the visual syntax of UML. This meta–language is called *Graphical Description Language*, abbreviated GDL.

A visualisation of the different fields in our domain related, is shown in figure 1.1.

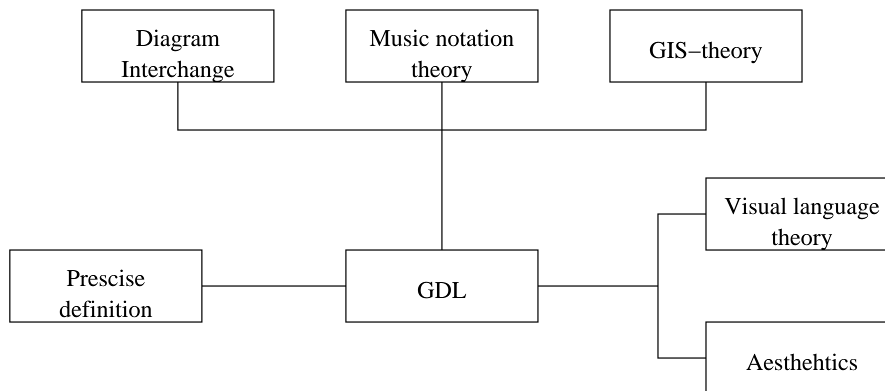


Figure 1.1: Domain model

1.2 Goal and Requirements

The goal of this thesis is to develop a meta–language that is capable of specifying the spatial syntax of a visual language.

Our primary target language is UML 2.0, which is a rich language of several different diagram types and thus a variety of notational forms. Since we investigate UML in particular, the meta–language will inevitably be influenced by that fact. In spite of that, we will try to make it as generic as possible so that may be applicable to other visual languages and notations as well.

The meta–language should be able to specify the spatial compositions of graphical symbols that lead to valid visual sentences in the language we are specifying. By a visual sentence, we understand an unordered collection of graphical elements in a spatial relationship to each other that convey something meaningful.

Another requirement we have, is *precision*. We want a language that is capable of specifying the spatial relationships of the graphical symbols at a precise level.

1.3 Design criteria

The topic of this thesis is language design. In design activities there is always important to have some criteria that guide the design activities to ensure a good design. Therefore we will now present some criteria, in addition to those in section 1.2, that we will use as guide–lines to ensure that we end up with a good design of our meta–language.

Donald Norman, in his book "The Design of Everyday Things" [45], describes several cases of bad design of everyday objects like door–handles, digital watches and telephones and explains why these designs are so bad, why users of these things make errors using them. Furthermore, he elaborates different principles that could make the design of everyday things better. Now clearly, a formal specification language is not an "everyday thing" to most people, but nevertheless it is intended for use by human beings and the knowledge of design principles that apply to everyday things could prove valuable even in this setting. We will use some of these principles as guidelines where they are applicable to language design.

Wexelblat [64] outlines, in his rather witty article, a number of steps that can be taken in programming language design to make the task of programming as difficult as possible. He elaborates why these different aspects on programming language design make different tasks so difficult for programmers. It could prove valuable to have those steps relevant to our domain in mind to avoid committing these design errors.

One of the criteria we will define, is what we will call *transparency*. By transparency we mean that the syntax and structure of the language should be as clear and easily understandable as possible to users of the language, it should not take a long time to learn. This implies a certain degree of simplicity and uniformity in the syntax and constructs of the language. It is, according to Wexelblat [64], the irregularities in a language that is difficult to learn and master. Programming and specification languages are artificial languages and with a little thought and consideration, there should be possible to avoid making a language that is too complex or irregular. Wexelblat also points out the fact that there are certain "standard" ways of the use of spaces and delimiters both in natural language and mathematics, and languages that deviates greatly from these "standard" ways are more error prone than others that conform.

We also want to design a language that is *compact* and *concise*, not a language that "explodes" with features. We want to strive to make a language that do not have too many nor too few constructs, just the constructs it needs to do what it is intended to achieve. One of Normans [45] principles is *constraints*. One should put constraints on the choices of what we can do with an object. When we put constraints on a design, we can greatly reduce the risk of users making errors and enhance the usefulness of a design. We should apply constraints so that, as Norman puts it "the user feels that there is only one thing to do - the right thing" [45].

1.4 Thesis structure

The following describes the structure of the thesis.

Chapter 2 Background This chapter introduces several theories for the specification of visual languages. In addition we review the *Diagram Interchange* specification. We also look at the role of aesthetics in visual languages/language design.

Chapter 3 Background from related disciplines This chapter draws lessons from the research field of geographical information systems (GISs) and the efforts to make spatial extensions to the *Structured Query Language (SQL)*.
This chapter also looks into music notation to see if there are any lessons to be learnt.

Chapter 4 Analysis This chapter provides an analysis of the graphical syntax of UML 2.0 in order to identify the issues our meta–language will have to be able to address.

Chapter 5 Graphical Description Language This chapter sees the design of GDL, Graphical Description Language.

Chapter 6 Defining a Visual Language This chapter uses GDL to specify a visual language, a subset of UML 2.0 called Tiny UML (T_{uml}).

Chapter 7 Discussion and Further Work This chapter evaluates the Graphical Description Language against the design goals and requirements and looks at the possibilities for further work.

Chapter 2

Background

We will in this chapter outline the different theories and techniques that have evolved in the field of visual language specification. We will concentrate on the research that has attempted to specify syntax specifications and descriptions for visual languages and on some visual programming languages/environments. We will not elaborate on the parsing algorithms given for the different formalisms, nor on various visual programming environments.

2.1 Grammatical approach

The earliest attempts to define a specification for visual languages, dating back to 1964, were grammatical in nature [40]. The first attempts were quite simple modifications of the phrase structure grammars that were used to specify string languages.

2.1.1 String Languages

This section reviews some theory and vocabulary related to string grammars.

A *production* is a form of rule that specifies the composition of the different grammatical elements [40]. A *non–terminal* symbol is a symbol that represent a compound structure in the language. A non–terminal can be composed of both non–terminal and *terminal* symbols. A terminal symbol is a symbol that represent the symbols that are part the alphabet of the language, Examples are numbers, a word, a letter or any other symbol. When we specify grammars, we often place the constraint

$$N \cap T = \varnothing$$

on the sets of non–terminal symbols (N) and terminal symbols (T).

Chomsky [12] divided phrase structure grammars into four different types based on the form of their productions. This now known as the *Chomsky hierarchy* [40].

Type 0, or unrestricted grammars allow for an unlimited number of terminal and non–terminal symbols on either side of the production.

Type 1, or context–sensitive grammars, are grammars that have the form

$$\alpha A \alpha' \rightarrow \alpha \beta \alpha'$$

A is a non–terminal symbol and β is a non–empty string and α and α' are strings that may be empty. The meaning of the production is that the non-terminal A can be replaced by the string β in the context of α and α'. So the result of a production thus relies on the content of the context symbols.

Type 2, or context–free grammars, are grammars that do not allow contexts. This is the type of grammars that is most often used in the specification of syntax for programming languages and has thus been studied in great depth. The productions in a context–free grammar are on the form

$$A \rightarrow \beta$$

which can be understood that the non–terminal A is to be replaced with β.

Type 3, or regular grammars. They have the form

$$A \rightarrow aB \text{ or } A \rightarrow a$$

where A and B are non–terminals and a is a terminal symbol.

2.1.2 String Grammars

2.1.2.1 Picture Description Language

One of the first formalisations of languages for the description and specification of visual languages was done by Shaw with his *Picture Description Language* (PDL) [58]. PDL is a linear, context–free string language.

A *picture primitive* is a n–dimensional pattern ($n \geq 1$). A primitive has two distinguished points, a *tail* and a *head*. The primitives can be concatenated at their tail and/or head only, to form more complex structures. The restriction that there is at most two point of concatenation, means that primitives can be represented as labelled directed edges of a graph, pointing from its tail to its head. This does not mean that there is an implied direction associated with the primitives as such, the abstraction using directed edges (i. e. arrows) is merely a way to distinguish between the tail and the head of a

primitive. The primitives correspond to the terminal symbols. Each primitive is member of a pattern class.

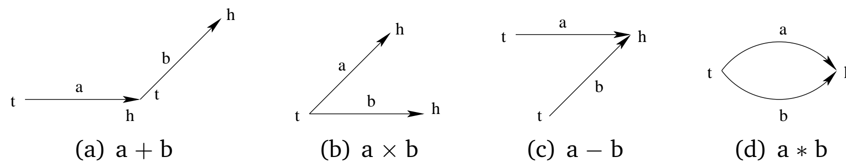
There are three different special primitives. A *don't care* primitive will match any primitive type. A *blank* primitive represents invisible primitives. The third special primitive, is a *null point primitive*, a primitive where the tail and head is the same point, i. e. the tail and head are identical. The null point primitive is denoted λ and consists only of its tail and head. In a graph, λ would be represented as a labelled node

In figure 2.1 we see the valid syntax of PDL specified in Backus–Naur Form (BNF). In this derivation, p represent any primitive class name (including λ). The symbol l is a label designator.

$$\langle S\rangle ::= \mathsf{p} \mid (\langle S\rangle\ \phi\ \langle S\rangle) \mid (\sim \langle S\rangle) \mid \langle L\rangle \mid (/\langle L\rangle)$$
$$\langle L\rangle ::= \langle S\rangle^{l} \mid (\langle L\rangle\ \phi\ \langle L\rangle) \mid (\sim \langle L\rangle) \mid (/\langle L\rangle)$$
$$\phi ::= + \mid - \mid * \mid \times$$

Figure 2.1: PDL syntax

PDL provides four binary *concatenation operators*. They define different tail/-head concatenations. Figure 2.2 shows the result of applying these operators to two primitives a and b. h and t in the figures designate the head and tail of the primitives. The $+$ operator connects the head of a primitive to the tail of another, as seen in figure 2.2(a). The $\times$ operator connects the tails of two primitives, as illustrated in figure 2.2(b). The $-$ operator connects the head of two primitives. This is illustrated in figure 2.2(c). The $*$ operator connects the tail of a primitive a to the tail of a primitive b and the head of a to the head of b, as can be seen in figure 2.2(d). The unary operator $\sim$ is a negation operator which reverses the tail and head of a primitive. The unary composition operator / is an operator where each primitive inside its scope refers to an identically labelled node outside its scope.



(a) a + b (b) a × b (c) a − b (d) a ∗ b

Figure 2.2: PDL concatenation operators

2.1.2.2 Positional grammars

Positional grammars are based on an observation about concatenation in string grammars. In string grammars, the role of concatenation is to indicate where the next symbol is relative to the current symbol [18,40]. In the theory of positional grammars, concatenation is therefore generalised into an arbitrary spatial relation REL_i which convey information about the relative position of the next symbol α_{i+1} with respect to the current symbol α_i [18,19]. Productions in a relational grammar thus have the form:

$$A \rightarrow \alpha_1 \; REL_1 \; \alpha_2 \; REL_2 \; \cdots \; REL_{n-1} \; \alpha_n, \Delta \tag{2.1}$$

Here, A is a non–terminal, the α_is are terminal or non–terminal symbols and REL_i are the before mentioned spatial relations which specify the position of α_{i+1} in relative to α_i. Δ is a rule which synthesises the attribute values of A from the values of the attributes of $\alpha_1, \cdots, \alpha_n$.

In [17], Costagliola *et al.* describe a formalism that is based on positional grammars, *Extended Positional Grammars*. This is a formalism that is based on an extension of the well–known LR–parsing technique.

Productions in an extended positional grammar are essentially the same as for ordinary positional grammars:

$$A \rightarrow \alpha_1 \; REL_1 \; \alpha_2 \; REL_2 \; \cdots \; REL_{n-1} \; \alpha_n, \Delta, \Gamma \tag{2.2}$$

The only part of this production that is different, is the set Γ which is a set that is used to dynamically insert new symbols into the visual sentence that is to be recognised.

2.1.3 Attributed multiset grammars

Attributed grammars were first devised by Knuth [36], as a means to extract the semantics from context–free languages.

In this category, the productions rewrite sets or multisets of symbols. The symbols have geometric and sometimes semantic attributes associated with them and rewriting can be controlled by constraints over the attributes on the right hand side of a production [40].

A multiset is a mathematical entity that is very much like an ordinary set. But unlike ordinary sets, multisets are allowed to have repeated elements [37,63]. An object may be an element of a multiset a finite number of times. The multiplicity of an object is relevant. In additions to the ordinary set operations, there are a few that are unique to multisets. An element which has a occurrences in multiset A and b occurrences in multiset B will have $a + b$ occurrences in $A \uplus B$, $max(a, b)$ occurrences in $A \cup B$ and $min(a, b)$ occurrences in $A \cap B$.

2.1.3.1 Grammar for mathematical notation

One of the first examples of an approach in this group is the *Coordinate Grammar* of Anderson [1]. He used this formalism to specify two–dimensional mathematical notation. The productions had the form:

$$A \rightarrow \alpha \text{ where } C\ F$$

The A is a non–terminal symbol, α is a non-empty string of symbols, C is a constraint over the attributes in α and F is an expression that computes the attributes of A with respect to the attributes of α. The symbols in the grammar each have six geometric attributes and one semantic attribute. It is worth noting that even though α is written as a string, the order of the symbols in α does not have any significance at all, as is the case in productions in string languages, so α is semantically a multiset.

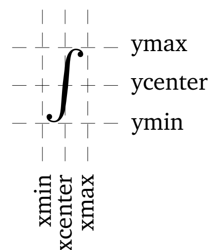


Figure 2.3: Coordinate grammar, integral

In figures 2.3 and 2.4, we see two examples on how mathematical symbols and expressions are placed in a grid that has 3x3 axes. The labels on the axes, *xmin*, *xcenter xmax*, *ymin*, *ycenter* and *ymax* are the six geometrical attributes of each symbol. Note that the center of a symbol or an expression is not necessarily the geometrical center but the *centre of interest* of the symbol. This can be seen in figure 2.4, where the attribute *ycenter* refers to coordinates above the geometrical centre along the *y*–axis for the horizontal bar of the fraction.

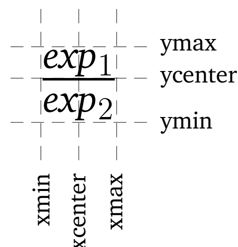


Figure 2.4: Coordinate grammar, division

Finally, we show an example on a production in this grammar. We take the derivation of a *DivTerm*, which is shown in figure 2.4 on the page before.

$$\begin{aligned}
DivTerm \longrightarrow \quad & Exp_1\ Exp_2\ hbar \quad \text{where} \\
& Exp_1.xmin > hbar.xmin \quad \& \\
& Exp_1.xmax < hbar.xmax \quad \& \\
& Exp_1.ymin > hbar.ymax \quad \& \\
& Exp_2.xmin > hbar.xmin \quad \& \\
& Exp_2.xmax < hbar.xmax \quad \& \\
& Exp_2.ymax < hbar.ymin \\
& DivTerm.xcenter = hbar.xcenter \\
& DivTerm.ycenter = hbar.ycenter \\
& DivTerm.meaning = (Exp_1.meaning/Exp_2.meaning)
\end{aligned}$$

Here, *DivTerm* corresponds to *A* on the left hand side in the general production shown above. The first line after the production arrow corresponds to 'α where'. The next lines correspond to the constraints in the production, the *C* in the general production described above. The last three lines correspond to the *F*, the expression that computes the variables of *A*.

In the last line of this production, we see the attribute *meaning*. This is the before mentioned semantic attribute of the symbols. It contains the mathematical meaning of the symbols and as we see here, the value of $DivTerm.meaning$ is computed on the basis of $Exp_1.meaning$ and $Exp_2.meaning$.

2.1.3.2 Picture Processing Grammars

Chang introduced *Picture Processing Grammars* [7]. This can be viewed as a generalisation of coordinate grammars in that it allows each symbol to have an arbitrary number of attributes (Anderson's Coordinate Grammar could only have six). The grammars describe the hierarchical structure of two dimensional pictures. Chang *et al.* further developed the SIL system [9] that handles icon–oriented grammars through the use of spatial operators for vertical concatenation (ˆ), horizontal concatenation (&) and overlap (+).

2.1.3.3 Picture Layout Grammars

Golin introduced *attributed multiset grammars* (AMG) [26]. An attributed multiset grammar is essentially a context–free picture processing grammar.

This grammar formalism was further extended to *Picture Layout Grammars* (PLG) [27,28]. A PLG is an AMG that is augmented with adjacency operators, or *production operators*. The operators and their meaning are listed in table 2.1 on page 14.

The symbols of an AMG has four attributes *lx*, *by*, *rx* and *ty*. *lx* is the point of the leftmost x–position the symbol occupies as defined by a coordinate system, *rx* is the rightmost x–position, *by* is the lowest y–position the symbol occupies and *ty* is the highest y–position.

If the object at hand is a shape, the attributes represent the extent of the object. In this case the relationships $lx \leq rx$ and $by \leq ty$ must hold.

If the object at hand is a line, the pair (*lx,by*) represent the left side of the line and the pair (*rx,ty*) represent the right side of the line. Golin points out that even though the endpoints are called *left* and *right*, there is not a requisite that $lx \leq rx$ or $by \leq ty$ hold.

The general form of production is as follows:

$$\begin{aligned} A \rightarrow \quad & \{B, C\} \\ & A.atrr = func_{op}(B.attr, C.attr) \end{aligned}$$

Where:

$$pred_{op}(B.attr, C.attr)$$

An example production that corresponds to the situation depicted in figure 2.5 is

$$A \rightarrow \text{over(B,C)}.$$

A is a non–terminal representing the drawing on the right hand side.

The operator *over* defines that its first argument located geometrically above its second argument. It is defined with the constraint $B.by \geq C.ty$, so that *B* is ensured to be located over *C*.

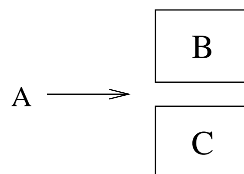


Figure 2.5: B over C

It is possible to define additional functions or addition to the predefined ones. As an example, look at the situation depicted in figure 2.6 on the following page which is specified by:

$$A \rightarrow \quad (B, C)$$

Where:

$$B.by == C.ty$$

This specifies that *B* must be over and exactly touching *C*.

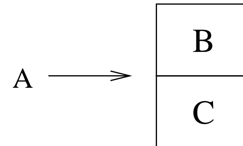


Figure 2.6: B over and touching C

Operator	**Meaning**
over(B,C)	B is over C
left_of(B,C)	B is to the left of C
tiling(B,C)	an arbitrary tiling
contains(B,C)	B contains C
group_of(B)	an arbitrary number of B's
adjacent_to(B,C)	B is adjacent to C
touches_L(B,C)	(lx,by) of B on the boundary of C
touches_R(B,C)	(rx,ty) of B on the boundary of C
points_from(B,C)	the left end of B is on C
points_to(B,C)	the right end of B is on C
labels(B,C)	B is adjacent to the line C
follow(B,C)	the right end of B is on the left end of C
join(B,C)	the right end of B is on the right end of C
fork(B,C)	the left end of B, is on the left end of C
parallel(B,C)	both ends of B and C match
reverse(B)	exchange the left and right side of B

Table 2.1: Picture Layout Grammar Production Operators

2.1.3.4 Constraint multiset grammars

Constraint multiset grammars are an extension of attributed multiset grammars, first introduced as *constraint set grammars* by Helm *et al.* [32]. Constraint multiset grammars were further formalised by Marriott [39], where he gives a general form for productions:

$$P ::= P_1, \cdots, P_n \textit{ where exists } P'_1, \cdots, P'_m \textit{ where } C$$

This rule says that the non–terminal P can be rewritten to the multiset of symbols $P_1, \cdots, P_n$ whenever there exists other symbols $P'_1, \cdots, P'_m$ such that all the symbols satisfy the constraints C.

Chok *et al.* [11] used constraint multiset grammars to specify rules for diagram beautification. Chok *et al.* [10] has also utilised this formalism to create tools that can create user interfaces directly from a specification written with a constraint multiset grammar.

2.1.4 Adjacency grammars

Jorge describes a family of *adjacency grammars* [35].

Adjacency grammars are grammars that build upon the *picture layout grammars* of Golin [26] the use of adjacency constraints and by allowing specification and parsing of geometrically overlapping pictures. These grammars are developed for what he calls *calligraphic interfaces*, which is a class of pen–based interfaces where drawing or sketching is the main input method, as opposed to classical interfaces which uses point–and–click or keyboard input.

Graphical primitives are divided into three categories. The first is a category of *point primitives*. A point primitive is a small shape that is characterised by a single point. Another category is *line primitives*, which are line like objects that serve as concatenation points in adjacency relations. The third an final category is *box primitives*. These are closed graphical shapes whose main attribute is the smallest enclosing *convex hull*. A convex hull can intuitively been viewed as a "boundary" of a set of points. This "boundary" can be used to approximate the shape of an object [29]. The convex hull of an object is indeed a good way of finding its shape, but it is rather computationally expensive. To circumvent this, Jorge [35] decides to use the enclosing rectangle known as a *Bounding Box*.

The symbols have a distinct type and a set of attributes, both geometrical and semantic. Figure 2.7 gives an example of an arc and its attributes. The arc itself has the attributes `sp` and `ep` which denote the start point and end point respectively. Furthermore, the bounding box (denoted `bb`), the dashed

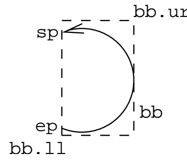


Figure 2.7: Visual symbol and some of its attributes

rectangle around the arc which extent is denoted by the attributes `bb.ll`, the lower left coordinate pair and `bb.ur`, the upper right coordinate pair. Semantic attributes are attributes that does not necessarily have any visual representation. This can for example be the type of a token or a graph–node's indegree and outdegree.

Jorge [35] describes and classify several different forms of adjacency relations based on the types of objects involved and how their attributes relate to each another.

A *line* adjacency is a relation that relate connection points of different graphical primitives. Some examples are an adjacency that concatenates lines that have a common endpoint, which is called a **chain** relation and an adjacency that relates a primitive, usually a string, to a line, called a **label** relation.

A *tiling* adjacency relation is a type of relation where we operate on objects of a plane surface. The objects occupy disjoint portions of the plane. Examples of adjacencies in this category are **above**, **below** and **to-the-right-of**.

Overlap, or *containment* adjacency relations are relations where the graphical objects involved occupy overlapping or nested regions of space. **Overlaps** and **contains** are examples of adjacencies in this category.

Metric adjacency relations relate the one–dimensional attributes of objects.

Logical adjacency relations are somewhat special, since they do not operate on the spatial attributes of their arguments. They can however, play an important part of visual languages. One example is from flowchart diagrams, where they are used to connect parts of the diagram that are disconnected. These parts are then logically connected through labelled connections points in the different diagrams.

Temporal adjacency relations are important in applications where time plays a central role. One example here is multimedia and interactive applications Another is GIS[1] applications that tracks objects in motion.

He then gives a definition of several adjacency operators that can be used for the specification of the spatial layout of symbols.

[1]Geographical Information Systems

Table 2.2 shows the different adjacencies grouped by class. It also shows the resulting graphical symbol of the adjacencies.

Adjacency class	Result	Adjacency
Line to Line	Line	`chain`
	Box	`closes`
	Point	`touches`
	Point	`intersects`
Line to Shape	Line	`pointsTo`
	Line	`pointsFrom`
	Box	`enters`
	Box	`exits`
	Line	`labels`
Shape to Shape	Box	`below`
	Box	`leftOf`
	Box	`contains`
	Box	`overlaps`
Logical	Item	`list`

Table 2.2: Adjacency relations

Jorge also introduces the concept of *fuzzy sets*. into his adjacency grammar to form *fuzzy relational adjacency grammars* (FRAGs).

Fuzzy sets were first introduced by Zadeh [67]. They can be considered as an extension to classical sets, by associating a degree of membership to each element. The degree of membership is a real number in the interval $[0, 1]$, where a value of zero correspond to non–membership (absolute) and a value of one correspond to full membership. Values in between denote a partial degree of membership.

Jorge argues that most of the research on visual languages has focused on precise spatial relations and that such an approach would not be feasible for specification and parsing of input data from calligraphic interfaces where the input can be ambiguous.

In "Sketching User Interfaces with Visual Patterns", Caetano *et al.* [6] describes an application for making user interfaces. The application utilises FRAGs to parse and generate code for a user interface. from data that is drawn from a calligraphic interface. Another useful application for fuzzy relational adjacency grammars is document layout specification, which is discussed in Pinto–Albuquerque *et al.* [50].

2.2 Logical approaches

2.2.1 Definite clauses

This approach uses *definite clauses* from first order logic. Here one have a collection of rules that define a predicate p. Here is an example of the form of rules ([40]):

$$p(T_1, \cdots, T_n) :- p_1(S_1^1, \cdots, S_1^{n_1}), \cdots, p_m(S_m^1, \cdots, S_1^{m_n}) \tag{2.3}$$

Here, T_i and S_i^j are terms. This rule may be understood to represent the logical formula:

$$p_1(S_1^1, \cdots, S_1^{n_1}) \wedge \cdots \wedge p_m(S_m^1, \cdots, S_m^{n_m}) \longrightarrow p(T_1, \cdots, T_n) \tag{2.4}$$

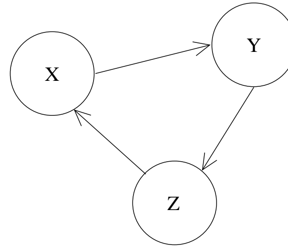


Figure 2.8: A simple directed graph

This type of formula is known as a *definite clause*. This can for example be used to define paths in a directed graph. Taking the graph of figure 2.8 as an example, the edges are defined as follows:

$edge(X, Y)$
$edge(Y, Z)$
$edge(Z, X)$

These edge–predicates hold if there is an edge going from node X to node Y, an edge going from node Y to node Z and finally, an edge going from Z back to X, respectively. Then, one can define the path from X to Z as:

$$path(X, Z) :- edge(X, Y) \wedge edge(Y, Z)$$

which reads 'the path from X to Z is defined as the edge going from X to Y and the edge going from Y to Z'.

2.2.2 Visual logical grammars

Visual logic grammars is a formalism that uses visual notation to reason about and define visual languages [42, 43]. The general idea is to introduce a new kind of term into logic, the *picture term*. Here pictures are handled as graph–like structures.
The underlying formal model is graphs with typed nodes. Typed edges denote relationship between nodes. These graphs are made according to the definition of a *picture vocabulary*.
Suppose we have a vocabulary which defines object types as

$$\{circle, line, label\}$$

and the relations defined as

$$\{touches : circle \times line, attached : label \times line\}$$

we would get a picture term graph as shown in figure 2.9.

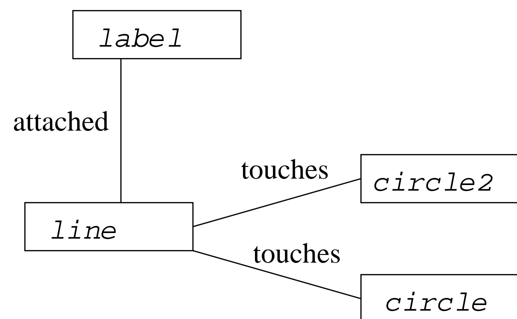


Figure 2.9: A picture term graph

This would correspond to the picture term that is shown in figure 2.10.

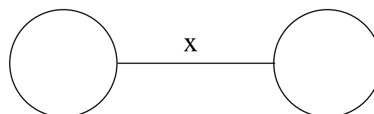


Figure 2.10: A picture term

2.3 MSC and SDL

We have had a look at two visual languages that are specified in a similar way to each other, namely MSC and SDL. Both are specified by the

International Telecommunications Union (ITU). We will briefly review their definition.

The *Specification and Definition Language (SDL)* is a language that is used for the specification of behaviour in real time systems [56]. The technique used to specify the language is based on a mixture of the meta–languages *Meta IV*, *BNF* and a set of *extension symbols* to BNF. The extensions are the following: **contains**, **is associated with**, **is followed by**, **is connected to** and **set**

The extension symbols operate on graphical symbols and define the relationship between its left–hand and its right–hand argument. For example, the symbol **contains** means that its right–hand argument is placed inside the symbol on the left side.

The symbol **is associated with** specifies that the left–hand argument is, both logically and visually, associated with the right–hand argument. Furthermore, the symbol **is followed by** specifies that the argument on its left side is followed by the argument on its right side. The last symbol, **is connected to**, describe a situation where the left argument is connected to its right side argument, both visually and logically.

Message Sequence Charts is a language that can model the message interchange of real time applications.

For *Message Sequence Chart* (MSC), the specification [33] states that

> *the meta–language consists of a BNF–like notation with the special meta–constructions: **contains, is followed by, is associated with, is attached to, above** and **set**.*

MSC is thus specified in a similar way as SDL is, except for the symbol **above**, that specifies that the left–hand argument is above the right–hand argument.

None of the meta–symbols of neither MSC or SDL that handle the spatial layout of graphical symbols are specified more than in informal text.

Figure 2.11 on the next page illustrates a message sequence chart. In this MSC, we have three instances, x, y and z. The instances send messages to each other, visualised by the labelled arrows. The instance x sends a message, labelled *rep* out to the environment (the environment is represented by the frame around the MSC).

2.4 Diagram Interchange

To be able to exchange UML diagrams between different tools, a mechanism was included in the first UML standard. UML has a strong emphasis on

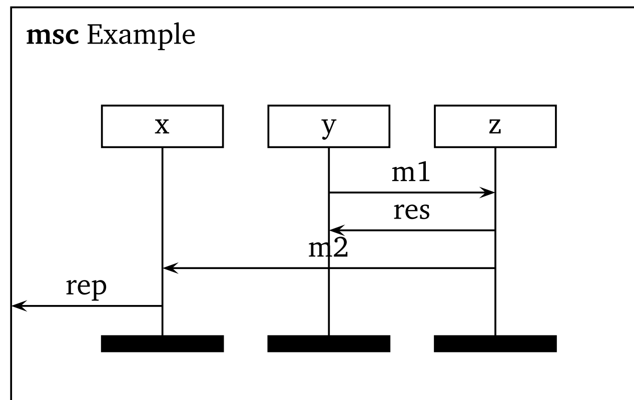


Figure 2.11: A message sequence chart

graphical representation and a deficiency in this first mechanism was that it only supported the definition of the different elements of a model and not the spatial organisation of the elements of the diagrams.

This should be enough information for tools that merely check the consistency of UML models or that generate code. Other tools that are more graphical oriented, do not benefit from this mechanism. The 'Diagram Interchange' (DI) format, described in [46] is a response to a request from the Object Management Group (OMG) for a mechanism that could solve these problems.

The submitters of the DI propose that it should not restrict the extensibility of UML. The mechanism should, if possible, not have any notion of concrete geometrical shapes of the elements in a diagram. That responsibility should lie with the UML tools that render the diagrams on screen or paper.

DI is based on the fact that most UML diagram types follow a schema known from graph theory, with nodes (that can be rectangles, circles or other shapes) and edges (lines that connect the nodes). The edges can have arrows at their ends and annotations (text attributes). The nodes can contain different compartments and annotations.
Most of the UML diagram types has a straightforward mapping to a graph. An exception here is the sequence diagrams, that do not map as naturally to a graph as the other diagram types.

2.4.1 The structure of Diagram Interchange

Diagram Interchange is specified as an extension to the current UML meta–model. It does not change the UML standard, it merely adds an extension package.

The core classes of the extended meta model are *GraphNode* and *GraphEdge*. The base class of these two classes, is *GraphElement*. The different elements are linked via a *GraphConnector* class. An excerpt taken from the DI meta model [46], showing the core classes, is shown in figure 2.12.

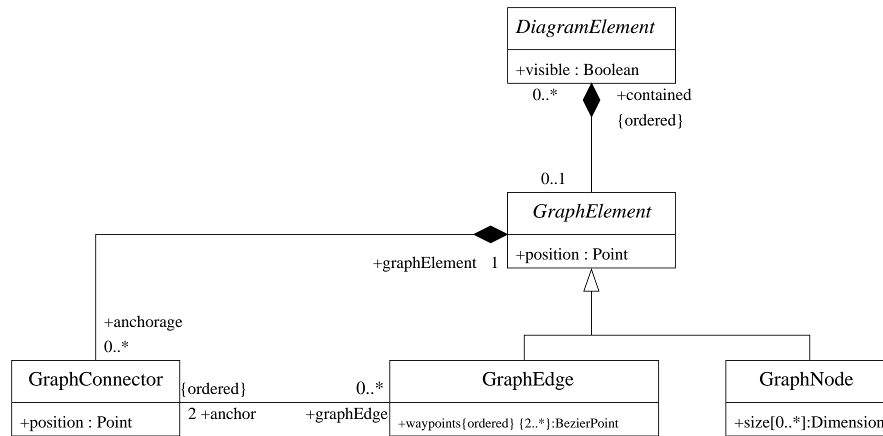


Figure 2.12: Excerpt of Diagram Interchange (DI) meta–model

A GraphElement can have any number of GraphConnectors. This way, any number of GraphEdges can connect to the GraphElements. A GraphEdge references two GraphConnectors, representing the endpoints of the edge. The connectors are ordered, with the first representing the first way–point and the second represents the last way–point of a GraphEdge.

The association between GraphElement and DiagramElement allows for the construction of classes, with GraphNodes representing the different parts of a class, i. e. the different compartments and so fort.

Figure 2.13 on the next page[2] shows the nesting of GraphElements, which outline the structure of a simple class. A more complex class may have contain more nested elements to represent inheritance, aggregation and other constructs.

The graphical representation of GraphElements is not stored explicitly in the (DI) meta–model. The knowledge to draw a class as a rectangle, for example, have to be taken care of by a drawing tool and a XSL style-sheet.

2.4.1.1 The Diagram

Diagram is a special node. It is the topmost node (GraphElement) in a diagram and thus recursively contains all other GraphElements. It has two attributes, a name and a viewport. The viewport is a point that indicates the

[2] taken from the DI specification [46]

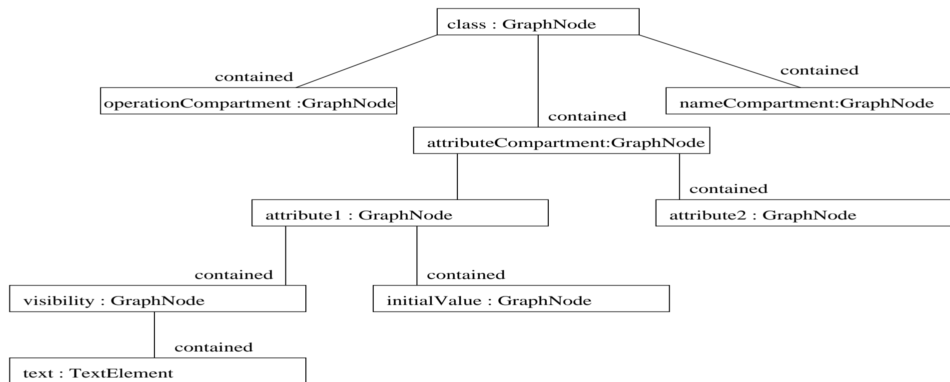


Figure 2.13: Example of a Class Diagram in DI

top left corner of the (currently) visible part of a diagram.
The type of the Diagram, e. g. StateDiagram, ClassDiagram etc., is stored in a SimpleSemanticModelElement. The Diagram references it through its semanticModel (these classes are not shown in the excerpt of the meta–model in fig. 2.12 on the facing page)

The DI metamodel allows for an additional semantic meaning by linking an element of an existing semantic model to a GraphElement via the abstract SemanticModelBridge.

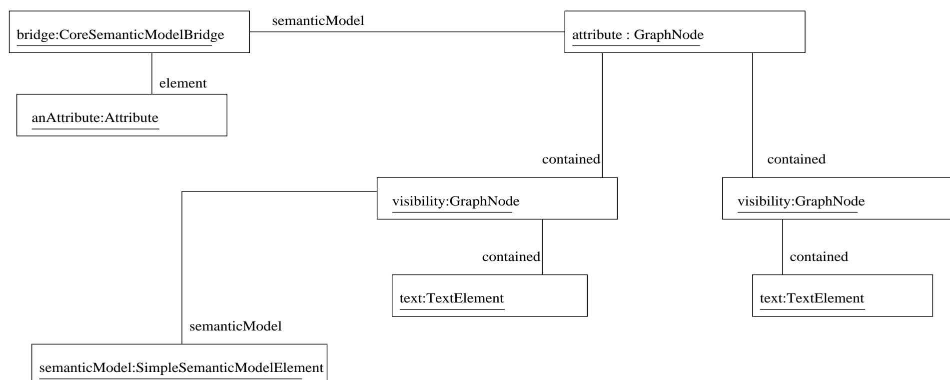


Figure 2.14: Example of a TextElement in DI

2.4.1.2 Representing the different diagram types

Through semantic links to the UML metamodel, DI can represent most of the UML diagram types. As mentioned above, most of the diagram types

of UML are represented well by a graph. Sequence diagrams however, are somewhat different.

In a UML class diagram for example, classes, interfaces and packages are represented by GraphNodes, while associations, inheritance (also known as generalisation) and dependencies are represented through GraphEdges. A class may have multiple compartments for attributes and/or operations. These compartments are represented, as shown in figure 2.13 on the page before, through nested GraphNodes with a link to a SimpleSemanticModelElement. This is because compartments are not part of the UML meta model. Compartments are only part of the representation of a class.

The end of edges, how they appear depending whether it is an aggregation, composition or generalisation, is controlled by the corresponding UML meta model elements.

The other diagram types are represented in a similar way.

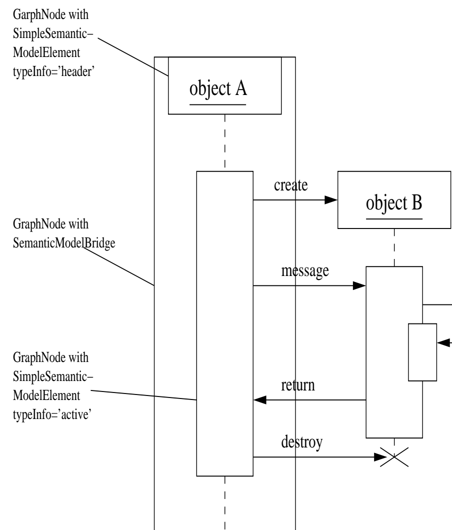


Figure 2.15: Example of Sequence Diagram in DI

Sequence diagrams are modelled with a GraphNode that has a semantic model bridge to the corresponding instance. This GraphNode contains other GraphNodes that represent the active section of the lifeline and a GraphNode that represent the rectangle at the top of the lifeline. An example depicting this, is shown in figure 2.15.

2.5 Aesthetics

In this context, aesthetics does not necessarily refer to the notion of 'aesthetically pleasing', but rather to criteria that have been defined and used in diagram/graph layout. In 'Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study' [52], Purchase *et al.* investigates which aesthetics have most impact on user comprehension of UML class diagrams.

Many CASE–tools which provide support for drawing UML diagrams, benefit from automatic graph layout algorithms. According to the authors, these algorithms are largely defined with respect to abstract graphs structures (nodes, edges). These structures often have no relationships to real world objects. They do not take into account some human computer issues relating to diagram comprehension.

Previous studies by the authors have shown that if one optimised the aesthetic of symmetry in a diagram and reduced the number of crosses and bends of lines, comprehension of diagrams could be improved greatly. Other aesthetics that may have influence for the comprehension of a diagram are variation in edge (path) lengths and flow. The experiments in [52] showed that diagrams with a medium variation in path lengths produce better results with regard to user performance than diagrams with greatly varying path lengths and diagrams with all path of the same length.
With regard to flow, which is described as which way for example inheritance arrows go, the results from this particular study actually was contradictory to an earlier study by the same authors [54]. This study found that the performance decreased in this study when the inheritance arrows pointed upwards (i.e. the superclass i above the subclass), when the previous results showed that this actually increased performance.

The authors of [54] have also conducted a similar study, 'UML collaboration diagram syntax: an empirical study of comprehension' [53], which addresses the same type of questions for collaboration diagrams that [54] did for class diagrams. The conclusions in this study concurred with the researchers expectations about which notational variant that would give the highest rate of comprehension. But when they looked at how the different notations scored with respect to how easy it was to identify errors, none of the notational variants gave any advantage over the other.

This research shows that one should not jump to quick conclusions about what is the best layout of a diagram.

What constitutes a good diagram layout, may vary with the task, domain, language and the person interpreting the diagram.

In creating a metalanguage for UML, one of the goals is to be able to eliminate at least some of the ambiguities that can arise in the notation. The

research of Purchase *et al.*. can prove valuable in defining certain criteria that can be implemented in the metalanguage to rule out some of the notational ambiguities/difficulties.

2.6 Summary

This chapter has presented some background material from the research field of visual language specification. The theories and approaches presented here are by no means a complete account of all the theories that exist. To stay reasonably within the limits of the thesis, we chose the theories that was most relevant to our own work.

For readers interested in learning more about visual language and visual languages in general, we refer to the excellent survey of Marriott *et al.* [40] and to Chang's article on visual languages [8].

Chapter 3

Background from related disciplines

This chapter presents some relevant background theory for the thesis in other disciplines than the theory of specification of visual languages that was presented in chapter 2. What they have in common with the theories presented in that chapter is that they deal with spatial data or the spatial presentation of data.

We look at the research field of spatial databases with focus on geographical information systems. In addition we look at some theory of musical notation.

3.1 Geographical Information Systems

There are many different applications that rely on spatial data. Notable examples are CAD/CAM system, image databases, VLSI design systems and geographical information systems (GISs) [20]. We will in this section concentrate on GIS or more precisely, the research field of spatial databases and the attempts to define a spatial SQL.

To enable users to query databases that contain spatial data, the research field of spatial databases has grown considerably for the last two decades.

3.1.1 Topological relationships

The study of point set topology has been the basis for many of the attempts to formally specify spatial relationships. There have been developed a number of theories that utilise this formalism to define the spatial relationships between objects of different types [13,14,16,57]. We will here present some

of the more notable in the field. In topology, the concepts of *interior*, *boundary* and *exterior* are well defined. The concepts can be used to define spatial relationships between two–dimensional objects in two–dimensional space ($\Re^2$).In this section, we will adopt the following notation convention: A and B denote point sets. A° denotes the interior of A, ∂A denotes the boundary of A and A^- denotes the exterior of A.

The 9–intersection(9IM) method is a model for binary spatial relations [57]. It applies to objects of the types area, line and point. These object types can be grouped together in six binary relationship categories (area/point, area/line, area/area, line/point, line/line, point/point). The topological relationships are characterised by the set intersections between the interior, boundary and exterior of A with those of B, as shown in the matrix in equation 3.1.

$$I(A,B) = \begin{pmatrix} A^\circ \cap B^\circ & A^\circ \cap \partial B & A^\circ \cap B^- \\ \partial A \cap B^\circ & \partial A \cap \partial B & \partial A \cap B^- \\ A^- \cap B^\circ & A^- \cap \partial B & A^- \cap B^- \end{pmatrix} \tag{3.1}$$

Each intersection is either empty ($\varnothing$) or non–empty ($\neg\varnothing$) depending on the topological relationship described. The *contains* relationship for example, would be specified with the following 9–intersection:

$$I(A,B) = \begin{pmatrix} \varnothing & \varnothing & \neg\,\varnothing \\ \neg\,\varnothing & \neg\,\varnothing & \neg\,\varnothing \\ \varnothing & \varnothing & \neg\,\varnothing \end{pmatrix} \tag{3.2}$$

The 9–intersection method is based on an earlier method, the 4–intersection (4IM) method. The 4IM method however, did not consider the exterior of objects, only the interior and the boundary. The spatial relations of the 4IM is represented by the following matrix:

$$I(A,B) = \begin{pmatrix} \partial A \cap \partial B & \partial A \cap B^\circ \\ A^\circ \cap \partial B & A^\circ \cap B^\circ \end{pmatrix} \tag{3.3}$$

The *dimension extended method* (DEM), developed by Clementini *et al.* [16], take into account the dimension of the intersections. It is based on the 4IM. The operator $dim(x)$ return the maximum dimension of the geometries in x. The dimensions can take the values $-1, 0, 1, 2$, where:

$$dim(x) = \begin{cases} -1 & \text{if } x = \varnothing \\ 0 & \text{if } x \text{ contains at least one point and no lines/areas} \\ 1 & \text{if } x \text{ contains at least one line and no areas} \\ 2 & \text{if } x \text{ contains at least one area} \end{cases}$$

Another method introduced by Clementini *et al.* [14,16], is a calculus–based method (CBM), which they prove is more efficient and powerful than the 4IM, 9IM and DEM. It specifies topological relationships using object calculus. They define five relationships and three boundary operators. For clarity, we will show the definitions of the relationships and boundary functions here. Let A and B be point sets. In the following equations we have the object calculus expression on the left equivalence arrow and the point set expression that defines it on the right side of the arrow.
The *touch* relationship is defined as the combination of two intersections, the intersection between the interior of the two point sets, which is the empty set and the non–empty intersection between the other regions of the two sets:

$$\langle A, touch, B\rangle \Leftrightarrow (A^\circ \cap B^\circ = \varnothing) \wedge (A \cap B \neq \varnothing) \tag{3.4}$$

Furthermore, we have the *in* relationship, which is defined as:

$$\langle A, in, B\rangle \Leftrightarrow (A \cap B = A) \wedge (A^\circ \cap B^\circ \neq \varnothing) \tag{3.5}$$

Figure 3.1 on the following page illustrates the *in* relationship, the circle is in the box.

The *cross* relationship applies to line/line and line/area combinations:

$$\begin{aligned}\langle A, cross, B\rangle \Leftrightarrow (dim(A^\circ \cap B^\circ) = max(dim(A^\circ), dim(B^\circ)) - 1) \\ \wedge(A \cap B \neq A) \wedge (A \cap B \neq B)\end{aligned} \tag{3.6}$$

overlap is a relationship that applies to area/area and line/line combinations:

$$\begin{aligned}\langle A, overlap, B\rangle \Leftrightarrow (dim(A^\circ) = dim(B^\circ) = dim(A^\circ \cap B^\circ)) \\ \wedge(A \cap B \neq A) \wedge (A \cap B \neq B)\end{aligned} \tag{3.7}$$

The *disjoint* relationship applies to all combinations of the three different object types:

$$\langle A, disjoint, B\rangle \Leftrightarrow A \cap B = \varnothing \tag{3.8}$$

There are three boundary operators, one for areas and two for lines.

The boundary operator b, when applied to an area A as a pair (A, b) will return the line ∂A.

The boundary operators f and t for a line L will when applied as the pairs (L,f) and (L, t) return the two points that represent the set ∂L.

Finally, we have the *dimension extended 9–intersection method* which is a combination of the DEM and the 9IM (DE+9IM). In [13], Clementini *et al.* proves that the combination of these two point–set methods is necessary to match the expressive power of the CBM.

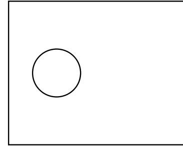


Figure 3.1: The *in* relationship

3.1.2 Spatial operators

There have been many attempts to define a query language that can handle spatial data. The most successful approaches seem to be those that have taken the standard query language for relational databases, SQL, as a basis [15, 21]. According to Clementini *et al.* [15], the reason for the success of the SQL based approaches, is that the contributors first developed an understanding of what requirements the spatial operators should satisfy. The OpenGIS Consortium has specified an extension to SQL that utilise the DE+9IM discussed in section 3.1.1 to handle spatial queries [49]. It has been adopted by well known database systems like Oracle and MySQL.

Clementini *et al.* [15], however stress that even though the OpenGIS extension contains several spatial operators, the set of operators is by no means complete and that more research in need to be done. They also describe a set of requirements the spatial operators should satisfy. One of these requirements is that the set of operators should be small. A large number of operators (the 9IM defines 56 relationships between 2–D geometric objects) would not be reasonable in a query language. The time required by users to master all the operators fully would simply be too long. *Generality* is another important requirement. The operators should be defined at a level of abstract geometrical data (points, line and regions), so that they are application independent. *Expressiveness* is another important requirement. The users should be enabled to formulate a wide range of spatial queries. The operators should be *consistent*, i. e. they should not give rise to ambiguities in computations or inconsistencies in the results to a query.

Shariff *et al.* [57] acknowledge that even though the spatial operators of a spatial query language have natural–language–like terms, like *connects*, *enters*, *leaves* and *near* for example, the formal definition of these terms rarely reflect the same meaning that people would associate with them when they are talking about a spatial relation. Even though there had been proposed many different models that tried to capture the semantics of spatial relations, these models most often stood only for themselves, which in turn led to languages that were mathematically well defined, but had little or no

link to the natural language semantics of the spatial operators. The work of Shariff *et al.* is an attempt to bridge that gap.

They performed an experiment with the purpose of investigating the understanding of English terms with respect to spatial relationship. They first developed a set of metric concepts to aid as a refinement to line/region relations. These concepts is *splitting*, which determines how the interior, boundary and exteriors of the regions and line are cut, *closeness*, which how far the boundary of a region is from different parts of a line and *approximate alongess* which is a combination of the closeness and splitting metrics.

The experiment consisted of a number of pictures with a region illustrating a park and an English-language sentence beneath. The participants were then asked to draw a line representing a road in relation to the park so that the resulting drawing would correspond to the English sentence. They then compared the different results to each other.

There has also been some research into the possibilities of making a temporal extension to SQL for use with databases that contains information on moving objects [66]. For this purpose, we would have to develop a set of temporal operators that are capable of retrieving data. If we known the present location, speed and heading of a car for example, we could perform a query about where the car would be in five minutes. There are many possible applications for temporal applications [65], the emergency services, delivery companies and taxis are but a small sample where these applications could be put to use. We will however not look further into this field, as we will not be looking at the temporal aspects of visual language specification.

3.2 Music notation

The modern, “normal” notation for music, called *orthochronic notation*[1], dates back to the beginning of the sixteenth century [55]. Before this period there had been no standard for music notation. That this particular form of notation was so successful, can according to Read [55] be considered as a consequence of the invention of music printing which occurred at about this point in time, not long after the invention of letter printing. An important benefit from the invention of music printing was the slow down of the evolution of notation, which made it “standardised”. The music notation as we know it today is thus not the result of a deliberate standardisation process. The advantage with the staff notation as we know it, is that it combines the visualisation of pitch and time in one process as many of the notation in our filed of computer science is. The symbols are characteristic for the subject

[1]from Greek *ortho*, meaning “correct” and *chronos*, meaning “time”

they represent and they are clear and concise (e. g. notes, rests and ornaments are clearly distinguishable). At last, but not least, the total number of symbols required is actually quite small [55].

A small example of standard music notation is shown in figure 3.2.

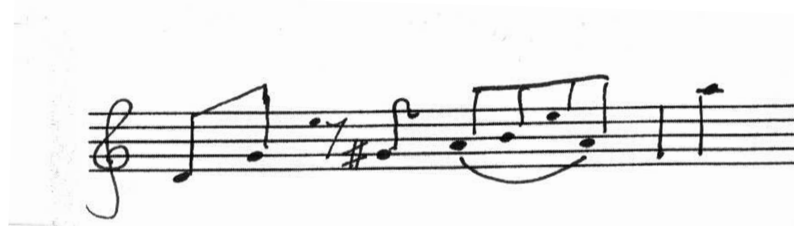


Figure 3.2: Standard music notation

As compositions grew increasingly more complex in the twentieth century, especially after the second world war, it became evident that the classical way of notation was inadequate for the new demands [62]. One aspect is rhythm. The rhythmic notation provided by the classical system is, with some exceptions, pulse–dominated, it presupposes a pulse. It is also regularly subdivided with only one geometric progression for all durations: 1, 2, 4, 8, 16, 32 etc. That is often not adequate for contemporary composers, as the rhythms they demand may be highly irregular, a–metrical or even pulse–less, according to Stone [61].

Another aspect is pitch. The classical western notation system is crafted for the tempered system. This system is based on half–steps between the different tones. After the second world war, there was a demand for a greater variety of pitches arose, for example quarter tone steps. This led composers to invent new symbols to indicate these steps. These symbols were of course different from composer to composer, so that there were no standards. Read [55] lists 24 of these symbols that indicates a quarter tone step in an upward or downward direction.

Some composers did not feel that the classical form of notation was able to express their music at all. They therefore invented their own notation based on their needs. Since this notation was not standardised anywhere, composers taking this approach soon found that they needed to include detailed instructions on the notation used in their compositions. Figure 3.3 shows an extreme example on how such a notation could look like. Without some explanation, it is quite hard to understand what the composer has intended. Some composers would here have given a detailed explanation, others would have stated that the notation is merely a guide to a performer and that the performer should improvise over it.

After the second world war, the quality and usability of electronic sound devices had increased so that composers were now able to create music electronically. Making music electronically, made composers independent of musicians and notation to a certain degree. A problem arose when scholars were to analyse the electronic pieces, look at its structure and so forth. To do so, one would have to rely on the sounds of the work only, as there were no This led Fennelly [24], based on experiments with composers and musicians, to develop a descriptive language that could be used to analyse this type of music.

This has been a very brief review of the field of music notation. Music notation is a very varied form of notation, ranging from the classical, "standard" notation to the notation used by the composers in the front of the avant–garde.

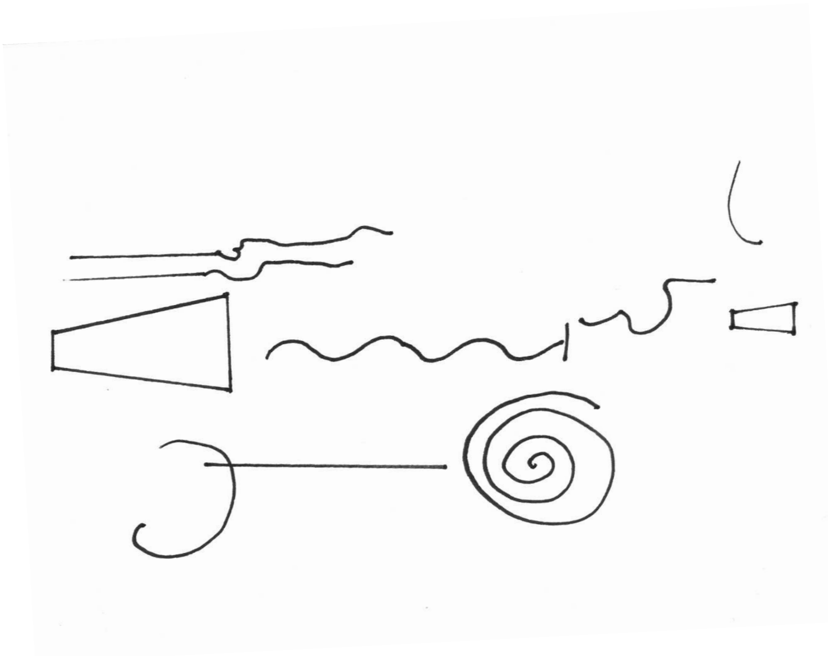


Figure 3.3: Example of avant–garde music notation

Chapter 4

Analysis

4.1 Introduction

The test case for this thesis is the *Unified Modeling Language, version 2.0* [48](UML 2.0). UML 2.0 [1] makes a good case for investigation, since it has no formal definition of the visual syntax.

The specification [48] does say how the different symbols of the language should look and what the semantics of the different relationships are. This is however done in plain text and not in any formal manner.

In this chapter we will investigate and analyse the visual syntax of UML to find what aspects of our meta–language will have to be able to address.

4.2 Categorisation

As an aid in finding the graphical problems, it is useful to make categories of graphical elements. In UML we have identified three different geometrical categories, *lines*, *text* and *objects*. The two first are self–explanatory but the latter one needs some explanation. Since this class of geometrical shapes must hold not only polygons, but rectangles with rounded corners, ellipses, circles etc., we have found that the term object sufficiently describes what we want to convey, a two–dimensional geometrical shape. When we put these in relation to each other, we get 9 categories of geometrical relationships.

However, there are three relationships which are permutations of some of the other, so we are left with six real relationships which are shown in table 4.1. These categories comprise what we will call the *geometrical level*.

[1] hereby refereed to as UML

Graphical primitive	Relation
Line	line – line line – text line – object
Text	text – text text – object
Object	object – object

Table 4.1: Geometrical relations

When we write diagrams using the above mentioned geometrical categories, for example into a diagram editor, it will most likely be represented internally with some kind of graph structure. For the editor, the underlying graph does not pose a problem for interpretation. Everything is clear and crisp. However, problems may arise when the diagram is to be interpreted by a human user. If the diagram is very complex, it is likely that misinterpretation will occur. We therefore introduce another level of analysis, which we will call the *communicative level*. At this level, we will try to find if there are any special issues that would need to be addressed to enhance human interpretation of the diagrams.

Graphical compactness is a category at this level where some information in a diagram is suppressed in order to emphasise the essence of the diagram. Another category at this level is what we will call *style*. By style we refer to the grouping of graphical elements, the direction of flow (see section 2.5 on page 25) in a diagram and other elements that can affect the layout of a diagram.

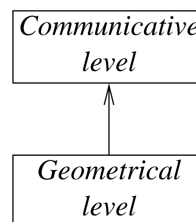


Figure 4.1: Analysis levels

The relationship between the levels, is illustrated in figure 4.1. As we can see, the communicative level relies on the geometrical level.

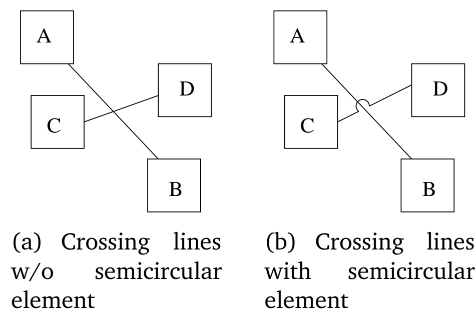
4.3 Geometrical level analysis

This section takes the categories in table 4.1 on the preceding page as its starting point and examines them one by one.

4.3.1 Line–line

One of the problems with lines, arise when they cross and is illustrated in figure 4.2(a). Here the line going from A to B is crossing the line going from C to D. In this case, the problem could be solved by rearranging the classes, but sometimes that is not possible.

It is often desirable to insert an additional graphical element, most often a semicircle, to ensure that the lines are distinguishable. i. e. to signify that the line crossing is purely graphical and does not have any logical implications. This is illustrated in figure 4.2(b). However, this addition of a semicircular element is only an optional requirement in UML [48].



(a) Crossing lines w/o semicircular element

(b) Crossing lines with semicircular element

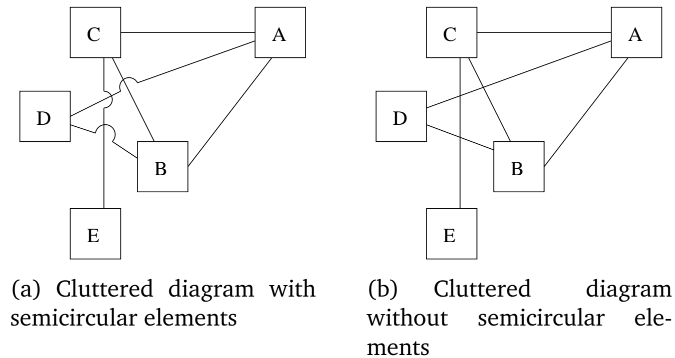
Figure 4.2: Line crossings

The insertion of a semicircular element to make lines distinguishable, can be a good aesthetic if the diagram is simple, as the diagrams in figure 4.2 are. If the diagram is a bit more complex, the semicircles can very easily make a diagram cluttered and more incomprehensible than without. As we can see in figure 4.3(a), this is a diagram that has a number of lines crossing. Some of the crossings are marked with semicircles to distinguish the different lines.

If the crossings are relatively close to each other, and there are many line crossings in a particular, perhaps small, area, the semicircles may make the diagram harder to comprehend. To determine how many crossings are too many and what “to close to each other” is, is however not an easy task.

Removing the semicircular elements, makes the diagram a bit easier to read, as we can see in figure 4.3(b), but it is still not satisfactory to a human reader.

To solve this, one would have to do something about the distribution of the boxes in the diagrams. This is discussed in later sections.



(a) Cluttered diagram with semicircular elements

(b) Cluttered diagram without semicircular elements

Figure 4.3: Cluttered diagrams

4.3.2 Line–text

In figure 4.4 we see an example of two text labels on an association line between two classes, A and B. Here, one can ask if the label 'n' is a name for the association between the classes, if it is a name for the role of this end of the association or if it represents the cardinality of this end of the association. As discussed in Morris *et al.* [44], an UML diagramming tool would have no problems in parsing this figure, knowing what is what in the diagram regardless of how the marks are placed in conjunction with the association line.

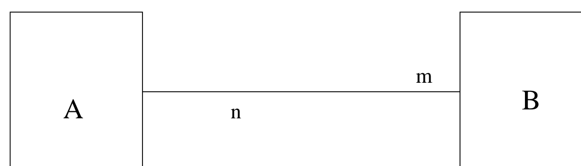
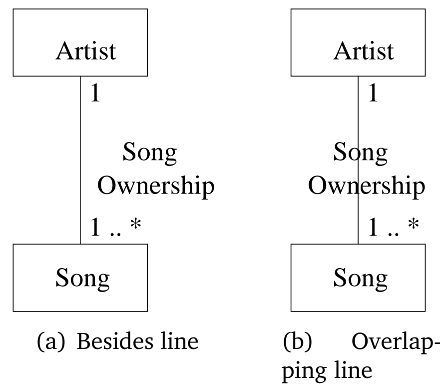


Figure 4.4: Association text labels

A human user however, could experience problems resolving this ambiguity without consulting the author of the diagram or using some parsing functionality of a tool to determine what the correct interpretation is.

The UML standard [48] states that "the association's name can be shown as a name string near to the association symbol, but not near enough to an end to be confused with the end's name".
Since the standard states that the name should be near the line, we interpret



(a) Besides line (b) Overlapping line

Figure 4.5: Placement of association text

that as it should not overlap the line. This is illustrated in figure 4.5. In addition, it is stated that the text should be sufficiently far from the endpoints of the line so that it will not be confused with the names on the ends.

In Purchase *et al.* [53], the authors tested how users preferred the placement of text in relation to lines. How would a notation where the text is rotated horizontally regardless of the direction of the line be considered compared to a notation where the rotation of the text always followed the direction of the line?

In figures 4.6 and 4.7, there are a few examples on a notation that has the text rotated to fit the direction of the lines. The experiment conducted in [53] found that most of the participants did not prefer this notations at all.

Their main objection to this form of notation, was as simple as that it made the text harder to read. If the diagram is on a sheet of paper, it is of course easy to just rotate the paper if the text is too hard to read. But it is a bit harder to do so if the diagram is displayed on a computer screen. Computer screens tend to be a bit larger and heavier than paper.

There was, however a small minority that preferred this notation because they thought that it was a more compact form of notation and that it made it more easy to follow associations between objects.

The examples in figures 4.8 on the next page and 4.9 on page 41 shows the form of notation that was preferred by a majority of the participants, illustrated with both class and sequence diagrams. This is the notation that also showed the best results at identifying correct diagrams with regard to a textual description.

The notation in figures 4.8 and 4.9 is the form of notation that has been utilised in the UML standard [48] and in the UML user guide by Booch *et*

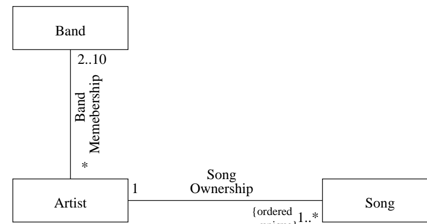


Figure 4.6: Complex text adjacency, class diagram

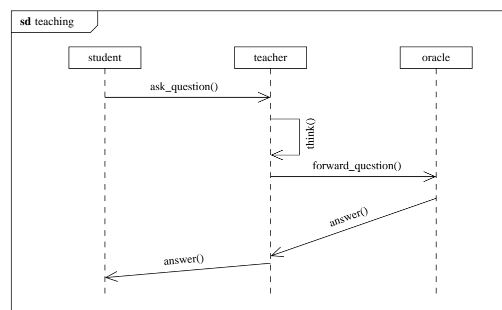


Figure 4.7: Complex text adjacency, sequence diagram

al. [5].

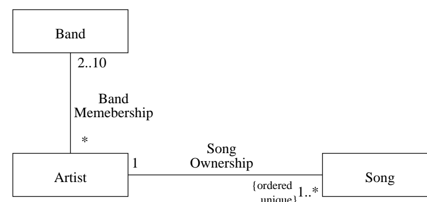


Figure 4.8: Horizontal rotation of text in adjacency to lines

4.3.3 Line–object

Almost all of the relationships between different types of objects in UML is expressed with the use of lines in some form. The lines are connected to the boundary of the objects and do not extend to the interior. The lines connecting two objects should not cross other objects. In figure 4.14 on page 43, we see two class diagrams where this is observed.

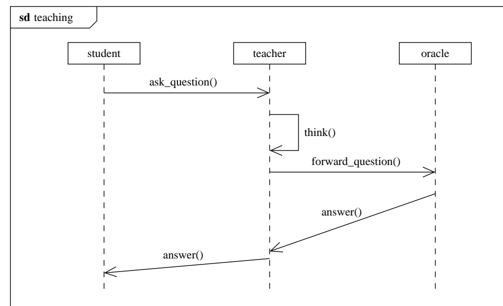


Figure 4.9: Horizontal rotation of text in adjacency to lines

4.3.4 Text–text

When we look at a string of text, it is easy to see that it can be treated as a line. At least in some cases. But text can also be viewed as an object, because it has an extent, i. e. it is not an one–dimensional entity as are lines.

The property that we most often is interested in when we look at text, is its meaning. We want to be able to read what it says. One reasonable constraint to put on a text entity in relation to another, is that the should not overlap. That actually goes for text versus any other graphical entity, for example line–text, that we discussed in section 4.3.2 on page 38.

4.3.5 Text–object

Figure 4.10 represents a simple UML class diagram. We see that the names of the classes is placed at different locations inside the class symbol. The UML standard [48] recommends that the name of the class is centred in the class symbol, so class *X* is drawn as recommended. Class *Y* has the class name the upper left corner. This is not considered as an error, but it is not optimal. So for a diagram to be coherent and easy to read, it is desirable that the symbols are drawn in the same manner.

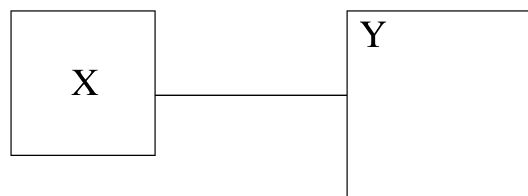


Figure 4.10: Example of insignificant differences in notation

However, looking at the diagram in figure 4.11, we see that the name of class *Y* is placed outside the class symbol. This is clearly a graphical error because the name of the class could then be interpreted as virtually any other text element that are used in class diagrams (that is, text that does not appear inside class symbols).

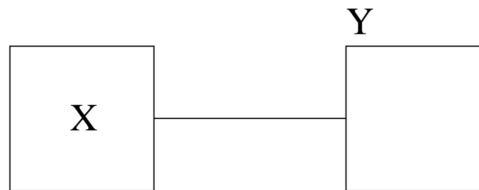


Figure 4.11: Example of significant differences in notation

So we need to be able to address the fact that the names of a class have to be inside the class symbol.

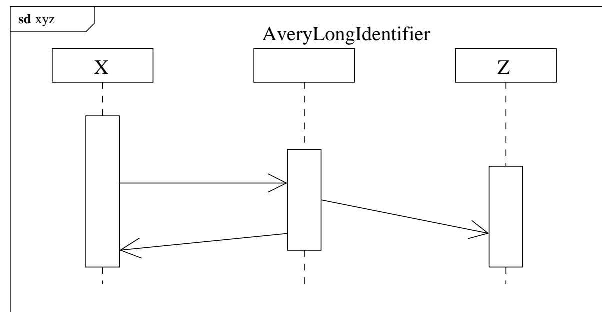


Figure 4.12: Acceptable variations in placement of text

With sequence diagrams we have to take another point into consideration. The heads of the lifelines, the boxes that represent the objects in the diagrams, normally are quite small compared to their counterparts in class diagrams. Therefore, if the text that is designating the name and type of the object is too long to fit into the box itself, it should be allowable to place the text outside and above the lifeline head, as illustrated in figure 4.12.

The situation illustrated in figure 4.13 on the next page is however not a good placement for long identifiers. Placed beneath the lifeline head, it is highly possible that it will be mistaken for some other element in the diagram.

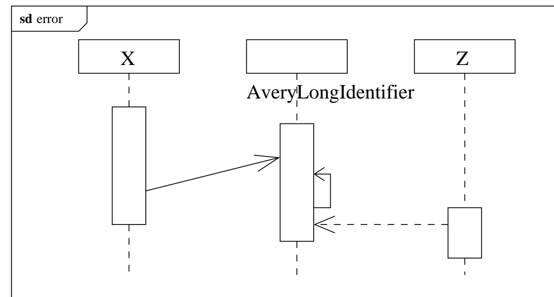
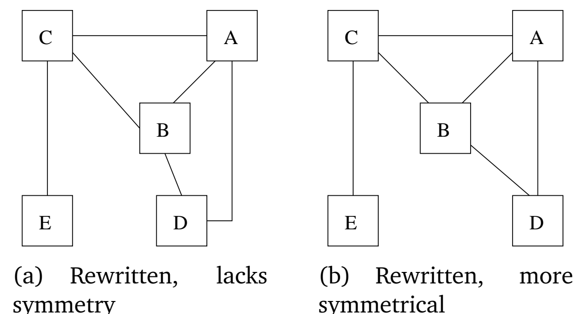


Figure 4.13: Erroneous placement of text

4.3.6 Object–object

In their article "Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study", Purchase *et al.* [52] perform some experiments to find out what aesthetics in a diagram makes it more or less comprehensive. They find that diagrams that have minimised the number of line bends and crossings and that have done something to optimise the symmetry of the diagram, produce the best results with regard to comprehension. In figure 4.14, we have rewritten the diagram from figure 4.3 on page 38 in two different ways to eliminate the crossings. The diagram to the left, fig. 4.14(a), has only one bend, but still appears a bit cluttered because of the lack of symmetry. The diagram on the right, fig. 4.14(b), has eliminated the bends and is rewritten a bit to be more symmetrical. This diagram appears more clean and comprehensive.

Much of the reason that this diagram is perceived to be more clear, is that the nodes are distributed more evenly and that the node "B", which is the node that have the most edges connected to it, is placed in the centre ("B" is the central node).



(a) Rewritten, lacks symmetry

(b) Rewritten, more symmetrical

Figure 4.14: Rewritten diagrams

4.4 Communicative level analysis

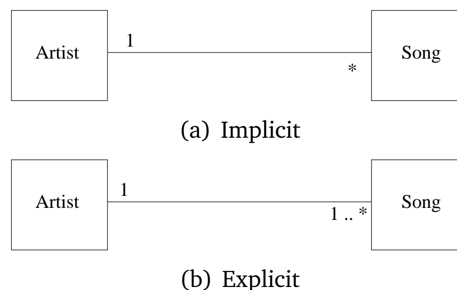
4.4.1 Graphical compactness

In [54], Purchase *et al.* perform an empirical study on which variant of a syntax variation for several different elements of a UML class diagram users prefer.

One of the elements tested, was how different notational styles for multiplicities was understood. In the UML standard, the general form of a multiplicity is defined to be on the form '*lower_bound .. upper_bound*'.

In the figures in 4.15, we see some variations on how to express multiplicities on an association. As the authors found, the subject preferred the notation in figure 4.15(b), because the multiplicities are made explicit. The notation in figure 4.15(a) implies, as the authors found, a $0..*$ multiplicity, not $1..*$. This is consistent with the UML standard [48], which states that if the lower bound of the multiplicity is equal to 0 and the upper bound is unspecified, one may use the alternative notation $*$ instead of $0..*$. If an association has a lot of adornments, using this alternative notation when appropriate could make a diagram easier to read.

Looking at the figures in 4.15, we also see another form of notation that does not use the 'lower .. upper'–form of notation. The UML standard states that if the lower bound is equal to the upper bound, an alternate notation is to use only the upper bound. In the figures in 4.15, we have used 1 instead of $1..1$.



(a) Implicit

(b) Explicit

Figure 4.15: Association multiplicities

A common practise when modelling, is to suppress or show the different compartments of a class symbol depending on what is modelled. If the diagram is meant to be used as a implementation diagram, i. e. to be used by a programmer, then one have to show all the attributes and operations of a class, according to Fowler *et al.* [25]. This is shown in figure 4.16 on the facing page.

Figure 4.16: Class, showing all attributes and operations

When, however the purpose of the diagram is to show the concept of a domain only, it is customary to suppress the attribute and operations compartments, because this is information that is not needed [25,41]. This is shown in figure 4.17. This is also how classes have been depicted throughout this chapter when only the name of the class has been of importance.

Figure 4.17: Class, all but name suppressed

A similar concept is shown in figures 4.18 and 4.19. Here we have a composite structure showing how a sliding bar icon is collaborating with a call queue in a call centre. The sliding bar shows operators how the call queue is developing (from [48]). Here, the standard gives two distinct ways of drawing. If we want to show the concept of the collaboration, we use the diagram in figure 4.18 and if we want to use it as an aid in implementation, we would use the diagram in figure 4.19.

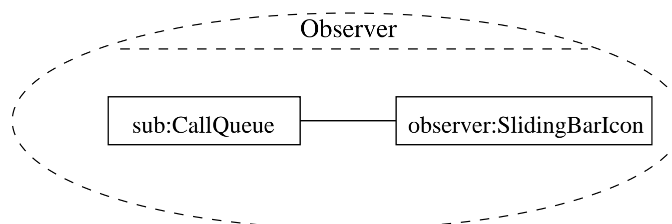


Figure 4.18: Composite structure, collaboration

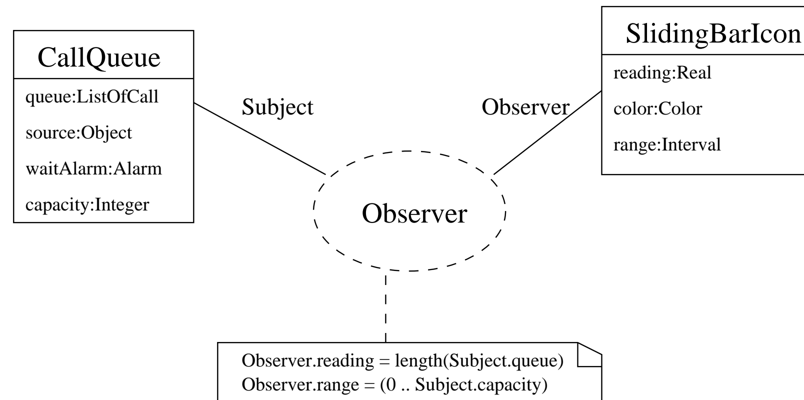


Figure 4.19: Composite structure, explicit collaboration

4.4.2 Style

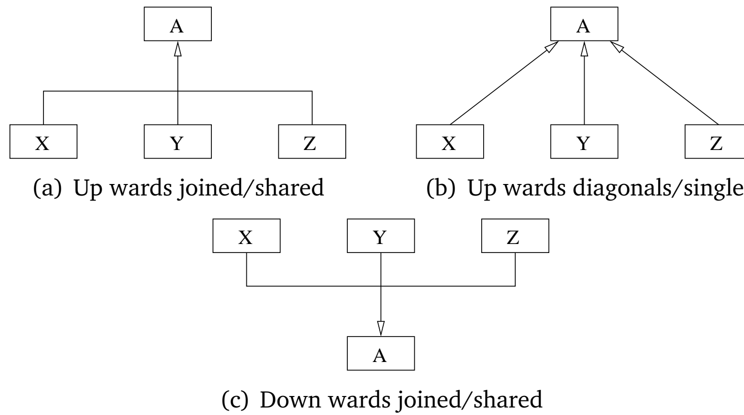
Another notation tested in [54], was which type of inheritance arcs produced the best results and in what direction the inheritance should go (upwards or down), called the direction of flow in the diagram. In figure 4.20, there are examples of three notation variants. Figure 4.20(a) and 4.20(b) shows the types of arcs that are used to signify inheritance. The notation in fig. 4.20(a) produced the best results when identifying correct diagrams and was preferred by most of the participants. The notation in fig. 4.20(b), however, produced better results when the task was to identify incorrect diagrams. One of the participants who preferred this notation said that this notation forced him to concentrate more on the meaning, because it was less intuitive than the variant in fig. 4.20(a).

With regard to the direction of flow, figures 4.20(a) and 4.20(c) shows the tested variants. The direction in fig. 4.20(a) produced better results with regard to identifying correct diagrams. The participants explained that they preferred this, because they read from top–to–bottom and it therefore was more natural to have the superclass above the subclasses. Again, the first variant, thought to produce better results than the second, does worse when it comes to identifying errors. The variant in 4.20(c) produced better results in this regard.

The UML standard [48] calls the style depicted in figures 4.20(a) and 4.20(c) “shared target style”. There is of no significance to the semantics of a diagram whether the shared or the single style is used.

Another UML diagram type that give users the possibility to use different drawing styles, is statemachines.

State machines utilise so called pseudo states to define entry and exit points



(a) Up wards joined/shared (b) Up wards diagonals/single

(c) Down wards joined/shared

Figure 4.20: Inheritance arcs and direction

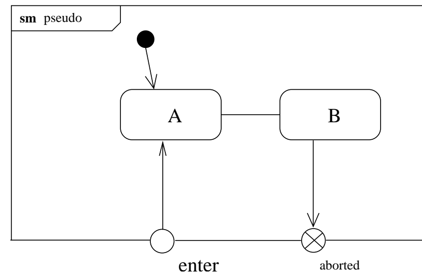


Figure 4.21: Placement of pseudo states on border

from a state machine. They also define initial (default), history and termination states.

The standard [48] says that the entry and exit points (represented by the pseudo states of a state machine) is shown with the appropriate symbol on the border of the state machine diagram, as shown in figure 4.21. An alternative notation is to place the symbol outside *or* inside the state machine diagram. This is shown in figure 4.22. Again, there is according to the standard, no semantic difference in the different ways to place pseudo states.

4.5 Summary

In this chapter we have provided an analysis of the graphical syntax of UML 2.0. We have highlighted issues that our meta–language will have to be able to address.

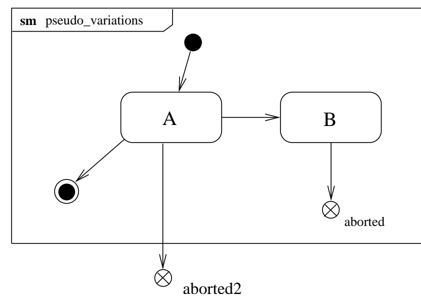


Figure 4.22: Pseudo states, alternative placement

Chapter 5

Graphical Description Language

5.1 Introduction

In this chapter we describe GDL, *Graphical Description Language* . GDL is a language that specifies valid syntactic constructions of a visual language.

In the initial work with the design of the language, we considered several different approaches. One approach would be to define everything from scratch ourselves, but as we pursued that approach, it soon became evident that we were actually reinventing the wheel. We then searched for other suitable methodologies that we could use as basis for our language. We found two possible candidates that we wanted to investigate further, the *Z notation* [60] and the *Object Constraint Language (OCL)* [47].

We decided to adopt the Z notation [60] as the basis for our language definition. The issues that led us to discard OCL as our basis and adopt Z is firstly that Z has existed longer than OCL and is thus more stable. Secondly, even though OCL is a precise specification language which has several features that are based on mathematics, it uses its own notation for this. We feel that the clarity of standard mathematical notation is preferable to the OCL version of these. Another issue is that we feel OCL is too bound to UML. Even though the test case for this thesis is UML 2.0, the intention with GDL that it should be more generic so that it can be used to specify other visual languages as well.

5.2 Brief introduction to Z

The Z Notation is based on typed set theory and uses standard mathematical notation to specify schemata [31, 59]. A schema is a collection of variables which are assigned a type and a collection of axioms that specify the relationship between these variables.

Schemata are notated vertically, with two compartments separated by a line.

The construct above specifies a schema named *S* with two variables above the short middle line, *x* and *y* which have the type of natural numbers Below the middle line we have a predicate which states that *x* is have to be smaller than *y* for this schema to hold. Z provides a convenient platform for GDL and we will use it in this chapter to define the predicates of GDL and in consequent chapters.

We will assume that the readers of this thesis has some knowledge of Z or mathematics beforehand, so we will not go further into details here and refer readers that are interested in learning more about Z to one of these excellent books [31, 59, 60].

5.3 Designating graphical elements

To be able to assign variable names to different graphical elements in diagrams, and to distinguish the variable names from textual elements that are actual parts of a diagram, we will adopt the following notation.

Variable names will be set in `typewriter font` and lines with arrows pointing from the variable to the graphical element it denotes will be set with a line consisting of dashes and points interleaved. As we will not typeset anything else in the example figures with typewriter font, the lines will be omitted if it is clear which graphical element the variable denotes. An example of this is illustrated in figure 5.1 on page 53. Here the two boxes and the texts A and B are parts of the diagram. `X` and `Y` are variables that designate one box each, designated by the “point–dashed” lines.

5.4 Visual symbols and attributes

To specify visual syntax, we need to be able to define how the different visual symbols relate to each other. Some relationships can be conveniently

be specified using definitions from topology. To specify other relationships that are not described as easily by topology precisely, we need the aid of geometrical attributes.

Using a grid with x and y axes, we adopt the technique devised by Golin [26] and assign the following four geometrical attributes to graphical symbols: *lx*, *rx*, *by* and *ty*. As described in section 2.1.3.3 on page 12, the attributes designate the leftmost x–position, the rightmost x–position, the lowest y–position and the topmost y–position of the visual symbol respectively. If the symbol is an object, the attributes designate the extent of the symbols and if it is a line, the attribute pair (*lx,by*) designate the left end of the line and the attribute pair (*rx,ty*) designate the right end of the line.

We need to be able to signify that lines have a start and end point, i. e. that it has a direction. The above mentioned attribute pairs is not capable of conveying this information. Lines are therefore given two attributes, named *tail* and *head*. These two attributes are inspired by the attributes of Shaw's [58] graphical primitives (see section 2.1.2.1 on page 8). Although the attributes in Shaw's formalism did not imply that the graphical primitives had a direction, out version of these attributes does. The attribute *tail* specifies the starting point of the line and the attribute *head* specifies the end point of the line.

We also need to signify the appearance of a symbol. This is solved by the attribute *appearance* and its value designate how the symbol looks. The values it can take, depends on the type of the symbol.

Lines can be *solid* or *dashed*. Objects can be *filled* or *transparent*. Filled is a value that the interior of the object has the same colour or pattern as the background of the diagram. The value *transparent* means that the object is essentially just a frame.

The attributes of the symbols are accessed using dot–notation. Thus, if we have a symbol *s*, we can access the attribute for the left–most x–position, *lx*, like this: *s.lx*. This can be read as "the *lx* of *s*".

Table 5.1 on the following page summarises the attributes.

5.5 The predefined predicates

GDL is a language that rely on predicates to specify the spatial relationship between the various graphical elements of the language. We provide a predefined set of predicates. Based on earlier research in this area [15], we will make the set of predefined predicates as small as possible. We will also try to be as precise as possible regarding the naming of predicates, keeping the research of Shariff *et al.* [57] in mind.

Type	Attribute	Description
Geometrical	*lx*	left–most x–position
	rx	right–most x–position
	by	lowest y–position
	ty	top–most y–position
Direction	*tail*	a line's start point
	head	a line's end point
Visualisation	*appearance*	how the symbol looks

Table 5.1: GDL symbol attributes

The predicates are designed in such a way that they can be read as a statement in natural language. We believe this to be a good principle make the understanding of them easier. Thus, a predicate stating *above(x, y)* can be read as "x is above y" and *disjoint(x,y)* can be read as "x is disjoint from y".

We will use topology to define the predicates where possible and the coordinate based method otherwise. This implies that we view the symbols as point–sets.

5.5.1 inside

Definition 5.1 (`inside`**)** *The* `inside`*(x,y) predicate defines that x is inside the object y.*

The `inside` predicate is specified in the schema *inside*.

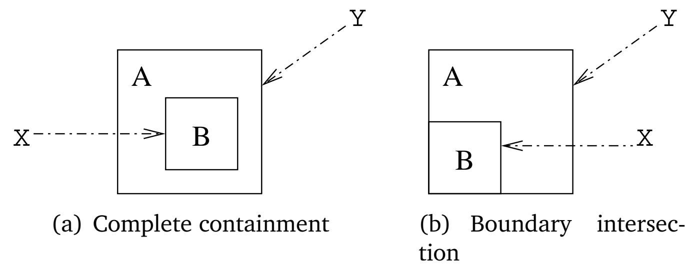
inside$[x, y : OBJECT]$

$x \subset y$

which specifies that x is a proper subset of y.

An example illustrating the inside relationship between two objects is shown in figure 5.1(a) on the next page, where the box designated by the variable `X` is contained in the box designated by the variable `Y`. The same relationship also apply to the two text fragments "A" and "B" in the diagram, where the text "A" is inside the box `Y` and "B" is inside `X`.

The two symbols may also have intersecting boundaries, but x must not be outside the boundary of y in any way. An illustration of this, can be found in figure 5.1(b) on the facing page.



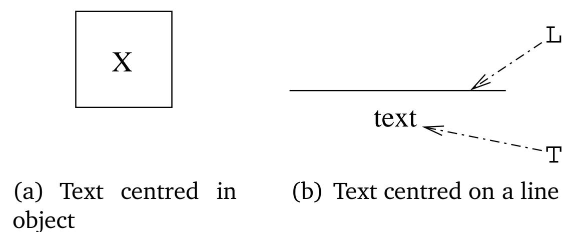
(a) Complete containment (b) Boundary intersection

Figure 5.1: Inside relationships (`inside(X, Y)`)

5.5.2 centerOf

Definition 5.2 (`centerOf`) *The `centerOf` (x,y) predicate defines that y should be at the centre of x.*

The `centerOf` predicate is used to describe that a graphical element should be at the geometrical centre of another graphical element. This does not mean that the graphical element *y* necessarily have to be inside the object *x*, but it can just as well be on the centre–point of a line. Figure 5.2(a) shows the use of `centerOf` in conjunction with `inside`, while figure 5.2(b) shows text that is centred on a line.



(a) Text centred in object (b) Text centred on a line

Figure 5.2: Use of `centerOf`–predicate

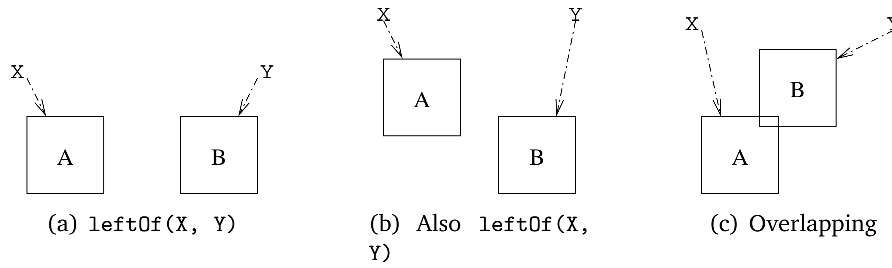
5.5.3 leftOf

Definition 5.3 (`leftOf`) *The `leftOf` (x,y) predicate defines that x is to the left of y.*

As we can see in the schema *leftOf* that specifies this predicate, it is defined that the right–most x–coordinate of the symbol *x* have to be smaller that the left–most x–coordinate of the symbol *y*.

$$\textit{leftOf}[x, y : \textit{OBJECT}]$$
$$x.rx < y.lx$$

Figures 5.3(a) and 5.3(b) are figures that satisfy this predicate, while figure 5.3(c) does not satisfy our condition.



(a) `leftOf(X, Y)` (b) Also `leftOf(X, Y)` (c) Overlapping

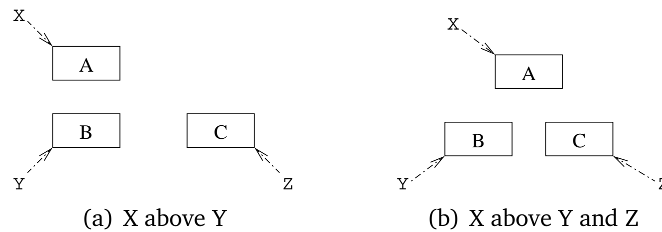
Figure 5.3: Different left-of adjacencies

5.5.4 above

Definition 5.4 (`above`) *The `above`(x,y) predicate defines x is spatially above y.*

The `above` predicate is a predicate that specifies that its first argument is above its second argument in the y–direction.

$$\textit{above}[x, y : \textit{OBJECT}]$$
$$x.by > y.ty$$



(a) X above Y (b) X above Y and Z

Figure 5.4: Different above adjacencies

5.5.5 overlap

Definition 5.5 (`overlap`) *The `overlap` (x,y) predicate defines that the symbol x is overlapping the symbol y.*

The overlap relationship is another spatial relationship that is suitable for topological definition.

$$\textit{overlap}[x, y : \textit{OBJECT}]$$
$$x \cap y \neq \varnothing$$

An example of the overlap relationship is illustrated in figure 5.3(c), where the two boxes overlap each other slightly.

5.5.6 disjoint

Definition 5.6 (`disjoint`) *The `disjoint` (x,y) predicate defines that the symbol x is disjoint from the symbol y.*

The disjoint predicate is a complement to the `overlap` predicate and is thus specified as follows:

$$\textit{disjoint}[x, y : \textit{OBJECT}]$$
$$x \cap y = \varnothing$$

Both figures 5.3(a) and 5.3(b) satisfy this predicate, while figure 5.3(c) does not, because this is a overlap relationship.

5.5.7 overlay

Definition 5.7 (`overlay`) *The `overlay` (x, y) predicate defines that the symbol x is overlaying the symbol y.*

Sometimes it is necessary to express that objects are layered. We use the `overlay` predicate for this purpose. This predicate describes an overlaying relationship. For two symbols *X* and *Y*, `overlay(X, Y)` means that *X* is visible but *Y* is not where the two objects overlap.

Figure 5.5 illustrates the `overlay` predicate. The box *X* is laid over the line *Y*, so that the line is not visible at the overlaying region.

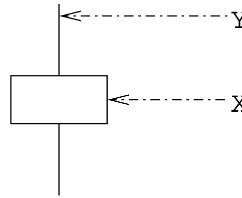


Figure 5.5: Overlay relationship

5.5.8 connectsTail and connectsHead

These two predicates connects lines with objects. We need two predicates to do so, because lines do have direction, as specified by the attributes *tail* and *head*.

Definition 5.8 (`connectsTail`) *The `connectsTail` (l,x) predicate specifies that the intersection between the line l and the symbol x is the point on the line designated by its attribute tail*

$$\textit{connectsTail}[l : \textit{LINE}, x : \textit{OBJECT}]$$
$$l \cap x = l.\textit{tail}$$

Definition 5.9 (`connectsHead`) *The `connectsHead`(l, x) predicate specifies that the intersection between the line l and the symbol x is the point on the line designated by its attribute head*

$$\textit{connectsHead}[l : \textit{LINE}, x : \textit{OBJECT}]$$
$$l \cap x = l.\textit{head}$$

Figure 5.6 on the facing page illustrates the concept of connection for the predicate
`connectsTail`. The connection point is shown as `L.tail`. The other end of the line *L* would then be *L.head* and had there been an object at that end, it would be connected to that object with `connectsHead`.

5.5.9 intersect

When we draw large and complex diagrams it is often hard, and sometimes even impossible, to draw lines between objects without having to have two or more lines cross. We therefore define a predicate `intersect`.

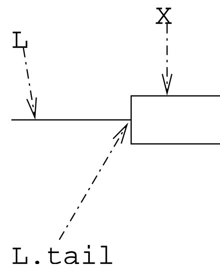


Figure 5.6: Connecting predicate (`connectsTail`)

Definition 5.10 (`intersect`) *The* `intersect` *(x,y) predicate defines that the line x is crossing the line y at a single point that is not the endpoint of y.*

Since lines are point sets as well, we can easily define intersection to be a single point, as we do in the schema *intersect*.

$intersect[x, y : LINE]$

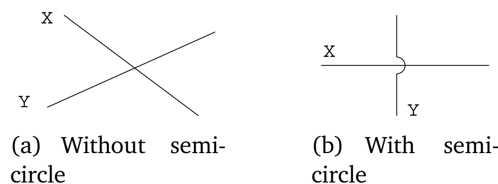
$\#(x \cap y) = 1$

$(x \cap y \neq y.head) \wedge (x \cap y \neq y.tail)$

Here we see that the cardinality of the set resulting from the intersection between the two lines is specified as being one. This is due to the fact that we consider the lines to be point sets (see section 5.4 on page 50) and the result is thus a single point where the two lines cross.

Examples of intersections are shown in figure 5.7.

Both of these figures satisfy the predicate `intersect`. Figure 5.7(b) is a special case which was discussed in section 4.3.1 on page 37 to visually signify that the intersection is purely a graphical intersection and that it does not have any logical implications. In spite of that, it still is an intersection between two lines graphically and thus it satisfies the predicate.



(a) Without semi-circle   (b) With semi-circle

Figure 5.7: Intersections

5.5.10 vertical and horizontal

These two predicates applies to lines.

Definition 5.11 (`vertical`) *The `vertical` (l) predicate defines that the line l is parallel to the y–axis of a coordinate system.*

Using the attributes of the line, we define the `vertical` predicate as follows for a line *l*:

$$\begin{array}{|l} \hline vertical[l : LINE] \\ l.lx = l.rx \\ \hline \end{array}$$

which unambiguously specifies that the line is vertical.

Definition 5.12 (`horizontal`) *The `horizontal` (l) predicate defines that the line l is parallel to the x–axis of a coordinate system.*

Again, using the attributes of the line, we get a definition of `horizontal` for a line *l* as:

$$\begin{array}{|l} \hline horizontal[l : LINE] \\ l.by = l.ty \\ \hline \end{array}$$

which unambiguously specifies that the line is horizontal.

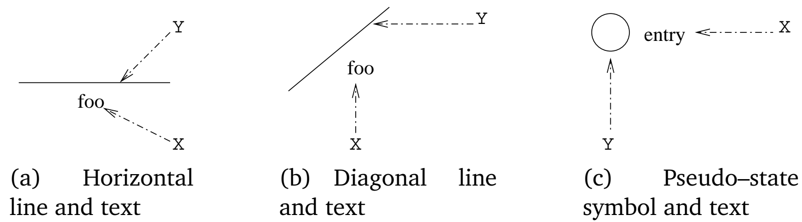
5.5.11 closeTo

Definition 5.13 (`closeTo`) *The `closeTo` (x,y) predicate defines that the symbol x is sufficiently close to the symbol y.*

`closeTo` is a predicate that we do not give a formal specification in Z. This predicate is of a type that Clementini *et al.* [15] would call a qualitative operator.

We adopt the notion that this predicate is true if its first argument is "close enough" to its second argument. This is an approach that is used both by Golin [26] and Jorge [35]. They introduce the concept of an adjacency region between the two symbol that should not contain any other objects.

Figure 5.8 on the facing page shows different `closeTo` adjacencies. In figures 5.8(a) and 5.8(b) we have a valid `closeTo` relationship between a line and a text, i. e. the text is close enough to the line in both cases. In figure 5.8(c) we see a pseudo–state symbol and a text. This also a valid construction.



(a) Horizontal line and text
(b) Diagonal line and text
(c) Pseudo–state symbol and text

Figure 5.8: Different `closeTo` realtions

5.6 Summary of predicates

Table 5.2 lists the predefined predicate set of GDL together with the type of relationship between visual symbols they apply to.

Relationship	**Predicate**
Object – object	inside
Any	centerOf
Object–object	leftOf
Object–object	above
Object–object, text–text or object–text	overlap
Any	disjoint
Any	overlay
Line–object	connectsTail
Line–object	connectsHead
Line–line	intersect
Line (unary predicate)	vertical
Line (unary predicate)	horizontal
Any	closeTo

Table 5.2: GDL predicates

Chapter 6

Defining a Visual Language

In the previous chapter we defined Graphical Description Language (GDL). This chapter will be devoted to the application of GDL, specifying the graphical syntax of a visual language.

A visualisation of the specification is shown in figure 6.1. As the basis, we have the Z notation, in which we will express the specification. Above that, we have GDL, which predicates we defined in chapter 5. At the top, we have T_{uml}, the language we will define in this chapter.

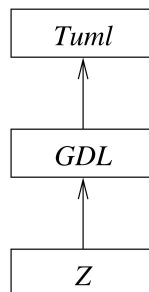


Figure 6.1: Structure of the specification

6.1 Tiny UML

We will show how GDL can be applied to a visual language by specifying the visual syntax of a subset of the *Unified Modeling Language, version 2.0* (UML). We will for clarity call this subset *Tiny UML*, abbreviated T_{uml}. The first and foremost reason that we only define a subset of UML, is that UML is a very large language encompassing a variety of different diagram types and it would prove too much for the scope of this thesis to define the visual

syntax of the entire language. Another reason is that we do not believe that it is necessary to define the entire UML to illustrate the potential of GDL.

UML 2.0 is defined using a meta–model that is notated in UML itself [48]. In this meta-model we can find all the concepts that users are familiar with. *Class*, *association*, *collaboration* and *generalization* are but a tiny selection. Some of the concepts of UML have a direct visual representation. Other concepts do not have a visual representation at all. The concepts of UML that we normally see in diagrams are typically defined with the aid of many other classes of concepts that have no inherent visual representation. One need only look through the UML 2.0 specification [48] to understand the complexity of the language definition.

The UML meta–model is quite large and defining all of the classes modelled in the UML meta–model would be impracticable and highly undesirable.Since T_{uml} is a subset of UML, it also has a meta–model. The meta–model of T_{uml} is a subset of the meta–model of UML in the same way as T_{uml} is a subset of UML.

The subset of UML that is T_{uml}, are the following diagram types:

- Class diagram
- Sequence diagram
- State Machines

6.2 Naming conventions

We will adopt the following naming conventions.

When we specify a schema for a concept that we can find in the meta–model, we will give that schema the same name as it has in the meta–model. That will normally mean that the name of the schema will start with an upper–case letter and the rest of the name will be in lower–case except in cases where we have a name that is composed of more than one word, a compound name, where the individual words will have upper–case letters, as in *ThisIsACompundName*. Schemata that do not have a counterpart in the meta–model but are defined as a kind of “utility” schemata, are given names that are written in all lower–case. When we refer directly to a symbol that is mentioned in the specification, it will be written in all upper–case, as in *SYMBOL*.

Variable names that occur in a schema, will be written in all lower–case.

In the UML specification [48], the specification for the various visual symbols are given in plain text. There is no formal definition of the appearance of the symbols. We adopt the practise of Diagram Interchange [46] by

not defining the actual appearance of the visual symbols in GDL, since they might change in the future.

We will, however give a figure depicting the symbols in question.

6.3 The graphical syntax of T_{uml}

This section gives a specification of the graphical syntax of T_{uml} using GDL. We will start with the top level schemata of the different diagram types and work our way down to the schemata at the lowest level.

6.3.1 Class diagrams

The class diagram is perhaps the most used and well known diagram type. We model the static, structural aspects of a system with this type of diagram.

This section will outline the GDL specification for class diagrams.

The top–level schema is *ClassDiagram*.

ClassDiagram

$cls : \mathbb{P}\, Class$
$assocs : \mathbb{P}\, Association$
$gens : \mathbb{P}\, Generalization$

$cls \neq \varnothing$
$\forall\, c1, c2 \in cls \mid disjoint(c1, c2)$
$\forall\, a \in assocs \bullet \exists\, c1, c2 \in cls \bullet a[c1, c2]$
$\forall\, g \in gens \bullet \exists\, c1, c2 \in cls \bullet g[c1, c2]$

Here we see that a class diagram consists of a set of classes, designated *cls*, a set of associations, designated *assocs* and a set of generalizations, designated *gens*.

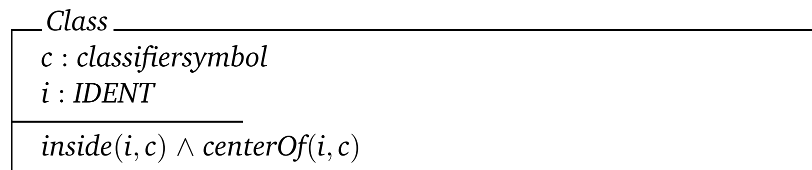
The predicate section of this schema, specifies that the set of classes cannot be empty. Further, it specifies that none of the classes that are members of this class diagram, may overlap, they must be disjoint. For all of the associations that are in the diagram, we specify that we have to apply the corresponding schema with the two classes that are part of that particular association. The same hold for all the generalisations that are part of the diagram. The schemata for these concepts are presented in later sections.

6.3.1.1 Class

A class specifies a classification of objects that have some common characteristics [48]. It specifies the name of the objects and what attributes and operations they are composed of.

The schema *Class* shows a specification for classes. A class is visualised by a classifier symbol, which is specified as a schema itself and by an identifier (which is the name of the class).

As we see in the predicate section of the schema *Class*, we specify that the name have to be inside the classifier symbol's name compartment and that it should be centred there. This rule is a response to the issues concerning classes we discussed in section 4.3.5 on page 41. A T_{uml} class is illustrated in figure 6.2.



Class

$c : classifiersymbol$
$i : IDENT$

$inside(i, c) \wedge centerOf(i, c)$

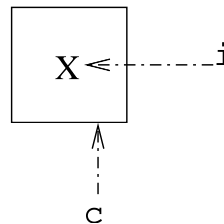


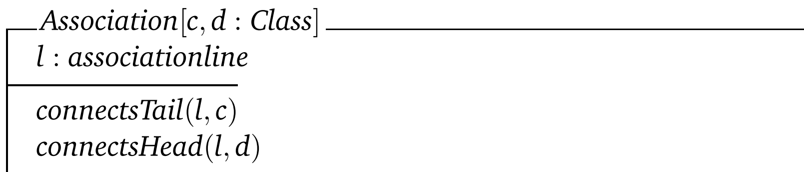
Figure 6.2: T_{uml} class symbol

6.3.1.2 Associations

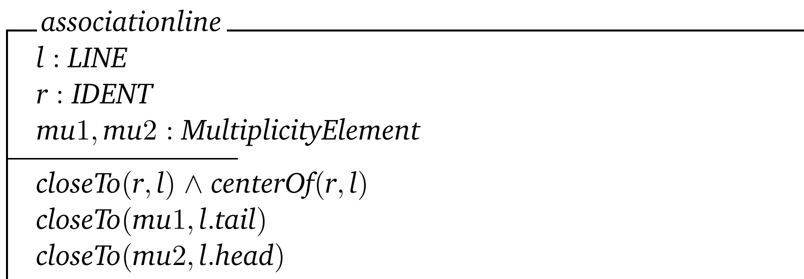
An association specifies a relationship between two classes.

The schema *Association* specifies an association.

The schema has two parameters, which both are of type *Class*. It also has one attribute, an association line. The predicate section specifies that the first parameter is connected to the tail–end of the association line and that the second parameter is connected to the head–end of the association line.



As illustrated in the schema *associationline*, an associationline is composed of a line symbol, an identifier and two multiplicity elements. The predicate section of that schema specifies that the identifier has to be close to the line and that it should be centred on the line. The multiplicity elements should also be close to the line, but not centred as the idenitifier, rather close to the tail and head ends of the line respectively.



Associations has multiplicities, represented by the type *MultiplicityElement*. Since a *MultiplicityElement* is a string, we can use BNF to specify it:

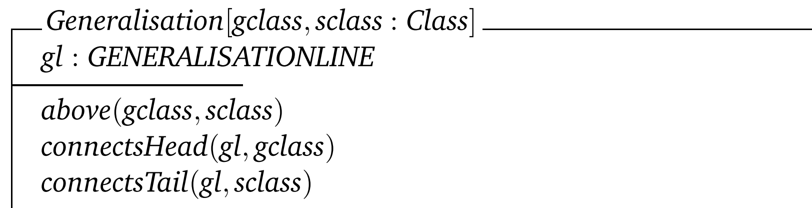
⟨*MultiplicityElement*⟩ ::= ⟨*multiplicity*⟩
⟨*multiplicity*⟩ ::= ⟨*multiplicity_range*⟩ ⟦ { ⟨*order_dsignator*⟩ } ⟧
⟨*multiplicity_range*⟩ ::= ⟦ ⟨*lower*⟩ .. ⟧ ⟨*upper*⟩
⟨*lower*⟩ ::= ⟨*integer*⟩ | ⟨*value_specification*⟩
⟨*upper*⟩ ::= ⟨*unlimited_natural*⟩ | * | ⟨*value_specification*⟩
⟨*order_designator*⟩ ::= ordered | unordered

This BNF–production, which is taken from the UML specification [48] and it specifies the legal format of multiplicities.

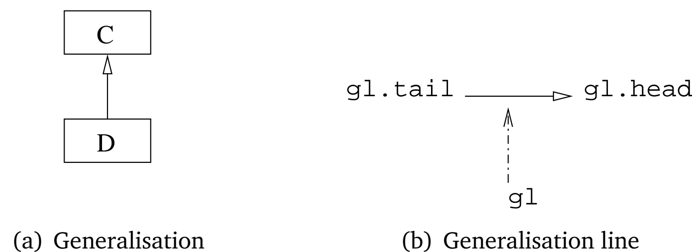
6.3.1.3 Generalisation

A generalisation is a relationship between twoclasses, where one of the classes is a generalisation of the other classes. The more specialised classes inherits the attributes and operations from the general class (this class is also known as a *super class* from object–oriented terminology).

The schema *ClassDiagram* specifies generalization.



As we can see, the schema defines only one attribute, a generalisation line, and two parameters which are of type Class. An example of a generalisation line can be seen in figure 6.3(b). The end of the line that is adorned with the hollow arrowhead is designated as the head end of the line. This end of the line is connected to the most general class, as we can see in the predicate section of the schema. The tail end of the line is thus connected to the more specialised class. In section 4.4.2 on page 46 we discussed some elements of layout style based on empirical research. One of the results from that research was that the preferred layout for generalisation is with the more general class above the more specialised classes. We therefore put the constraint on the general class that it should be above the special class, as seen in the first line of the predicate section.



(a) Generalisation (b) Generalisation line

Figure 6.3: Generalisation in T_{uml}

6.3.2 Sequence Diagrams

Sequence diagrams are a special kind of interaction diagram. Interaction diagrams are used to model and analyse the dynamic aspects of systems. UML has several different types of interaction diagrams in addition to sequence diagrams, but only sequence diagrams are part of T_{uml}.

Sequence diagrams are used to model and analyse the sequencing of message passing between instances in the system concerned.

This section will outline the GDL specification of sequence diagrams.

A T_{uml} sequence diagram with the different elements designated is illustrated in figure 6.4 on the facing page.

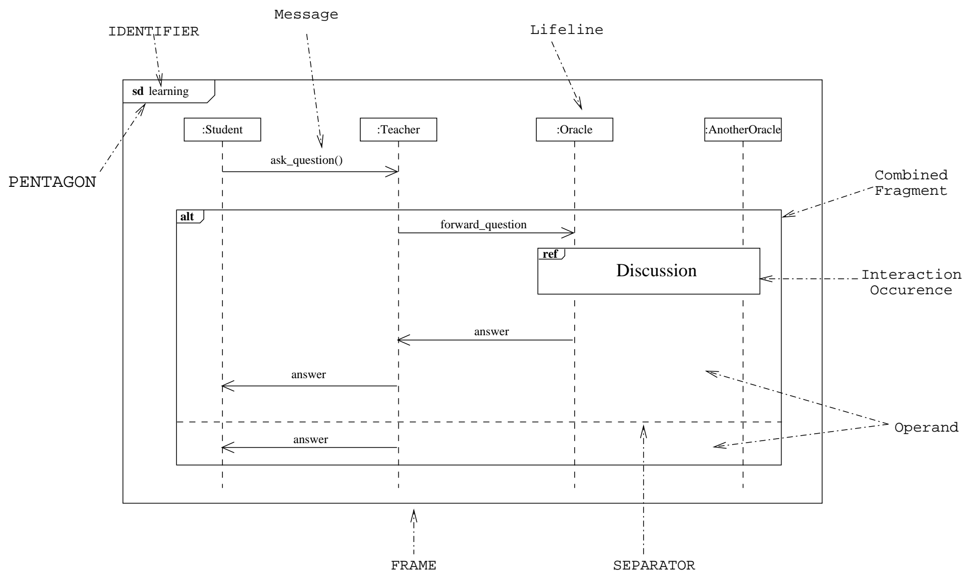
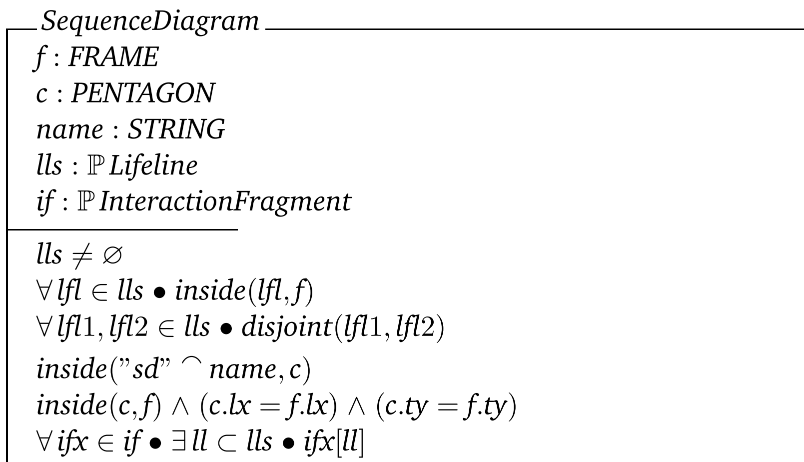


Figure 6.4: T_{uml} sequence diagram

A sequence diagram is represented by a frame with a compartment in the upper left corner that contains the interaction identifier. The schema *SequenceDiagram* specifies an interaction. Besides the frame and the name compartment, which contains the name of the interaction preceded by the string **sd**, we see that an interaction also has a set of life lines.



SequenceDiagram

$f : FRAME$
$c : PENTAGON$
$name : STRING$
$lls : \mathbb{P}\, Lifeline$
$if : \mathbb{P}\, InteractionFragment$

$lls \neq \varnothing$
$\forall lfl \in lls \bullet inside(lfl, f)$
$\forall lfl1, lfl2 \in lls \bullet disjoint(lfl1, lfl2)$
$inside(\text{"}sd\text{"} \frown name, c)$
$inside(c, f) \wedge (c.lx = f.lx) \wedge (c.ty = f.ty)$
$\forall ifx \in if \bullet \exists ll \subset lls \bullet ifx[ll]$

The predicate section of the schema *SequenceDiagram* defines that the set of lifelines that are part of this interaction cannot be empty. This makes sense because there will not be any interaction if there are no one that can interact. One could argue that we should put the constraint on the set of lifelines that it should have more than one member, but we see it as

possible that someone want to model the interaction an instance has with itself also known as a *self–call*. Furthermore, we demand that all the lifelines are contained within the frame of the interaction.

We also specify a constraint on the layout of lifelines within the frame. We do not want any of the lifelines to overlap each other. If they did, a diagram would appear very cluttered. This to address the issues described in section 4.3.6 on page 43, where we at the distribution of elements in diagrams. We do, however not impose any constraint that they have to be completely disjoint, because if there are many lifelines in a diagram, they might be placed very close to each other, maybe even touching each other. This is not an optimal situation however and one should consider a decomposition of the sequence diagram into smaller parts to solve this.

The frame has a name compartment, in which the name of the interaction is contained, preceded by the string “sd”. This identifies the interaction. We add a constraint to the name compartment, that it have to be inside the frame and that the leftmost x–coordinate and the upper y–coordinate have to be equal to its counterparts of the frame.

$$
\begin{array}{l}
\textit{InteractionFragment}[ll : \mathbb{P}\,\textit{Lifeline}] \\
\hline
c : \mathbb{P}\,\textit{CombinedFragment} \\
o : \mathbb{P}\,\textit{InteractionOccurence} \\
m : \mathbb{P}\,\textit{Message} \\
\hline
\#m + \#o + \#c = 1
\end{array}
$$

The schema *InteractionFragment* specifies interaction fragments. An interaction fragment is an abstraction of several interaction units [48]. An interaction fragment groups together several fragments of an interaction, as its name indicates.

As we can see, the schema contains three sets, one of type *CombinedFragment* (see section 6.3.2.2 on the facing page), one of type *InteractionOccurence* (see section 6.3.2.3 on page 70) and one of type *Message* (see section 6.3.2.4 on page 71).

The predicate section of the schema *InteractionFragment* specifies that the sum of the cardinalities of the three sets have to be equal to one. This is a modelling of a syntactic choice, which expresses that an interaction fragment is either a combined fragment, an interaction occurrence or a message.

6.3.2.1 Lifeline

A lifeline is a construct that represent the participants in an interaction. Each participant has its own lifeline.

Lifeline

$lh : lifelinehead$
$l : LINE$

$vertical(l)$
$l.appearance = "dashed"$
$above(lh, l)$
$connectsHead(l, lh)$

The schema *Lifeline* defines two variables, one of type *lifelinehead* and one of type *LINE*. The line's *appearance* attribute is given the value dashed. Furthermore, we apply some constraints on the line, namely that it have to be vertical and below the lifeline head (or, as the predicate actually reads, the lifeline head is above the line). The last predicate states that the *head*–end of the line and the head are connected.

The type *lifelinehead*, which is actually defined as a schema itself, is not a type that can be found in the T_{uml} meta–model. It is a part of the lifeline itself, containing its name. We have defined it as a schema for itself to separate out the particularities concerning this symbol.

lifelinehead

$ls : LIFELINEHEADSYMBOL$
$li : LIFELINEIDENT$

$inside(li, ls) \vee above(li, ls)$

The schema *lifelinehead* specifies a constraint on the spatial relationship between the "head symbol" and its identifier. The identifier can be either inside the symbol or above it. This is analogous to what we discussed in section 4.3.5 on page 41, where we discussed the placement of identifiers for lifelines. We do not give a specific constraint on how long the string have to be before it could be placed above the head, because that will depend on the size of the lifeline head and the font size of the string. This will have to be up to a tool creator or a modeller to decide.

6.3.2.2 Combined Fragment

A combined fragment is a definition of different interaction fragments [48]. Its type depends on its interaction operator. A combined fragment can overlap a number of lifelines, thus specifying the operation carried out on these life lines (i. e. interaction fragments).

$$
\begin{array}{l}
\textit{CombinedFragment}[lls : \mathbb{P}\,Lifeline] \\
\hline
cfs : COMBINEDFRAGMENTSYMBOL \\
nc : PENTAGON \\
opr : INTERACTIONOPERATOR \\
ops : \mathbb{P}\,Operand \\
sep : \mathbb{P}\,SEPARATORSYMBOL \\
\hline
cfs.appearance = transparent \\
inside(opr, nc) \\
inside(nc, cfs) \wedge (nc.lx = cfs.lx) \wedge (nc.ty = cfs.ty) \\
\forall\, ll \in lls \bullet overlap(cfs, ll) \\
\#sep = (\#opr - 1) \\
\forall\, o1, o2 \in opr \bullet \exists\, sp \in sep \bullet above(o1, sp) \wedge above(sp, o2)
\end{array}
$$

The schema *CombinedFragment* specifies combined fragments.

The symbol designating a combined fragment has its *appearance* attribute set to transparent. Furthermore, the pentagon is placed at the same position as the pentagon is for the frame of sequence diagrams. The interaction operator is specified inside the pentagon.

The schema has as its input parameter a set of lifelines. The predicate section of the schema specifies that each of the elements in that set, has to be overlapped by the combined fragment symbol.

The schema also defines a set of *Operands* and a set of separator symbols. The operands are represented by the compartments in a combined fragment. If there are more than one operand, the operands are divided by a separator symbol. In figure 6.4 on page 67 there are two operands, separated by a dashed line, the separator symbol. The predicate section of this schema specifies that the number of elements in the set of separator symbols must equal to the number of elements in the set operands subtracted by one. Further, it is specified that the operands are organised vertically with the separator symbols in between.

6.3.2.3 Interaction Occurrence

An interaction occurrence is a reference to an *Interaction* [48]. That is, the content of the interaction that is referenced by the interaction occurrence is copied into the place where the interaction occurrence is.

We will only be concerned with the graphical aspect of this.

$$\begin{array}{l}
\textit{InteractionOccurrence}[lls : \mathbb{P}\,Lifeline] \\
\hline
ics : COMBINEDFRAGMENTSYMBOL \\
nmc : PENTAGON \\
name : STRING \\
\hline
ics.appearance = filled \\
inside(nmc, ics) \land (ics.lx = nmc.lx) \land (ics.ty = nmc.ty) \\
inside(\text{"}ref\text{"}, nmc) \\
inside(name, ics) \land \neg\, inside(name, nmc) \\
\forall\, ll \in lls \bullet overlay(ics, ll) \\
\forall\, ll' \notin lls \bullet overlay(ll', ics) \lor disjoint(ll', ics)
\end{array}$$

As we see in the schema *InteractionOccurence*, it is represented by the symbol for a combined fragment. The difference here, is that its operator is the keyword **ref**. The name of the occurrence is contained in the main area of the symbol.

The pentagon, containing the operator **ref** is placed inside the combined fragment symbol, in its upper left corner. The name of the interaction occurrence is to be placed inside the main symbol, but not inside the pentagon. The schema has one input parameter, a set of lifelines. This set contains the lifelines that are part of the interaction occurrence. All of the lifelines that are elements of this set, are specified to be overlaid by the symbol of the interaction occurrence. Thus, as specified in the last line of the schema, the lifelines that are not part of this input set are not part of the interaction occurrence and are thus not overlaid by its symbol. Instead, they may themselves overlay the interaction occurrence symbol or they may be disjoint from it.

6.3.2.4 Message

A message is a visualisation of communication between two lifelines. There are several types of messages, distinguished by their visual representation. Asynchronous messages have solid lines with an open arrowhead. Synchronous messages have the open head replaced with a closed arrowhead. There is usually associated a return message with a synchronous message and this is represented by a dashed line with an open arrowhead.

$$\begin{array}{l}
\textit{Message}[ll1, ll2 : Lifeline] \\
\hline
msgid : STRING \\
ms : MESSAGESYMBOL \\
\hline
closeTo(msgid, ms) \land disjoint(msgid, ms) \land centerOf(msgid, ms) \land \\
\quad horizontal(msgid) \\
(connectsTail(ms, ll1) \land connectsHead(ms, ll2))
\end{array}$$

The schema *Message* specifies a message. It consists of two symbols, the message symbol and an identifier. The identifier is the name of the message, often it represent a call for a procedure in the recipient instance. The predicate section of this schema states that the message identifier can not be absent and furthermore, that the identifier have to be centred on the message symbol, not overlapping it. Figure 6.5 illustrates a T_{uml} message symbol. The t and h in the figure denotes the *tail* and *head* attributes respectively.

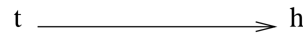
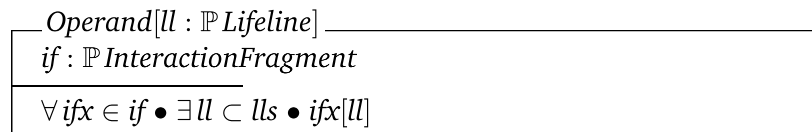


Figure 6.5: T_{uml} message symbol

The first parameter, *ll1*, is connected to the tail end of the message symbol. This is the sending end of a message. The second parameter, *ll2* is connected to the head end of the message symbol and is at the receiving end of the message. In figure 6.4 on page 67, we can see an example of message passing. There, *ll1* would be the lifeline labelled *:Student* and *ll2* would be the lifeline labelled *:Teacher*. The message originates at *:Student* and ends at *:Teacher*.

6.3.2.5 Operand

As we see in the schema *Operand*, an operand contains a set of interaction fragments. An operand has one input parameter, a set of lifelines. Operands allow for nesting of interaction fragments, which is illustrated by interaction occurrence “Discussion” in figure 6.4 on page 67.



The predicate section of this schema specifies that for all the interaction fragments that are contained in this operand, there has to exists a proper subset of the input set of lifelines, so that the interaction fragment is applied to it.

6.3.3 State Machines

As with sequence diagrams, state machines are used to model the dynamic aspects of a system.

While sequence diagrams are used to model the interaction, i. e. message passing between different instances, state machines are used to model the

different states a system can be in and the different (legal) transitions between the states.

This section will outline the GDL specification for state machines. Figure 6.6 on page 75 illustrates a state machine and its different parts.

StateMachine

$$
\begin{array}{l}
sms : FRAME \\
pn : PENTAGON \\
stn : STRING \\
ss : \mathbb{P}\, State \\
trs : \mathbb{P}\, Transition \\
pss : \mathbb{P}\, PseudoState \\
\hline
sms.appearance = transparent \\
inside(pn, sms) \land (pn.lx = sms.lx) \land (pn.ty = sms.ty) \\
\forall\, ps \in pss \bullet overlay(ps, sms) \\
\forall\, s \in ss \bullet inside(s, sms) \\
\forall\, s1, s2 \in ss \bullet disjoint(s1, s2) \\
\forall\, t \in trs \bullet \exists\, s1, s2 \in ss \bullet t[s1, s2]
\end{array}
$$

As we can see, the schema specifies that a state machine consists of a frame, a pentagon and a string which is the name of the state. Furthermore, we have a set of states, a set of transitions and a set of pseudo states.

The predicate section of this schema, defines that the appearance–attribute of the frame should be transparent and that the pentagon have to be inside and in the top left corner of the frame. For the set of pseudo states, it is specified that all the elements of it have to overlay the frame of the state machine. This is a restriction made to T_{uml} that is not made in UML. This was discussed in section 4.4.2 on page 46. In UML, the pseudo state symbols may be either inside the state machine, on its frame or outside it.

It is specified that all the states that are part of this state machine, must be contained within the frame, in the same manner as was done for lifelines in sequence diagrams.

The last line in the predicate section of *StateMachine* specifies that each transition, there have to two states that the transition is applied to.

6.3.3.1 State

The schema *State* specifies the appearance of states.

The schema defines two variables, a state symbol and a string, which is the name of the state. The predicate section of this schema defines that the name has to be contained within the state symbol.

$$
\begin{array}{|l}
\textit{State} \\
\hline
ss : STATESYMBOL \\
sn : STRING \\
\hline
inside(sn, ss) \\
\hline
\end{array}
$$

6.3.3.2 Pseudo states

A pseudo state is an abstraction for a number of different types of states. The semantics of a pseudo state depends on its kind, which in turn determines the visual representation of the pseudo state. The ten different kinds of pseudo states can be seen listed in the schema.

The schema *PseudoState* specifies a pseudo state.

$$
\begin{array}{|l}
\textit{PseudoState} \\
\hline
kind? : kinds \\
kinds : \{entryPoint, exitPoint, initial, deepHistory, shallowHistory, \\
\quad join, fork, junction, terminate, choice\} \\
fpsym : kinds \twoheadrightarrow \mathbb{P}\, PSEUDOSTATESYMBOL \\
psym : fpsym(kind?) \\
txt : STRING \\
\hline
closeTo(txt, psym) \\
\hline
\end{array}
$$

The schema declares two variables, a set containing the kind of pseudo states and a set containing the symbols that a pseudo state can have. It also declares a total surjective function *fpsym*, which maps the kind to the corresponding symbol. That is to say, that for every element in the set of pseudo state symbol there are at least one element in the set of pseudo state kinds that maps to it. This also means that there are not necessarily one symbol for each kind, but that one symbol may designate more than one. The variable *psym* is the assigned the symbol of the particular state as a result of applying *fpsym* with the input variable *kind*.

A pseudo state may be labelled, as specified with `closeTo` in the predicate section of the schema.

6.3.3.3 Transition

A transition is a relationship between two states. It is a visualisation of the transition that brings the state machine from one state to another.

$$
\begin{array}{|l}
\hline
\textit{Transition}[s1, s2 : \textit{State}] \\
ts : \textit{TRANSITIONSYMBOL} \\
ti : \textit{TransitionText} \\
\hline
\textit{closeTo}(ti, ts) \\
\textit{connectsTail}(ts, s1) \land \textit{connectsHead}(ts, s2) \\
\hline
\end{array}
$$

A transition has two input parameters, two states. The schema also defines two variables, the transition symbol itself and a text that specifies what triggered the transition and what its effect is. The transition text specified in BNF:

$\langle \textit{TransitionText} \rangle$ ::= trigger / effect

The identifier is specified to be close to the transition symbol. From the input set of states, there have to be two states, *s1* and *s2*, that is connected to the transition symbol. One of the state symbols is connected to the tail end of the transition symbol. This is the originating state of the transition. The other state symbol is connected to the head end of the transition symbol and is thus the resulting state of the transition.

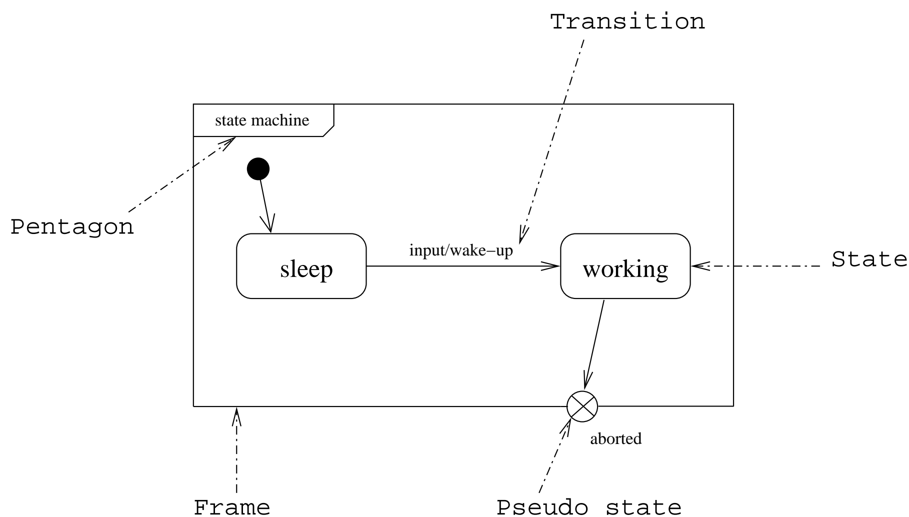


Figure 6.6: T_{uml} state machine

Chapter 7

Discussion and Further Work

We aimed to design a language that could act as a meta–language for the specification of visual languages, a language which could be used for formal definition of spatial syntax without having to be concerned with the graphical primitives.

To find out what issues the language had to address, we did an analysis of UML 2.0, which is a complex language that encompasses wide array of different diagram types. We felt that UML 2.0 would be a suitable language to analyse, since it does not have a formal specification for spatial syntax.

We then defined GDL, Graphical Description Language, a language for specifying spatial syntax. Using GDL, we defined T_{uml}, a language that closely resembles UML 2.0 and therefore faces the same challenges when it comes to the spatial syntax.

In this chapter we evaluate GDL with regard to the design criteria described in section 1.3. We also evaluate the language with regard to the analysis done in chapter 4.

7.1 Evaluation of Requirements

In section 1.2 on page 3 we presented some requirements that we felt the language had to meet in order for us to achieve the goal of this thesis.

This section investigates if the requirements were met.

The analysis in chapter 4 is performed after a categorisation of relationships between different types of visual symbols (see table 4.1 on page 36). This made it easier to identify the different issues we has to address. Based on this, we designed the predicates of GDL. The predicates are listed in table 5.2.

7.1.1 The geometrical level

In section 4.3.1 on page 37 we discussed the relationship between lines. We needed to be able to specify that lines may cross each other. This is solved by the `intersect` predicate. We also discussed the practise of inserting a graphical element, a semi–circle, at the intersection point to signify that the lines only intersect graphically, not logically, as illustrated in figure 4.2(b) on page 37. We have not addressed this explicitly, because we feel that this is not an issue that is crucial to the layout of a diagram. As noted in this section, these semi–circles may contribute to that a diagram looks cluttered, as illustrated in figure 4.3(a) on page 38.

In section 4.3.2 we analysed the line–text relation. The main issue here is that a string of text is in locates in close proximity to a line. This is solved by the predicate `closeTo`. Although this predicate is specified somewhat loosely in plain text, it is our belief that this is sufficient. Used in conjunction with the predicate `centerOf` and the attributes *tail* and *head*, we are able to specify where on the line the text is located. With the `disjoint` predicate, we are able to precisely specify that the text should not overlap the line. The `horizontal` predicate is able to specify that the text should always be horizontally rotated to enhance readability.

In section 4.3.3 on page 40 we looked at the relationship between lines and objects. We have been able to express that lines connects to objects, using the connective predicates `connecsTail` and `connectsHead`. These predicates connects a line with an object, each at one end of the line.

With regard to the text–text relationship discussed in section 4.3.4 on page 41, we are able to specify that text strings should not overlap each other. This is the only issue we looked at regarding this relationship.

The text–object relationship, which we discussed in section 4.3.5 on page 41, is another relation that is easily specified with GDL. Using the `inside` predicate, we are able to place a string of text inside an object. We can further specify that it have to be centred using the `centerOf` predicate. With regard to T_{uml} lifelines, specified in section 6.3.2.1 on page 68, we can specify that the identifier may be inside the lifeline head or above it, depending on its length.

In section 4.3.6 on page 43 we analysed the object–object relationship. In GDL, we are able to specify that objects may be above each other or to the left or right of each other. We are also able to specify that an object may be inside another.
The issue regarding the distribution of objects and symmetry of a diagram, is however not something GDL is able to address.

7.1.2 The communicative level

The issues discussed in section 4.4 on page 44, concerning the communicative aspects of diagrams are partially solved. We are able to specify the direction of flow in a diagram, as illustrated in figures 4.20(b) on page 47 and 4.20(c) on page 47, using the `above` predicate. We should also be able to specify the same in the vertical direction using the `leftOf` predicate.

7.1.3 Requirement of Precision

In section 1.2 on page 3 we also stated that we wanted a language that could precisely specify the spatial relationships between symbols with some precision. We will now discuss if that requirement is met.

Since we are using set theory and predicate logic as the basis for GDL, this implies a certain degree of precision. By defining the visual symbols as point sets, we were able to use theories developed with topology for use with geographical information systems to determine the topological relationship between objects. This gives improved precision for specifying spatial relationships. However, not all of the spatial relationships we needed were suitable for topological definition. This issue had to be solved with other means.

We found the solution to this problem in the technique devised by Golin [26]. Here we assume a two–dimensional grid on the diagram and assign four attributes designating two x–positions and two y–positions to the symbols. By constraining the relationships between the different attributes of the symbols in a relation, we were able to specify the spatial relationships that did not fit into a topological definition.

With regard to the requirement of precision, there are three relationships we need to explain further. The `overlay`–predicate (specified in section 5.5.7 on page 55) is a predicate we did not give a formal specification using the above mentioned techniques. If we were to specify this predicate formally with Z, we would have to assign the symbols in this relationship an attribute that would specify its position on an z–axis of the coordinate system. This would have added a third dimension to the diagram, which was not desirable since UML is graphically a two–dimensional language. But we still needed to be able to express that a symbol is actually covering another. We could have adopted a concept of layers, similar to that of many image processing applications, but this is on closer inspection not an option either. We will explain why. The graphical interaction between lifelines and interaction occurrences may be viewed as a form of weaving, which can be seen in figure 7.1 on the following page. Here we can see two lifelines (*X* and *Y*) and two interaction occurrences (*Foo* and *Bar*). As we can see, the interaction occurrence *Foo* is covering the line of *X* and is covered by the line of *Y*. *Bar*

is covered by the line of *X* and covers the line if *Y*. Thus, *X* is part of *Foo* and *Y* is not. Conversely, *Y* is part of *Bar* and *X* is not part of this interaction occurrence. We would not have been able to address this situation using layers. We therefore decided to rely upon a textual description of this relationship.

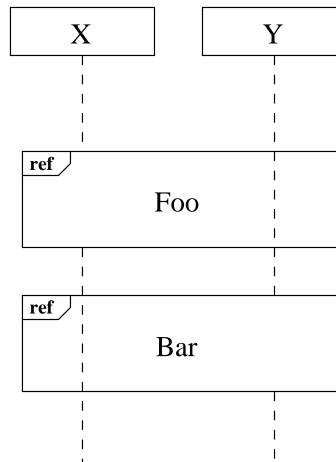


Figure 7.1: "Weaving" of graphical symbols

The `centerOf`–predicate is another relationship that was not specified formally with Z. The reason for this, is that "center of" can be interpreted in more than one way. We can for example have that a symbol *x* have to be centred in an object *y* or we may have that a text string is centred on a line, as illustrated in figure 5.2(b) on page 53. Therefore, we opted to specify this predicate in plain text.

The predicate `closeTo` was not specified using Z either. The reason for this, is that what is "sufficiently close" (see definition 5.13 on page 58) may vary. Because of this, we felt that it would be better to define it using plain text.

7.2 Evaluation of design

In section 1.3 on page 3 we gave some criteria that we felt we had to follow in order to ensure that we would be able to design a good enough language. We will now review the design of GDL with respect to those criteria.

As described in chapter 5, we chose the Z notation as the basis for our meta–language. By choosing the Z notation [60] we get a predefined, formal framework to build upon.

With respect to our criteria of *transparency*, we feel that our design fulfil this criteria. As we noted in section 1.3, Wexelblat [64] points out that there are certain "standard" ways of notation and that large deviations from such

standards could make a language harder to comprehend that it really needs to be. The mathematical and logical notations are standardised notations and most computer scientists have some training in both mathematics and logic. The schemata of Z may require some background, but our experience is that this concept is quickly comprehended.

The form of Z notation that we have used, notating the schemata vertically with the variables and predicates divided into two compartments, provide a convenient structuring mechanism, which gives us the opportunity to structure the specification into logical parts, as we did with the specification of T_{uml} in chapter 6.

The use of mathematical notation gives us a certain amount of compactness in the specification. Mathematics is a language that is capable of saying a lot with few symbols. Although the symbols are abundant considering mathematics as a whole, our experience is that there is not that many symbols that are used in practise (see the schemata through chapter 6).

7.3 Further work

This section will outline the possibilities for on the solutions presented in this thesis.

7.3.1 Applicability to other languages

The set of predefined predicates of GDL currently counts 13 predicates that specify different spatial relationships. If GDL is to be applied to other visual languages than UML, it is possible that this set of predicates need to be expanded or that the definitions of the predicates may be changed.

GDL is designed with UML 2.0 in mind. The predicates relating symbols are therefore designed with regard to the issues we found in the analysis of UML 2.0. We believe, however, that the predicate set is a set that is generic enough to address issues found in other languages as well. Languages that spring to mind as possibly suitable for GDL specification are the *Specification and Definition Language*(SDL) [56] and *Message Sequence Charts*(MSC) [33].

7.3.2 Metrics

In section 4.3.6 on page 43 discussed the layout and distribution of elements in a diagram. Presently, GDL does not address these issues.

The factors that influence comprehension of diagrams can be controlled. There has been a lot of research on this in the field of computational geometry (see Battista *et al.* [2]). The factors are the *orthogonality* of nodes and edges, the *number of line crossings*, minimising the number of *line bends* and the direction of flow in a diagram. Purchase [51] addresses these issues and argues that many previous approaches to diagram aesthetics tend to have an extreme approach to diagram aesthetics, for example having no line crossings or maximising the symmetry of the diagram. This also tend to be done informally. Purchase's own research in [51] is an attempt to provide quantifiable metrics for seven different graph drawing aesthetics.

As we did not address these issues in our work, one could look at the possibility for implementing a functionality to GDL that put constraints on the of graphical symbols. These parameters could then be adjusted to suit the language that is to be specified.

7.4 Concluding remarks

This thesis has provided a meta–language that is capable of specifying the graphical syntax of UML. We have illustrated its use by specifying a subset of UML which we called T_{uml}. T_{uml} consists of three diagram types, class diagrams, sequence diagrams and state machines.

We have built upon previous research in the field of visual languages and made a meta–language that is capable of addressing the various notational forms of UML diagrams. The way GDL is constructed, with the Z notation as its basis, should make it easy to extend to apply to other languages.

Bibliography

[1] Robert H. Anderson. Two–dimensional mathematical notation. In King Sun Fu, editor, *Syntactic Pattern Recognition, Applications*, volume 14 of *Communications and Cybernetics*, pages 147–177. Springer Verlag, 1977.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[3] Andreas D. Blaser and Max. J. Egenhofer. A visual tool for querying geographic databases. In *Proceedings of the working conference on Advanced visual interfaces*, pages 211–216. ACM Press, 2000.

[4] Christine Bonhomme, Claude Trépied, Marie-Aude Aufaure, and Robert Laurini. A Visual Language for Querying Spatio–Temporal Databases. In *Proc. seventh ACM intl. symposium on Advances in geographic information systems*, pages 34–39. ACM Press, 1999.

[5] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison–Wesley, 1999.

[6] Anabela Caetano, Neri Goulart, Manuel J. Fonseca, and Joaquim A. Jorge. Sketching User Interfaces with Visual Patterns. In *Proc. of the 1st Ibero–American Symposium in Computer Graphics, SIACG 2002*. Eurographics Portugese Chapter, 2002.

[7] Shi-Kuo Chang. Picture processing grammar and its applications. *Information Sciences*, 3:121–148, 1971.

[8] Shi-Kuo Chang. Visual Languages: A Tutorial and Survey. *IEEE Software*, 4:29–39, 1987.

[9] Shi-Kuo Chang, Michael J. Tauber, Bing Yu, and Jing-Sheng Yu. A visual language compiler. *IEEE Transactions on Software Engineering*, 15(5):506–525, 1989.

[10] Sitt Sen Chok and Kim Marriott. Automatic Construction of User Interfaces from Constraint Multiset Grammars. In *Proc. 11th Intl. IEEE Symposium on Visual Languages*, pages 242–249, 1995.

[11] Sitt Sen Chok, Kim Marriott, and Tom Paton. Constraint–based Diagram Beautification. In *IEEE Symposium on Visual Languages*, pages 58–61, 1999.

[12] Noam Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2:137–167, 1959.

[13] Eliseo Clementini and Paolino Di Felice. A comparison of methods for representing topological relationships. *Information Sciences*, 3:149–178, 1995.

[14] Eliseo Clementini and Paolino Di Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, 90:121–136, 1996.

[15] Eliseo Clementini and Paolino Di Felice. Spatial operators. *ACM SIGMOD Record*, 29(3):31–38, 2000.

[16] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end–user interaction. In *Proc. 3rd Intl. Symposium on Large Spatial Databases*, LNCS 692, pages 277–295. Springer Verlag, 1993.

[17] Gennaro Costagliola, Vincenzo Deufemia, Filomena Ferruci, and Carmine Gravino. Using Extended Positional Grammars to Develop Visual Modeling Languages. In *Proc. of the 14th Intl. conference on Software engineering and knowledge engineering*, pages 201–208. ACM Press, 2002.

[18] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Genny Tortora. Positional Grammars: A Formalism for LR–Like Parsing of Visual Languages. In Kim Marriott and Bernd Meyer, editors, *Visual Language Theory*, pages 171–191. Springer Verlag, 1998.

[19] Gennaro Costagliola, Sergio Orefice, Giuseppe Polese, Genny Tortora, and Maurizio Tucci. Automatic Parser Generation for Pictorial Languages. In *Proc. IEEE Symposium on Visual Languages*, pages 306–313, 1993.

[20] Max J. Egenhofer. Extending SQL for Cartographic Display. *Cartography and Geographic Information Systems*, 18(4):230–245, 1991.

[21] Max J. Egenhofer. Spatial SQL: A Query and Presentation Language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86–95, 1994.

[22] Max J. Egenhofer. Spatial-Query-by-Sketch. In *Proc. IEEE Symposium on Visual Languages*, pages 60–67, 1996.

[23] Franck Favetta and Marie-Aude Aufaure-Portier. About Ambiguities in Visual GIS Languages: a Taxonomy and Solutions. In *Fourth Intl. Conference on Visual Information Systems (VISUAL'2000)*, Lecture Notes in Computer Science, pages 154–165. Springer Verlag, 2000.

[24] Brian Fennelly. A descriptive language for the analysis of electronic music. In Benjamin Boretz and Edward T. Cone, editors, *Perspectives on Notation and Performance*, pages 117–133. W. W. Norton & Company, 1976.

[25] Martin Fowler and Kendall Scott. *UML Distilled: a brief guide to the standard object modeling language*. Addison–Wesley, second edition, 2000.

[26] Eric J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

[27] Eric J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, 2(4):371–393, 1991.

[28] Eric J. Golin and Steven P. Reiss. The Specification of Visual Language Syntax. In *IEEE Symposium on Visual Languages*, pages 105–110. IEEE Press, 1989.

[29] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[30] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[31] Ian Hayes, editor. *Specification Case Studies*. Series in Computer Science. Prentice Hall, 1987.

[32] Richard Helm, Kim Marriott, and Martin Odersky. Building visual language parsers. In *Proc. ACM Conf. Human Factors in Computing*, pages 105–112, 1991.

[33] ITU-T. Message Sequence Chart (MSC), ITU–T Z.120. Technical report, International Telecommunications Union, Geneva, August 2001.

[34] Kurt Jensen. A Brief Introduction to Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of the TACAS'97 Workshop*, volume 1217 of *LNCS*. Springer Verlag, 1997.

[35] Joaquim A. Jorge. *Parsing Adjacency Grammars for Calligraphic Interfaces*. PhD thesis, Rensselaer Polytechnic Institute, Dec. 1994.

[36] Donald E. Knuth. Semantics of context–free languages. *Mathematical Systems Theory*, 2:127–145, 1969.

[37] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison–Wesley, third edition, 1997.

[38] Kenneth C. Louden. *Compiler Construction: principles and practice*. PWS Publishing Company, 1997.

[39] Kim Marriott. Constraint Multiset Grammars. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 118–125, 1994.

[40] Kim Marriott, Bernd Meyer, and Kent B. Wittenburg. A survey of visual language specification and recognition. In Kim Marriott and Bernd Meyer, editors, *Visual Language Theory*, pages 5–85. Springer, New York, 1998.

[41] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis & Design*. Marko Publishing, first edition, 2000.

[42] Bernd Meyer. Pictures Depicting Pictures. On the Specification of Visual Languages by Visual Grammars. In *Proc. IEEE Workshop on Visual Languages*, pages 41–47. IEEE Computer Society Press, 1992.

[43] Bernd Meyer. Picture Deduction in Spatial Information Systems. In *IEEE Symposium on Visual Languages*, pages 23–30. IEEE Computer Society Press, 1994.

[44] Stephen Morris and George Spanoudakis. UML: An Evaluation of the Visual Syntax of the Language. In Ralph H. Sprague, Jr., editor, *Proc. 34th Annual Hawaii International Conference on System Sciences*, pages 1223–1232. Computer Society, 2001.

[45] Donald A. Norman. *The Design of Everyday Things*. Basic Books, 2002 edition, 2002.

[46] OMG. *UML 2.0 Diagram Interchange, version 2.0*, January 2003. OMG document ptc/03-07-03.

[47] OMG. *UML 2.0 OCL Specification*, October 2003. OMG document ptc/03-10-14 Adopted Specification.

[48] OMG. *UML 2.0 Superstructure Specification*, 2003. OMG document ptc/03-08-02, final adopted specification.

[49] Open GIS Consortium, Inc. *OpenGIS Simple Features Specification for SQL, revision 1.1*, May 1999.

[50] Maria Pinto-Albuquerque, Manuel J. Fonseca, and Joaquim A. Jorge. Visual Languages for Sketching Documents. In *IEEE International Symposium on Visual Languages*, pages 225–232, 2000.

[51] Helen C. Purchase. Metrics for Graph Drawing Aestethics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

[52] Helen C. Purchase, Linda Colpoys, Matthew McGill, and David Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*. ACS, 2001.

[53] Helen C. Purchase, Linda Colpoys, Matthew McGill, and David Carrington. UML collaboration diagram syntax: an empirical study of comprehension. In *Proc. of the 1st Intl. Workshop on Visualizing Software for Understanding and Analysis, 2002, (VISSOT'02)*, pages 13–22. IEEE Press, 2002.

[54] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, and Carol Britton. UML class diagram syntax: An empirical study of comprehension. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*. ACS, 2001.

[55] Gardner Read. *Music Notation: A manual of modern practice*. Taplinger, second edition, 1979.

[56] SDL Forum. SDL–2000. Technical report, SDL Forum Society, 1999. SDL Forum Version of ITU–T Z.100(11/99) Specification and Definition Language (SDL).

[57] A. Rashid B. M. Shariff, Max J. Egenhofer, and David M. Mark. Natural–language spatial relations between linear and areal objects: The topology and metric of english–language terms. *Intl. Journal of Geographical Information Science*, 12(3):215–246, 1998.

[58] Alan C. Shaw. A Formal Picture Description Scheme as a Basis for Picture Processing Systems. *Infromation and Control*, 14:9–52, 1969.

[59] J. Mike Spivey. *Understanding Z: A Specification Language and its formal semantics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[60] J. Mike Spivey. *The Z Notation: A Reference Manual*, second edition, 1998.

[61] Kurt Stone. Problems and methods of notation. In Benjamin Boretz and Edward T. Cone, editors, *Perspectives on Notation and Performance*, pages 9–31. W. W. Norton & Company, 1976.

[62] Kurt Stone. *Music Notation in the Twentieth Century*. W. W. Norton & Company, first edition, 1980.

[63] Apostolos Syropoulus. Mathematics of Multisets. In Cristian Calude, editor, *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *Springer Lecture Notes in Computer Science*, page 347, January 2001.

[64] Richard L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proc. 2nd Intl. Conference on Software engineering*, pages 331–336. IEEE Computer Society Press, 1976.

[65] Ouri Wolfson, Sam Chamberlain, Kostas Kalpakis, and Yelena Yesha. Modeling Moving Objects for Location Based Services. In *Developing an Infrastructure for Mobile and Wireless Systems. NSF Workshop IMWS 2001*, volume 2538 of *LNCS*, pages 46–58. Springer-Verlag, 2002.

[66] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *Statistical and Scientific Database Management*, pages 111–122, 1998.

[67] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.