GenGEd:

Visual Definition of Visual Languages based on Algebraic Graph Transformation

vorgelegt von
Diplom-Informatikerin
Roswitha Bardohl

Vom Fachbereich 13 – Informatik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurswissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr. K. Obermayer
Berichter:	Prof. Dr. H. Ehrig
Berichter:	Prof. Dr. G. Engels

Tag der wissenschaftlichen Aussprache: 17. Dezember 1999

Berlin 1999
D 83

to feedback bernie

Abstract

The overall aim of this thesis is the support of software development techniques that involve several kinds of visual languages. These visual languages are used for software engineering and programming. Visual programming languages are usually tightly integrated with visual environments. Whenever the visual means of expressions are changed, time- and cost-intensive re-implementations are necessary which can be avoided if generators for visual environments are available. This thesis is one step towards this goal.

The visual environment GENGED supports the visual definition of visual languages. The resulting visual language specification is the parameter of a visual language editor for the manipulation of visual sentences.

Visual languages definable by using GENGED are not restricted to visual programming languages. The goal of this environment is to support numerous kinds of visual languages. These can be diagrammatic, iconic or form-based languages used for software modeling or specification as well as visual languages for teaching children. Although we focus on the definition of visual syntax rather than semantics of visual languages, we distinguish abstract and concrete syntax. This distinction is a first step towards an extension concerning semantic definitions based on abstract syntax.

The visual language specification that can be defined by using GENGED consists of an alphabet and a grammar. The alphabet is represented by an algebraic graph signature, the grammar is represented by an algebraic graph grammar. Moreover, the productions of the grammar are used as edit commands of the visual language editor supporting syntax-directed manipulation of sentences. So far, we show that algebraic graph grammars and graph transformation are suitable for the modeling of visual languages.

Zusammenfassung

Das Gesamtziel dieser Arbeit ist die Unterstützung von Software-Entwicklungstechniken, die verschiedene Arten von visuellen Sprachen integrieren. Diese visuellen Sprachen werden eingesetzt für Software-Engineering und Programmierung. Visuelle Programmiersprachen sind normalerweise fest in visuellen Programmierumgebungen integriert. Eine zeit- und auch kostenintensive Reimplementierung ist notwendig, wenn sich die visuellen Mittel ändern. Dies kann durch Bereitstellung von Generatoren für visuelle Umgebungen verhindert werden.

Die visuelle Umgebung GENGED unterstützt die visuelle Definition von visuellen Sprachen. Die resultierende Sprachspezifikation ist Parameter eines Editors zur Manipulation von visuellen Sätzen über der Sprachspezifikation.

Visuelle Sprachen die mit Hilfe von GENGED definiert werden können, umfassen nicht nur visuelle Programmiersprachen. Der Schwerpunkt der GENGED-Umgebung ist, verschiedene Arten von visuellen Sprachen zuzulassen. Dazu gehören Diagrammsprachen, Ikon-Sprachen, aber auch tabellenorientierte Sprachen, die zur Software-Modellierung oder -Spezifikation benutzt werden, aber auch visuelle Sprachen zur Lehre von Kindern. Obwohl wir uns auf die Definition der visuellen Syntax von visuellen Sprachen konzentrieren anstatt auf deren Semantik, unterscheiden wir bereits zwischen abstrakter und konkreter Syntax. Diese Unterscheidung ist ein erster Schritt in Richtung Erweiterbarkeit. Die abstrakte Syntax ist die Basis zur Definition der Semantik.

Die visuelle Sprachspezifikation, die mit Hilfe von GENGED definiert werden kann, besteht aus einem Alphabet und einer Grammatik. Das Alphabet ist durch eine algebraische Graphsignatur repräsentiert, die Grammatik durch eine algebraische Graphgrammatik. Die Produktionen der Grammatik definieren die Editierkommandos des Editors für visuelle Sätze. Mit Hilfe dieser Kommandos können syntaxgesteuert Sätze der Sprache manipuliert werden. Insofern zeigt diese Arbeit, daß algebraische Graphgrammatiken und Graphtransformation zur Modellierung von visuellen Sprachen geeignet sind.

Foreword

In the early years of my work at the Technical University of Berlin I developed the language $\mathcal{GVT}$ for the visualization of terms over algebraic specifications. At that time I recognized already how important and exciting the visualization problem is, especially in the field of software modeling and specification. Then, I became a computer science assistant at the department *Theoretical Computer Science/Formal Specification*, short *TFS*, and I got the opportunity to work on this thesis.

My deepest gratitude goes to my supervisor Prof. Dr. H. Ehrig for his continuous attendance and his constructive support. He gave me a lot of valuable comments to improve draft versions of this work. Furthermore, I'm much obliged to my co-supervisor Prof. Dr. G. Engels and my colleague Gabi Taentzer for exciting discussions and valuable hints.

Special thanks to my colleagues, especially to Mrs. H. Barnewitz and Mrs. K. Schlicht for their help in any situation.

I wish to express my deepest thanks for numerous discussions, competent suggestions and proof reading of this thesis to Andy Schürr, Bernd Meyer, Bernie Bardohl, Claudia Ermel, Magnus Niemann, Manuel Koch, Manuel Schwarze, Mark Minas, Martin Große-Rhode, Nicole Bardohl, Julia Padberg, Leila Ribeiro, Olga Runge, Stefan Gruner, Thorsten Schultzke, and Vanderlei Moraes Rodrigues.

Finally, I thank my family and my friends for their patience and understanding. For a long time I had known one thing only, my work, but they all stayed at my side. Umm Kulthumm sings, "The sleep doesn't extend the life And the longest night doesn't shorten it, either. I hope she is right.

Contents

Chapter 1

Introduction

Originally, the term *visual language* referred to sign languages for deaf people. Nowadays, this term describes visual means of expression, i.e. languages using drawings or graphics. Outside the computer science community, the use of visual means for the development of architectural designs, circuit designs, etc. has a long ongoing tradition. Text-oriented languages for modeling chemical processes or programming electronic components are currently being replaced by visual languages, too. Inside the computer science community, visual languages are used for rather different purposes, like knowledge modeling, database query and design, and the object-oriented (structured) analysis and design of software systems. Languages for visual modeling are often accompanied by visual programming languages, visual database query languages and so forth.

While visual languages are used for many different tasks, how to define the term *visual language* is a problem still to be resolved. Usually, visual – in contrast to textual – languages offer multidimensional representation of language elements. However, it is difficult to find one common principle that distinguishes visual from non-visual (programming) languages. Some so-called visual programming languages, such as GRAIN [CRM78] (supporting storage of pictures in and retrieval of pictures from a relational database system), manipulate visual objects but have a textual representation. Others, such as Entity Relationship diagrams, have a visual representation but are considered modeling languages rather than programming languages. Yet others, like PIGS [PN83] (imperative programming language), are definitely real programming languages but offer graphical representation as an add-on feature to a traditional textual programming language (Pascal).

In the literature, several definitions and visual-language classifications can be found for visual programming languages but not for visual languages in general. In [Shu90], for example, three categories of visual programming languages are proposed. These are visual languages for (1) processing of visual (or image) information, (2) supporting visual interaction, and (3) programming with visual expressions. Regarding this attempt to classify and characterize visual (programming) languages as somewhat problematic, we consider a variant of Myers's definition of a visual programming language [Mye90]. This definition states that a visual programming language is a programming language with at least some constructs that have either an inherently or a superimposed multidimensional representation. Also, Myers's classification is not expedient for our purpose, which is to characterize visual languages in order to define them in visual terms.

The differences in classifications led Schiffer [Sch98] to define a visual language as a formal language with a visual syntax. He states that the visual syntax is described by an alphabet and a kind of formalism to enable visual sentences to be built over the alphabet. Schiffer's definition is very restrictive because not all visual (modeling) languages are formal languages in the sense that they offer formal semantics. Although some visual languages are not formal languages, they are widely accepted.

The classification system of Burnett and Baker [BB94] reflects the research being conducted in the visual-language community. Even if this focuses on visual programming languages and not on visual languages in general, it is given consideration in this thesis.

Classification system for visual programming languages Visual programming languages and visual programming environments are classified by the type of visual representation used and the underlying programming paradigms [BB94]. Other aspects taken into account are the available language features, language purpose and theoretical background. The following (two-dimensional) visual representation types are usually distinguished:

diagrammatic representations with two-dimensional shapes as object visualizations, and polylines as (binary) relationship visualizations

iconic representations, where spatial relationships between icons like "above" or "adjacent" play the role of explicit connections between objects of diagrammatic representations

form-based representations such as used in spreadsheet applications and spreadsheet-based general-purpose programming languages

Visual representations are not normally only two-dimensional. Three dimensions are possible if x, y and z coordinates are considered. More dimensions are conceivable, e.g. for programming-by-example languages. The aspect of time also can be regarded as a dimension. Although we concentrate on two-dimensional representations, we often use the term *multidimensions*.

Concerning the underlying *programming paradigms*, the classification system in [BB94] lists categories like concurrent language, constraint-based language, data-flow language, form-based language and rule-based language. Note that visual graph transformation languages such as AGG [TER99], PAGG [Göt87] or PROGRES [SWZ99] belong to the category of rule-based visual programming languages or, more precisely, to the subcategory of diagrammatic rule-based visual programming languages. Mostly, they are hybrid languages because they offer visual means and text. We will show that graph transformation languages offer a suitable formalism for defining visual languages. Such visual languages are related to all the representation types distinguished in [BB94].

It depends on the kind of visual language and its purpose whether it is formal or semiformal. In any case, visual languages relate to a specific visual representation type, but they can be interpreted in a specific way as well. This means that all visual languages have a specific meaning, which is mostly formal in the case of (visual) programming languages.

Abstract and concrete syntax of visual programming languages Very often a visual programming language is defined by a textual one, as in the case of PIGS [PN83] and PictorialJanus [KS90]. They merely offer a graphical representation for a traditional textual programming language like Pascal in PIGS, and Janus [Sar93] [1] in PictorialJanus. A common feature of all visual program-

[1] Janus is a textual and constraint-oriented language.

ming languages is that they distinguish between visual representations, called *layout* or *concrete syntax*, and meaningful parts, called *logic* or *abstract syntax*. The abstract syntax of PictorialJanus programs, for example, corresponds to that of textual Janus programs. The concrete syntax is defined in terms of pictorial objects with spatial relationships between them. Note that the abstract syntax of programs is the basis for their semantics.

We concentrate on the visual syntax of visual languages. Such visual languages may be programming languages, but also modeling languages. Modeling languages do not always offer formal semantics [2]. However, the formalism we propose allows us to express some (static) semantical aspects. This is the reason why we talk about the visual definition of *visual languages* and not only about their syntax. Besides, we differentiate between concrete and abstract syntax in order to draw a clear distinction between the two syntactical descriptions. This distinction is influenced by the work of Andries, Engels, Rekers and Schürr [AER96, RS96].

Formalisms for visual languages Although visual languages are used extensively in several applications areas, there is still no standard visual formalism to define their syntax. Many different formalisms for the specification of visual languages have been investigated for more than two decades now [MM97]. These formalisms can be divided into grammar-like and non-grammar-like formalisms. The grammar-like formalisms cover all kinds of graph grammars [Göt87, MV93, AER96, GP98, BTMS99] and positional grammars [CLOT98]. Non-grammar-like formalisms are given, e.g., by logic-based formalisms [HM90a, Mey92, Haa96] and algebraic formalisms [DÜ96].

As regards (visual) languages, their syntax must obviously be defined by a grammar. All currently used grammars for syntax specification of visual languages, such as picture-layout grammars [Gol91a, Gol91b], or constraint multiset grammars [Mar94], use textual rather than visual concepts. Hence, multidimensional representations have to be coded into one-dimensional strings. Using graph grammars and graph transformation instead, we have a much more natural and even visual formalism for this purpose. Note that the visual definition of a visual language is an action resulting in a specification of the specific visual language. As we concentrate on a formalism behind the visual definition,

[2]Compare, e.g., the discussions on the semantics of the Unified Modeling Language [Rat99b].

namely that of algebraic graph grammars and transformation, we also talk about *visual language specifications*. Using alphabets, as proposed by Schiffer in his definition of a visual language, and graph grammars as the formalism for the specification of visual languages, we give the following definition:

> *"A visual language specification consists of an alphabet and a graph grammar. The grammar contains a start sentence and a set of productions. Then, a visual language is given by the set of all visual sentences which can be derived by applying grammar productions to the start sentence."*

Algebraic graph grammars and transformation In general, graph grammars provide an intuitive description for the manipulation of graphs. Graph grammars give a formal specification which can be executed, provided that corresponding tools are available. They are called *algebraic* because the manipulated graphs are special algebras and the transformation of graphs is based on algebraic constructions. More precisely, graph transformations are defined via gluing constructions of graphs which are pushouts in the sense of category theory. Thus, graph transformation combines static grammar descriptions with a dynamic component. Static and dynamic information are based on one joint formal calculus, namely, that of category theory.

Several graph-transformation formalisms can be found in the literature, compare for example [Roz97]. We use algebraic graph structure grammars and transformation as proposed in [LKW93]. Visual sentences are represented by graph structures or, more precisely, by attributed graph structures. We use the single-pushout approach to graph transformation [EHK$^+$97] for specifying the abstract syntax of visual languages as well as for the transformation of visual sentences.

General algebraic graph structures subsume several graph classes, such as hypergraphs used in the DiaGen approach [Min98], and typed graphs occurring in Progres [SWZ99]. In the DiaGen approach, visual languages are specified textually by hypergraph grammars. In Progres, the abstract syntax of visual languages is specified visually, however, using graphs as visual means of expressions. Pictorial objects and spatial relationships that affect the concrete syntax of a specific visual language are associated afterwards.

So far, we have discussed several aspects of visual (programming) languages and their specification. In the following sections, we focus on corresponding visual (programming) environments.

1.1 Visual Environments

Visual programming languages are usually embedded in and tightly integrated with visual programming environments [BGL95]. There are a wide variety of visual programming languages and environments for different application purposes. The reader is referred to [Sch98] for a survey of current visual programming languages and environments.

Popular commercial examples of visual programming environments for textual programming languages are, e.g., *Delphi* (for Borland C), *Visual Basic* (for Microsoft Basic) and *VisualWorks* (for ParcPlace Smalltalk). These languages are user-interface-programming extensions of textual programming languages. The corresponding environments offer visual means of expression mainly for constructing user interfaces. The program code has to be defined textually.

Visual programming environments for visual programming languages are, e.g., *LabVIEW* (National Instruments), *Prograph* (Pictorius Inc.) and *VisualAge* (IBM). These visual languages are based on textual programming languages. They are used for different application purposes. Furthermore, the corresponding visual programming environments offer more visual means than text for programming purposes.

Visual programming environments such as those mentioned above are helpful for rapid prototyping. Such environments provide tightly implemented visual means. Whenever these visual means change, a time- and cost-intensive re-implementation is necessary. The desire to solve this problem was one of the main motivations for this thesis.

Usually, a visual programming environment offers an integrated set of tools allowing compilation or interpretation, debugging and (sometimes) animation and verification of visual programs. The kind of tools available in an environment depends on the specific application purpose. If we take an environment for the visual language of Petri-nets [Pet62, BRR87], or variants of them like high-level or abstract Petri-nets [PER95, Pad96], an animation component can be expected to be available allowing us to watch the firing of transitions. Also, the verification of Petri-nets is desirable. The situation is slightly different as regards the visual language of class diagrams [Rat99b] where modeling is the main purpose.

The application domain of the visual (programming) environment introduced in this thesis is the visual definition of visual languages. These may be programming, modeling or even specification languages. We therefore present a visual programming environment that matches the following definition:

> *A visual programming environment is an integrated set of tools supporting editing, analysis and execution of visual programs.*

In a visual programming environment supporting the visual definition of visual languages, the programs are given by visual language specifications, called *VL specifications*. A VL specification, comprising an alphabet and a grammar, can be edited and analyzed with respect to the underlying formalism. The execution perspective then relates the interpretation of VL specifications in the sense of editing visual sentences. In order to avoid confusion, we consider two visual environments: a visual environment called *VL specification tool*, and a visual environment called *VL editor*. The VL specification tool supports the visual definition of visual languages; it *generates* VL specifications. The VL editor allows visual sentences to be edited with respect to a specific VL specification.

Editing modes If we look at visual language editors, we find there is a choice of editing modes, namely, syntax-directed or free-hand editing. In the latter case, a language-specific scanner and parser must be available. With graphs and graph grammars as the underlying formalism for VL specifications, a graph parser has to decide the membership problem. For a given graph and graph grammar, the parser tries to find a derivation sequence from the grammar's start graph to the given graph. This problem is undecidable for general grammars [Sal73]. Hence, only those restricted classes of graph grammars can be used for which this problem is more or less efficiently decidable. The reader is referred to [BTMS99] for a survey of visual language graph parsing approaches. The development or generation of scanners and parsers is not necessary if syntax-directed editing is supported.

We prefer not to restrict the expressiveness of (graph) grammars specifying visual languages. This means that we favor syntax-directed editing. Note that the visual environment *Rational Rose* [Rat99a] implementing the Unified Modeling Language (UML) [Rat99b] also provides the user with syntax-directed editing for the visual parts. Free-hand editing is allowed for textual expressions. Another reason for using syntax-directed editing is that it offers user-guidance, which is helpful for editing visual sentences. Moreover, this editing mode forces the user to obey certain rules that are necessary to define correct visual sentences.

Similar to *Rational Rose*, we offer both editing modes in the VL specification tool as well as in VL editors. Syntax-directed editing is allowed for the visual parts of a visual language, and free-hand editing for the textual parts. These textual parts are defined as textual attributes that belong to a symbol, like a class name of a class symbol such as occurs in class diagrams [Rat99b].

Syntax-directed editing and grammar productions Syntax-directed editing means that the grammar productions can be interpreted differently [BTMS99]. The language-generating productions are extended by language-editing productions. Both kinds of productions define the edit commands of a syntax-directed VL editor. In order to be able to distinguish properly between (1) language modeling with productions and (2) programming with productions, we will use the following terminology:

1. (*grammar-based*) the productions of a grammar are used to derive all sentences of a language from a given start sentence. The language semantics of a grammar is the set of its generated sentences.
2. (*transformation-based*) a rewriting or transformation system takes a given set of facts as input and uses its productions to transform them into a desired state. The semantics of a rewriting system is a binary relation over a given domain of sentential forms.

Note that a transformation system with empty start sentence also defines a grammar from which languages can be generated. However, language-generating productions together with language-editing productions are used as the edit commands of a VL editor. The advantage of this strategy is that each VL editor acquire an operational semantics for the language-specific parts. Language-specific user interactions do not have to be specified separately.

So far, we have outlined the topics to be addressed in this thesis. The aims and results are summarized in the following section.

1.2 Aims and Results

The overall goal of this thesis is to improve the kind of software development that involves visual techniques. Such techniques employ all kinds of visual languages such as occur in software engineering and programming.

The first aim is to develop a conceptual model for the visual definition of visual languages based on algebraic graph transformation. The developed conceptual model distinguishes between the abstract syntax and the concrete syntax of visual languages. Each visual language is specified by an alphabet and a grammar. The abstract syntax of an alphabet is formally defined by an attributed graph structure signature. It is enriched by a graphic signature and a constraint-satisfaction problem connected to the concrete syntax. This yields the visual syntax of an alphabet. Visual sentences such as occur in a grammar and are derived by applying grammar productions, are algebras with respect to the alphabet. Additionally, they satisfy the corresponding constraint-satisfaction problems. The abstract syntax level of a visual language is formally defined, whereas the concrete syntax level is defined semiformally. According to the classification system of Burnett and Baker [BB94], this topic contributes to the field "theory of visual languages".

The second aim is to develop a suitable visual environment that demonstrates the feasibility of the conceptual model. We introduce the GENGED[3] environment comprising a VL specification tool and a VL editor. Both tools are based on the conceptual model, namely, on the formal calculus of algebraic graph transformation with respect to the abstract syntax, and on graphical constraint solving with respect to the concrete syntax. Hence, we show that algebraic graph grammars – a visual language is defined by an algebraic graph grammar – and transformation can be usefully applied to visual languages. Moreover, GENGED provides a rule-based style of programming by offering language-specific productions. These productions define the edit commands of the corresponding editors.

[3] GENGED is used as abbreviation for GENeration of Graphical EDitors.

The editors have an operational semantics based on algebraic graph transformation. According to the classification system of Burnett and Baker [BB94], this topic contributes to the field "environments and tools for visual programming languages".

The third aim is to show that a variety of visual languages can be captured by the conceptual model implemented by the GENGED environment. Given the underlying graph structure grammar and graph structure transformation-based formalism, the most suitable visual languages are graph-like languages. These can be programming languages, but may also be software modeling and specification languages or any other graph-like visual language. Referring to the visual representation types proposed in the classification system of Burnett and Baker [BB94], we focus on diagrammatic visual languages. Nevertheless, we also specify a box-like language representing the iconic representation type in [BB94]. It is further conceivable that form-based languages can be specified using GENGED because of the powerful concepts it offers.

1.3 Organization of the Thesis

In the previous sections, we have pointed out some of the aspects relevant for this thesis. Moreover, we have introduced some classification criteria that can be found in the literature and have outlined the aims and results of this thesis. The other chapters are organized as shown in Figure 1.1 and discussed below.

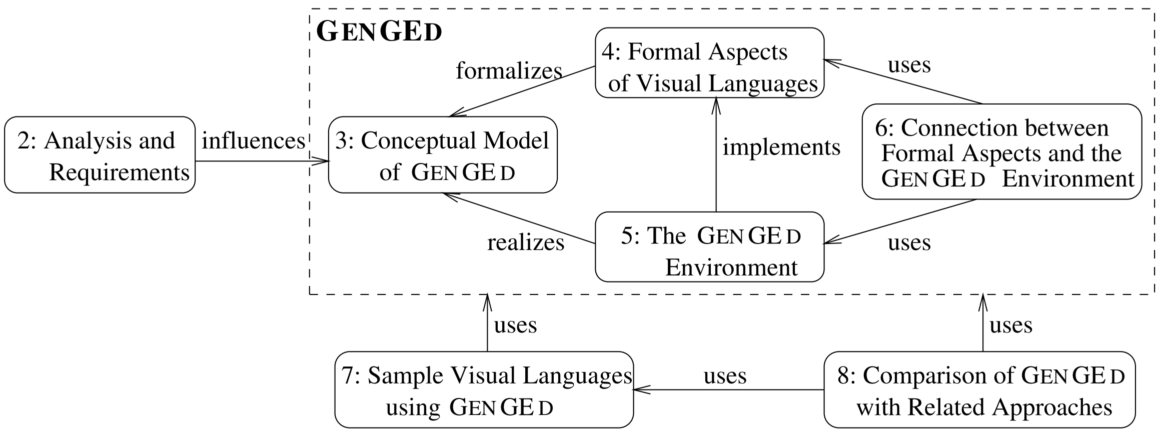


Figure 1.1: Organization of the thesis.

In order to establish detailed requirements, we analyze in Chapter 2 several existing approaches supporting the definition of visual languages. Most of the aspects mentioned at the beginning are discussed in detail. Furthermore, we analyze parts of the class diagram language well-known from UML [Rat99b]. This language is the basis for our case study.

The analysis and requirements treated in Chapter 2 influence the conceptual model of the GENGED approach presented in Chapter 3. This model relates to the visual language specifications as well as the implementation. The modeling of the class diagram language in this chapter forms the illustrational basis for Chapters 4 and 5.

In Chapter 4, the concepts of visual language specifications are formalized. The formalization is based on algebraic graph grammars and transformation, which are combined with constraint-satisfaction problems. This chapter not only deals with the formal aspects of the GENGED approach, the reader also gains a deeper insight into the graph transformation paradigm and its expressive power. This chapter can be read out of context. The concepts presented in Chapter 3 are reviewed briefly.

In Chapter 5, we introduce the GENGED environment that realizes the conceptual model of Chapter 3. The formal aspects of this conceptual model are treated in Chapter 4. Hence, the GENGED environment implements the formal aspects as well. In Chapter 5, we focus on the data structures of a VL specification, namely, the alphabet and the grammar as well as the interfaces to the tools used. The GENGED user interface illustrates the behavior of the environment.

We connect theory to practice in Chapter 6, discussing how the GENGED environment addresses the formal requirements. Note that Chapter 6 builds entirely on Chapters 4 and 5.

In Chapter 7, some visual languages are modeled briefly using the GENGED approach. Although the class diagram language has been looked at in other chapters, we summarize and extend its specification here. In addition to the class diagram language, we specify parts of a statechart language as well as Nassi-Shneiderman diagrams.

At the beginning of this thesis, we mentioned some approaches to the definition of visual languages. Although we do not specifically set out to draw a comparison, in Chapter 8, we compare two related approaches with the GENGED approach. This comparison relates not only to the definition of visual languages but to the expressiveness of the underlying formalism as well.

Finally, in Chapter 9, we draw conclusions and suggest possible future directions of research.

Chapter 2

Visual Definition of Visual Languages – Analysis and Requirements –

Alphabets of textual languages usually comprise some basic objects like words. Grammars or, more precisely, grammar productions describe how text sentences can be derived or generated from these words. The productions are almost defined using the Backus-Naur-Form (BNF). But what are the constituents of an alphabet for a visual language (VL), and which kind of grammar is suitable? In textual languages, the basic objects are usually placed one "beside" the other. Of course, in VLs the placement of basic objects offers more expressiveness than in textual languages. It is a question whether the placement of basic objects occurring in VLs can be specified by the alphabet, too. Furthermore, a formalism like BNF as used for textual languages surely is not sufficient for VLs because of the multidimensional character of VLs.

A visual programming environment (VPE) is an integrated set of tools that supports editing, analysis, and execution of visual programs. In a VPE with the purpose to support the visual definition of VLs, the visual program is a VL specification. Thus, the corresponding environment comprises an editor, an analysis tool, and a compiler for these VL specifications. The latter two tools are based on semantics definitions of VLs. The semantics of VLs is not in the focus of this thesis. However, using a suitable grammar formalism for VL specifications it would possible to analyze specified VLs with respect to that formalism. Omitting semantics definition, then the execution process concerns editing of visual sentences. Hence, a VPE for VLs comprises at least a VL specification tool and a VL editor. Such a VL editor can be generated by the environment or, it can be a generic editor where the parameter is a VL specification.

This chapter is organized as follows. In Section 2.1 we first list some formalisms for VLs that can be found in literature. This forms the basis for the discussions of this chapter. In Section 2.2 we analyze some visual sentences, each one belongs to a specific visual language. Thereafter, in Section 2.3, we discuss a grammar-based formalism for the specification of VLs. Some related approaches with tool support are analyzed in Section 2.4. Note that none of existing approaches come up with comprehensive tools supporting the visual definition of VLs. In Section 2.5 we discuss the topics concerning tool support. In Section 2.6 we introduce a further VL, namely, a simplified version of UML class diagrams. This case study together with the discussions before form the requirements of this thesis.

2.1 Formalisms of Visual Language Specifications

Many different formalisms for the specification of VLs have been investigated for more than two decades now [MM98b]. Grammar-like formalisms range from early approaches like web and array grammars [Ros76], and shape grammars [Gip75] to recent formalisms like positional grammars [COP+93], relation grammars [FTTV96], relational grammars [Wit92, WW96], attributed multiset grammars [GR90], and several types of graph grammars [Cou90, Roz97]. There are also a variety of non-grammar-like formalisms including algebraic approaches [DÜ96] and logic-based approaches [HM90a, Mey92, Haa96]. No matter which type of framework is selected, there are still a lot of important choices to be made [MM97]:

- **What are the basic objects?** Some formalisms use only lines or points as basic objects, while others allow almost arbitrary complex symbols, which means that the borders between *abstract syntax* and *concrete syntax* are shifting.

- **How are spatial relations handled?** In many formalisms they are given as uninterpreted, abstract relations, whereas other methods use implicit relations that are expressed as conditions on attribute values.

- **Are spatial relations interpreted?** An interpretation of a spatial relation can either be symbolic or arithmetic (in terms of the underlying geometry). Interpreting spatial relations allows us to express general geometric axioms (e.g., the transitivity of spatial inclusion) and thus enriches the expressiveness of the framework.

- **What is the general mathematical (data) structure used to represent a visual sentence?** In many approaches graphical objects are aggregated as arbitrary sets or multisets, whereas other methods use some tiling of the space (normally a grid-like structure) on which objects have to be arranged.

These choices represent only some of the basic design decisions to be made. Yet it is clear that they deeply influence the expressive power and the computational complexity of the resulting specification method. A comparison of different formalisms even within a single framework like grammar-based formalisms is thus very difficult. This is contrarily to textual languages where the basic objects to be manipulated are given as a sequence of textual symbols.

In this thesis, we do not concentrate on a comparison between several approaches but we are looking forward to an approach which allows for a wide spectrum of extensions. Although we do not consider the semantics of VLs we have to have in mind that the approach we are looking forward can be easily extended also for semantics definition. Having this aim in mind, we will distinguish between the *abstract syntax* and the *concrete syntax* of visual languages. The abstract syntax is the basis to define the semantics of a visual language, whereas the concrete syntax concerns the layout. Moreover, the concrete syntax may change during the time whereas the abstract syntax remains unchanged, and vice versa. However, the relations between these kinds of syntactical descriptions have to be taken into account. These observations lead to the first requirement:

Requirement 2.1.1 (*Visual Syntax of Visual Languages*)
The visual syntax of visual languages comprises the abstract syntax and the concrete syntax. The abstract syntax determines the logical meaning and the concrete syntax determines the layout of visual languages. The two syntactical descriptions have to be connected in a suitable way. △

Having the requirement for the visual syntax, the original question remains open: what are the basic objects of a visual language and how are they connected? In the following section we analyze some visual sentences in order to achieve a general model allowing to cover several visual languages under different aspects.

2.2 Examples of Visual Sentences

The classification system of visual programming languages proposed by Burnett and Baker in [BB94] distinguishes visual representation types, namely, diagrammatic, iconic and form-based representations. The aim of this thesis is to find a suitable model for all these representation types. Doubtless, the differences between these types are not always clear. Figure 2.1 illustrates four visual sentences, each one belongs to a specific VL. Each language differs with respect to its expressiveness, i.e. its logic (abstract syntax) and its layout (concrete syntax). The basic objects and relations of the corresponding languages are analyzed in the following.

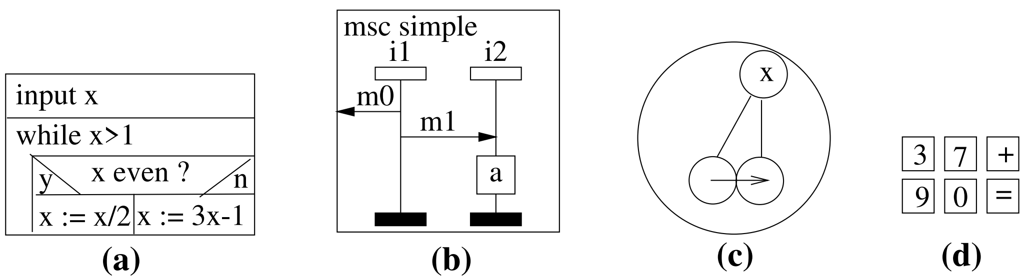


Figure 2.1: Nassi-Shneiderman diagram (a), Message Sequence Chart (b), VEX diagram (c) and an iconic sentence (d).

Nassi-Shneiderman diagrams (NSDs) are used for representing structured programs [NS73]. Each statement occurring in a program is visualized by a box.

From the visualization point of view, the boxes (the statements) are attached in a way to obtain rectangular diagrams. Hence, NSDs are given by a box-like language because of the attaching boxes. Although we use the term Nassi-Shneiderman *diagrams*, with respect to the visual representation type of [BB94], the boxes may be icons that are related by spatial relations like “adjacent”. However, some more spatial relations must be considered, namely, that the composed statement boxes define one rectangle.

The NSD illustrated in Figure 2.1 (a) represents a program consisting of two sequential statements: a single statement (*input x*) and a *while* statement. The while statement is nested; it comprises an *if* statement which covers a *yes* statement and a *no* statement. All statements are coupled and visualized in a way that the program is represented by a rectangle.

The basic objects of NSDs are statements. Each statement is related to another one in a specific way. E.g. the *while* statement requires only one further statement, whereas the *if* statement requires two statements, one for the *yes* case and one for the *no* case. These requirements express a “nested” relation. A single *text* statement requires no further statement. Each statement can be related to another one by a “sequence” relation. The “sequence” relation expresses the relations of statements on one layer.

Message sequence charts (MSCs) allow describing interactions between entities [ITU93]. A MSC describes which messages are interchanged between process instances (entities), and what internal actions they perform. A MSC is visualized by a rectangle. The entities of a MSC are visualized by timelines, messages by arrows, and actions by rectangles. Each of these representations are spatially related to the environment rectangle by an “include” relation. Further spatial relations are “begin” and “end” with respect to the message arrows, and “adhere” with respect to action rectangles adherent to specific timelines.

The visual language of MSCs can be seen as a combination of box-like and graph-like language. The box-like character is given by the rectangles used to visualize the environment and the actions. The graph-like character is given by (message) arrows connecting the (entity) timelines. Using the terms proposed by the classification system of [BB94], MSCs combine diagrammatic and iconic representations.

The MSC illustrated by Figure 2.1 (b) represents two entities called *i1* and *i2*. The message *m1* is sent from entity *i1* to *i2* when the message *m0* is sent from entity *i1* to the environment. The action *a* is executed by entity *i2* after it received the message *m1*. The environment is named.

The basic objects of a MSC are rectangles, timelines and arrows. The rectangles are ambiguously because they indicate either the environment or an action. This causes the interpretation of the "include" relation relating the main objects, i.e. the entity timelines, with the environment rectangle. An action rectangle belongs to a specific entity timeline; it must not be related to the environment rectangle separately. There are three notable kinds of (abstract) relations concerning a message. A message may relate one entity, or two entities, or it may relate one entity and the environment. The corresponding spatial relations are then interpreted in terms of the underlying geometry. A further relation concerns the time. It is important to observe the place where a message occurs in order to capture the time relation. E.g. the message called *m0* in Figure 2.1 (b) befalls sooner as the message called *m1*. All these basic objects and relations are important to notice with respect to the abstract syntax. All spatial relations must be interpreted.

In contrast to NSDs where the statements describe a set of sequential tokens the textual expressions belonging to basic objects occurring in one MSC must be unique. These unique names indicate the objects with respect to both, the abstract syntax as well as the concrete syntax.

Visual λ expressions (VEX) belong to a visual language for λ-calculus expressions [CHZ95]. It comprises pictorial objects like circles, lines, and arrows. Spatial relations are given by "inside" and "touches". In the terms of the classification system of [BB94], this kind of visual language is like the MSC language a mixture of a diagrammatic and iconic representation type although the circles can be named.

In VEX each variable identifier is represented by an empty circle (labeling text may be contained) that is connected by a line to a so-called *root node*. A root node is also an empty circle, with one or more lines touching it, leading to all identifiers representing the same variable. A root node may be internally tangential to another circle. Then it represents a parameter of a λ-abstraction. If it is not included by any other circle, it denotes a free variable. A circle representing a λ-abstraction contains its parameter circle and a VEX (sub) diagram as its body. An application of two expressions is depicted by two externally tangential circles with an arrow at the tangent point. The head of the arrow lies inside the argument circle, and the tail of the arrow lies inside the abstraction to be applied.

Figure 2.1 (c) illustrates a VEX expression for $\lambda x.xx$. It consists of a large circle representing a λ-abstraction. The large circle comprises one root node denoted by *x* which is internally tangential to the λ-abstraction. Its application is depicted by the two externally tangential circles with the arrow at the tangent point.

The VEX language comprises few basic objects and few spatial relations which must be interpreted. In order to capture the abstract syntax of a λ expression the spatial relations between the basic objects are interpreted in terms of the underlying geometry. In contrast to MSCs where the basic objects, except for the rectangle, represents a certain item, all positions of basic objects in a VEX expression and their spatial relations are important to capture the corresponding expression with respect to its abstract syntax.

Arithmetic expressions. Figure 2.1 (d) illustrates one visual sentence describing arithmetic addition between two operands that have the same number of digits. This kind of visual language is proposed in [COL95] as an iconic language. It can also be seen as a kind of form-based visual language with respect to the classification system of [BB94].

The basic objects of arithmetic expressions are rectangles, each one includes a number or an arithmetical sign. The rectangles are spatially related by "horizontal" and "vertical" expressions. These relations must be interpreted. Moreover, it is important to observe the position of each rectangle in order to achieve a result for such expressions. These positions may be expressed as attributes of the basic objects.

Summary

In this subsection we proposed four visual sentences, each one corresponds to a specific kind of visual language. These visual languages differ with respect to their abstract and concrete syntax. A distinction between these two kinds of syntactical descriptions was already required in Requirement 2.1.1. However, we analyzed the basic objects and the spatial relations between them. We mentioned the corresponding logical items according to their well known semantics.

The visual sentences regarded offer different basic objects and relations between the objects. Some of the basic objects are complex pictorial objects like the entity timelines in MSCs, some others are simple rectangles like the statement boxes of NSDs. However, because of the computational complexity it is not useful to use lines and points as basic objects but more comprehensive objects as they are important to build up the abstract syntax. Furthermore, having in mind the visual definition of visual languages it seems to be useful that spatial relations are also reflected with respect to the abstract syntax. Usually, such spatial relations are not directed like the "sequence" relation occurring in NSDs. According to the abstract syntax we need a possibility to express such dependencies. This observation causes the requirement to use graphs with nodes and directed edges as mathematical data structure for visual sentences.

Using graphs as internal data structure for visual sentences, then the nodes represent the basic objects and the edges represent the spatial relations. Consequently, the formalism to describe the syntax of visual languages is given by a graph grammar. This leads to the following requirement:

Requirement 2.2.1 (*Sentence' Representation, Graph Grammar Formalism*)
The abstract syntax of visual sentences is represented by graphs. Nodes represent the basic objects and directed edges represent the spatial relations. The concrete syntax of visual sentences is determined by pictorial objects embodying the basic objects. Graphical constraints between pictorial objects embody the spatial relations.

Based on this representation model, graph grammars are used as formalism for visual language specifications. This formalism is extended by graphical constraint solving for the spatial relations. △

In general, visual languages possess multidimensional representations of language elements. The examples we proposed in this section, however, are concerned only with two dimensions, these are x, y coordinates of pictorial objects. Visual languages may be also three-dimensional; then a further z coordinate is regarded. Furthermore, a time dimension might taken into account investigating programming-by-example languages where the program is a sequence of two- or three-dimensional pictures. Hence, we speak about multidimensional representations of visual languages. These multidimensional representations are reflected by constraint relations between language elements.

In this section we analyzed several visual sentences. Furthermore, we required graphs as internal representation model and graph grammars as specification formalism. But how to describe visual languages by graph grammars? In the next section we discuss several grammar formalism for visual languages.

2.3 Grammar Formalism of Visual Language Specifications

A visual language (VL) can be described by an alphabet and a grammar formalism in order to build up visual sentences from the alphabet. In the previous section we required graph grammars as formalism and graphs as internal representation model for visual sentences. But how are the constituents of a VL specification, namely, an alphabet and a grammar, described?

A grammar for textual languages usually consists of a set of (non-terminal and terminal) symbols, a non-terminal start symbol, and some grammar productions. Grammar productions are used to describe, generate, and recognize languages and their sentences. Commonly, a grammar yields a language comprising all terminal sentences derived by productions from the given start sentence. Thereby, we distinguish between intermediate derivation results in the form of non-terminal sentences and terminal sentences that cannot be rewritten. The operational semantics of a grammar is the set of its generated terminal sentences. The components of a grammar used in grammar-based formalisms for modeling VLs should look similar to grammars generating textual languages. This means that, based on an alphabet of symbols, a grammar should comprise a start symbol (or a complete start sentence) and several productions.

In textual languages, symbols are usually placed one “beside” the other. In VLs, symbols can be arbitrarily placed, however, their placement is mostly defined in terms of spatial relations. These spatial relations may already be part of the alphabet or they may be described by productions. In order to find a suitable representation model for grammar productions let us have a look to Figure 2.2 showing the context-free productions of a grammar generating the Nassi-Shneiderman language; it is taken from [BTMS99]. For illustrational reasons we use a formalism similar to Backus-Naur-Formalism (BNF) for the productions of the NSD grammar. In the productions' left-hand sides (LHSs) one non-terminal symbol occurs which can be transformed to the connected symbols (visual sentences) of the right-hand sides (RHSs). The alternatives are separated by a vertical line.

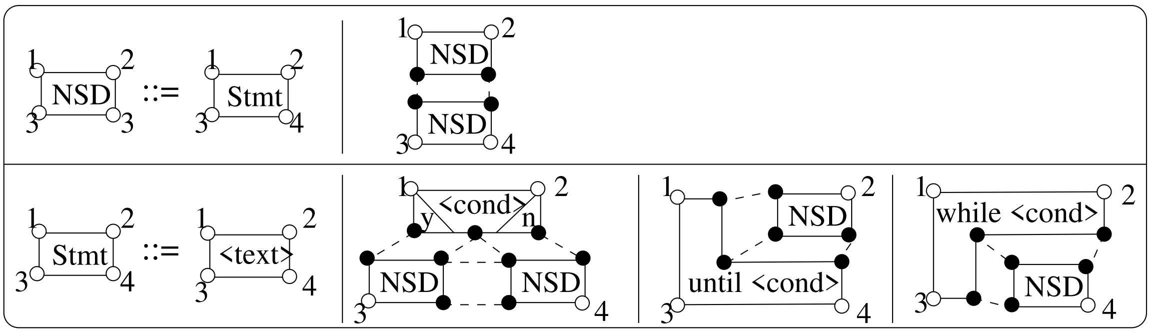


Figure 2.2: Representation model for context-free productions of a NSD grammar.

The upper productions of Figure 2.2 describe the possible derivation from the non-terminal NSD symbol, the lower productions describe the derivation from a non-terminal Stmt symbol. We extend the BNF by black and white (attaching) dots as well as dashed lines to illustrate the spatial relations due to the concrete syntax (the layout). The dashed lines between black dots indicate the gluing of the corresponding box dots. The numbered white dots occurring in the LHS and RHS symbols represent the identification of the corresponding dots. It is a question how such productions can be described visually where in addition two syntactical descriptions must be considered, namely, the abstract syntax and the concrete syntax as well.

Most approaches to VL modeling restrict the grammars to context-free productions. The reason is that they offer efficient parsing facilities in supporting tools. Nevertheless, context-free productions are surely not sufficient to describe VLs. Regarding the MSC language, e.g. we expect a production supporting the insertion of a message between two entities looks like that one shown in Figure 2.3. In its LHS, two entities are available represented by timelines. These timelines occur also in the RHS of the production; they are identified by numbers. Note that this production contains terminal symbols only.

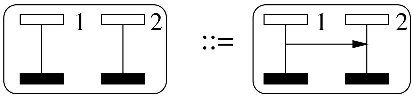


Figure 2.3: Context-sensitive MSC-production expressing the insertion of a message.

The production in Figure 2.3 is context-sensitive because the LHS requires the availability of terminal symbols, namely, the timelines, when the production is applied to a given visual sentence. In general, grammar productions should not be restricted to context-free productions. This observation together with Requirement 2.2.1 leads to a further requirement:

Requirement 2.3.1 (*Grammar Productions*)
The graph grammar productions must not be restricted to be context-free in order to allow the specification for a great variety of visual languages. △

In the next section, three related approaches with additional tool support is treated. Two of these approaches offer graph grammar formalisms to define VLs. A third approach comes up with a visual specification tool.

2.4 Related Approaches with Tool Support

It seems that graphs are a suitable representation model and graph grammars are a suitable formalism for VL definition. This is the reason why we restrict our analysis to grammar-based approaches, mainly to graph grammars. We consider also positional grammars because of the corresponding tool support allowing in some way the visual definition of VLs:

1. DIAGEN (DIAgram GENerator) [Min97, Min98, BTMS99] uses hypergraphs and hypergraph grammars as description model for VLs. DIAGEN is a generator for VL editors allowing syntax-directed (DIAGEN I) as well as free-hand editing (DIAGEN II). A VL is specified using a common text

editor. Nevertheless, DIAGEN offers also the possibility to specify textually animated diagram languages [MG97] as well as to specify user interactions (DIAGEN I) of a VL editor being generated; a suitable interaction model is proposed in [VM94, MV95].

2. VLCC (Visual Language Compiler Compiler) [COL94, COL95, CLOT97] uses positional grammars, a kind of string grammars, for defining VLs. VLCC offers a visual specification tool for defining the alphabet of a VL. The grammar is defined rather textually than visually. VLCC offers also the possibility to define the semantics of some VLs because it concentrates on programming languages. For some others, for example, software modeling languages, it can be checked whether sentences are correctly edited in a generated VL editor or not.
3. PROGRES (PROgramming with Graph REwriting Systems) [Sch91, SWZ99, BTMS99] constitutes a well-established branch of rule-based visual specification or programming languages. The language is designed in order to distinguish between data definition and manipulation activities. Additionally, it does not rely on the rule-based paradigm of programming for all purposes, but supports also imperative programming with control structures. Altogether, PROGRES is a hybrid VL with a mixture of tightly integrated graphical and textual elements. Any graphical representation has a text representation, too. Note that the main goal of PROGRES is to program with graph rules and not to define arbitrary VLs by VL-specific visual means.

The visual definition of VLs is influenced by the kind of underlying (grammar) formalism, and further-on, by the kind of editing mode that should be available in a VL editor. For example, allowing free-hand editing, suitable scanners and parsers must be available for which the word problem is determinable. Scanners and parsers can be omitted if syntax-directed editing is possible. The aspect of conceivable editing modes available in a VL editor is regarded in Section 2.5. In the following we discuss in more detail the related approaches with respect to the specification of VLs.

DIAGEN The grammar of a visual language consists of a set of (non-terminal and terminal) symbols, a start sentence and a set of productions. All these components are described textually. The description of non-terminal and terminal symbols can be seen as the alphabet of the language. Each symbol occurring in the alphabet is described by a hypergraph. More precisely, a hyperedge defines the symbol's type and its layout, the number of tentacles define the attachments and the (hyper)nodes define the connecting entities. Note that the symbol's type is used to describe the abstract syntax level, the attachments and the connecting entities describe the concrete syntax level. The description of symbol layouts is done explicitly by the definition of graphical primitives like rectangles, lines, etc. The attachments are defined using graphical constraints like "equal-Position", etc.

Based on the hypergraph representation of symbols occurring in an alphabet, the visual language grammar is defined by a hypergraph grammar as underlying formalism. Such a hypergraph grammar can be illustrated as in Figure 2.2. For example, the $\boxed{\text{NSD}}$ symbol in the LHS of the upper production is defined by a hyperedge whose type is given by the term "NSD"; its layout is given by the box (the rectangle). This hyperedge has (implicitly) four tentacles, each one is connected to a hypernode defining the connecting entities (black and white dots).

In DiaGen context-free hypergraph grammars as well as context-free hypergraph grammars with embedding productions are supported. An embedding production is defined similar to a context-free production: the RHS of an embedding production is the same as its LHS but with an additional hyperedge representing a symbol. The application of an embedding production thus simply adds a hyperedge to a given context. Application of productions to a hypergraph representing a visual sentence is similar to that used in string grammars: if the production's LHS is a subgraph of the hypergraph, this subgraph is removed and replaced by an isomorphic image of the production's RHS. The resulting hypergraph is said to be derived from the first one. The application of productions is described by transformation rules which are also part of the textual specification of a VL. Transformation rules as well as grammar productions comprise some features for pretty printing a visual sentence by graphical constraints. This is due to the attachments and the connecting entities (the nodes) of a hypergraph.

VLCC The components of a VL grammar are given by a set of (non-terminal and terminal) symbols, a start symbol, and context-free productions. Further constituents are some binary relation identifiers expressing how symbols are syntactically composed, and attribute inference rules belonging to each production. Binary relation identifiers are based on a syntactic model proposed in [CLOT97]. The syntactic models classify VLs due to connection-based and geometric-based languages. The models reflect two modalities to compose visual sentences: graphical objects are either connected or spatially arranged. According to the classification system proposed in [BB94], the connection-based model corresponds to diagrammatic VLs and the geometric-based model corresponds to iconic and form-based VLs. The attribute inference rules define syntactic attribute values of the LHS's non-terminal symbol that depend on the syntactic attribute values of the symbols occurring in the production's RHS.

The application of (positional) grammar productions to a visual sentence is similar to string grammars. But it is driven by so-called pictorial evaluation rules. These rules maintain the substitutions of the syntactic attributes occurring in the production's inference rule. Positional grammars as used by VLCC are an extension of string grammars. Special textual symbols and operators as well as special-purpose attributes are used to indicate spatial relations between the symbols used in the grammar productions. Note that these spatial relations are not mapped to graphical constraints.

PROGRES In [RS96] it is explicitly distinguished between the abstract syntax and the concrete syntax of VLs. The abstract syntax is captured by an *abstract syntax graph* (ASG), the concrete syntax is described by a *spatial relation graph* (SRG). A *correspondence graph* representing a visual sentence establishes the connection between the nodes of the ASG and its layout given by the SRG. In our opinion it is not adequate to define three grammars for one VL, however, a distinction between the two kinds of syntactical descriptions is necessary (compare Requirement 2.1.1).

In [AER96] a concrete PROGRES specification is sketched for MSCs where both syntactical descriptions are coupled by *represent* edges occurring in productions. This means that each MSC production models the abstract syntax as well as with the concrete syntax. The nodes representing abstract items have at least one graphical correspondence. Note that the directed edges indicating relations on the abstract syntax level do not have a graphical counterpart.

A visual (graph) language is specified in PROGRES in two steps [AER96]. First, in a *graph scheme*, the types of nodes, edges and attributes are declared that may occur in a graph of the considered language. A graph scheme is defined textually. As PROGRES is a powerful graph transformation language it offers lots of features to program with rules. One of these features is given by the availability of cardinality constraints. Such cardinality constraints can be used, for example, to express that each message occurring in MSCs has exactly one origin as well as one destination. The graph scheme of a PROGRES specification already defines a graph language. The graph scheme by itself does not fully define a specific (visual) graph language; it is defined by the second part in the PROGRES specification of a graph language, which is the set of productions obeying the type restrictions imposed by the graph scheme.

Summary

In this section we have regarded three approaches for the definition of VLs. In the VLCC approach, syntactic models as proposed in [CLOT97] must be taken into account by a designer of a VL. These models distinguish different visual representation types of VLs, namely connection-based and geometric-based languages. In our opinion, there is no unambiguous classification of VLs according to their visual representation types.

Neither the PROGRES approach nor the DIAGEN approach consider syntactical models in the way of the VLCC approach. Furthermore, in PROGRES the language productions are not restricted to be context-free. Note that in PROGRES there is no distinction between non-terminal and terminal symbols. This is due to the transformation-based approach mentioned in the introduction of this thesis. The transformation system defined using PROGRES describe editing operations of a (syntax-directed) editor.

Common to the three approaches is the fact that they allow for the definition of types. These are symbol types in DIAGEN and VLCC and additional edge types indicating relations in PROGRES. Having in mind alphabets over which visual sentences can be defined it seems to be useful to follow the PROGRES approach. This means that an alphabet of a VL describes the symbol types and the relation types. Instances of these types occur then in the start sentence and in grammar productions as well as in visual sentences that are derived from grammar productions. This leads to the following requirement:

Requirement 2.4.1 (*Visual Language Specification*)
A visual language is specified by an alphabet and a grammar. The alphabet describes the symbol types and the relation types of a visual language. Instances occur in the start sentence and in grammar productions that specify how language-specific visual sentences can be derived. △

So far, we examined some aspects important for VL specifications. In the following section we discuss in more detail tool support for VLs concerning the visual definition of VLs as well as editing modes in a VL editor.

2.5 Tool Support for Visual Languages

The main goal of DIAGEN and VLCC is the generation of VL editors for free-hand editing. This means that efficient parsing facilities are in the fore which restrict the expressiveness of VL grammar productions. It is a question whether we should strive for free-hand editing or whether syntax-directed editing is acceptable. If syntax-directed editing should be supported, more productions must be available than only language-generating productions. Language-generating productions, e.g. are illustrated by Figure 2.2. Note that VL editors occurring in visual environments like *Rational Rose* [Rat99a] (for UML [Rat99b]) are mostly syntax-directed because of missing standard visual symbols and relations between them, and because there is a lack of efficient parsing algorithms. Nevertheless, the editing of textual symbol attributes like a class name is not syntax-directed.

Before we discuss the two kinds of editing modes (syntax-directed and free-hand editing) in a (generated) VL editor, we first consider the possibility for the visual definition of VLs.

2.5.1 Visual Definition of Visual Languages

The idea of defining the syntax of VLs visually is not new. Beside the graph transformation-based approach of PROGRES [SWZ99, BTMS99] where mainly the abstract syntax of a VL is described visually by means of graphs, we have to mention the positional grammar approach implemented by VLCC.

The visual language compiler-compiler VLCC [COL95] comes up with a (partly) visual specification tool for the generation of a visual environment allowing editing and execution of visual sentences. Using VLCC, the language designer can define VLs by specifying the appearance of non-terminal and terminal symbols using a symbol editor. This editor offers two modes: symbol mode and drawing mode. In symbol mode, the designer can transform an image into an alphabet symbol by adding syntactic attributes. A symbol or an image is defined as a terminal or non-terminal symbol.

A VL grammar is defined by a positional grammar, and the semantics of a VL by semantic rules associated to positional grammar productions. In [COL95] the visual definition of all components is proposed: the parameterized grammar editor (PGE) consists of a symbol editor allowing to create non-terminal and terminal symbols. The PGE consists of a production editor, a relation selector and an attribute-inference-rule (AIR) selector. It is said that the selectors let the designer select relations and AIRs from class libraries. These class libraries contain specific predefined relations and AIRs for a particular VL class. The selected elements can be used to compose the productions. However, the semantics of a VL must be defined textually for each production.

In the implementation available, the grammar productions and the semantic rules must be defined textually. The grammar productions are specified under consideration of the positional grammar formalism and the chosen syntactic model of the corresponding visual representation type for the VL which is being specified. The textual specification is based on a YACC-like notation.

Summary

The main goal of this thesis is the visual definition of VLs. Such a definition can be supported by a visual environment we will call *specification tool*. The specification tool should comprise an alphabet editor and a grammar editor similar to those proposed in [COL95]. This leads to the following requirement:

Requirement 2.5.1 (*Visual Definition of Visual Syntax*)
The visual definition of visual languages is supported by a visual language specification tool. △

Visual sentences over a VL specification can be edited in a VL editor. In the following section we discuss conceivable editing modes.

2.5.2 Editing Modes of Visual Language Editors

Visual language editors may offer two different editing modes [BTMS99]: *syntax-directed editing* and *free-hand editing*. Syntax-directed editors provide a set of editing operations. Each of these operations is meant to modify the meaning of the visual sentence in a certain aspect. Editing operations modify the visual sentences' abstract syntax; its concrete syntax and thus its visual appearance is updated accordingly.

Syntax-directed editing is supported by the *grammar-based* and the *transformation-based* approach. The grammar-based approach uses an explicit grammar for defining a visual language and its abstract syntax independent of any editor. Editor operations directly modify a sentences' derivation tree and, thereby, indirectly the generated sentence. A typical example is DIAGEN I where syntax-directed editing is supported. The editing operations allowing the insertion, deletion, moving, and grouping of symbols or sentence parts, are internally transformations on the derivation trees. Their application is controlled by interaction automata. An automaton recognizes an interaction pattern, i.e. a sequence of external or internal events.

The transformation-based approach omits an explicit grammar for specifying a VL. Editing operations are defined by transformations on the abstract syntax which also define the VL at the same time. The VL is thus defined by its editor. Generally, such editors do not keep track of visual sentences' derivation histories. The PROGRES system is a typical example of such a system which represents visual sentences by attributed graphs and editing transformations by productions of programmed attributed graph grammars.

The advantage of the grammar-based approach is the use of the grammar as a concise syntax definition. Basic editing operations (e.g. insertion and deletion of simple objects) can be derived from this grammar. More complex editing operations may be defined by additional transformations that can be checked via this grammar in order not to change the language. The advantage of the transformation-based approach is the possibility of modeling easily intermediate, temporally incorrect visual sentences in terms of the syntax of the VL which is not directly represented. Such intermediate, syntactically incorrect visual sentences, which the user soon completes to syntactically correct ones, are often required for convenient editing. The transformation-based approach, moreover, even comprises the grammar-based one if we disregard the grammar-based approach's concise syntax definition by a grammar since the grammar-based approach also needs additional transformations to define complex editing operations.

Syntax-directed editors can also be distinguished by the way how the user interacts with the editor. The select&apply paradigm as used with most of these editors, requires the user to select an editing operation and some pictorial objects. As result, the editing operation is applied to the visual sentence by using the selected objects as parameters. There is no need to explicitly specify this interaction paradigm since it is inherent in the graph transformation approach. However, it forces the user to be completely aware of the underlying graph transformation system. If those details should be hidden from the user, interactions have to be more elaborated. The editor specification has to describe interaction patterns to be translated to graph transformations. This approach is realized in DIAGEN I.

Visual language editors providing free-hand editing are graphical editors directly manipulating the visual sentence. The editor modifies a visual sentence's concrete syntax. The editor becomes a VL editor by manipulating only pictorial objects used by the VL and by offering a parser. A parser is necessary for checking the correctness of visual sentences, for recreating the concrete syntax' derivation history, and for updating the abstract syntax accordingly.

The advantage of free-hand editing compared to syntax-directed editing is to define a VL by a concise graph grammar only; editing operations can be omitted. The editor does not force the user to edit visual sentences in a certain way since there is no restriction to predefined editing operations. This may turn out to be a disadvantage since editors permit to create any visual sentence; editors do not offer explicit guidance to the user. Furthermore, free-hand editing requires a scanner and a parser and is thus restricted to kinds of VLs and graph grammars that offer efficient parsers.

Summary

The approaches analyzed in Section 2.4 offer syntax-directed editing (except for VLCC) and free-hand editing. In the DIAGEN and VLCC approach, the syntactic structure of a visual sentence is - similar to string grammars - represented by a syntax tree. The non-terminal symbols together with their connecting entities are inner nodes. Terminal symbols together with their connecting entities are leaves. In the DIAGEN approach the hypergraph representation of a visual sentence is additionally equipped with a (graphical) constraint tree which conforms to the derivation structure of the visual sentence. Because of the transformation-based approach of PROGRES there is no derivation tree for the visual sentence but a graph representing the abstract syntax and the concrete syntax of a visual sentence.

In VL editors generated by DIAGEN and PROGRES, the layout adjustment of a sentence after application of productions is done by constraint solving. Syntax-directed editing changes the internal data structure in a syntactically correct fashion and specifies exactly how to update the physical representation via the constraint system. Direct manipulation of graphical objects is usually allowed within the limits imposed by the constraints and is guaranteed not to change the interpretation of the visual sentence. In VL editors generated by VLCC an automatic layout adjustment is not realized. The user has to take care to define correct sentences according to the visual syntax description.

The main effort of this thesis is the visual definition of VLs and not the generation of VL editors or visual environments. Hence, we favor the syntax-directed editing mode for VL editors; scanners and parser can be omitted. Syntax-directed editing is supported by the grammar-based approach and by the transformation-based approach. As discussed above, the latter one comprises the grammar-based approach. All together, this leads to the following requirement:

Requirement 2.5.2 (*Syntax-Directed Editing*)
In order to get an operational semantics of VL editors we require syntax-directed editing. Therefore, we favor the transformation-based approach, where editing operations are defined by transformations on the abstract syntax. △

Note that according to this requirement, VL editors are not generated. Instead, we have to speak about a *generic* VL editor whose parameter is given by a VL specification. The VL specification defines the editing operations of the VL editor. Thus, the editor gets an interpretative (or operational) semantics. In the following, we analyze in more detail the editing operations of a language-specific VL editor. The VL we regard is a simplified version of UML class diagrams.

2.6 Analysis of the Class Diagram Language

UML [Rat99b] offers a comprehensive VL, called class diagrams, to describe object-oriented classes and their structural relationships. A class diagram is used to model the static structure of a system. It consists of a number of classes and structural relationships between them.

It is possible to define several structural relationships between different classes, like associations, generalizations, etc. Associations may be enhanced by attributes, association classes, role names and cardinality. Generalizations represent the relationship between superclasses and subclasses, i.e. between a more general class and a more specific class. Thus, the specific class is fully consistent with the superclass and adds additional information. Aggregation is a special form of binary association representing the whole-part relationship. Composition is a form of aggregation for n-ary associations which implies strong ownership and coincident lifetime of a part with the whole. For structuring complex systems, packages may be used, which are groups of class model elements. Packages may be nested. In order to keep this case study small and understandable, we do not regard aggregation, composition and packages. One class diagram is shown in Figure 2.4.

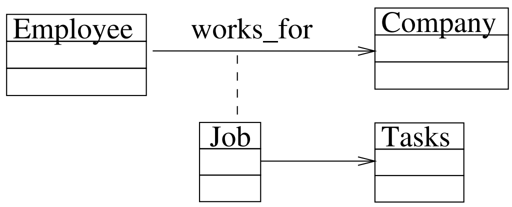


Figure 2.4: Class diagram example.

Classes may be visualized by one or three rectangles; in this section we consider three rectangles. The structural relationships are visualized by different types of lines and arrows. A binary association is given by a line or an arrow; the latter one indicates a directed association which is regarded in this section and employed in Figure 2.4. An arrow and a vertical dashed line indicate an association class. Another kind of arrow is used to indicate the structural relationship of generalization. Figure 2.5 shows all the symbols of the class diagram language treated in this section. Note that we concentrate on the editing operations on a class diagram editor. Therefore, we do not distinguish between the abstract and concrete syntax of class diagrams and use the term *symbols*.

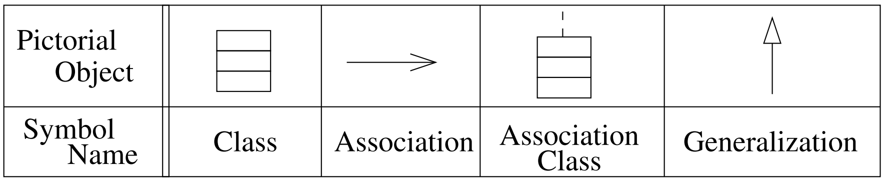


Figure 2.5: Symbols of class diagrams.

Structural relationships like associations or generalizations relate to classes which are their adherent symbols. This means, moving a class on the screen should lead to the adaption of the relationship means. This requires the explicit definition of spatial relations between symbols. Using graphs as internal representation model, spatial relations are represented by directed edges according to the *abstract syntax* and graphical constraints according to the *concrete syntax*. It is conceivable that structural relationship symbols are in the source and the adherent (class) symbols are in the target of corresponding directed edges.

As stated in Requirement 2.5.2, in a VL editor we favor syntax-directed editing following the transformation-based approach. This leads us to consider in more detail the desired editing operations which define the abstract syntax of class diagrams. Direct manipulation of the pictorial objects, given by moving or resizing, is allowed within the limits imposed by the graphical constraints. Furthermore, it is guaranteed that direct manipulation does not change the interpretation of the diagram.

In the following we consider how symbols occurring in class diagrams may be manipulated; these symbols are illustrated by Figure 2.5. Thereby, we discuss the effects resulting from inserting and deleting symbols and consider some dependencies resulting from editing of datatype symbols like the class name.

Class Symbol

Classes are represented by three connected rectangles of the same width. Each rectangle can include a datatype value, these are a class name, an attribute list and a method list. The size of each rectangle depends on the size of its datatype value.

Insertion If a class symbol is inserted in a class diagram, the user must define a certain position. Usually it is required that the new class symbol does not overlap with some already existing class symbols. However, overlapping class symbols do not violate the syntax of class diagrams. Hence, we prefer to allow overlapping of class symbols.
The graphic of a class symbol is not unique. Therefore, the user is forced to specify a unique class name if a new class symbol is inserted by a corresponding production. Then the LHS of a production allowing the insertion of a class symbol is empty, and the RHS comprises the class symbol and a variable designating the class name. This variable indicates that the user must define a class name when applying the production.
In a class diagram the class names are required to be unique. Note that in order to express the uniqueness of class names we need an application condition which is coupled with the production.

Deletion It should not be possible to delete a class symbol which is related to another class symbol. Since the corresponding relationship symbol does not connect two class symbols any more, the result of deleting a connected class symbol would lead to an incorrect class diagram. A class symbol which is not related to another class symbol can be deleted. In order to express this we need application conditions which are coupled with the production allowing the deletion of a class symbol. Note that these application conditions force the user to delete the relationship symbols before the production allowing the deletion of a class symbol can be applied. However, if a class symbol is deleted it is required that all its datatype values are deleted, too.

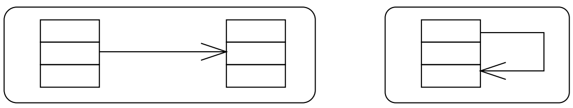
Dependencies Changing the class name of a class symbol may force a changing of the upper rectangle's width. In general, changing the width of one rectangle requires to change the width of the other two rectangles due to the fact that all three rectangles should have the same width. The same is true for attributes (methods) occurring within an attribute list (method list).

The height of a rectangle may also be changed. E.g. the rectangle in the middle of a class symbol is used to include the attribute list. Usually, an attribute list consists of a finite set of attributes, each one is put into one row. The height of the attribute rectangle depends on the number of rows and the height of each row. Inserting a further attribute results in a resizing of the middle rectangle of the class symbol. Furthermore, if the expansions run from top to bottom also the position of the lower rectangle is changed.

Binary Association Symbol

Binary associations are represented by arrows between class symbols. This means that an association symbol always begins and ends at a class symbol. Note that the direction of association arrows is not fixed and has no meaning. For layout adjustment the user should be able to extend a simple association arrow to a polyarrow.

Insertion Inserting an association symbol requires the existence of one or two class symbols where the arrow can begin and end at. Figure 2.6 illustrates the two situations which may occur if an association symbol is inserted. Figure 2.6 (a) shows two classes which are related by an association given as an arrow. Figure 2.6 (b) illustrates an association symbol defined on one class symbol; the association is given by a polyarrow. Because of the different layouts, two different productions allowing the insertion of an association symbol are useful. Note that the insertion of an association symbol requires the existence of one or two class symbols. This may be expressed in the LHSs of the corresponding productions; Figure 2.6 illustrates the conceivable RHSs for both situations.



(a) (b)

Figure 2.6: Association between two classes (a) and association defined on one class (b).

We assume that class symbols are always enhanced with unique class names. Inserting an association symbol between two class symbols, it is not absolutely necessary that the new association symbol gets a unique name, too. The class names are unique and thus, an association is unique, provided that only one association relates the two class symbols. However, a production allowing the insertion of an association name must be available. Due to the concrete syntax the name must be placed according to the direction of the association arrow. Note that the insertion of an association name requires the existence of an association arrow. As before, this may be expressed by the LHS of the corresponding production; the RHS of the production may comprise the association symbol and a variable indicating the association name required from the user.

Deletion The deletion of a binary association symbol causes no problem. This means that only the existence of an association symbol is required by the LHS of the corresponding deletion production; the RHS is empty.

Dependencies The length of an association arrow may be defined by the length of the association name. Of course, this depends on the direction of the arrow. However, for readability reasons the association name should be always aligned horizontally.

Association Class Symbol

Association symbols may be enhanced with class symbols. This is represented by an association arrow together with a vertical dashed line which is connected to an associated class symbol. Usually, a vertical dashed line is connected to the top of a class symbol, but another attachment point of the class' contour does not violate the syntax.

Insertion It is unusual to insert an association symbol together with its associated class symbol at once. During class diagram editing we can assume that an association symbol is extended by an associated class symbol. This assumption requires the existence of an association symbol which may be expressed by the LHS of the corresponding production. The RHS may comprise the associated class symbol and a new class symbol. It is required that the association name is not changed by the production provided that the name is existent.

Deletion The deletion of an association class symbol causes no problems. However, the LHS of the corresponding deletion production requires the existence of this symbol; the RHS is empty.

Dependencies Although a dashed line is usually aligned vertically, it must get a horizontal alignment if the attached arrow has nearly a vertical alignment.

Generalization Symbol

The structural relationship of a generalization symbol is visualized by an arrow; the top is given by an unfilled triangle. The generalization symbol should have the same properties as an association symbol. This means that it begins and ends at a class symbol. However, in contrast to an association symbol, a generalization symbol always occurs between two class symbols.

Insertion Class symbols may be specialized by other class symbols. This kind of structural relationship is expressed by a generalization symbol visualized by an arrow with a triangle as its top. To define a production allowing the insertion of such a generalization symbol, several situations must be regarded as illustrated by Figure 2.7.

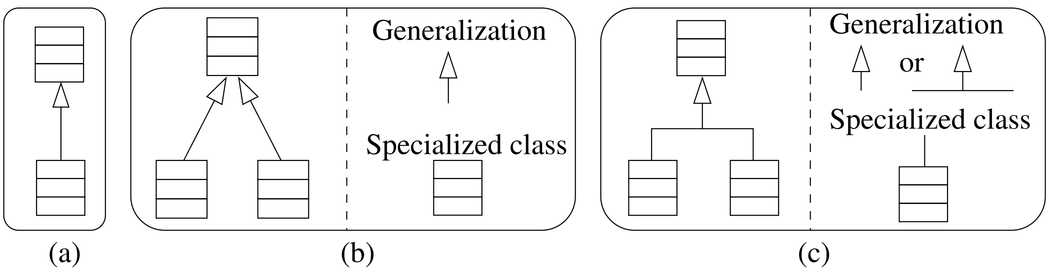


Figure 2.7: Generalization of a class symbol: one specialized class symbol (a) different layouts for multiply specialized class symbols (b) and (c).

Figure 2.7 (a) illustrates the result of inserting a generalization symbol between two class symbols. The corresponding production allowing the insertion of the generalization symbol can be defined similar to that allowing the insertion of an association symbol. This means that in the LHS of the corresponding production, the existence of two class symbols is required; the RHS looks like Figure 2.7 (a). Note that it is also conceivable to require only the existence of one class symbol in the LHS; the RHS is not changed.

Figures 2.7 (b) and (c) show two specialized class symbols with different layout. Figure 2.7 (b) shows the result after applying the production allowing the insertion of a generalization symbol as mentioned above. This result is achieved with respect to a requirement that class symbols do not overlap. The desired kind of representation is illustrated by 2.7 (c) where additionally the generalization symbol's single components are shown. This kind of representation seems to be difficult to express by corresponding grammar productions. It requires the availability of if-then-else constructs as well as lots of graphical constraints. Probably this is the reason why commercial tools like *Rational Rose* [Rat99a] provides the user with the pictorial objects as illustrated in Figure 2.7 (b).

Deletion A production allowing the deletion of a generalization symbol is similar to that allowing the deletion of an association symbol. Regarding the kind of visual representation as shown in Figure 2.7 (b), the deletion is as difficult to describe as the insertion.

Dependencies Usually, a generalization symbol is enhanced with a name. This name is to be treated similar to the association name. This means that the name is aligned horizontally. It is placed beside the generalization symbol.

Summary

In this section we analyzed some conceivable editing operations of a class diagram editor. The main attention was directed to the expressiveness of productions. Note that we did not distinguish non-terminal and terminal symbols. Instead we mentioned only terminal symbols in the productions. Thereby, we did not directly distinguish between nodes representing pictorial objects, however, we used the term *symbol* instead.

In Figure 2.5 we only mentioned the symbols of the class diagram language. According to Requirement 2.2.1 these symbols are represented by nodes of the underlying graph. The spatial relations are not mentioned explicitly, but discussed with respect to editing operations. In order to allow syntax-directed editing following the transformation-based approach, where additionally the VL editor should get operational semantics, all editing operations must be specified by grammar productions. Thus, it is useful to define the relations once by an alphabet and to profit from this alphabet during the definition of productions.

One important feature we required for inserting and deleting class symbols are application conditions coupled with the corresponding productions. We assume that such application conditions may be useful in order to express some semantical situations which may or may not occur if the production is applied.

The following chapter introduces our conceptual model for the visual definition of VLs taking into account all these features.

Chapter 3

The Conceptual Model of GENGED

This chapter introduces the conceptual model of GENGED, a visual environment for visual languages (VLs). The environment supports the visual definition of VLs and the editing of VL sentences. Accordingly, the environment comprises two components: the GENGED specification tool and a generic visual language editor (VL editor).

The visual definition of visual languages results in visual language specifications (VL specifications), each one comprises an alphabet and a grammar. The concepts of VL specifications are implemented by the GENGED environment. Hence, these concepts are in the focus of this chapter. The internal representation model of sentences as well as their editing belong to these concepts, too. The dependencies between concepts and environment are illustrated by Figure 3.1.

The GENGED specification tool comprises an alphabet editor and a grammar editor for establishing the constituents of VL specifications. VL specifications are the input of the generic VL editor. In Chapter 2 we required syntax-directed editing for a VL editor. Hence, grammar productions occurring in VL specifications have to be defined in such a way that they can be used as edit commands of the VL editor. Doing this, the VL editor gets an operational semantics with respect to language-specific editing.

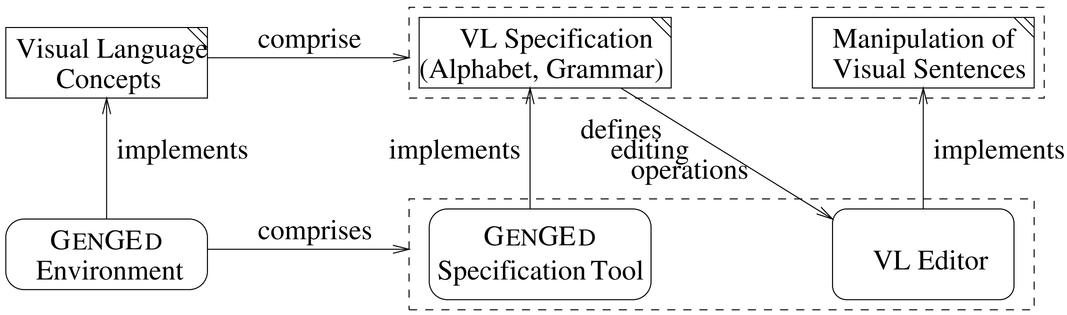


Figure 3.1: Dependencies of the visual language concepts and the GENGED environment.

VL sentences which can be edited using the VL editor consist of symbols and links. Note that links express the relations between symbols according to the abstract and the concrete syntax level of VLs. With respect to the abstract syntax level, symbols are represented by graph nodes, and links are represented by graph edges. Thus, VL sentences are represented by graphs and the editing of VL sentences is supported by graph transformation. In order to define VLs at the concrete syntax level (using the GENGED specificiation tool) we developed a suitable and explicit description model for graphical objects and constraints. This description model together with the representation and the transformation concepts underlying our framework ensures that the abstract syntax as well as the concrete syntax of VLs is correctly kept in all necessarily steps.

Symbols and links occur not only in VL sentences, but in the alphabet and the grammar as well. An alphabet is interpreted as a type system for symbols and links. A grammar consisting of a start sentence and a set of productions is based on the alphabet. Each grammar production comprises a *left-hand side* (LHS) and a *right-hand side* (RHS). We propose context-sensitive productions which can be enhanced with *negative application conditions* (NAC). Such conditions express certain situations that inhibits the application of a grammar production to a VL sentence if this situation occurs in the VL sentence. The constituents of a production (LHS, RHS, NAC) are *visual sentences*, i.e., instances of the alphabet. Note that also VL sentences are visual sentences. VL sentences are such visual sentences which are derived from grammar productions.

The development of the VL concepts is influenced by visual definition aspects including feasibility as well as designer-friendly aspects. Our concepts are additionally influenced by the analysis and requirements carried out in the previous Chapter.

This chapter is organized as follows. We start with the description model of graphics and constraints in Section 3.1. Symbols and links occur in the alphabet and the grammar of VL specifications, and in VL sentences. Hence, in Section 3.2, symbols and links are introduced autonomously. The concepts of the alphabet, the grammar and VL sentences are then straight forward. The alphabet is explained in Section 3.3, and the grammar is proposed in Section 3.4. In order to illustrate the concepts, we model parts of the class diagram language which has been analyzed in Chapter 2.6. In Section 3.5 we show how VL sentences can be derived from the applicaton of grammar productions. The GENGED environment implements the concepts mentioned so far. Hence, in Section 3.6 we only briefly introduce the concepts of the GENGED environment.

3.1 Graphical Objects and Graphical Constraints

Graphical objects and constraints form the basis for the concrete syntax level of visual languages (VLs). This causes the development of a small but suitable description model. In Section 3.1.1 we suggest the graphical objects used in this thesis. Additionally, we regard datatype objects. Such datatype objects determine the attributes of symbols like the class name of a class symbol.

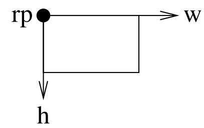
Graphical objects can be spatially related to each other with respect to user-defined attachments. We model these attachments as access operations on graphical objects in Section 3.1.2. This modeling concept allows us for deriving constraint variables usually occurring in graphical constraints. These constraints express the spatial relations between graphical objects. In Section 3.1.3 we suggest an interface to constraints and constraint-satisfaction problems.

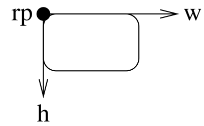
3.1.1 Graphical Objects

The graphical objects we regard in this thesis are primitive graphics like rectangles, arrows, etc., but also datatype objects. We omit properties like filling possibilities, line styles, colors, etc. However, each graphical object occupies a rectangular area. The area is determined by its reference point rp, its width w

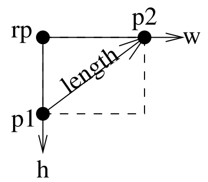
and its height h. We illustrate the area as a dashed rectangle in the following. Note that a rectangle corresponds exactly to this area. In order to support also the treatment of iconic VLs, we offer the possibility to handle bitmap graphics as well. Bitmap graphics can be treated like rectangles, but they cannot defined to be invisible.

A **rectangle** (a rounded rectangle) is defined by its reference point rp, its width w and height h. A rectangle can be moved and scaled. A rectangle can be defined to be invisible; then, we call it **place holder**. In Chapter 7, we show that the invisibility of rectangles is helpful for the visual definition of VLs.

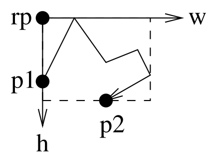




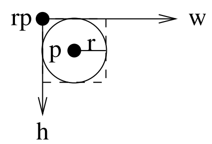
An **arrow** (line) is defined by the two points $p1$ and $p2$ and its *length*. An arrow can be moved and lengthened. The area an arrow (line) occupies, is illustrated by the dashed rectangle.



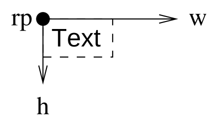
A **polyline** (-arrow) is a set of lines. A polyline has a certain begin point $p1$ and end point $p2$. It can be moved at all and lengthened in particular.



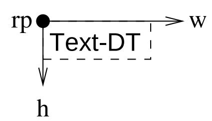
A **circle** is defined by a center point p and a radius r. These properties determine the area a circle occupies. A circle can be moved and scaled.



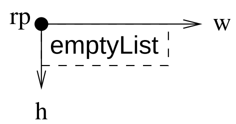
A static **text** object is given by a string like “Text”, text size and text font. These properties define the area a string object occupies. A static string object can be moved.



A **string** datatype object is defined by a value, text size, and text font. In contrast to static text, the value can be changed by productions.



A **list** datatype object denotes a list of string objects which can be arranged vertically or horizontally in the list object. Although a list is empty by default, for illustrational reasons it is useful to put one string like “emptyList” in its top. A list object can be changed (edited) by productions.



We suggested few graphical objects because we do not like to model a fully low-level graphics editor, but to concentrate on the visual definition of VLs. We are sure that the graphical objects proposed are sufficient to solve this task. Common to all graphical objects is a rectangular area they occupy. This leads to the following concept.

Concept 3.1.1 (*Graphical Objects*)
A *graphical object* consists of geometrical points. It is determined by object-specific properties and the properties of the area the object occupies as well. The properties of the area is given by the reference point rp, the width w and the height h. The reference point determines the position of a graphical object. The width together with the height determines the object's size. $\triangle$

The object-specific properties and the area a graphical object occupies concern attachments which are relevant for defining spatial relations between graphics or more precisely, for constraint definitions. In the following section we model the properties of graphics as access operations on graphics. Note that the object properties and thus the access operations are important for constraint definitions which are explained afterwards.

3.1.2 Access Operations on Graphical Objects

In contrast to low-level graphic editors, we permit graphical constraints in order to relate graphical objects. Therefore, attachment points must be available for each graphical object. These attachments belong to object-specific points, circumferences, and the area an graphical object occupies. In our description model for graphics and constraints, attachments correlate to access operations on graphics. Moreover, graphics and access operations determine constraint variables used for constraint definitions. These facts are formulated in the following concept.

Concept 3.1.2 (*Access Operations on Graphical Objects*)
Given a graphical object. *Access operations* on the graphical object support the access on certain object coordinates, its circumference and its areas.

Given a graphical object g and some access operations ac on this object. Each access operation on a graphical object determines a (constraint) variable which is denoted by $g.ac$. $\triangle$

The access operations (attachments) which are used through the rest of this thesis are illustrated by Figure 3.2. Figure 3.2 (a) shows the access operations available on all graphical objects according to the area an object occupies. These operations reflect the compass coordinates. For example, the upper left corner is denoted by .nw indicating the *north-west* corner; by .se, the *south-east* corner is meant. Note that the corner .nw indicates also the graphic's reference point, i.e. its position. Lines and arrows can be additionally accessed by their begin (.beg) and end (.end) points as illustrated in Figure 3.2 (b). The access on the circumference of a graphical object and its area is also possible. The circumference of graphics is determined by .circ, and the access on their area is described by .area in Figure 3.2 (c) and (d), respectively.

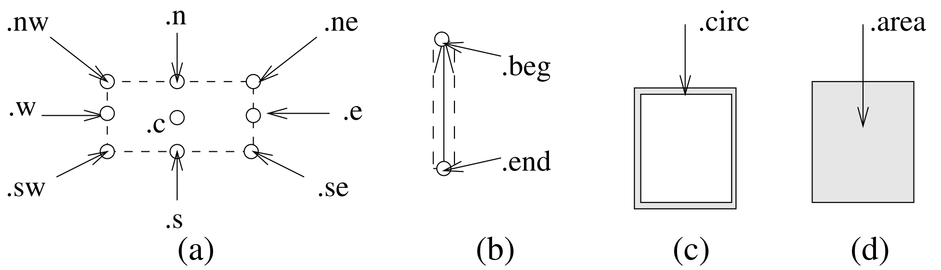


Figure 3.2: Access operations available on all graphics (a), especially on lines and arrows (b), on circumferences (c), and on areas (d).

The access operations on graphics presented in Figure 3.2 are not complete: we neither illustrated the access on single (x,y) coordinates nor the access on the width and height. However, constraint variables which correspond to access operations (attachments) form the basis for constraint definitions.

3.1.3 Graphical Constraints and Constraint Networks

In low-level graphic editors, it is possible to group graphical objects, however, a further access on the grouped objects is forbidden. Such a behavior is not desired in a VL editor. Hence, we propose constraints in order to relate graphical objects.

In Chapter 2.2 we used terms like “adjacent”, “nested”, and “include” to denote spatial relations between graphics. Thereby, we caused the user’s intuition with respect to the semantics of those constraints. Usually, constraints are defined on object-specific properties like coordinates. For example, in order to express that a graphic $g1$ is “above” another graphic $g2$, their y coordinates must be taken into account, thus the vertical adjustment: $g1.y > g2.y$. This constraint allows for intersection of both graphics. The x coordinates of the graphics are not taken into account. This example is a short introduction into graphical constraints. We call constraints like $g1.y > g2.y$ *low-level constraints* (LLCs). Constraints like $g1$ “above” $g2$ are called *high-level constraints* (HLCs). Their semantics is defined by LLCs.

In our opinion, working with LLCs is not very user-friendly. Hence, we require the availability of HLCs. Because this requirement concerns the GenGEd implementation this topic is shifted to Chapter 5 that introduces the implementation. In this section we mostly employ LLCs like “equalPosition” indicating equal positions. Furthermore, we focus on constraints and on constraint-satisfaction problems but not on the numerous solving algorithms. Specific constraint solver and solving algorithms have been proposed e.g., in [Bor90, Bor94, San95]. However, together with the graphical objects proposed in Section 3.1, constraint-satisfaction problems are relevant for the concrete syntax level of visual languages.

In general, constraints extend low-level graphic editors by ensuring desired constellations. The durable specification of relations between graphics allows easy modification of the whole graphic because a local modification usually does not destroy the structure. The main emphasis of a constraint solver for an interactive graphical editor is to find permanently solutions such that the corresponding constraint-satisfaction problem is satisfied. In the following we introduce our terminology for constraint-satisfaction problems.

Constraint A constraint is a declarative description of a boolean operation between one or more constraint variables. Constraints do not express an order of the calculation because constraints are undirected relations.

Equalities and inequalities are special cases of constraints specifying an (in)equality in the mathematical sense. An inequality constraint specifies a restriction of an open or closed interval of numbers.

A graphical constraint is a numerical constraint which is graphically interpreted.

Constraint Network A constraint network results from the combination of constraints which share constraint variables. A constraint network can be acyclic or cyclic, as well as over- or under-specified. There are several possibilities for the treatment of such cases.

An over-specified network contains more constraints than constraint variables. Usually, an over-specified network is contradictory, so there may be no solution. If there is no solution, it is possible to inform the user about this fact and to force him to remove or change some constraints. Alternatively, weights can be distributed to constraints in order to get at least a partial solution. The partial solution does not satisfy all constraints but the most important constraints (from the user point of view).

An under-specified network contains more constraint variables than constraints. Such a network usually is not unique. So there may be many solutions which is obstructively in graphical applications. In order to get one solution there is the principle of least astonishment proposed in [Gri96]. That means, the chosen solution preserves locality so that it is near by the previous solution.

Constraint Satisfaction A solution of one constraint is given by concrete values for the corresponding constraint variables. A solution satisfying all constraints which occur in a constraint network is called *constraint satisfaction*. Whenever all constraints are satisfied, the values are assigned to the constraint variables.

Summarizing the explanations of this section yields the following concept:

Concept 3.1.3 (*Constraint-Satisfaction Problem*)
A constraint network comprises a set of constraint variables and a set of constraints sharing the constraint variables. Each constraint variable has a certain domain; a value from this domain can be assigned to the constraint variable.

A constraint-satisfaction problem is given by a constraint network which has to be satisfied. A constraint-satisfaction problem is satisfied if there is a variable assignment for each constraint variable such that all constraints are satisfied. $\triangle$

Graphical objects and constraint-satisfaction problems determine the concrete syntax level of visual languages. On the one hand, constraints may be used to define one complex symbol graphic. On the other hand, constraints must be used to determine the links between symbol graphics. However, not only the concrete syntax level of symbols and links is important, but the abstract syntax level as well. Therefore, in the next section we consider symbols and links with respect to both syntactical levels.

3.2 Symbols and Links

In Chapter 2.2 we considered several diagrams. In our terminology, these diagrams are visual sentences of different visual languages (VLs). Each of these sentences comprises visual symbols that are spatially related. Most of these symbols are attributed by some values. For example, an entity in a message sequence chart (MSC) has a unique name as well as each message between entities. Also class symbols as they occur in class diagrams can be attributed by a class name, an attribute list and a method list. This causes us to distinguish two kinds of symbols as well as two kinds of spatial relations. The latter one we will call *links* furtheron.

Pure graphical symbols like a class symbol of a class diagram define the lexical symbols of VLs. Attribute symbols like the class name define the *textual* attributes of lexical symbols. Due to Requirement 2.2.1, each symbol is defined with respect to its abstract syntax and its concrete syntax. This means that the abstract syntax items are equipped with their symbol graphics.

Concept 3.2.1 (*Symbols*)
A *symbol* is defined by a *symbol name* and a *symbol graphic*. The symbol name indicates the abstract syntax level, and the symbol graphic indicates the concrete syntax level of a symbol. The symbol name is coupled with its symbol graphic.

A symbol is represented by a labeled and attributed graph node. The label represents the symbol name and the node attribute represents the symbol graphic.

A symbol graphic may consist of several graphical objects like rectangles and lines which are related by graphical constraints, called *symbol constraints*. The symbol constraints determine a constraint-satisfaction problem with constraint satisfaction. The constraint variables occurring in the constraint-satisfaction problem are constructed with respect to Concept 3.1.2. Additionally, the constraint variables keep the symbol name.

A pure graphical symbol is considered as a *lexical symbol*. An *attribute symbol* is a symbol used as “textual” attribute of a lexical symbol. Like a lexical symbol, it consists of a symbol name and a symbol graphic capturing the corresponding textual expression (value). An attribute symbol is always linked to a lexical symbol. △

We will illustrate this concept by an example:

Example 3.2.2 (*Symbols*)
Figure 3.3 shows some symbols adapted from the class diagram language as analyzed in Chapter 2.6. A class symbol (Class) and an association symbol (Assoc) used in class diagrams are lexical symbols. A class name (CN) and an association name (AN) are their “textual” attributes. Not only lexical symbols have a certain layout, but also attribute symbols, i.e., textual expressions. They are visualized with respect to text font and text size.

Syntax	Lexical Symbols		Attribute Symbols	
Abstract	Class	Assoc	CN	AN
Concrete		$\longrightarrow$	"ClassName"	"AssociationName"

Figure 3.3: Lexical and attribute symbols of class diagrams.

△

Box concept of symbol graphics

The *concrete syntax* of symbols (symbol graphic) is given by an invisible box enclosing all graphical objects like rectangles, circles, etc. used to build a symbol graphic. The positions of the objects are set relatively to the position of its enclosing box. The position of the box is set relatively to the next layer which is also interpreted as a box. The next layer is the drawing area where the box is placed.

Concept 3.2.3 (*Box Concept of Symbol Graphics*)
A symbol graphic consists of an invisible box enclosing the set of graphical objects belonging to the symbol.

A box is treated like a primitive invisible rectangle with respect to its position and size (cf. Section 3.1). The positions of each enclosing graphical object, defined by their reference points (the upper left corners), are set relatively to the position of the box. △

The size of a box is influenced from constraints between graphical objects enclosed or, more precisely, from solutions for the corresponding constraint-satisfaction problems. In the following we exemplify the box concept and constraints by an example:

Example 3.2.4 (*Box Concept and Constraints*)
Three rectangles, denoted by $recti$_Class, $i \in \{1, 2, 3\}$, define a symbol graphic for a class symbol. They are illustrated in Figure 3.4. The invisible box enclosing the three rectangles is presented as a dashed rectangle. The positions of the class symbol rectangles are set relatively to the position of the enclosing box which is given by the reference point (upper left corner). Note that the box position ([3,1]) is irrelevant for the positions of enclosed graphics. The box position is set relatively to the next layer which is the drawing area. However, the box size is calculated from sizes and positions of the enclosed objects. In this example the box size is [2,3].

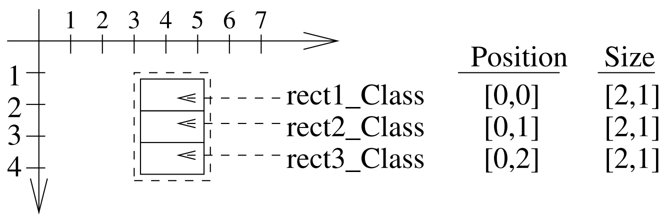


Figure 3.4: Symbol graphic of a class symbol.

In order to ensure correct positions and sizes for a class symbol graphic, several constraints are defined on corresponding constraint variables. We achieve these variables with respect to the access operations on graphics suggested in Concept 3.1.2. For the class symbol graphics we need only the upper and lower corners of the involved rectangles:

rect1_Class.sw = rect2_Class.nw; rect2_Class.sw = rect3_Class.nw;
rect1_Class.se = rect2_Class.ne; rect2_Class.se = rect3_Class.ne;

△

The box concept allows resembling treatment of graphical objects independently whether it is a symbol, a drawing area, a sentence part, etc. Each box can be used for constraint definitions, especially for constraints between symbol graphics. These kind of constraints concern the concrete syntax level of links between symbols. The abstract syntax of links is given by directed edges between symbols or, more precisely, between the nodes representing the symbols.

According to the two kinds of symbols we distinguish between two kinds of links. A connection link bounds two lexical symbols, whereas an attribution link bounds a lexical and an attribute symbol. This leads to the following concept for links. Note that Requirement 2.2.1 is met.

Concept 3.2.5 (*Links*)
A *link* between two given symbols is defined by a named *link edge* and some *link constraints*. The named link edge indicates the abstract syntax level, and the link constraints indicate the concrete syntax level of a link.

A *link* is represented by a labeled directed edge between the two nodes representing the symbols. The edge label represents the link name. Link constraints extend the constraint-satisfaction problems of the symbol graphics involved. The corresponding constraints are satisfied.

A *connection link* is a link between two lexical symbols. An *attribution link* is a link between a lexical and an attribute symbol. △

We will explain the Concept 3.2.5 by an example:

Example 3.2.6 (*Links*)
Figure 3.5 shows two connection links and one attribution link. These links are adapted from the class diagram language analyzed in Chapter 2.6. Each link is represented by a named edge between each two symbol names indicating the abstract syntax (nodes). With respect to the concrete syntax of the connection links, “equalPosition” constraints are used in order to relate the symbol graphics involved. These constraints are illustrated by dashed lines and marked with “=”

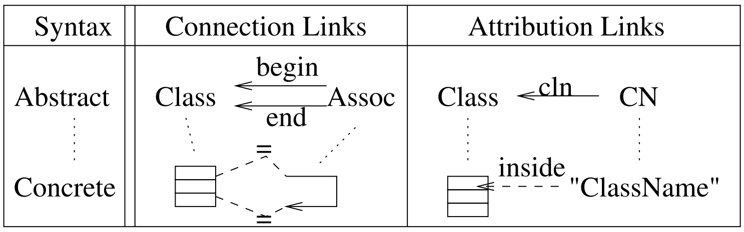


Figure 3.5: Connection and attribution links used in class diagrams.

in Figure 3.5. The constraint of the attribution link indicates that the “ClassName” is “insight” the rectangle. This directed high-level constraint (HLC) is illustrated by a dashed arrow. △

Note that we use graphical constraints to build symbol graphics consisting of more than one graphical object as well as to establish the link constraints. However, important is the distinction between the two kinds of symbols and links as well as the two kinds of syntactical representations, namely, the abstract and the concrete syntax of symbols and links. In the following sections we focus on the latter distinction. We introduce the concepts for the constituents of VL specifications (alphabet, grammar). Furthermore, we show how sentences can be derived from grammar productions.

3.3 The Alphabet

An alphabet of a visual language (VL) is defined by symbols and links between symbols and additionally by a constraint-satisfaction problem. The alphabet is interpreted as a type system for the instances occurring in visual sentences. Under consideration of the VL modeling aspects, we distinguish lexical symbols and attribute symbols as well as connection links and attribution links we introduced in Section 3.2.

Due to Concept 3.2.1, a user-defined symbol type consists of a symbol name for the abstract syntax and a symbol graphic for the concrete syntax. The symbol graphic is given by a box graphic as proposed in Concept 3.2.3. The position of a box is not relevant in an alphabet but its size as well as the positions and sizes of the enclosed graphical objects. These graphical objects and additionally the box size yield unique constraint variables which correlate to the access operations on graphics proposed in Concept 3.1.2. Graphical objects occurring in one symbol graphic are coupled by (symbol) constraints. The corresponding constraint variables and constraints define a constraint-satisfaction problem for the specific symbol. The constraint variables get concrete values according to the user-defined symbol graphic. Doing this, there is a *subassignment* of constraint variables. The subassignment ensures that symbol instances have first of all this predefined layout.

Link types are defined between two symbol types. A link type consists of a link edge between symbol names (the abstract syntax) and some constraints between symbol graphics (the concrete syntax). It is required that link edges are acyclic. The set of (symbol) constraint variables possibly is extended if link constraints are defined. In this case we have a disjoint union of constraint variables and constraints. In contrast to the subassignment of constraint variables expressing the layout of symbol types, the definition of symbol and link constraints do not result in a subassignment of constraint variables. Altogether, a constraint-satisfaction problem for a specific alphabet is constructed with respect to the concrete syntax.

Concept 3.3.1 (*Alphabet*)
An *alphabet* establishes a type system for symbols and links of a visual sentence, i.e., an instance of the alphabet. With respect to the abstract syntax level, an alphabet is represented by an attributed graph: nodes represent the symbols, edges represent the links. Node attributes represent the datatypes, a symbol may be enhanced with. The concrete syntax level extends the abstract syntax level by further graphical attributes and a constraint-satisfaction problem. In order to achieve unique graphical objects on the instance level, each constraint variable comprises the corresponding symbol names. The constraint-satisfaction problem of an alphabet is achieved as follows:

Given a (user-defined) symbol graphic and corresponding constraint variables (cf. Concept 3.1.2). That variables which express the size of the box and the positions and sizes of enclosed graphical objects get concrete values from their specific domain (cf. Concept 3.2.3; *subassignment of constraint variables*). The corresponding symbol constraints and link constraints are defined over these constraint variables.

We distinguish lexical and attribute symbol types according to Concept 3.2.1 as well as connection and attribution link types according to Concept 3.2.5.

The symbol and link names are unique in one alphabet. Furthermore, link edges are acyclic. △

The requirement for acyclic link edges concerns the expressiveness of directed edges with respect to instances occurring in visual sentences. The edges defined between a source and a target symbol (name) require on the instance level for a source symbol instance the existence of a target symbol instance. As we will show by the concepts of the GENGED environment in Section 3.6, the allowance of cycles for link edges is obstructive with respect to the generation of so-called alphabet productions.

Let us explain the alphabet concepts by an example. Like in the previous sections we consider the language for class diagrams or, more precisely, a small part of its alphabet.

Example 3.3.2 (*Alphabet of Class Diagrams*)
Figure 3.6 illustrates two connection links between the Assoc symbol and the Class symbol (cf. Examples 3.2.2 and 3.2.6). The symbol graphic of a Class symbol together with the corresponding constraint-satisfaction problem is already explained in Example 3.2.4 The symbol graphic of the Assoc symbol is determined by an arrow with begin point and end point. These points are used as constraint variables, namely, arrow_Assoc.beg and arrow_Assoc.end. There is a subassignment of these variables according to the positions of the arrow points which are set relatively to the corresponding enclosing box similar to the Class symbol graphic (cf. Example 3.2.4). No constraints are defined for the association graphic.

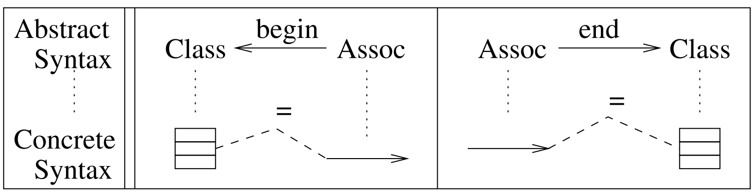


Figure 3.6: Two connection links.

Two links are shown in Figure 3.6, each consists of an edge with respect to the abstract syntax. For each link edge constraints are defined expressing how the symbols are spatially related. They are indicated by the dashed lines. Assuming a constraint variable box_Class.circ correlating to the circumference of a Class symbol graphic we are able to formulate the following two constraints for these links:

begin link:	arrow_Assoc.beg = box_Class.circ;
end link:	arrow_Assoc.end = box_Class.circ;

△

For a VL designer it is important to work out the relevant symbols and links of a visual language. This influences strongly the definition of an alphabet. Important are the link edges requiring for each visual sentence, that the existence of a source symbol implies the existence of a target symbol. In the case of class diagrams, e.g., each association arrow must be connected with one or two classes according to the begin and end links. A situation where one of these links is not defined, is not correct with respect to the alphabet.

Remark 3.3.3 (*Alphabet Graph*)
In order to clarify the dependencies of link edges, it is possible to draw the abstract syntax level of an alphabet as a graph. Symbol names are represented by labeled nodes and named link edges are represented by labeled edges between the labeled nodes. △

We already used the notions of graphs in the figures. The labeled graph nodes represent the symbol names and the labeled graph edges represent the link names.

Instances of symbols and links occur in visual sentences and therefore, in the constituents of the language-specific grammar. In the following section we introduce the concepts of VL grammars. Thereby, we concentrate on the expressiveness of grammar productions with respect to the abstract syntax level rather than on the concrete syntax level.

3.4 The Grammar

A VL grammar is similar to a string grammar in the sense that the grammar consists of a start sentence and a finite set of productions. In this section we concentrate on grammar productions defining the edit commands of a syntax-directed VL editor. Thereby, we distinguish *generating productions* and *editing productions*. Generating productions concern the generation of symbols and links. Editing productions concern the deletion and the exchange of symbols and links. They may as well express more comprehensive language-specific edit commands.

The start sentence is a visual sentence comprising instances of symbols and links with respect to a given alphabet. This means that symbols and links are typed over the alphabet. Concrete graphics are obtained by instantiating the constraint-satisfaction problem of the alphabet. The instantiation process includes the subassignment of constraint variables. Furthermore, an instance of the alphabet is a visual sentence iff there is a constraint satisfaction. The start sentence as well as constituents of grammar productions are visual sentences:

Concept 3.4.1 (*Visual Sentences*)
Given an alphabet of a specific visual language. A *visual sentence* comprises several symbol and link instances with respect to the alphabet.

With respect to the abstract syntax level, each symbol instance (respectively link instance) is typed over the corresponding symbol (respectively link) of the alphabet. Furthermore, for each link edge type with source and target symbol types, the existence of a source symbol implies the existence of a target symbol.

With respect to the concrete syntax level, the set of instantiated symbols and links yields a constraint-satisfaction problem for the sentence with constraint satisfaction. △

In the following we give a brief example of a visual sentence. We describe how sentence' constraint-satisfaction problems are obtained as well.

Example 3.4.2 (*Visual Sentence*)
Given the alphabet of the class diagram language as sketched in Example 3.3.2. Figure 3.7 illustrates the abstract and concrete syntax of a visual sentence with

respect to the alphabet. According to Concept 3.2.1 the symbols consist of a symbol name and a symbol graphic. We used the numbers 1, 2 and 3 for the symbol names. Class and Assoc indicate the symbol types from which the symbols are instantiated. Thus, we have the necessarily information for the link edges (cf. Concept 3.2.5), namely, the types of source and target symbols.

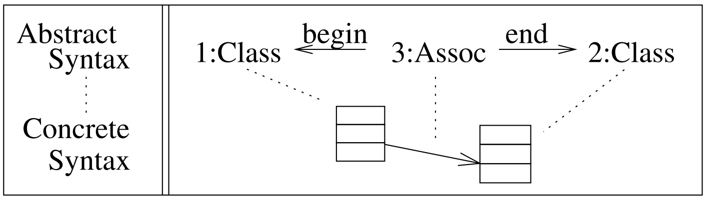


Figure 3.7: Abstract and concrete syntax of a visual sentence.

In order to visualize a visual sentence, we construct the concrete syntax as follows. Given the abstract syntax of a visual sentence. For each symbol a constraint-satisfaction problem is generated with corresponding subassignments of instantiated constraint variables. These subassignments result in the default layout of symbols but do not determine their positions. The positions (and sizes) are expressed by link constraints which are instantiated from the alphabet for each link edge, too. Solving the constraint-satisfaction problem of the visual sentence yields a solution. Due to Concept 3.2.1 the constraint variables indicate certain properties (including positions and sizes) of specific symbol graphics. Hence, the symbol graphics are generated from constraint variables and their (solution) values. The nodes representing the symbols are attributed by their layout information such that the sentence can be visualized. △

Similar to textual grammar productions, a VL production consists of a left-hand side (LHS) and a right-hand side (RHS). A grammar production describes how symbols and links in its LHS can be replaced by symbols and links in its RHS if the production is applied to a given visual sentence. Negative application conditions can be bound to productions in order to present the application context in certain situations.

Concept 3.4.3 (*Visual Language Grammar*)
A *VL grammar* consists of a start sentence and a finite set of productions defining the edit commands of a VL editor. We distinguish generating and editing productions.

A *grammar production* consists of a unique production name, optionally a list of parameters indicating attribute symbols, a left-hand side (LHS), a right-hand side (RHS) and optionally a set of negative-application conditions (NACs).

The symbols in the LHS should be mapped symbols in the RHS; this is called *production morphism*. For each NAC it is required, that all symbols in the LHS are mapped to symbols in the NAC; this is called *NAC morphism*. △

Example 3.4.4 (*Grammar Production*)
Figure 3.8 illustrates one production for class diagram editing. Both, the production's LHS and the RHS are visual sentences. The production parameter indicates a value of an attribute symbol, namely, a concrete class name if the production is applied to a given visual sentence. This value is represented by the variable cn. In the production's RHS, the variable *cn* is placed "inside" the class name rectangle according to the concrete syntax level of the corresponding attribution link.

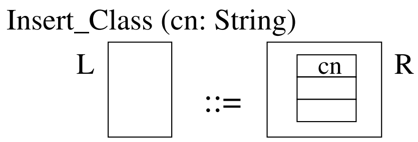


Figure 3.8: Visualization of the production for the insertion of a class symbol and a class name.
△

The example above gives a brief introduction to grammar productions. In the following we explain in more detail the constituents of such grammar productions and what they express.

Production parameters

A production parameter is always determined by a variable and the abstract syntax type of the corresponding attribute symbol type. It is linked to a lexical symbol under consideration of the corresponding alphabet link. It depends on the expressiveness of the production in which of its constituents (LHS, RHS, or NACs) the parameter is placed.

Example 3.4.5 (*Production Parameter*)
The abstract and concrete syntax of the production shown in Figure 3.8 is depicted in Figure 3.9. The abstract syntax of the production parameter is internally mapped to the LHS of the production. We used the term *abstr(cn)* in order to make clear the parameter's abstract syntax. Usually, the parameter's abstract syntax is visualized and thus, it is kept in the corresponding symbol graphic. For illustrational reasons only, the attribute symbol *2:CN* in the RHS keeps additionally the parameter's abstract syntax which is indicated by the dashed arrow. With respect to the alphabet, the symbol *2:CN* is linked to *1:Class* by a link edge.

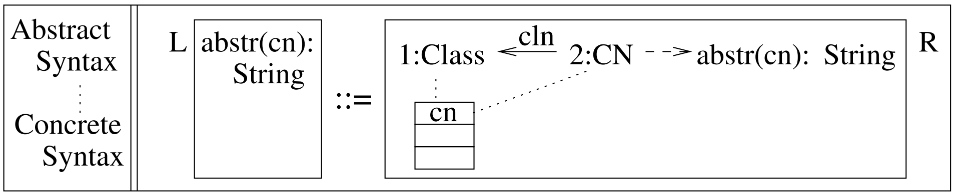


Figure 3.9: Abstract and concrete syntax of a production.

△

Remark 3.4.6 (*Attribute Symbols*)
Attribute symbols like *String* in Figure 3.8 are given by build-in datatypes, i.e. by implemented components. It depends on the corresponding implementation, which datatype operations are available and hence, which datatype productions can be determined visually. △

Production morphisms

Usually, the LHS and/or the RHS of a production is not empty. Whenever both sides contain symbols and links, it is necessary to map some symbols in the LHS to symbols in the RHS. This mapping is called *production morphism*. It is used to express which symbol should be preserved and which one should be deleted if the production is applied. A production morphism is visualized by numbers placed "beside" the symbols.

Example 3.4.7 (*Production Morphism*)
Figure 3.10 illustrates a production expressing the insertion of an association. Both classes have to be preserved if the production is applied to a given sentence. This is required by the production morphism between the symbols in

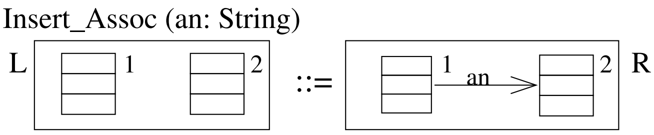


Figure 3.10: Visualization of a production morphism.

the production's LHS and the symbols in the RHS, indicated by equal numbers beside the symbols. △

Negative application conditions (NACs) and NAC morphisms

NACs are useful to express situations which have to be excluded if the production is applied. Note that the LHSs of productions are not sufficient to describe all desired situations but the NACs [HHT96]. It is possible to bound numerous NACs to one production. This bounding is similar to a production morphism. For each NAC there is a NAC morphism between symbols in the production's LHS and symbols in each NAC.

Example 3.4.8 (*NAC and NAC Morphism*)

Figure 3.11 illustrates a production for deleting a class symbol. The class symbol in the production's LHS is not mapped to a symbol in the RHS because there is none. Two NACs are bound to the production by NAC morphisms illustrated by the number *1* written beneath the symbol graphics. Both NACs require that the class symbol in the LHS is not linked to an association symbol, neither by a begin link (NAC-1) nor by a end link (NAC-2).

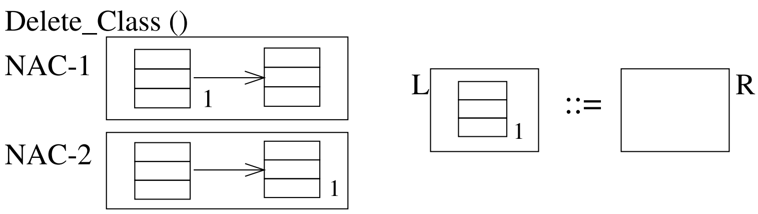


Figure 3.11: Visualization of a production with NACs and NAC morphisms. △

Production parameters and NACs

The combination of parameters and NACs offer a powerful mechanism in order to express certain situations by a production. For class diagram editing, e.g., it is possible to express that a class with a class name can be inserted only if it is not yet in the current sentence. The corresponding production is illustrated by Figure 3.12. Note that there is a NAC morphism with respect to the abstract syntax similar to the production morphism explained in Example 3.4.5.

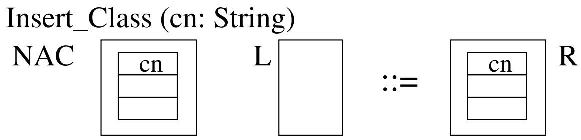


Figure 3.12: Production for the insertion of a class symbol.

These are all concepts with respect to grammar productions. Preconceiving the concepts, without doubt, the definition of productions is a design decision. This design decision depends as well on the designer's point of view and the definition of the alphabet. However, in the following sections we discuss the edit commands of a syntax-directed VL editor. We distinguish these edit commands with respect to language-generating productions and language-editing productions.

3.4.1 Language-Generating Productions

Language-generating production express the insertion of symbols and links. This means that usually the RHS contains more symbols the the LHS. In the previous section we already proposed two examples. One is given by the production called *Insert_Class* (cf. Figure 3.12), another one is given by the production called *Insert_Assoc* (cf. Figure 3.10). Yet, these productions do not allow to insert, e.g. an association class symbol.

Before we are able to show a suitable production expressing the insertion of an association class symbol we must introduce the corresponding part of an alphabet. This means, we have to determine the symbol and link types; they are shown in Figure 3.13. We modeled a symbol called AssocClass which is linked to a class symbol three times. Except of the AssocClass symbol and the link edge called assocClass we already discussed the remaining symbols and links (cf. Example 3.3.2). The AssocClass symbol and the links are determined in the same way.

The two class symbols occurring in the LHS of the production shown in Figure 3.14, are mapped by a production morphism to two class symbols in the

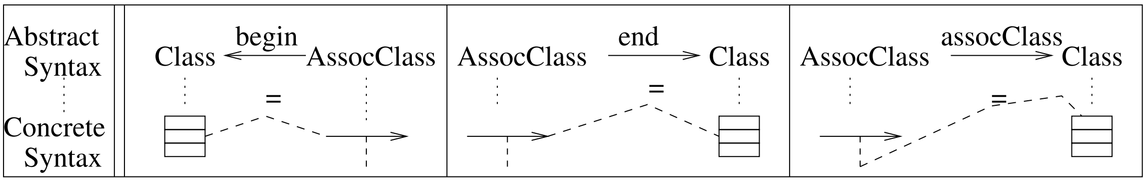


Figure 3.13: Alphabet part for an association class symbol.

RHS. The RHS comprises in addition two new generated symbols, namely, the association class symbol and a third class symbol which is linked to the association class symbol.

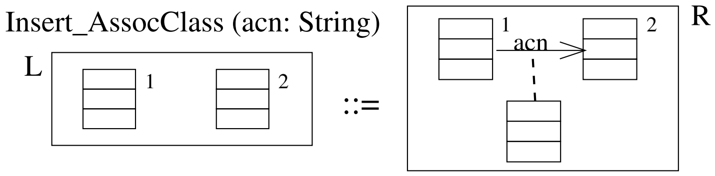


Figure 3.14: Visualization of the production for inserting an association class symbol.

In Figure 3.15, the abstract syntax level of the production Insert_AssocClass is shown. This level depicts the important constituents of the production. The production parameter is kept in the LHS and identically mapped by a production morphism to the RHS, similar to the class name parameter discussed in Example 3.4.5. The symbols 1:Class and 2:Class are mapped by the production morphism as well. The 3:Class symbol and the 4:AssocClass symbols are added in the RHS. The further 5:ACN symbol denotes an instance of the attribute symbol ACN defining the name of an association class symbol. We assume that in the class diagram alphabet the ACN symbol is linked to the AssocClass symbol.

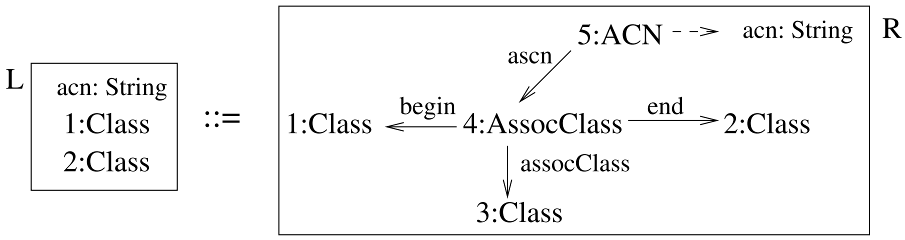


Figure 3.15: Abstract syntax of the production for inserting an association class symbol.

The production presented by Figure 3.15 makes clear that symbols are added but not changed. This is different to language-editing productions.

3.4.2 Language-Editing Productions

Language-editing productions concern the deletion and the exchange of symbols. They may as well express more comprehensive language-specific edit commands of a syntax-directed editor. A deletion production has been already proposed by Figure 3.11. A production allowing the exchange of symbols will be explained now. For example, during class diagram editing it should be possible to exchange a (binary) association symbol by an association class symbol. In Chapter 2.6 we required exactly such a production. Thereby, the association name should not be changed. The corresponding production is illustrated by Figure 3.16.

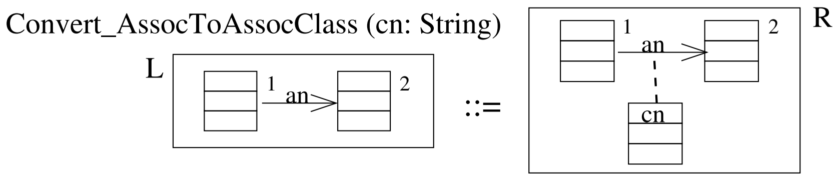


Figure 3.16: Visualization of the production for exchanging an association symbol by an association class symbol.

In contrast to Figure 3.14, this production requires not only the presence of two class symbols in its LHS but an association symbol and an association name (an:String) as well. This association symbol is exchanged by an association class symbol in the RHS of the production. The production parameter indicates the class name which is added in the RHS together with the corresponding class symbol. The variable an occurring in the LHS as well as the RHS expresses that the association name must be preserved. The abstract syntax of the production is presented by Figure 3.17. The dotted arrows depict the exchange of the corresponding symbols.

So far, we introduced all concepts concerning a VL grammar as well as language-generating and language-editing productions. Thereby, we implicitly mentioned the application of productions where we aimed at the reader's intuition with respect to the accomplishment. In the following section we explain

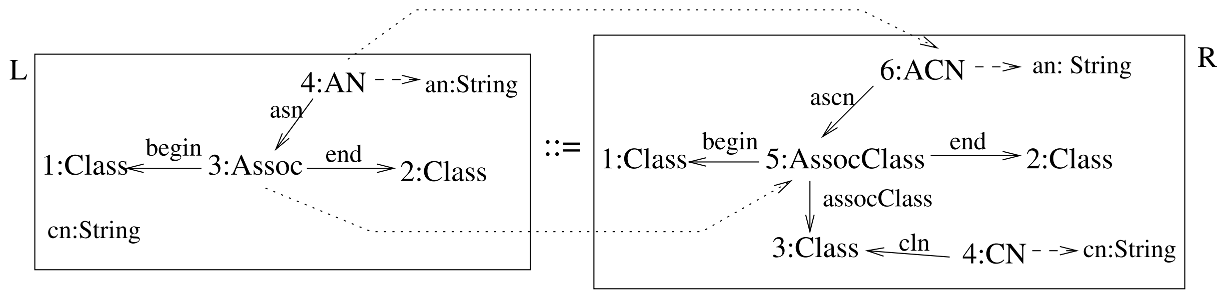


Figure 3.17: Abstract syntax of the production for exchanging an association symbol by an association class symbol.

explicitly how one given VL sentence can be derived from the application of grammar productions.

3.5 Manipulation of Visual Sentences

Visual sentences can be edited by applying productions of the VL grammar. A visual language sentence (VL sentence) is derived in a finite sequence of derivation steps by applying grammar productions Then, the visual language is defined by the set of all VL sentences. In this section we consider the concepts of applying grammar productions to a given sentence. Note that a VL sentence is a visual sentence with respect to Concept 3.4.1 but derived from grammar productions.

The application of productions to a given sentence is carried out if all symbols in the production's LHS are mapped by a match morphism to symbols in the sentence which is to be transformed. The sentence is transformed under consideration of both, the production morphism and the match morphism. This means that by a derivation step some symbols may be preserved, some others may be inserted and further others may be deleted. The links occurring in the production sides are transformed implicitly.

Concept 3.5.1 (*Match Morphism*)
Given a production and the sentence the production is applied to. A *match morphism* is determined by a mapping: all symbols occurring in the LHS of the production are mapped to symbols occurring in the sentence. Only those symbols can be matched having the same symbol type. △

A match morphism together with a production morphism specifies how the sentence can be derived from the production. The symbols which are mapped by the match morphism determine a subsentence which is embedded in a certain context, namely, that of the visual sentence where the production is applied to. Due to the production and match morphisms, the symbols and links occurring in the RHS of the production are exchanged in this context. Note that a sentence can only be derived from the production if the NACs are fulfilled, provided that the production comprises some NACs. Before we establish the corresponding concepts we will illustrate the application of productions by an example:

Example 3.5.2 (*Production Application*)
Figure 3.18 illustrates the application of a production shown in part (a). This production allows us to insert an association symbol. By the production morphism (illustrated by the equal numbers in the LHS and RHS of the production) the class symbols in the LHS are mapped to the class symbols in the production's RHS.

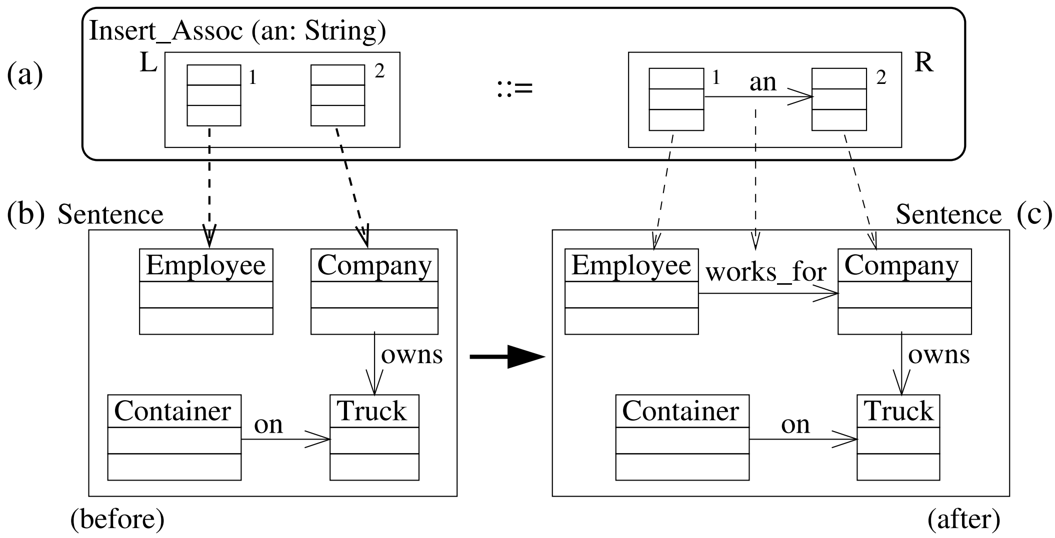


Figure 3.18: Application of a production.

Given the sentence depicted by (b) in Figure 3.18. By the match morphism (illustrated by dashed arrows) the symbols in the production's LHS are mapped to the symbols in the sentence (b). The production morphism preserves these class symbols and inserts the association symbol in between. A name (e.g. works_for) for the variable an indicating an association name (of type String) has to be determined. The derivation results in the sentence (c). The link is set implicitly, however, with respect to the corresponding alphabet (cf. Example 3.3.2). The link edge is introduced as well as the link constraints are solved for the resulting visual sentence.

During the derivation, the match morphism as well as the production morphism are uniquely continued. The continuation of the match morphism is illustrated by the dashed arrows running from RHS to the (c) part. It maps all symbols in the production's RHS to that of the resulting sentence under consideration of the match morphism. The continuation of the production morphism is not illustrated, however, each symbol occurring in (b) are uniquely mapped to the symbols in (c). △

The reader may have recognized that the symbol graphics of the production may differ with respect to positions and sizes to that of the sentence the production is applied to. This is due to our interpretation concepts. We interprete the symbol graphics of a production as placeholder for a specific symbol of a certain type. E.g., the two class symbols in the production's LHS are typed over Class. Although both class symbols have a certain layout with respect to the constraint-satisfaction problem of the alphabet (cf. Example 3.4.2) the layout is not important, neither for the match morphism nor for the production morphism. Although productions, sentences and morphisms are represented on the concrete syntax level – from the user's point of view – the derivation of a sentence is obtained at the abstract syntax level. This means that the concrete syntax of symbols and links is not relevant as the concrete syntax is completely neglected by the morphisms.

Concept 3.5.3 (*Production Application on the Abstract Syntax Level*)
The application of a production p to a given sentence *s-before* yields a resulting sentence *s-after*, provided that *s-after* can be obtained from *s-before* as result of the following basic steps:

1. **Choose** an occurrence for all symbols available in the production's LHS and **check** the application conditions. If they are fulfilled, the occurrence of the LHS is called a *match morphism*.
2. **Remove** the occurrence of the LHS (except for those which are mapped by the production morphism to symbols in RHS) from the sentence as

well as all *dangling link edges*, i.e. all edges which have lost their source and/or their target symbol.

3. **Glue** the symbols according to the morphisms, that is, construct the disjoint union of symbols. This yields an intermediate graph *s-inter*.
4. **Check** whether for each link edge type a corresponding source symbol and link edge exists. **Remove** the source symbols in *s-inter* and the resulting dangling edges if at least one link edge is not available.
5. **Continue** the production and match morphism, i.e., identify the occurrences of the symbols in *s-inter* with the occurrences of the symbols in RHS and with the occurrences of the symbols in *s-before*. This yields the sentence *s-after*.

△

The concept above is an informal description of the single-pushout approach underlying our framework [EHK+97] [1]. This kind of informal description is adapted from [Tae96] (using the double-pushout approach for the transformation of graphs). Let us explain the concept by an example:

Example 3.5.4 (*Production Application on the Abstract Syntax Level*)
Given the part of the alphabet explained in Example 3.3.2. Two links are given between the Assoc and the Class symbol, namely, begin, end: Assoc $\rightarrow$ Class. Assume further the production *DeleteClass* as presented by Figure 3.11, however, without the negative application conditions. Figure 3.19 (a) illustrates the abstract syntax level of this production.

The production is applied to the sentence *s-before* shown in (b). This sentence comprises two class symbols and an association symbol. The production morphism is given by an empty mapping. We choose an occurrence for the 1:Class symbol according to Step 1. No NAC must be checked such that this occurrence is already a match morphism. Due to Step 2, the 1:Class symbol and the begin link edge are deleted. Thereafter, Step 3 is executed, i.e., the symbols are glued with respect to the morphisms. In this case no symbols are glued. Nevertheless, Step 3 yields the intermediate sentence *s-inter* shown in (c).

[1] The formal calculus is reviewed in Chapter 4.1.

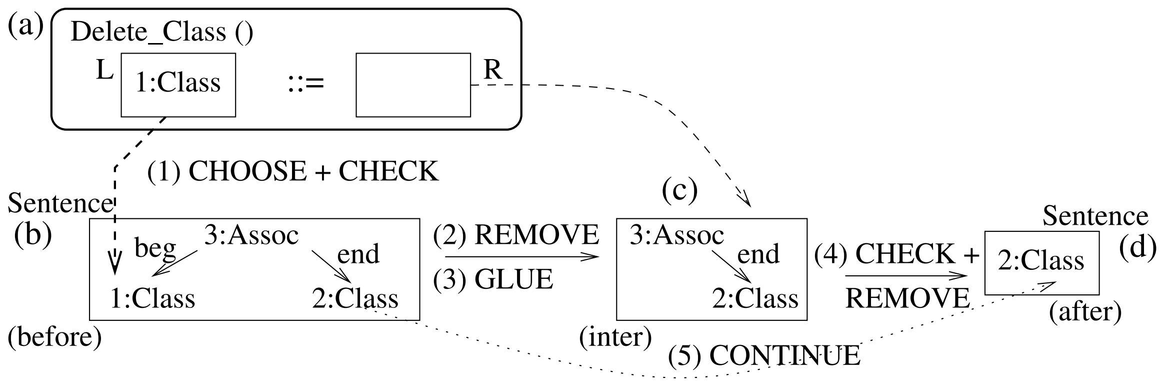


Figure 3.19: Abstract syntax: application of a production.

According to Step 4 it is checked whether source symbols exist for each link edge type. This is the case because the begin link does not exist any more although an Assoc symbol is available. This causes the removing of the Assoc symbol and the end link edge. The morphisms are continued according to Step 5. Note that the production morphism is an empty mapping so as the continuation. All these steps yield the derived sentence *s-after* presented in (d). The abstract syntax of the sentence derived from the production is equipped with its corresponding layout as explained in Example 3.4.2. △

Applying a deletion production effects the deletion of symbols which are linked to other symbols. This may lead to unexpected results as discussed in the example above. A similar conflict occurs as shown in Figure 3.20. Two class symbols are in the production's LHS but only one is mapped to the class symbol in the RHS by the production morphism. By the match morphism all symbols of the production's LHS must be mapped to the given sentence. Thus, mapping the symbols in the way as illustrated in Figure 3.20 effects the deletion of the class symbol.

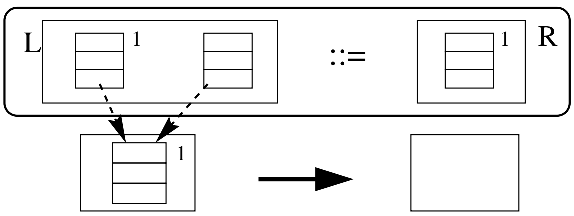


Figure 3.20: Deletion with conflict.

The unexpected results emerging from morphisms is one of the reasons for introducing negative application conditions (NACs). The advantage of NACs is, that they are more expressive than the LHSs of productions, but they model explicitly all edit commands of a VL editor as well. The user of a VL editor has not to follow the try&error mechanism as necessary in editors allowing free-hand editing. The user of our VL editor gets an explicit guidance of how to edit a sentence using the productions. Furthermore, the VL editor gets an operational semantics with respect to language-specific tasks. These and other features concern the GENGED environment proposed in the following section.

3.6 The GENGED Environment

The GENGED environment is a visual environment supporting the visual definition of visual languages satisfying Requirement 2.5.1. All the concepts that have been introduced so far have to be implemented by the GENGED environment. Hence, the GENGED environment comprises a specification tool and a VL editor as well.

The GENGED specification tool supports the editing of VL specifications. The VL editor obtains a specific VL specification and assists the editing of visual sentences. The VL editor does not directly depend on the environment. It can be also used as stand-alone system. The relations between editing, the GENGED components and resulting structures are shown in Figure 3.21.

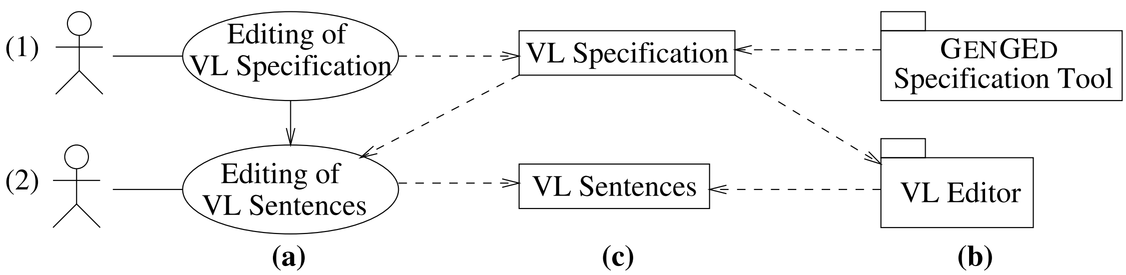


Figure 3.21: The GENGED framework.

Figure 3.21 (a) shows an UML activity diagram with two kind of actors, namely, a language designer (1) and a common user (2). The component diagram in Figure 3.21 (b) depicts the GENGED components. VL specifications and VL sentences are data structures achieved. They are illustrated in part (c). The dashed arrows indicate the data flow.

Due to the constituents of VL specifications, the GENGED specification tool consists of an alphabet editor and a grammar editor. The user-defined alphabet establishes a type system for instances as they occur in visual sentences. Visual sentences are constructed from algorithms generating alphabet productions. These alphabet productions form the language-specific edit commands of the grammar editor. All components of the GENGED environment have to be implemented in Java [2] in order to achieve platform-independence.

The Alphabet Editor supports the definition of symbol and link types according to Concept 3.3.1. The symbol graphics of lexical symbols can be drawn in a way similar to common graphic editors. In contrast, the alphabet editor supports the definition of graphical constraints. Such constraints may be used to define specific properties of one graphic like a fix size. Constraints must be used to couple several primitive graphical objects like rectangles, lines, etc. if they are used for one symbol graphic. This ensures the uniform layout of instances. Several access operations on graphics are implemented by the alphabet editor. These access operations have been proposed in Concept 3.1.2. They concern attachments that form the basis for constraint definition. In addition to the drawing of symbol graphics, the alphabet editor supports the treatment of bitmap graphics that can be read from a file. As proposed in Section 3.1 the properties of bitmap graphics resemble the properties of rectangles except for the invisibility property.

Attribute symbols like strings, integers, lists, etc., are built-in datatypes. They are implemented in Java and offer suitable methods for their access. Each datatype has a visual counterpart on the user interface. The graphical counterpart can be chosen by the user. Predefined graphical properties like text color can be changed. Note the we denote bitmap graphics, datatype graphics, and drawn graphics in a uniform way as symbol graphics with respect to graphical properties. Once the user has created a certain symbol graphic, he or she is asked to define a unique symbol name according to Concept 3.2.1. The naming of symbols is supported by a corresponding dialog and a small text editor. The resulting symbol graphic effects a box expression according to Concept 3.2.3 and a constraint-satisfaction problem as well. The defined symbols are listed at

[2]"Java" is a trademark of Sun Microsystems, Inc.

least by their symbol names in a specific palette such that they can be watched and chosen by the user for further evolutions.

Links can be defined between two different symbols. According to Concept 3.2.5 we distinguish between connection links and attribution links. During link definition it is checked whether the requirements are met. These requirements concern the attribution links that must not be defined between two attribute symbols. The definition of a link edge between the two involved symbol names is supported by a corresponding dialog. This dialog keeps all symbol names that can be chosen by the user and declared to be a source or a target symbol. Furthermore, the user is asked about a link name. A link is completed by graphical constraints between the corresponding symbol graphics. These constraints establish the spatial relations. Suitable attachments have to be defined by the user according to Concept 3.1.2. Once the user has created a link between two symbols, the link is listed at least by its name in a specific palette such that it can be watched and chosen be the user for further evolutions.

The evolutions we mentioned concern the symbols and the links as well. The alphabet editor allows to change already created symbol graphics and link constraints. Changing symbol-specific and/or link-specific graphical properties results in an update of the corresponding symbol and/or link. The user is informed about accruing inconsistencies.

The user-defined alphabet is stored in a file. The sequently toggled grammar editor reads this file and generates its own language-specific edit commands. These commands are called *alphabet productions*, explained in the following.

Alphabet productions are generated from the user-defined alphabet. These productions express either the insertion of specific symbol(s) (cf. Section 3.4.1) or the deletion of symbol(s) (cf. Section 3.4.2). Similar to grammar productions (cf. Concept 3.4.3), every alphabet production consists of a LHS, a RHS and optionally NACs. The symbols in the LHS are mapped to symbols in the RHS (respectively NACs) by a production morphism (respectively NAC morphism).

The LHS, the RHS and each NAC of an alphabet production are visual sentences according to Concept 3.4.1. They contain instances of symbols and links defined in the alphabet. Hence, the dependency of symbols must be regarded for the construction of a visual sentence. This effects NACs for such alphabet productions that allow the deletion of symbol(s).

The generation of alphabet productions is based on the dependency relation of symbols (cf. Concept 3.4.1). This dependency relation concerns the abstract syntax level. Each visual sentence of an alphabet production is visualized according to its concrete syntax. The corresponding constraint-satisfaction problem has to be satisfied. The dependency relation concerns not only lexical but attribute symbols as well. These attribute symbols are represented by the user-defined variable (using the alphabet editor) which functions as a placeholder in an alphabet production.

The Grammar Editor is a generic editor that takes an alphabet as input or, more precisely, the user is able to choose a specific file. Once read the file, the process of generating alphabet productions is triggered. The alphabet productions are then the language-specific edit commands of the grammar editor. These commands together with language-unspecific commands allow for creating a VL grammar as proposed in Concept 3.4.3. The generated alphabet productions are listed by their names in a palette such that the user can chose them for application. The application of alphabet production concerns the manipulation of visual sentences similar to Concept 3.5.3. The difference is given by production parameters that are not available in alphabet productions.

Some generated alphabet productions may express already intended grammar productions. We are aware that in general the grammar productions have to be much more expressive. However, if the user considers an alphabet production as a suitable grammar production, the alphabet production can be declared as grammar production. It is then listed in the palette indicating the grammar productions.

The alphabet productions support the editing of visual sentences occurring in grammar productions (LHS, RHS, NACs) and the start sentence as well. The editing is supported in a syntax-directed way, namely, by the select&apply mechanism. According to Concept 3.5.1, symbols of the alphabet production's LHS must be matched to symbols in one visual sentence of the grammar production. Initially, such a visual sentence is empty. Thus, the match morphism is empty. Symbols and links can be inserted by applying alphabet productions. Additionally, they can be deleted by applying a suitable deletion production.

The select&apply mechanism represents one kind of the grammar editor's language-unspecific commands. Further language-unspecific commands concern the definition of production morphisms, NAC morphisms and production headers. We regard these commands as language-unspecific because they are defined for every grammar. The morphisms can be set and deleted component-wise, i.e., for each symbol. The definition of a production name and parameters is supported by a text editor. The production parameters have to be consistent with a user-defined variable indicating the corresponding attribute symbol.

Because of the generated alphabet productions used as edit commands, the grammar editor offers a rule-based style of programming following the *transformation-based* approach mentioned in Chapter 1 and in Requirement 2.5.2. However, the definition of the grammar (and the alphabet) results in a VL specification which is the input of the intended VL editor.

The intended VL Editor supports the manipulation of visual sentences with respect to a specific VL specification. It is, like the grammar editor, a generic editor. This means that the parameter of the editor is a VL specification offering the language-specific edit commands. The language-unspecific commands concern the select&apply mechanism underlying the transformation-based approach of editing as proposed in Section 3.5.

In general, the VL editor is similar to the grammar editor. In contrast, the VL editor has only one drawing area for applying grammar productions starting at the start sentence of the grammar. The application of grammar productions is supported according to Concept 3.5.3. If a production with parameters is applied then a corresponding text editor allows the definition of a value with respect to the underlying datatype.

Using object-oriented techniques as supported by Java, the grammar editor and the VL editor are derived from an abstract class that defines the common behavior and user interface of the editors. Thus, the VL editor as proposed so far, is the smallest build. However, the major aim of this thesis is the visual definition of visual languages. This aim is achieved as the reader may have recognized. In the following two Chapters the concepts proposed in this section are refined. Chapter 4 shows the formal aspects of VL specifications and the derivation of sentences as well. Chapter 5 introduces the available GENGED environment.

Chapter 4

Formal Aspects of Visual Languages

This chapter introduces the formal aspects of visual languages given by algebraic graph transformation and graphical constraint solving. In general, the formalism of algebraic graph transformation combines statical descriptions of graph-like structures with a dynamic component. Thus, statical and dynamical information can be described in one joint calculus. The special power of algebraic graph transformation is given by its sound mathematical foundation [Ehr79, Löw93, EHK$^+$97]. With algebraic and categorical methods valid statements over graph transformation can be made as e.g. rule-based graph rewriting, termination (for non-deterministic graph rewriting), consistency checks of instance classes, construction of dynamical application conditions from static consistency conditions, and invertibility, parallel and sequential composition of productions.

Algebraic graph transformation has been extended by attribution concepts in [LKW93, HMTW95, CL95]. Attribution concepts allow to enrich graph-like structures by datatype informations. We show in this chapter that algebraic graph transformation with attribution concepts offers a sound theoretical background for the complete and precise specification of visual languages.

A visual language specification (short *VL specification*) consists of a visual alphabet and a visual grammar:

- A visual alphabet is represented by an algebraic signature (an attributed graph structure signature AGS) and a constraint-satisfaction problem (CSP). Graphical symbols are described by sort symbols which have a certain layout. The layout of a symbol is determined by its attribute and some graphical operations. Graphical links are described by operation symbols together with graphical constraints formulating the CSP. Symbol and link constraints are defined on the symbols' attributes denoting the layout.
- A visual grammar consists of a start sentence and productions. The start sentence as well as each constituent of a production, namely, the left-hand side (LHS), the right-hand side (RHS) and negative application conditions (NACs) are algebras with respect to the visual alphabet of a specific visual language. Each algebra has to satisfy the corresponding CSP, instantiated from the CSP of the alphabet.
- A visual language (VL) then is the set of all visual sentences which can be derived by applying the grammar productions starting at the start sentence S. We denote the visual sentences which are derived by applying grammar productions as *VL sentences*.

This chapter is organized as follows: In Section 4.1 we give a brief survey about the techniques used. The visual language concepts as proposed in Chapter 3 concern the components of a VL specification, namely, an alphabet and a grammar. The formalization of these concepts and the application of grammar productions are the topics of Section 4.2.

4.1 Basic Notions

Like the approaches [Koc99, Wag97, CL95, LKW93] our approach is algebra-driven, i.e. attributed graph structures are introduced as algebras of *attributed graph structure signatures*. Therefore, before starting with attributed graph structures, we first review the basic notions of universal algebra in Section 4.1.1. Attributed graph structures and their transformation are then reviewed in Section 4.1.2. In this section we do not only introduce the basic notions but we show additionally why we have to extend them for the specification of visual languages.

In the focus of this thesis is the graph transformation based treatment of visual languages (VLs) including their *abstract syntax* (the logical part) as well as their *concrete syntax* (the layout part). The latter one is determined by graphics and graphical constraints yielding constraint-satisfaction problems proposed in Section 4.1.3.

4.1.1 Universal Algebra

In this section we review some basic notions from [EM85] used in this chapter.

Definition 4.1.1 (*Signature, Algebra, Homomorphism*)
A *signature* SIG $= (S, OP)$ consists of a set of sort (symbols) S and a family of operation (symbols) $OP = (OP_{w,s})_{w \in S^*, s \in S}$.

A *partial* SIG-*algebra* $A = (A_S, op^A)$ is a S-indexed family $(A_s)_{s \in S}$ of carrier sets together with an OP-indexed family of partial mappings $(op^A)_{op \in OP}$ such that $op^A : A_{s_1} \times ... \times A_{s_n} \multimap A_s$ if $op \in OP_{s1 \ldots s_n, s}$. We denote by $dom(op^A)$ the domain of the partial mapping op^A.
A SIG-algebra A is *total* if all mappings $(op^A)_{op \in OP}$ are total.

A partial homomorphism $h : A \multimap B$, briefly SIG-*morphism*, between two SIG-algebras A and B is a sort-indexed family of partial mappings $h = (h_s : A_s \multimap B_s)_{s \in S}$ such that $op^B \circ h(x) = h \circ op^A(x)$ for all $x \in dom(op^A)$.
A SIG-morphism is *injective* (*surjective, bijective*) if the mappings h_s for all $s \in S$ are injective (surjective, bijective). △

Example 4.1.2 (*Algebra of a Signature*)
Given Σ_{Nat}, the signature for natural numbers. The syntactic term algebra T_{Nat} is a total algebra with respect to Σ_{Nat}:

Σ_{Nat} =

sorts	N	$T_{Nat} = \{s^n(z) \mid n \in \mathbb{N}\}$
opns	z: $\rightarrow$ N;	$z^{T_N} = z$
	s: N $\rightarrow$ N;	$s^{T_N}(s^n(z)) = s^{n+1}(z), (n \in \mathbb{N})$.

A partial algebra is given by, e.g. $(\mathbb{N}, 0, +1)$ with $+1 : \mathbb{N} \rightarrow \mathbb{N}; +1(n) = n+1$, for $n > 3, n \in \mathbb{N}$. It is partial because the successor operation $+1$ is not defined for all natural numbers. $\triangle$

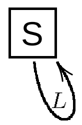
All partial algebras and all homomorphisms of one fixed signature define the category ***Alg(*SIG*)*** where objects are partial SIG-algebras and morphisms are partial SIG-morphisms. If we restrict objects to total algebras we get the subcategory ***TAlg(*SIG*)***.

Remark 4.1.3 (*Graph Structure Signatures*)
Whenever signatures comprise unary operations only, they can be represented as graphs so as the corresponding algebras. The sort symbols (respectively elements of the carrier sets) are depicted as nodes and the operation symbols (respectively operations) as edges.

Let us illustrate this assertion by an example. Both signatures in the figure below contain unary operations. The signature Σ in the left of the figure below contains unary operations only. Therefore, it can be represented as a graph as described above. The signature $Graph_\Sigma$ given in the right of the figure below specifies common graphs. Also this signature contains unary operations only and hence, it can be represented as a graph:

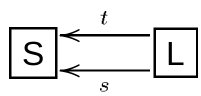
Σ =

sorts	S
opns	L: S $\to$ S;



$Graph_\Sigma$ =

sorts	S, L
opns	s, t: L $\to$ S;



Both examples are relevant in our approach. Σ is used as meta model for the specification of visual languages consisting of symbols and links between symbols: S denotes the symbols and $L : S \to S$ denotes the links between symbols. In the implementation (cf. Chapter 5), this representation is mapped onto the graph representation illustrated by $Graph_\Sigma$. This kind of representation is the used by the AGG system executing the transformations of graphs. $\triangle$

4.1.2 Attributed Graph Structure Transformation

An *attributed graph structure* is a partial algebra to an *attributed graph structure signature*. An attributed graph structure signature is separated into two parts. One of them describes the graph structure part, the other one the attribute part. Attribution operations connect the elements of the graph structure part with elements of the attribute part. The attribute part usually is given by arbitrary datatype signatures, like for boolean, natural numbers, etc. These signatures may have non-unary operations.

Definition 4.1.4 (*Attributed Graph Structure Signature* AGS)
An *attributed graph structure signature* AGS = $GS + AS + AOP$ consists of

- a *graph structure signature* $GS = (S_{GS}, OP_{GS})$, where OP_{GS} contains unary operations only,
- a signature $AS = (S_{AS}, OP_{AS})$, called *attribute signature*,
- an S-indexed family $AOP = (AOP_s)_{s \in S}$ of attribution operations, where $S \subseteq S_{GS}$.

△

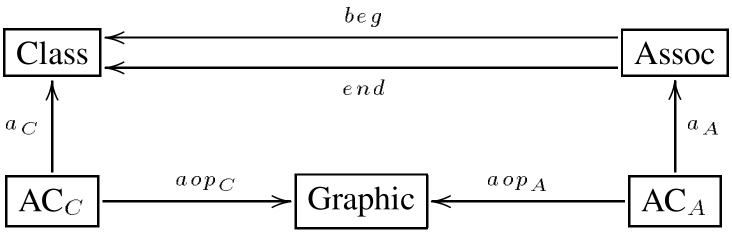
An attributed graph structure is a partial algebra of an attributed graph structure signature AGS, where the GS-algebra is total and the AS-algebra may be partial. The mappings for the attribution operations given in AOP have to be total as well, since we require for each symbol a certain layout (attribute). However, as we will show, the layout is usually changed by productions, i.e. this is an update of attributes. In [CL95] attribute carriers are introduced where attributes are updated by removing the attribute carrier holding the value and adding a new carrier pointing to the new value. Therefore, we have attribute carrier sorts (AC) and connection operations (a) in a graph structure signature which are not directly relevant for modeling a visual language.

Example 4.1.5 (*Attributed Graph Structure Signature* AGS)
Class diagrams as introduced in Chapter 3 can be specified by an attributed graph structure signature. In this example we only regard classes and associations between classes defining the GS part of the signature. Both symbols are attributed by a corresponding graphic.

classdiagram =

GS	**sorts**	Class, Assoc, AC_C, AC_A
	opns	beg: Assoc $\to$ Class; end: Assoc $\to$ Class;
		a_C: AC_C $\to$ Class; a_A: AC_A $\to$ Assoc;
AS	**sorts**	Graphic
AOP	**opns**	aop_C: AC_C $\to$ Graphic; aop_A: AC_A $\to$ Graphic;

The corresponding attributed graph structure signature can be represented as a graph according to Remark 4.1.3. We add the attribute sort and the attribution operations:

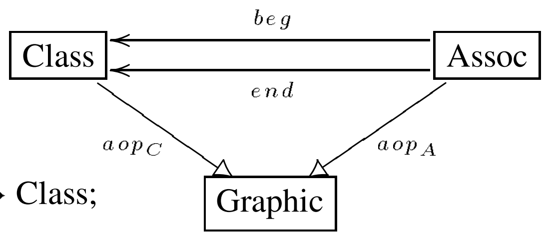


△

Convention 4.1.6 (*Attribute Carrier*)
For illustrational reasons we omit the attribute carriers in the following examples. Instead we use arrows with a triangle in the arrow-head instead. The graph structure signature of Example 4.1.5 and its graph representation then looks as follows:

classdiagram =

GS	**sorts**	Class, Assoc
	opns	beg, end: Assoc $\to$ Class;
AS	**sorts**	Graphic
AOP	**opns**	aop_C: Class $\to$ Graphic;
		aop_A: Assoc $\to$ Graphic;



△

For attributed graph structures as defined now, we have total operations and partial homomorphisms for the graph structure part but partial operations and total homomorphisms for the attribute part.

Definition 4.1.7 (AGS-*Algebra and* AGS-*Morphism*)
An attributed graph structure with respect to an attributed graph signature AGS = $GS + AS + AOP$ is a tuple $A = (A^{GS}, A^{AS}, AOP^A)$, where $A^{GS} = (A^{GS}_S, A^{GS}_{OP})$ is a total GS-algebra, $A^{AS} = (A^{AS}_S, A^{AS}_{OP})$ is a partial AS-algebra, called *attribute algebra*, and AOP^A is a family of total *attribution operations* $aop^A : A^{GS}_s \to A^{AS}_{s'}$ for all $aop : s \to s' \in AOP$.

Given two attributed graph structures A and B with respect to an attributed graph structure signature AGS. An *attributed graph structure morphism* $h : A \mathrel{\circ\!\!\rightarrow} B$, briefly AGS-morphism, is a pair of partial and total homomorphisms $h = (h^{GS}, h^{AS})$ which preserve attributions AOP, that is $aop^B \circ h^{GS} = h^{AS} \circ aop^A$ for all $aop : s \to s' \in AOP$. We denote by $h^{GS} : A^{GS} \mathrel{\circ\!\!\rightarrow} B^{GS}$ the partial homomorphism between the graph structure algebras with respect to GS. If h^{GS} is total, then h is called total homomorphism.

$$\begin{tikzcd} A^{GS}_s \arrow[r, "h^{GS}_s"] \arrow[d, "aop^A_s"'] \arrow[dr, phantom, "="] & B^{GS}_{s'} \arrow[d, "aop^B_{s'}"] \\ A^{AS}_s \arrow[r, "h^{AS}_s"'] & B^{AS}_{s'} \end{tikzcd}$$

Attributed graph structures with respect to AGS and AGS-morphisms define the category AGS. △

Let us illustrate attributed graph structures and morphisms by an example:

Example 4.1.8 (AGS-*Algebra and* AGS-*Morphism*)
Figure 4.1 illustrates two class diagrams, the left one is called A, the right one B. Both are attributed graph structures, i.e. algebras with respect to the signature presented in Example 4.1.5. We do not present the whole attribute algebra (AS-algebra) but the attribution (labeling) operations of the graph structure items which are illustrated by leading numbers and corresponding sorts of the signature.

The morphism between both algebras A and B is defined component-wise according to the GS and the AS part. It is illustrated by the leading numbers of the graph structure items, i.e. *1:Class* (*2:Class*) in A is mapped onto *1:Class* (*2:Class*) in B. △

Productions describe how a given attributed graph structure shall be transformed into another one. A production is applicable if there is an occurrence of

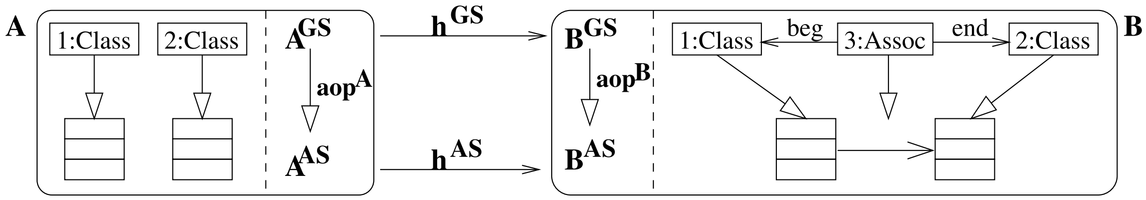


Figure 4.1: Morphism between two class diagrams.

the left-hand side (LHS) into a given attributed graph structure G, i.e. we have to find the LHS in G:

Definition 4.1.9 (*Production, Occurrence*)
A production p is a partial AGS-morphism $p : L \multimap\!\!\to R$ from an attributed graph structure L, called the left-hand side (LHS) or precondition, to an attributed graph structure R, called the right-hand side (RHS) or postcondition, where p is required to be the identity on the attribute part.

An occurrence m, also called *match*, of a production p into a given attributed graph structure G is a total AGS-morphism from L into G, i.e. $m : L \to G$. $\triangle$

The intuition for this definition is as follows: The items which are not mapped from the LHS to the RHS are deleted by the application of the production. The items in the RHS which are not in the image of the homomorphism are added. The items in the domain of the homomorphism are the objects which must be present for the production to be applicable and preserved by the application. We require that the attribute part is mapped identically by the production. This is due to the fact that we consider an attribute algebra which holds all graphics for a specific visual language. Changing a graphical attribute means to delete the corresponding attribute carrier (node) and to add a new one holding the new graphic.

The existence of an occurrence specifies that a given attributed graph structure contains a subgraph that satisfies all preconditions of the production. Hence, occurrences are required to be total. Note that the attribute part in a production is allowed to be different from the attribute part of the occurrence since we allow arbitrary homomorphisms here. This gives us the freedom to use syntactic algebras, for example term algebras (cf. Example 4.1.2) on the production level, and concrete algebras on the level of graph structures. Occurrences can be used to specify substitutions on the attribute part, whose evaluation results in concrete calculations in the transformation process.

Following the ideas of [CL95] and [LKW93], the application of productions at occurrences is defined by the pushout of the two involved AGS-morphisms. The construction of the pushout can be done component-wise for the graph structure and the attribute part by the following construction:

Construction 4.1.10 (*Production Application*)
The application of a production $p : L \multimap R$ at an occurrence $m : L \to G$ in an attributed graph structure G, denoted by $G \overset{p,m}{\Longrightarrow} H$, is given by a pushout in the category ***Alg(*** AGS***)***. It is defined as follows and illustrated by Figure 4.2:

1. Construct the pushout of $p^{GS} : L^{GS} \to R^{GS}$ and $m^{GS} : L^{GS} \to G^{GS}$ in ***Alg(GS)***, the category of GS-algebras and partial homomorphisms [Löw93].
2. Since $L^{AS} = R^{AS}$ also $H^{AS} = G^{AS}$, and $L^{AS} \overset{=}{\longrightarrow} R^{AS} \overset{m^{AS}}{\longrightarrow} H^{AS} = L^{AS} \overset{m^{AS}}{\longrightarrow} G^{AS} \overset{=}{\longrightarrow} H^{AS}$ is a pushout in ***Alg(AS)***, the category of partial AS-algebras and total homomorphism [Bur96].
3. Add the attribute assignments (aop^H) as unique extensions of the resulting diagrams [LKW93].

$$
\begin{tikzcd}
& L_{s'}^{AS} \arrow[rr, "id"] \arrow[dd, "m_{s'}^{AS}"] & & R_{s'}^{AS} \arrow[dd, "m_{s'}^{AS}"] \\
L_s^{GS} \arrow[ur, "aop^L"] \arrow[rr, "p_s^{GS}"] \arrow[dd, "m_s^{GS}"] & & R_s^{GS} \arrow[ur, "aop^R"] \arrow[dd, "m_s^{GS\star}"] & \\
& G_{s'}^{AS} \arrow[rr, "id"] & & H_{s'}^{AS} \\
G_s^{GS} \arrow[ur, "aop^G"] \arrow[rr, "p_s^{GS\star}"] & & H_s^{GS} \arrow[ur, "aop^H"] &
\end{tikzcd}
$$

Figure 4.2: Pushout construction.

△

In [Löw93] the construction of a pushout in ***Alg(GS)*** is formally defined. We introduce now informally the concepts of the pushout constructions following the equivalent but easier construction of [Kor96].

Remark 4.1.11 (*Pushout Construction*)
The necessary steps for constructing a pushout are the following:

1. The construction of the pushout of $p^{GS} : L^{GS} \to R^{GS}$ and $m^{GS} : L^{GS} \to G^{GS}$ in ***Alg(GS)*** is done by
   (a) Construct the pushout component-wise in Set^P, i.e. for the carrier sets of the involved algebras. Although the commutativity of the homomorphism is ensured for the carrier sets, the result H is not always a total algebra. The transformation of H into a total algebra requires a further step:
   (b) Delete the items of carrier sets for which the operations of the algebra H are not defined, such that the result is a total algebra.
2. The attribute algebra is not changed by a production as we consider one fixed attribute algebra. Only the attribution operations (AOP) hold the values of the items available in the graph structure part. After transforming the graph structure part the attributes of the remaining items are "calculated". This means, that the old attribute carriers are deleted and a new carrier is inserted pointing to the new value.

△

Let us illustrate productions, occurrences and the application of a production, i.e. a *direct derivation*, by an example.

Example 4.1.12 (*Production, Occurrence, Production Application*)
Figure 4.3 illustrates one production p in the upper part. It expresses the insertion of an association. The homomorphism is indicated by equally named graph structure objects. The production is applied to a given attributed graph structure G. The occurrence m from p to G is also indicated by same names. The transformation is done according to Construction 4.1.10. All items of the production's LHS are identically mapped onto items of the RHS. Nothing is deleted by the application of this production. The item denoted by *3:Assoc* is not in the image of the production morphism. The items in the domain of the homomorphism, denoted by *1:Class* and *2:Class* inclusive their attributes, are not touched during the transformation process but the new item of the RHS is added.

According to Convention 4.1.6 not the whole attribute algebras are presented for each attributed graph structure but only the attribution operations. The attributes of the production sides are given by variables of sort *Graphic*, whereas the attributes of G and H are given by concrete values (concrete graphics). The

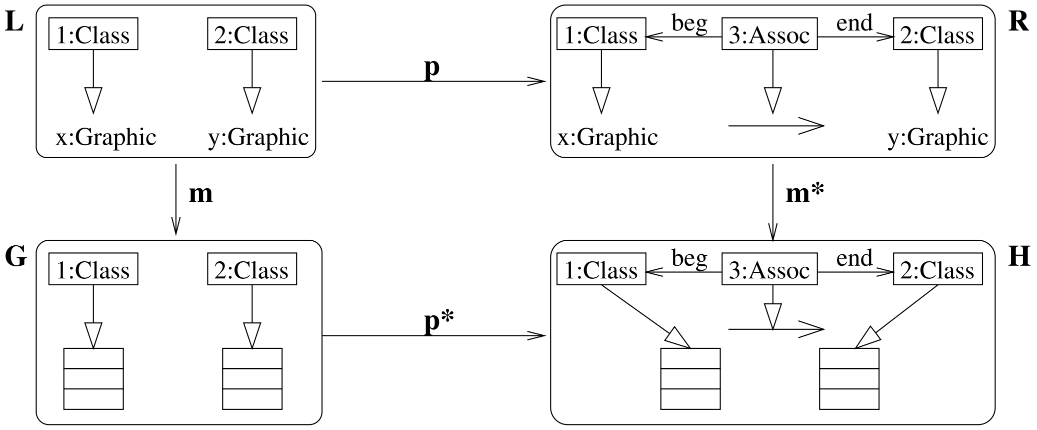


Figure 4.3: Example production and its application.

production allows to match classes without regarding their concrete layout. Because the variables x, y occur in both production sides, the matched attributes are preserved by the transformation. The variables occurring in the production's LHS are instantiated by the match with the concrete graphics of G. These graphics are not exchanged by the production but the association of the production's RHS is inserted in H. Neither the attributes of the classes are changed by the production nor the association arrow is placed as desired; it is placed at an arbitrary position. $\triangle$

The example above illustrates that more features are necessary in order to model visual languages with respect to their logical part (the *abstract syntax*) and their layout (the *concrete syntax*).

Conflicts may occur if productions express deletion. In order to illustrate such conflicts we review the deletion with conflicts from [EHK$^+$97] for simple graphs without attributes. Figure 4.4 depicts a situation where the production p deletes item 2 and preserves item 1 but by the match m items 1 and 2 are mapped to the same item $1, 2$. The result of the derivation is the empty graph, that is, the conflict between preservation and deletion is solved by deletion. This property of the approach is caused by the order of the construction steps: first the (conflicting) items are glued and in the case of conflicts as described they are deleted. Similar, the problem of the dangling edge on the right of Figure 4.4 is solved: it is deleted, too.

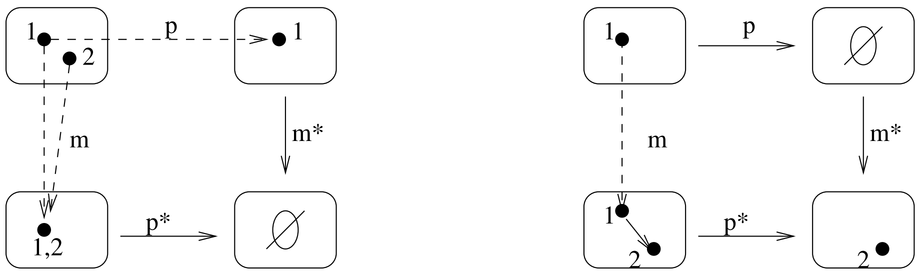


Figure 4.4: Deletion with conflicts in a common graph.

It would be possible to restrict the matches such that conflicts as described above are avoided. This restriction is called *gluing condition*. However, in the standard single-pushout (SPO) approach matches need not fulfill the gluing condition. This may lead to unexpected results, as will be illustrated by the following example:

Example 4.1.13 (*Deletion with Conflicts*)
Assume a production allowing the deletion of a class symbol as illustrated in the upper part of Figure 4.5. This production is being applied to the given attributed graph structure G. Applying the production effects in the deletion of the association *3:Assoc* together with the operations *beg* and *end* in G (according to the pushout of attributed graph structures mentioned in Remark 4.1.11). This means, that the application of this production leads to side-effects (deletion) not specified by the production.

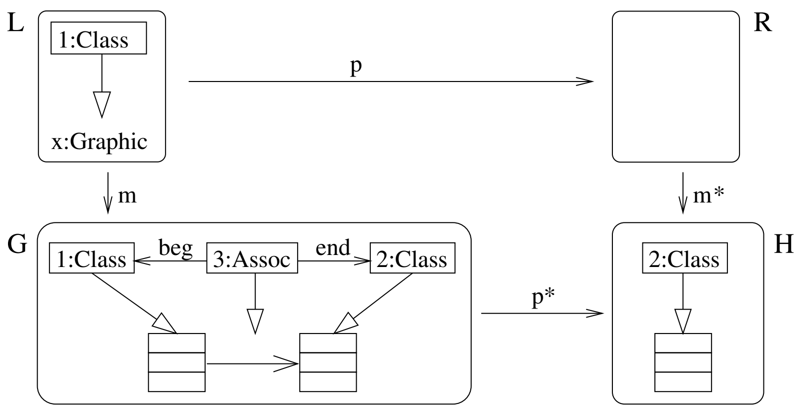


Figure 4.5: Deletion with conflicts in an attributed graph structure.

△

Application conditions for productions were first introduced in [HW95] and in [HHT96] concerning the graph structure part (GS part) only. Just as attributed graph structures are split into a graph structure and an attribute part, application conditions for attributed productions are splitted as well. Application conditions concerning attributed graph structures are expressed by AGS-morphisms (cf. [HMTW95, Koc99]).

An application condition consists of a finite set of conditional constraints over an attributed graph structure. The idea of a conditional constraint is to require structures in a graph structure G under the assumption of the existence of a certain substructure in G.

Definition 4.1.14 (*Conditional Constraint, Application Condition*)
Let AGS be an attributed graph structure signature and L be an AGS-algebra.
A *conditional constraint* $cc_L = (c : L \to X, c_X)$ over L consists of

- a total morphism $x : L \to X$
- an I-indexed set of total morphisms $c_X = (c_i : X \to C_i)_{i \in I}$, where I may be empty as well. If I is empty, cc is a *negative application condition* (NAC).

A total AGS-morphism $m : L \to G$ *satisfies* a conditional constraint cc_L over L, denoted by $m \models_L cc_L$, if for each total AGS-morphism $p : X \to G$ with $p \circ x = m$ there is an $i \in I$ and a total AGS-morphism $q : C_i \to G$ with $q \circ c_i = p$.

$$\begin{tikzcd} C_i \arrow[drr, "q"'] & X \arrow[l, "c_i"'] \arrow[dr, "p"] & L \arrow[l, "x"'] \arrow[d, "m"] \\ & & G \end{tikzcd}$$

An *application condition* A_L over L consists of a finite set of conditional constraints over L. A total AGS-morphism $m : L \to G$ *satisfies* A_L if $m \models_L cc_L$ for each $cc_L \in A_L$. △

A *conditional production* is a production equipped with an application condition over the LHS and over the RHS. The ones for the LHS are preconditions that have to be met before application of the production. The conditions over the RHS have to be fulfilled afterwards. A *conditional direct derivation* is then a direct derivation where the match meets the application conditions of the LHS and the co-match meets the application conditions of the RHS.

Definition 4.1.15 (*Conditional Production*)
A *conditional production* is a pair $cp = (p, A(p))$ consisting of a production $p = (L \rightarrowtail R)$ and an application condition $A(p) = (A_L, A_R)$ over L and R.

The conditional production is *applicable* to a graph G at a match $m : L \to G$ and co-match $m^* : R \to H$ if $m \models A_L$ and $m^\star \models_R A_R$. In this case, the *conditional direct derivation* is given by the direct derivation $G \overset{p,m}{\Longrightarrow} H$ of the production p at m. △

Example 4.1.16 (*Conditional Production*)
Figure 4.6 illustrates a conditional production with two negative application conditions (NACs): the production allows us to delete a class symbol whenever the class is not linked with an association symbol. This is expressed by the two NACs in A_L: the morphism x_1 maps *1:Class* in L to *1:Class* in A_L, and the morphism x_2 maps *1:Class* in L *2:Class* in A_L.

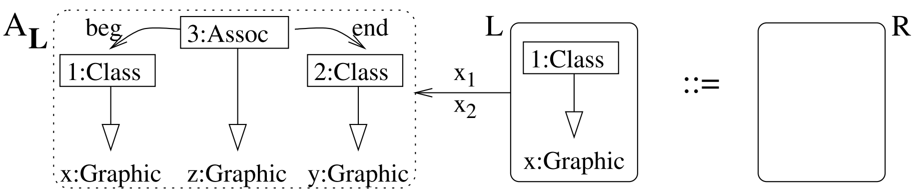


Figure 4.6: Conditional production allowing the deletion of a class symbol.

△

As already mentioned, negative application conditions are more powerful than the gluing condition. This is the reason why we are able to model the gluing condition by NACs. Moreover, NACs can be used for several expressions. E.g., assume a class diagram signature and an attribute signature of strings, a class is attributed by a string indicating the class names. A desirable production expresses the insertion of a class together with a class name given as variable of sort *String*. The variable, let us call it *cn:String*, occurs in the LHS as well as in the RHS of the production. The RHS comprises additionally a class item which is then attributed by the variable. Nevertheless, in class diagrams it is required that class names are unique. Such a requirement can be also expressed by a NAC as in Section 4.2.

All productions are collected in a grammar. Usually, a grammar consists additionally of a start sentence. The grammar productions are language generating as well as pure editing productions like the production allowing the deletion of a class. The sequential application of grammar productions to attributed graph structures is done by attributed graph structure transformation. This process is called *derivation sequence* generating an attributed graph structure. An attributed graph structure language is then defined as the set of all sentences which are generated by a derivation sequence.

Definition 4.1.17 (*Conditional Grammar, Derivation Sequence, Language*)
A *conditional grammar* is a pair $GG = (S, cp = (p, A(p))_{p \in P})$, where S is an attributed graph structure and $(p, A(p))_{p \in P}$ is a P-indexed set of conditional productions. P is a finite set of productions.

We denote by $Der(GG) = \{S \stackrel{p_0,m_0}{\Rightarrow} G_1 \stackrel{p_1,m_1}{\Rightarrow} \ldots \stackrel{p_n,m_n}{\Rightarrow} G_{out} \mid p_i \in P, m_i$ are total matches, $n \in \mathbb{N}\}$ the set of all derivation sequences starting at S using conditional productions of P applied according to the matches.

An attributed graph structure language $Lan(GG)$ is given by the set of all sentences G_{out} which are generated by a derivation sequence:
$Lan(GG) = \{G_{out} \mid S \Rightarrow \ldots \Rightarrow G_{out}\}$. $\triangle$

4.1.3 Constraint Satisfaction Problem

Constraints occur in several application areas as given by common languages, simulation, graphics, user interfaces, etc. In general, constraint-based systems can be classified by *Logic Programming*, *Constraint Programming*, and *Constraint Logic Programming* (CLP).

Prolog is one well known *Logic Programming* language. It is based on unification and back tracking. It is not strived for finding a numerical solution as it is required in graphical environments. The main emphasis of *Constraint Programming* is exactly to find a numerical solution from a constraint system. Such a constraint system consists of a set of equations and inequations. *Constraint Logic Programming* combines the features of *Logic Programming* (unification) and *Constraint Programming* (numerical solution), and extends them by the possibility to formulate constraints. Thereby, the pattern matching mechanism of the unification is replaced by a more general operation, which is called *Constraint Satisfaction*.

For each of these classes of constraint-based systems numerous constraint solving algorithms can be found in the literature. An overview is given in [Gri96]; this is briefly reviewed in Chapter 5. The definitions proposed in the following are adapted from [DvB97].

Definition 4.1.18 (*Constraint Satisfaction Problem*)
A *constraint-satisfaction problem* (CSP) is given by

- n variables $X = \{x_1, \ldots, x_n\}$ for $n \in \mathbb{N}$
- a (finite) domain D_i of possible values for each variable x_i, $i = 1, \ldots, n$
- a set of t relations (the constraints) $\{C_{S_1}, \ldots, C_{S_t}\}$, where S_i are arbitrary n-tuples of variables in X.

A *constraint* C_S over a tuple of variables $S = (x_1, \ldots, x_r)$ is a subset of the product of the variable domains (i.e. $C_S \subseteq D_1 \times \ldots \times D_r$). The number r of variables on which a constraint is defined denotes the arity of the constraint.

Given a list of variables, i.e. $X = (x_1, \ldots, x_n)$ and their domains D_i, $1 \leq i \leq n$. A n-tuple $\Gamma = (a_1, \ldots, a_n)$ with $a_i \in D_i$ is an assignment for each variable x_i with a concrete value a_i. This variable assignment is also called *instantiation*. We use $\Gamma(x_i) = a_i$ to denote the value of x_i under the assignment Γ.

A constraint C_S defined over a tuple of variables $S = (x_1, \ldots, x_r)$ is *satisfied* by a variable assignment iff $(\Gamma(x_1), \ldots, \Gamma(x_r)) \in C_S$.

A variable assignment Γ is called *solution of a* CSP if it satisfies all constraints of the CSP. △

Let us illustrate this definition by an example:

Example 4.1.19 (*Constraint Satisfaction Problem and Solution*)
Given variables $X = \{x_1, x_2, x_3, x_4\}$ with domains $D_i = \{1, 2, 3\}$, $\forall x_i \in X$. In the following we list three constraint tuples and constraints:

1. $S_1 = (x_1, x_2, x_3)$,
   $C_{S_1} = \{(1,1,1), (1,1,3), (1,2,3), (1,3,2), (2,1,3),$
   $(2,2,2), (2,3,1), 3,1,2), (3,2,1), (3,3,3)\}$
2. $S_2 = (x_2, x_4)$,
   $C_{S_2} = \{(1,2), (2,1), (2,3), (3,1), (3,3)\}$

3. $S_3 = (x_3, x_4)$,
   $C_{S_3} = \{(1,2), (1,3), (2,2), (3,1), (3,2)\}$

One solution is given by the assignment $\Gamma_1 = (3, 2, 1, 3)$.
The set of all solutions is given by

$$Sol(X) = \{(1,1,1,2), (1,1,3,2), (1,2,3,1), (2,1,3,2), \\ (2,3,1,3), (3,1,2,2), (3,2,1,3), (3,3,3,1)\}.$$

No solution is, e.g. $\Gamma_Y = (1, 2, 3, 3)$ because $\Gamma_Y[S_3] = (3, 3)$ but $(3, 3) \notin C_{S_3}$.
$\triangle$

The following definition concerns the (sub) assignments of constraint variables which is used in Section 4.2.2 to establish the default layout of symbols.

Definition 4.1.20 (*Subassignment*)
Given a set X of variables and their domains D_i, $1 \leq i \leq n$. We assume $Y \subset X$ and Γ_Y as an assignment of Y. Then, Γ_Y is called a *subassignment* of X. $\triangle$

A constraint restricts the possible values of a variable under consideration of other variable values. For simplifying a CSP, often a binary CSP is considered with unary and binary constraints. Each n-ary CSP can be transformed into an equivalent binary constraint. The advantage of binary CSPs is the possibility to represent them by graphs. Each variable represents a node and each constraint an undirected edge. Unary constraints represent loops. Directed edges represent the variable order in a tuple over which the constraint is defined. This kind of CSP representation is called *constraint graph*. Such constraint graphs will be used in Section 4.2.2.

Graphical constraints, attributed graph structure signatures and attributed graph structures are the main building blocks for visual syntax modeled using GENGED. How they are used is topic of the next section.

4.2 Visual Language Specifications

In this section we propose the formal aspects of visual language specifications (briefly *VL specifications*) given by a combination of algebraic graph structure transformation and constraint-satisfaction problems. According to our conceptual model presented in Chapter 3, the specification of a visual language (VL) comprises a visual alphabet and a visual grammar. The visual language itself is defined to be the set of all visual sentences that can be derived by applying grammar productions to a given start sentence; then the sentences are called *VL sentences*. Table 4.1 shows the relationships between the formal aspects and the visual language aspects.

Formal Aspects	**Visual Language Aspects**
Attr. Graph Structure Signature AGS + CSP	Visual Alphabet
Attr. Graph Structure (Algebra) w.r.t. AGS	Visual Sentence
Attr. Graph Structure Grammar	Visual Grammar
Attr. Graph Structure Language	Visual Language

Table 4.1: Relationship between formal and language aspects.

The visual alphabet is represented by an attributed graph structure signature AGS and a constraint-satisfaction problem CSP. Visual sentences are algebras (attributed graph structures) with respect to the visual alphabet. They occur in the constituents of a visual grammar as well as they may be VL sentences which are derived by applying grammar productions. Due to our concepts proposed in Chapter 3.2, we distinguish lexical and attribute symbols as well as connection and attribution links. Moreover, due to Requirement 2.2.1, we distinguish the abstract and the concrete syntax of both, signatures and algebras. This yields an attribution situation shown in Figure 4.7.

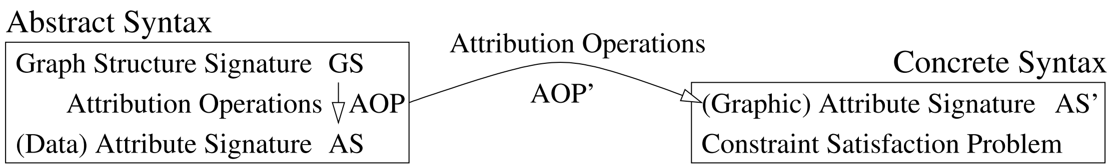


Figure 4.7: Abstract and concrete syntax levels of an alphabet.

The abstract syntax of VLs is completely based on the techniques proposed in Section 4.1.2. The abstract syntax of (lexical and attribute) symbols is specified by sort symbols in the corresponding graph structure signature GS, the abstract syntax of connection links by operation symbols in GS. The attribute signature AS specifies the datatypes such as String or Integer. These datatypes are coupled with the attribute symbols by attribution operations in AOP.

The concrete syntax of VLs (the layout) is specified by a graphic signature and a constraint-satisfaction problem CSP. All (abstract) sort symbols occurring in GS are attributed by a data attribute sort in AS and also by a graphic sort in AS' such that we get a double attribution (cf. Figure 4.7). In our approach the two syntactical descriptions represent the visual syntax of VLs.

The abstract syntax of visual languages is briefly proposed in Section 4.2.1. The visual syntax including the abstract and concrete syntax, is the topic of Section 4.2.2.

4.2.1 Abstract Syntax

The *abstract syntax* of a visual language (VL) describes its logical level. Especially the semantics of a VL and the implementation in terms of programming code is based on the abstract syntax.

In our approach, the abstract syntax of a VL specification is represented by an attributed graph structure signature (the alphabet) and an attributed graph structure grammar. These constituents are already discussed in Section 4.1.2. We have used attributes of type *Graphic* and have shown that we need more powerful concepts in order to express correct positions and sizes for the concrete syntax level. In this section we consider datatypes like String, Integer, etc. which are the datatype parts of the attribute symbols. Attribute symbols like the name of a class symbol are now modeled by the attribute carriers (cf. Convention 4.1.6).

Abstract Syntax of an Alphabet

The abstract syntax of an alphabet is given by an attributed graph structure signature according to Definition 4.1.4. According to Concept 3.3.1, the links have to be cycle-free.

Definition 4.2.1 (*Cycle-free Signatures*)
Given a signature $\Sigma = (S, OP)$.

- A sort symbol s_1 depends on a sort symbol s_2, denoted by $s_2 < s_1$, if there is an operation $op : s_1 \to s_2$ in OP.
- The dependency relation $<_\Sigma$ is the transitive closure of $<$.
- A signature Σ is called *cycle-free* if the dependency relation $<_\Sigma$ is irreflexive.

$\triangle$

Cycle-free signatures are required by alphabets, namely, for the connection links between lexical symbols.

Definition 4.2.2 (*Abstract Syntax of an Alphabet*)
The abstract syntax of an alphabet ($\text{ALPH}_{\text{Abstr}}$) is given by an attributed graph structure signature AGS = $GS + AS + AOP$ (cf. Definition 4.1.4), such that

- $GS = (S_{GS}, OP_{GS})$ is a cycle-free graph structure signature specifying the symbols and links
- $AS = (S_{AS}, OP_{AS})$ is an arbitrary signature specifying the datatypes of attribute symbols
- $AOP = (AOP_s)_{s \in S}$ is an S-indexed family of operations $aop : s \to s'$ coupling attribute symbols $s \in S$, $S \subseteq S_{GS}$, with datatype symbols s' in S_{AS}.

$\triangle$

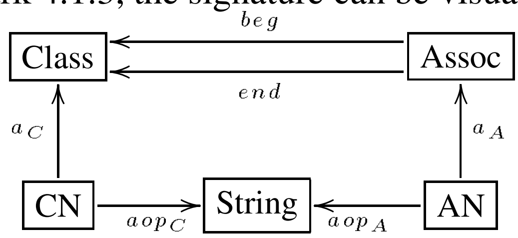
Let us illustrate this definition by an example:

Example 4.2.3 (*Abstract Syntax of an Alphabet*)
The main constituents of class diagrams we regard in this example are classes, binary associations (cf. Example 4.1.5) and corresponding attributes. That is, a class is attributed with a class name (CN) and an association (Assoc) is attributed with an association name (AN). We use the datatype String for these attribute symbols. The items CN and AN are attribute symbols and hence they are part of GS, whereas String is the datatype sort in AS:

classdiagram$_{Abstr}$ =

GS	**sorts**	Class, Assoc, CN, AN	
	opns	beg: Assoc $\to$ Class;	end: Assoc $\to$ Class;
		a$_C$: CN $\to$ Class;	a$_A$: AN $\to$ Assoc;
AS	**sorts**	String	
AOP	**opns**	aop$_C$: CN $\to$ String;	aop$_A$: AN $\to$ String;

According to Remark 4.1.3, the signature can be visualized as a graph:



△

Abstract Syntax of Visual Sentences

Visual sentences are algebras with respect to an attributed graph structure representing the alphabet.

Definition 4.2.4 (*Abstract Syntax of Sentences and Sentence Morphisms*)
Given the abstract syntax of an alphabet AGS = $GS + AS + AOP$. An abstract sentence is an attributed graph structure (cf. Definition 4.1.7) consisting of

- a total GS-algebra $A^{GS} = (A^{GS}_S, A^{GS}_{OP})$ describing the symbols and links of a visual sentence
- a partial AS-algebra $A^{AS} = (A^{AS}_S, A^{AS}_{OP})$ describing the datatypes of the attribute symbols
- a family AOP^A of total operations $aop^A : A^{GS}_s \to A^{AS}_{s'}$ for all $aop : s \to s' \in AOP$ coupling attribute symbols with their datatype symbols.

Given two visual sentences A and B with respect to the abstract syntax of an alphabet. A *sentence morphism* $h : A \circ\!\!\to B$ is an attributed graph structure morphism (cf. Definition 4.1.7). It is a pair of partial and total homomorphisms $h = (h^{GS}, h^{AS})$ which preserve attributions AOP, that is $aop^B \circ h^{GS} = h^{AS} \circ aop^A$ for all $aop : s \circ\!\!\to s' \in AOP$. We denote by $h^{GS} : A^{GS} \circ\!\!\to B^{GS}$ the partial homomorphism between the graph structure algebras with respect to GS. If h^{GS} is total, then h is called total homomorphism. △

In order to illustrate the abstract syntax of visual sentences and morphisms between them, let us consider an example:

Example 4.2.5 (*Abstract Syntax of Sentences and Sentence Morphisms*)
Figure 4.8 illustrates two class diagrams, the left one is called A, the right one B. Both are algebras with respect to the abstract syntax of the alphabet presented in Example 4.2.3.

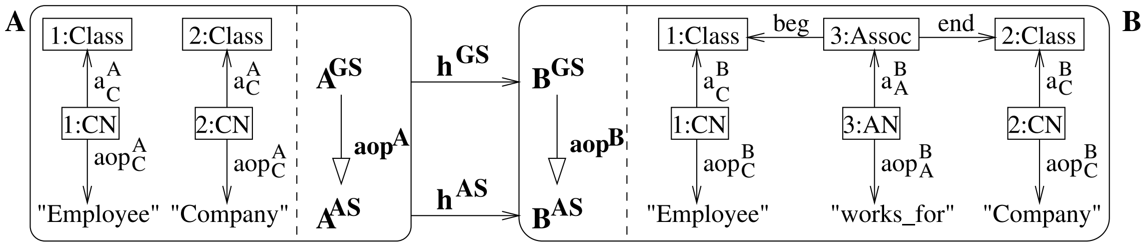


Figure 4.8: Morphism between the abstract syntax of two class diagrams.

The morphism between both algebras A and B is defined component-wise according to the GS and the AS parts. It is illustrated by the leading numbers of the graph structure items. This means that *1:Class*, respectively *2:Class*, in A are mapped to *1:Class*, respectively *2:Class*, in B. △

Abstract Syntax of a Visual Language Grammar
The abstract syntax of a visual language grammar is given by a conditional grammar as introduced by Definition 4.1.17. It comprises a start sentence and a set of conditional productions describing how a given visual sentence shall be transformed into another one. We extend now the productions introduced in Definitions 4.1.9 and 4.1.15 by production names and optionally some production parameters. Each parameter is given by a variable of a certain datatype that is specified in the alphabet's attribute signature AS. It is required that the production names are unique in one grammar.

Definition 4.2.6 (*Abstract Syntax of a Visual Language Grammar*)
The abstract syntax of a visual language grammar *VL-GG*$_{Abstr}$ = (S, P) is given by the abstract syntax of a start sentence S and the abstract syntax of a set of productions P.

Each production consists of a production header and a production body. The production body p may be a production as proposed in Definition 4.1.9 or a conditional production as proposed in Definition 4.1.15. The production header comprises a unique production name and optionally a set of parameters due to datatypes of attribute symbols. Each parameter is given by either a variable or a constant of a specific sort $s' \in AS$. The variable or the constant are in the left-hand sides of productions. The production name defines the production morphism which is an AGS-morphism (cf. Definition 4.1.7 and 4.2.4). △

Usually, variables that occur in the RHS of a production have to be defined already in the LHS in order to obtain a pushout if the production is applied. In most articles this fact is implicitly treated but it is explicit required in [TER99]. In our approach such a variable indicates an attribute symbol. Let us illustrate the abstract syntax of a production by an example:

Example 4.2.7 (*Abstract Syntax of a Production*)
Given the alphabet of Example 4.2.3 and the production with production header *InsertClass (cn: String)*. Figure 4.9 illustrates the abstract syntax of the corresponding production body.

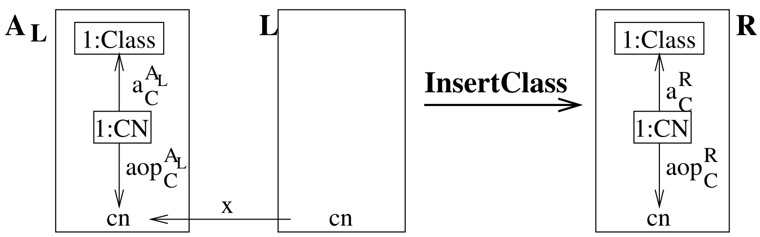


Figure 4.9: Abstract syntax of production *InsertClass (cn: String)*.

The production comprises a negative application condition A_L, a left-hand side L and a right-hand side R (cf. Example 4.1.16). The negative application condition states that the class and the class name should not be available in the given sentence. The variable of the production parameter *cn* is the only part presented in L. Then we have a total AGS-morphism $x : L \to A_L$ and a total AGS-morphism $h :=$ *InsertClass*: $L \to R$. △

Abstract Syntax of a Visual Language

The abstract syntax of a visual language is defined to be the set of all sentences which are derived by applying grammar productions. A production is applied to a given visual sentence according to Construction 4.1.10.

Definition 4.2.8 (*Abstract Syntax of a Visual Language*)
The abstract syntax of a visual language $L(VL\text{-}GG_{Abstr})$ is an attributed graph structure language as introduced in Section 4.1 (cf. Definition 4.1.17).

The language of the abstract syntax of visual sentences is a sublanguage of all visual sentences, i.e. $L(VL\text{-}GG_{Abstr}) \subseteq \boldsymbol{Alg}(\text{AGS})$. △

The application of productions has been already described in Section 4.1. Because the application of grammar productions modeling visual syntax is similar to these considerations we omit an example and concentrate on the visual syntax of visual languages in the following section.

4.2.2 Visual Syntax

The visual syntax of a visual language extends the abstract syntax by layout descriptions for symbol graphics, and constraint-satisfaction problems expressing positions and sizes of the symbol graphics. We call these extensions the *concrete syntax* level. As already mentioned in the beginning of this section (cf. Figure 4.7), the symbol graphics are modeled as attributes of abstract syntax items. These are lexical as well as attribute symbols of a visual language.

The concrete syntax of symbols and links is based on the box concept for symbol graphics, their access operations, and graphical constraints as proposed in Chapter 3.1. For each symbol there is exactly one symbol graphic and some symbol constraints. For each link there is at least one graphical constraint. These constraints are built up over a set of constraint variables (cf. Section 4.1.3). The constraint variables reflect the graphic-specific access operations. In this sense we have a bijective correspondence between constraint variables and access operations on graphics modeling attachment points.

We start with the description of the dependencies between graphics and constraints. Thereafter, we explain how the visual syntax is specified along the constituents of a VL specification. We consider again the notion of alphabet, sentences, grammar and visual language. We use "Conceptual Definitions" instead of "Definitions" because the concrete syntax of visual languages is not given completely by formal notions.

Graphical Objects and Constraints

The main features of a graphic algebra G as proposed in Chapter 3.1 concern primitive graphics and a layout for datatypes (cf. Chapter 3.1.1), the box concept for symbol graphics (cf. Chapter 3.2), and access operation on graphics (cf. Chapter 3.1.2). In the following we sketch the corresponding graphic signature $\Sigma(G)$.

Conceptual Definition 4.2.9 (*Graphic Signature* $\Sigma(G)$)

The main features of the graphic signature, called $\Sigma(G)$, are given as follows:

$\Sigma(G)$ =

sorts Graphic, Point, Size, Real, String, ListOfString, ListOfGraphic, ...

opns `/* Graphical Primitives */`
primitive: Point Size ... → Graphic;

`/* Datatypes */`
text: Point Size String ... → Graphic;
list: Point Size ListOfString ... → Graphic;

`/* Symbol Graphic */`
box: Point Size ListOfGraphic → Graphic;

`/* Access Operations */`
ac_R: Graphic → Real;
ac_P: Graphic → Point;

△

The signature $\Sigma(G)$ comprises all sort and operation symbols which are necessary to build up a graphic represented by the sort Graphic. Graphical primitives like rectangles, lines, etc. are represented by the operation primitive. It is required, that they contain at least a position (Point) and a size (Size) as argument sorts. Non-primitive graphics are built up by the operations text, list, and box. The latter one determines a symbol graphic according to Concept 3.2.3; the enclosed primitives are represented by the sort ListOfGraphic. The dots used for graphical primitives and datatypes indicate that also other layout features may be specified, like colors or line thickness.

Conceptual Definition 4.2.10 (*Access Operations on Graphics*)
The access operations ac_R and ac_P given in the signature $\Sigma(G)$ (cf. Conceptual Definition 4.2.9) indicate possible attachments of graphics like coordinates, circumferences, and areas (cf. Concept 3.1.2). △

Graphical constraints are used to specify spatial relations between graphics according to the abstract syntax of links. We consider constraints for positions and sizes of graphics. Not only binary but also n-ary constraints are conceivable to express spatial relations. As mentioned in Section 4.1.3, n-ary constraints can be transformed into equivalent binary constraints which can be modeled by constraint graphs.

Conceptual Definition 4.2.11 (*Constraint Graph*)
A constraint-satisfaction problem (cf. Definition 4.1.18) with binary constraints is given by a *constraint graph* where each constraint variable represents a node and each constraint represents a directed edge. △

We will use constraint graphs for illustrational reasons. Moreover, they define the "interface" between the definitions of Section 4.1.3 and the constraints as they are treated in this section.

Example 4.2.12 (*Constraint Graph*)
Given variables $X = \{x_1, x_2\}$ with domains Real and Point, respectively. Let $C_{S_1} = \{(x_1, x_2) \mid x_1 = x_2\}$ a constraint over the tuple $S_1 = (x_1, x_2)$. The corresponding constraint graphs are illustrated by Figure 4.10; the operation symbols dom_R and dom_P, respectively, indicate the attribution of variables. A solution is given by the set of tuples over the variable domains which satisfy the constraint C_{S_1}.

Figure 4.10: Constraint graphs.

△

We are coming back to graphics and their access operations. Each access operation defines a constraint variable occurring in constraint-satisfaction problems. The domain of a constraint variable is given by the co-domain of the corresponding access operation such that there is a bijective correspondence between the domains. access operations on graphics and the domains of constraint variables (cf. Conceptual Definition 4.2.10).

Conceptual Definition 4.2.13 (*Bijective Correspondence*)
Figure 4.11 illustrates the bijective correspondence between the domains of constraint variables and co-domains of the access operations on graphics.

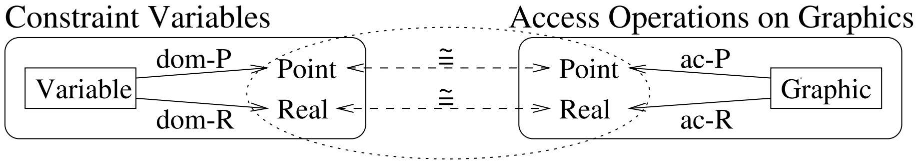


Figure 4.11: Bijective correspondence between domains of constraint variables and co-domains of access operations on graphics.

△

In the following we explain how the concrete syntax of alphabets, sentences and visual languages is achieved from the graphics, the attachments/access operations, and the constraint variables and constraints occurring in constraint-satisfaction problems.

Visual Syntax of an Alphabet

Given the abstract syntax of an alphabet as defined in Section 4.2.1. The concrete syntax is an extension of the abstract syntax by a language-specific graphic signature, some attribute carriers and corresponding (attribution) operations, and a constraint-satisfaction problem. Subassignments of some constraint variables define the default layout of symbols occurring in visual sentences (algebras).

Conceptual Definition 4.2.14 (*Visual Syntax of an Alphabet*)
Given ALPH$_{\text{Abstr}}$, the abstract syntax of an alphabet, i.e. an attributed graph structure signature AGS = $GS + AS + AOP$ (cf. Definition 4.2.2). The visual syntax of an alphabet (ALPH$_{\text{Visual}}$) extends this signature yielding an (double) attributed graph structure signature AGS'= $GS' + AS' + AOP'$ and a constraint-satisfaction problem CSP:

- GS' = $(S_{GS'}, OP_{GS'})$ contains GS plus further sorts and operations:
  - $\forall\ s \in S_{GS}$ in GS $\exists$ attribute carrier sort AC_s in $S_{GS'}$
  - $\forall\ s \in S_{GS}$ in GS and $AC_s \in S_{GS'}$ $\exists\ a'_s : AC_s \rightarrow s \in OP_{GS'}$

- AS' = $(S_{AS'}, OP_{AS'})$ contains AS plus a (language-specific) graphic signature (cf. Conceptual Definition 4.2.9) where:
  - $S_{AS'}$ at least comprises the sorts Graphic, Point, and Real
  - $\forall\ s \in S_{GS}$ in GS we have the following operations in $OP_{AS'}$:
    * one symbol graphic: box_s: Point Size ListOfGraphic $\rightarrow$ Graphic;
    * primitive operations: primitive_s: Point Size . . . $\rightarrow$ Graphic;
    * access operations: ac_R: Graphic $\rightarrow$ Real; and ac_P: Graphic $\rightarrow$ Point;

- AOP' extends AOP by further operations:
  AOP' = $(AOP'_s)_{s \in S}$ with $S \subseteq S_{GS'}$, is an S-indexed family of attribution operations $aop'_s : AC_s \rightarrow Graphic$.
- CSP is a constraint-satisfaction problem as introduced in Definition 4.1.18 where we have for each operation $op \in OP_{GS}$ at least one (graphical) constraint and suitable constraint variables:

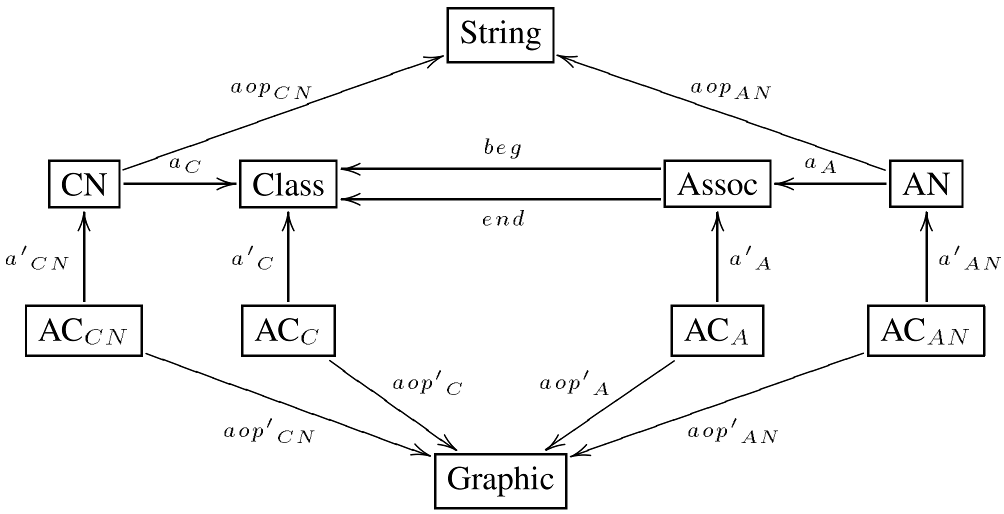
  - For each operation op_s: Point Size . . . $\rightarrow$ Graphic in AS', $s \in S_{GS}$, belonging to a symbol graphic (box and primitives), and for each access operation ac_R, ac_P, the set of constraint variables X is given by: X = {op_s.ac_R} $\cup$ {op_s.ac_P}
  - The domains of possible values for each variable x in X are given by $\mathbb{R}$ (for the sort Real) and [$\mathbb{R}$,$\mathbb{R}$] (for the sorts Point, Size).
  - Constraints are given over the set of constraint variables. This means that $C_s = \{C_{S_1}, \ldots, C_{S_t}\}$ is a set of t symbol constraints, where S_i are arbitrary tuples of variables in X.
  - There is a subassignment $\Gamma = (a_1, \ldots, a_n)$, $n \in \mathbb{N}$, for some constraint variables x_i in X, indicating positions and sizes. These constraint variables x_i correspond to primitive graphics and corresponding access operations. Additionally, there is a subassignment for each box_s.size, $s \in S_{GS}$.

△

The abstract syntax of an alphabet is extended by several sort symbols and operations describing the concrete syntax of visual languages. Thereby, the abstract syntax of datatypes defined in ALPH$_{\mathrm{Abstr}}$ (like String) is coded into sort Graphic. The abstract (data) attribute carriers indicate attribute symbols, already existing in S_{GS}. So we are able to insert for each sort symbol in S_{GS} a further attribute carrier for the graphics. These attribute carriers are coupled to the sorts indicating symbols by further operation symbols.

The language-specific graphics are specified by the signature AS'. This signature influences the CSP of an alphabet where unique constraint variables are obtained with respect to the access operations on graphics. This is already sketched in Conceptual Definitions 4.2.10 and 4.2.13. In the following example we explain the dependencies of graphics and constraints for a specific alphabet in more detail.

Example 4.2.15 (*Visual Syntax of an Alphabet*)
Given the abstract syntax of the alphabet of class diagrams (cf. Example 4.2.3); its graph representation is illustrated in the upper part of the figure below. On the abstract syntax level, the attribute carrier sorts CN and AN (of type String) indicate the attribution of lexical symbols, namely, Class and Assoc.



Extending the abstract syntax level of the alphabet by its concrete syntax, for each symbol (Class, Assoc, CN, AN) a graphic attribute carrier is introduced as well as suitable operations in $OP_{GS'}$ (not illustrated in the graph above) and AOP' such that we obtain the following signature for the visual syntax of our alphabet where the indices A and C are used as abbreviations for Class and Assoc, respectively:

classdiagram$_{Visual}$ =

GS	**sorts**	Class Assoc CN AN
	opns	beg: Assoc $\rightarrow$ Class; end: Assoc $\rightarrow$ Class;
		a_C: CN $\rightarrow$ Class; a_A: AN $\rightarrow$ Assoc;
AS	**sorts**	String
AOP	**opns**	aop_C: CN $\rightarrow$ String; aop_A: AN $\rightarrow$ String;
GS' = GS +	**sorts**	AC_C AC_A AC_{CN} AC_{AN}
	opns	a'_C: AC_C $\rightarrow$ Class; a'_A: AC_A $\rightarrow$ Assoc;
		a'_{CN}: AC_{CN} $\rightarrow$ CN; a'_{AN}: AC_{AN} $\rightarrow$ AN;
AS' = AS +	**sorts**	Graphic, Point, Size, Real, String, ListOfGraphic
	opns	`/* Symbol Graphics */`
		rect1_Class: Point Size $\rightarrow$ Graphic;
		rect2_Class: Point Size $\rightarrow$ Graphic;
		rect3_Class: Point Size $\rightarrow$ Graphic;
		box_Class: Point Size ListOfGraphic $\rightarrow$ Graphic;
		arrow_Assoc: Point Size Point Point $\rightarrow$ Graphic;
		box_Assoc: Point Size ListOfGraphic $\rightarrow$ Graphic;
		`/* Symbol Graphics for Datatypes */`
		text_CN: Point Size String $\rightarrow$ Graphic;
		box_CN: Point Size ListOfGraphic $\rightarrow$ Graphic;
		text_AN: Point Size String $\rightarrow$ Graphic;
		box_AN: Point Size ListOfGraphic $\rightarrow$ Graphic;
		`/* Access Operations */`
		.x, .y, .width, .height: Graphic $\rightarrow$ Real;
		.n, $\ldots$.se, .c, .beg, .end, size: Graphic $\rightarrow$ Point;
AOP' = AOP +	**opns**	aop'_C: AC_C $\rightarrow$ Graphic;
		aop'_A: AC_A $\rightarrow$ Graphic;
		aop'_{CN}: AC_{CN} $\rightarrow$ Graphic;
		aop'_{AN}: AC_{CN} $\rightarrow$ Graphic;

The symbol graphics defined in AS' are shown in Figure 4.12. These are the graphics for the lexical symbols Class, Assoc and for the attribute symbols CN, AN. They are built up by the symbol-specific operations in AS'. For the attribute symbols, default values enclosed by the boxes are depicted; we omit to specify also these default values because they are used for illustrational reasons only.

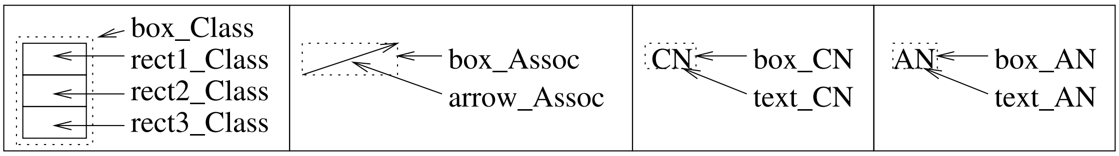


Figure 4.12: Symbol graphics for sorts Class, Assoc, CN, AN.

From the graphic operations occurring in AS' and their access operations (cf. Section 3.1.2) we obtain unique constraint variables such that there is a bijective correspondence (cf. Conceptual Definition 4.2.13). In the signature above we sketched some access operations that are available for each graphic. Those are important to reach a subassignment for some constraint variables expressing the default layout of symbols used for the visual sentences. These subassignments concern the position of primitives which is set relative to the position of the enclosing box, and the box size (cf. Concept 3.2.3 in Chapter 3).

Independent of any subassignment, graphical constraints are defined on constraint variables. Some of these constraint variables and some constraints (symbol and the link constraints) are shown in the constraint graph (cf. Definition 4.2.11) of Figure 4.13.

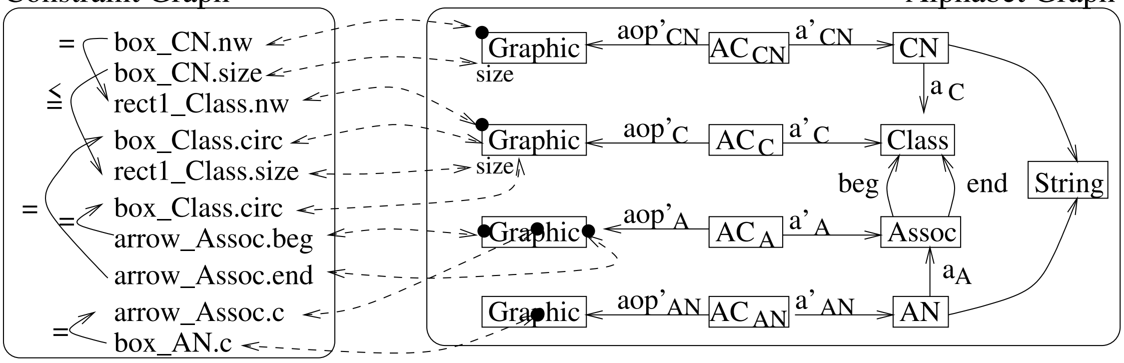


Figure 4.13: Bijective correspondence between an alphabet and a constraint graph.

According to Definition 4.2.13, there is a bijective correspondence between the domains of the constraint variables and the co-domains of the access operations on language-specific graphics; domains and co-domains are not shown explicitely in Figure 4.13.

The access operations are indicated by dots. For example, the access operation box_CN.nw denotes the upper left corner of a graphic (the *n*orth-*w*est corner of a graphic for a class name), the access operation box_Class.circ denotes the *circ*umference of a class graphic.

We used few graphical constraints only; except for the $\leq$ constraint, all constraints require equal positions. Usually, there are some more constraints necessary. The class graphic, e.g. consists of three rectangles that are related by (symbol) constraints; these are omitted. △

The example above illustrates in more detail the double attribution concept mentioned in the beginning of this section (cf. Figure 4.7). This attribution concept is suitable whenever the layout of one alphabet, given by the signature AS' and the corresponding constraint-satisfaction problem, should be changed. It would be also conceivable to hold several layouts in a list such that we have one abstract syntax for the meaningful parts of an alphabet, but one can switch between the layouts. This feature is desirable but adjourned to the future.

Visual Syntax of Visual Sentences

We have shown in the previous section that the abstract syntax of visual sentences are algebras (attributed graph structures) with respect to the abstract syntax of an alphabet. The visual syntax of visual sentences are also algebras with respect to the corresponding visual alphabet. We require that these algebras satisfy the constraint-satisfaction problems derived from the CSP of the alphabet.

Conceptual Definition 4.2.16 (*Visual Syntax of a Visual Sentence*)

Given a visual alphabet AGS' $= GS' + AS' + AOP'$ which is an attributed graph structure signature according to Definition 4.1.4, and a constraint-satisfaction problem CSP as defined in Conceptual Definition 4.2.14.

A visual sentence is an attributed graph structure A' (cf. Definition 4.1.7) with respect to the visual alphabet which satisfies the constraint-satisfaction problem of the visual sentence. According to Definition 4.1.7, the visual sentence $A' = (A^{GS'}, A^{AS'}, A^{AOP'})$ consists of

- a total GS'-algebra $A^{GS'} = (A_S^{GS'}, A_{OP}^{GS'})$ (abstract syntax of symbols and links, and attribute carrier sets for the symbol graphics),
- a partial AS'-algebra $A^{AS'} = (A_S^{AS'}, A_{OP}^{AS'})$ (abstract syntax of datatype values, and the symbol graphics)
- a family $AOP'^{A'}$ of total attribution operations $aop'^{A'} : A_s^{GS'} \to A_{s'}^{AS'}$ for all $aop' : s \to s' \in AOP'$ (coupling the abstract syntax of symbol attributes with their values, and coupling the element of each graphic attribute carrier set with a graphic whose properties satisfy the sentence CSP)

The constraint-satisfaction problem $\text{CSP}^{A'}$ for a visual sentence A' is derived from the constraint-satisfaction problem CSP of the alphabet, especially with respect to the GS'-part of algebra A':

- For each symbol as exists a constraint-satisfaction problem, called CSP_{as}: $\forall\ as \in A_s,\ A_s \in A_S^{GS'},\ \exists\ \text{CSP}_{as}$ which is derived from the corresponding constituents of the alphabet CSP (constraint variables, constraints, subassignments).
- Given two symbols $as \in A_s$ and $as' \in A_{s'}$, with A_s and $A_{s'}$ in $A_S^{GS'}$, and the corresponding constraint-satisfaction problems CSP_{as} and $\text{CSP}_{as'}$. For each link $op^A(as) = as'$ and $op^A \in A_{OP}^{AS'}$ exists a constraint-satisfaction problem, called CSP_{op^A}:
  Let $g = aop'^A(as)$ in $A_{op}^{AOP'}$ and $g' = aop'^A(as')$ in $A_{op'}^{AOP'}$, $g, g' \in A_{Graphic}$. Then, $\forall\ op^A(as) = as'$ in A_{op}, $A_{op} \in A_{OP}^{GS'}\ \exists\ \text{CSP}_{op^A}$ which is derived from the corresponding constituents of the alphabet CSP (constraint variables, constraints).
- A^{CSP} results from the disjoint union of all generated constraint-satisfaction problems: $A^{\text{CSP}} = \bigcup_{A_s \in A_S^{GS}, as \in A_s} \text{CSP}_{as} \uplus \bigcup_{op^A \in A_{OP}^{GS}} \text{CSP}_{op^A}$.

According to the derivation steps, subassignments of some variables in the CSP of an alphabet result in subassignments of the corresponding variables in A^{CSP}. These subassignments define the default layouts of graphics in a sentence which are changed if link constraints have to be satisfied. △

Usually, there is more than one solution for one constraint-satisfaction problem (cf. Example 4.1.19). We need only one solution which will be chosen from the set of solutions. In the following we illustrate the visual syntax of a visual sentence by an example:

Example 4.2.17 (*Visual Syntax of a Visual Sentence*)
Given the visual syntax of the alphabet for class diagrams as proposed by Example 4.2.15. Figure 4.14 illustrates one visual sentence with respect to these specifications. It is an algebra satisfying the derived constraints (we assume one solution Γ for the constraint-satisfaction problem).

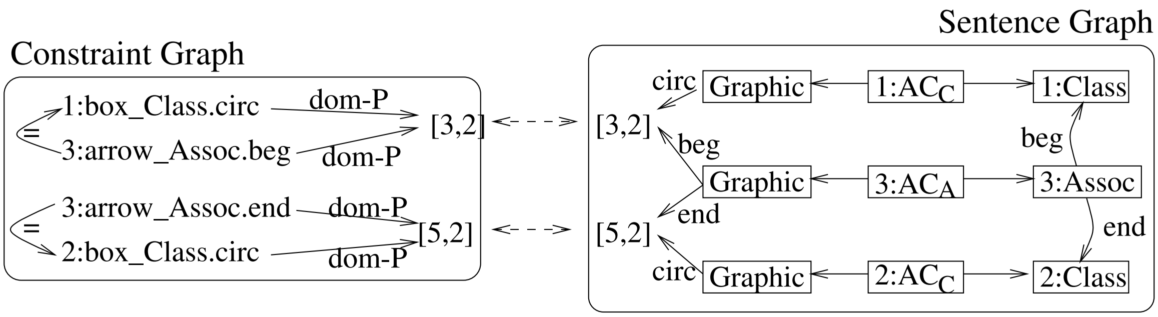


Figure 4.14: One visual sentence.

The visual sentence consists of two class symbols and one association symbol. The carrier sets for the corresponding names (class name, association name) are empty. The symbols of the sentence are indicated according to its *abstract syntax* by *1:Class*, *2:Class* and *3:Assoc* in the right of the sentence graph. Each symbol is coupled with its symbol graphic indicating the *concrete syntax* by (total) attribution operations $aop'^{A'}$ illustrated by left directed edges. According to our conceptual model proposed in Chapter 3 and Conceptual Definition 4.2.9 the symbol graphics are usually described by box expressions. For illustrational reasons we denote these expressions simply by Graphic.

A simplified version of the sentence' constraint-satisfaction problem is illustrated in the left of Figure 4.14. We restrict ourself to the link constraints. Nevertheless, we have a bijective correspondence (cf. Conceptual Definition 4.2.13) between the domains of the constraint variables (dom_P) and the corresponding co-domains of the graphics' access operations (ac_P) such that $dom(x_i) = ac\_P(g), \forall\, x_i \in X, g \in A'^{AS'}_{Graphic}, i = 1, \ldots, n, n \in \mathbb{N}$. $\triangle$

Morphisms between visual sentences are restricted to the abstract syntax because in general we do not have morphisms between the concrete syntax of visual sentences. This is due to the fact that the positions and sizes of symbol graphics are determined by solutions of the constraint-satisfaction problems for visual sentences (cf. Conceptual Definition 4.2.16). The constraint-satisfaction problems for visual sentences result from instantiating the corresponding constraint-satisfaction problem of the alphabet. This instantiating process is similar to object-oriented techniques but its formal treatment as it would be necessary for morphisms is out of the scope of this thesis. Thus, we restrict morphisms between visual sentences to their abstract syntax.

Conceptual Definition 4.2.18 (*Visual Morphism*)
Given a visual alphabet AGS' = $GS' + AS' + AOP'$ with a constraint-satisfaction problem CSP, and two visual sentences A and B with respect to the visual alphabet (cf. Conceptual Definition 4.2.16).

A *visual morphism* between visual sentences A and B is restricted to a partial morphism on the sentences' abstract syntax (cf. Definition 4.2.4). △

Visual Syntax of Visual Language Grammars
The concrete syntax of a visual language grammar is based on its abstract syntax (cf. Definition 4.2.6). Thus, a visual language grammar *VL-GG* $= (S, P)$ consists of a start sentence S and a set of (conditional) productions P.

As mentioned before, in general we do not have morphisms between visual sentences, so we do not have production morphisms in general. So, the production morphisms are restricted to the abstract syntax of visual sentences (cf. Conceptual Definition 4.2.18).

Conceptual Definition 4.2.19 (*Visual Syntax of a VL Grammar*)
The visual syntax of a visual language grammar *VL-GG*$_{Visual}$ = (S, P) is given by the visual syntax of a start sentence S and the visual syntax of a set of (conditional) productions P (cf. Definition 4.2.6). △

Note that due to Definition 4.2.6, the production names define the morphisms according to the abstract syntax of productions (cf. Definition 4.1.7 and 4.2.4).

Example 4.2.20 (*Visual Syntax of a Production*)
Given the visual syntax of the alphabet proposed by Example 4.2.15 and a production with production header *insertClass(cn:String)*. Figure 4.15 illustrates the visual syntax of the corresponding production body.

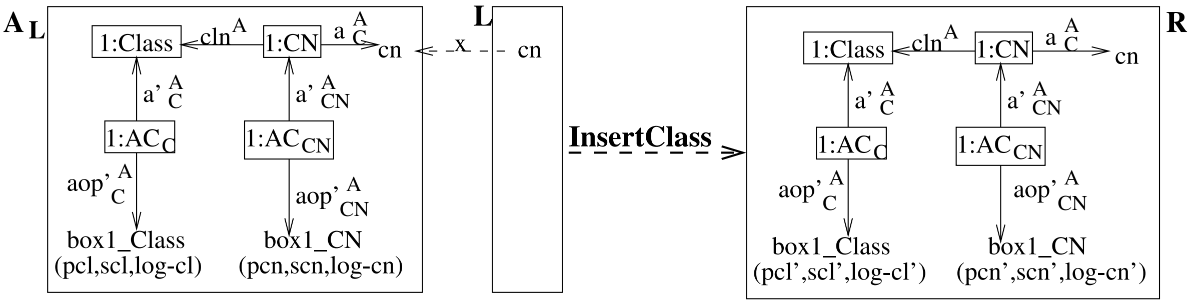


Figure 4.15: Visual syntax of the production *insertClass(cn:String)*.

Similar to Example 4.2.7, the production comprises a negative application condition A_L, a left-hand side L and a right-hand side R. The negative application condition states that the class with the class name must not be in the visual sentence where the production is applied to.

The symbol graphics in A_L and R are given by graphical expressions with variables for the positions, sizes and list of graphics. According to Conceptual Definition 4.2.16, these expressions (and the constraint-satisfaction problems) are built up from the alphabet. Although we have morphisms (*insertClass, x*) on the abstract syntax level, there are at least different positions (with respect to the concrete syntax level) in the visual sentences holding a class symbol and a class name in A_L and in L. △

In the example above we used terms as graphical expressions. In fact, the graphical expressions are visualized in the GENGED environment (cf. Chapter 5). So the visual sentences occurring in a grammar production are built up according to Conceptual Definition 4.2.16 (including the generation of constraint-satisfaction problems) such that the sentences can be visualized after constraint solving.

Visual Syntax of a Visual Language
The visual syntax of a visual language is given by the set of all visual sentences that can be derived by applying grammar productions starting at the start sentence occurring in a visual grammar.

Conceptual Definition 4.2.21 (*Visual Syntax of a Visual Language*)
The visual syntax of a visual language *L(VL-GG$_{Visual}$)* is an attributed graph structure language as introduced in Section 4.1 (cf. Definition 4.1.17) but extended by its concrete syntax yielding a double attribution. △

All visual sentences occurring in *L(VL-GG$_{Visual}$)* consist of an abstract syntax level (cf. Definition 4.2.8) which is extended by the layout, i.e. the concrete syntax level (cf. Conceptual Definition 4.2.16. Because we do not have morphisms on the visual syntax level in general, the derivation of visual sentences is slight different to that for the abstract syntax.

The application of a production p to a given visual sentence G is done as follows. Like production morphisms, also the match morphisms from a production's LHS to G are restricted to the abstract syntax. Whenever some symbols occurring in the domain of a match are linked, the match is completed on the abstract syntax level. The construction of a pushout is done on the abstract syntax level (cf. Sections 4.1 and 4.2.1). This leads to an intermediate visual sentence G'. All symbols which are preserved by the production maintain their symbol graphics with their properties (especially position and size). All symbols which are added get their default layout given by a subassignment of constraint variables originally defined by the visual alphabet (cf. Conceptual Definition 4.2.16). However, according to Conceptual Definition 4.2.14 there is no subassignment of constraint variables indicating the position of symbol graphics; in order to obtain a defined attribute (because of the required total attribution operations) we choose an arbitrary position.

For the intermediate visual sentence G, a constraint-satisfaction problem is built up (cf. Conceptual Definition 4.2.16). This constraint-satisfaction problem must be solved. Whenever there is no solution, we restore the previous state. This means that the production cannot be applied. When there is a solution, the graphical attribute carriers keeping the old values are deleted and new attribute carriers are added keeping the attributes according to the solution (cf. Remark 4.1.11). This yields the visual sentence H which can be visualized.

Figure 4.16 illustrates the application of a production called *InsertAssociation*. This production allows the insertion of an association symbol.

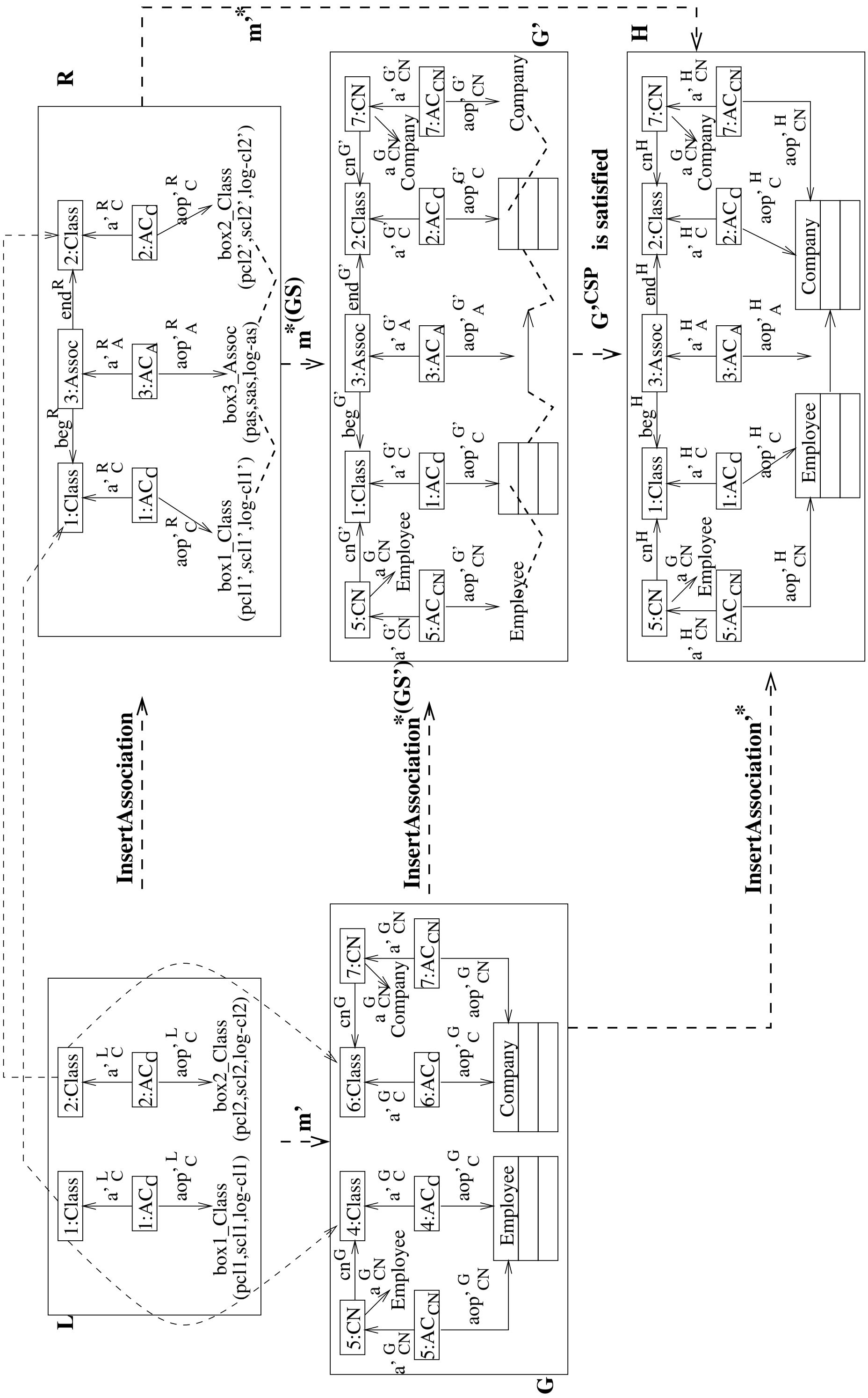


Figure 4.16: Application of the production *InsertAssociation*.

Chapter 5

The GENGED Environment

A visual language is defined by a visual language specification (short *VL specification*) consisting of an alphabet and a grammar. Accordingly, the GENGED specification tool which is part of the GENGED environment and which supports the visual definition of visual languages, comprises an alphabet editor and a grammar editor as well. Figure 5.1 illustrates the GENGED environment including the tools used, and the data flow. The dashed arrows indicate the use relations.

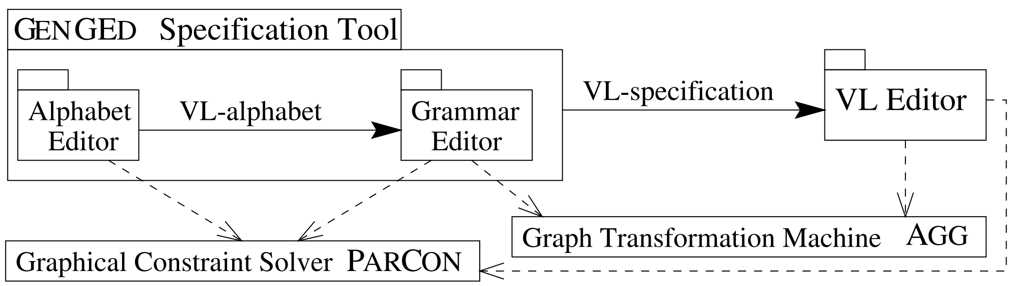


Figure 5.1: The GENGED environment.

The alphabet editor establishes an alphabet which is the input of the generic grammar editor. Given an alphabet, the grammar editor supports the visual definition of a grammar. As known from textual languages, a grammar consists of a start sentence and a set of productions. The editing of *visual sentences* (or diagrams) like the start sentence and the left-hand sides as well as the right-hand sides of grammar productions, is supported by *alphabet productions*. Alphabet productions are generated automatically from the user-defined alphabet. They are the edit commands of the syntax-directed grammar editor. The VL specification (alphabet and grammar) is the input of the generic *VL editor*. The grammar productions define the edit commands of this VL editor that supports the syntax-directed editing of *VL sentences*.

In our conceptual model (cf. Chapter 3) we distinguished the abstract and concrete syntax of visual languages. The abstract syntax defines the meaningful part of a visual language whereas the concrete syntax defines its layout. Moreover, we proposed graphs as internal data structure, and graph transformation together with graphical constraint solving as computational model. The abstract syntax items used for the symbols of a visual language are enhanced with graphics. Graphical constraints define spatial relations between the symbol graphics according to the links between the corresponding abstract syntax items. The abstract syntax level is the basis for the transformation of visual sentences. Whenever a sentence is transformed, the graphical constraints are solved such that the abstract syntax items can be enhanced with the corresponding graphics. These graphics are built up with respect to the given solution after constraint solving. Note, visual sentences occur in alphabet productions and in grammar productions as well. VL sentences that are derived from grammar productions are visual sentences, too.

In this chapter we give a brief survey of the GENGED environment. A comprehensive description can be found in the diploma thesises of M. Schwarze [Sch99] (alphabet editor) and M. Niemann [Nie99] (grammar and VL editor). Short descriptions of the GENGED environment can be found in [Bar98, BE99, BNS00, NB00]. Nevertheless, the development of GENGED is based on the conceptual model of Chapter 3. The transformation of sentences is accomplished on the abstract syntax level by the graph transformation system AGG [TER99]. The graphical constraints between the graphics define a constraint satisfaction problem solved by the graphical constraint solver PARCON [Gri96]. In Section 5.1, we first introduce the tools used together with their interfaces. Thereafter, in Section 5.2, we sketch the design of GENGED. In Section 5.3 we propose the user interface of GENGED illustrated by several screen shots.

5.1 The Tools Used

AGG [TER99] is a general purpose (programming) language where the (rule-based) graph transformation paradigm is incorporated into the object-oriented paradigm of programming. It is a hybrid programming language offering visually as well as textually represented elements. Graph transformation on the visual point of view is combined with Java [1] code for flexible attribute computations. AGG has a formal foundation based on the single-pushout (SPO) approach to graph transformation. The formal semantics of graph transformation is given in terms of category theory by a single categorical construction known as a *pushout* in an appropriate category of attributed graphs with partial morphisms - hence the name *Single-Pushout (SPO) approach.* This approach has been reviewed in Chapter 4.1 for arbitrary graph structures. The AGG graph model is one conceivable graph structure. It covers labeled and attributed hierarchical graphs. For the user interface typed and attributed directed graphs are made available. This graph model is used by GENGED.

Graphical constraints are relations between variables over a certain domain. They are called *graphical constraints* if the parameters of the variables are graphical expressions like x, y coordinates, width, height, etc. In the literature numerous constraint solving algorithms and systems based on constraint solving can be found. Well-known systems are given, e.g., by *Sketchpad* [Sut63], *Thinklab* and *Thinklab II* [Bor81, FMB90, SMFB93]. Nevertheless, all constraint solvers and algorithms differ with respect to the underlying formalisms according to the following problems.

- Logical connection of constraints (e.g. conjunction, disjunction).
- Treatment of cyclic and non-directional constraint nets.
- Treatment of dynamical changes of constraint nets (*incremental solver*).

PARCON [Gri96] is an executable constraint solver developed especially for solving graphical constraints. This constraint solver has enough capacity to express complex situations. Moreover, it offers the necessary speed for practical applications [PSG95]. In the case of under-specified constraint systems the principle of *least astonishment* is realized. A special treatment for over-specified systems is not realized, yet.

[1] "Java" is a trademark of Sun Microsystems, Inc.

In this section we introduce both systems, namely, AGG and PARCON. We start with the description of AGG. In Section 5.1.2 we propose PARCON.

5.1.1 Graph Transformation by AGG

Figure 5.2 illustrates the components (packages) of the AGG system. The AGG system offers comprehensive means of editing indicated by the packages Editor and GUI. It allows the construction of a graph grammar with rules which can be applied to a so-called *host graph*. The editing features are not used by GENGED. The used components are given by the graph model (package Basis), attribution features (package Attributes), match search (package Util) and transformation (package ALR). By AGG arbitrary graph rules are allowed including insertion, deletion and gluing of graph parts as well as complex attribute computations based on Java computations. Moreover, negative application conditions (NACs) can be added to rules in order to restrict their applicability to certain situations. All these features are used by GENGED, however, with a slightly different graph model.

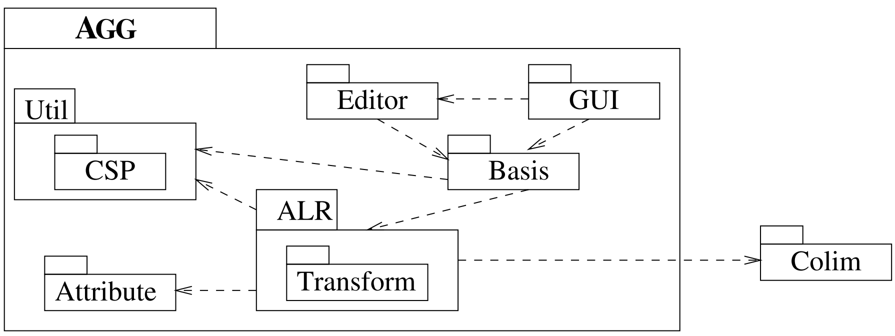


Figure 5.2: Components of the AGG environment.

Graph model The AGG-system supports two different notions of a graph which are closely related. On the one hand, this is the notion of so-called simple graphs, that are graphs covering nodes and arcs. The other idea of a graph is a hierarchical one. Both graph models are implemented by the package called Basis. Since the GENGED-system does not make use of hierarchical graphs but of simple graphs, we concentrate on those.

A *simple graph* consists of two disjoint sets containing the *nodes* and the *arcs* of a graph. These are called *objects* of the graph. Every arc represents a directed connection between two nodes. They are called the *source* and *target* nodes of the arc. Any graph object may be associated with exactly one *type* out of a given type set. An AGG graph can have multiple arcs of the same type between a single pair of nodes, because every arc has an identity on its own, just like a node does. This fact distinguishes the underlying graph notion from another popular notions where an arc is described just as a relation between nodes. One graph is illustrated by Figure 5.3 [2]. The nodes are represented by rectangles and arcs by ellipses with corresponding source and target illustrated by arrows. The numbers followed by names indicate the objects and the object types, respectively.

Figure 5.3: A class diagram modeled by AGG.

Attribution Nodes and arcs of graphs can be extended by attributes. These attributes correspond to common data attributes which have to be evaluated during a transformation step. The attribute component Attribute consists of one *attribute manager* and an arbitrary number of *attribute handlers*. Each handler implements the semantical domain of a specific type of attributes. With a graph object having multiple attribute entries, each one may be handled by a different attribute handler. By default, the attribute component offers one attribute handler that is able to cope with arbitrary Java expressions. One attributed graph is illustrated by Figure 5.4. The nodes and arcs are represented as before. The attributes are written beneath the typed graph objects and coupled by arrows. The names in the left indicate the types of attributes whereas the right ones indicate their values.

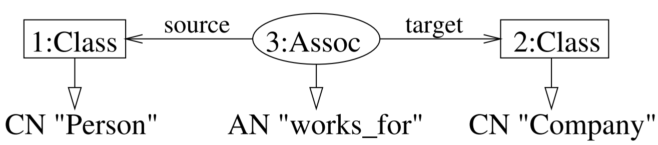


Figure 5.4: An attributed class diagram modeled by AGG.

[2]The graph of Figure 5.3 is an algebra with respect to the Signature $Graph_\Sigma$ in Remark 4.1.3.

Match morphism In order to apply a graph rule to a host graph a match must be found matching all attributed nodes and arcs in the left-hand side of the rule to the current host graph. The graph matching problem (package Util) is formulated as a constraint-satisfaction problem (as it is done for graphical constraints in GenGEd). The advantage of such a formulation is its independence of any concrete graph matching algorithm.

Transformation The host graph can be transformed by applying rules. During a transformation step this component (package ALR) controls the match morphisms between attributed nodes and arcs of rules and the host graph. The transformation of nodes and arcs, namely, the calculation of the resulting graph, is supported by the implemented categorical library, called Colim [Wol97]. Attribute variables that occur in the left-hand side of the rule, are assigned with the concrete values of the host graph. During a transformation step the transformation component controls the assignment of variables and the attribution of involved nodes and arcs with the result of attribute evaluation.

5.1.1.1 Implementation Issues

The AGG system is designed in an object-oriented way, and it is implemented in Java. We have to mention, that there are several interfaces but until now there is no common interface available for GenGEd. The availability of such an interface may be supported using the facade pattern [GHJV95] allowing the construction of one interface for a specific application that is based on other interfaces.

Graph grammar A graph grammar can be generated using the constructor GraGra(). The result is an empty grammar, i.e. without rules but with an empty start graph. The access to the start graph is given by the method Graph getGraph(). Rules can be generated using the method Rule createRule(). The left-hand side and the right-hand side of a rule are graphs. The access to these graphs is supported by the methods Graph getLeft() and Graph getRight(). Nodes and arcs occurring in graphs are usually labeled with a type information. For the creation of types the method Type createType() is available (see below).

A rule is defined by a morphism. It can be set by the method void addMapping(GraphObject o, GraphObject i). The source and target graphs of a morphism can be retained by Graph getOriginal() and Graph getImage(), respectively. The domain and co-domain of each morphism is available by Enumeration getDomain() and Enumeration getCodomain().

Modeling visual languages allows defining application conditions. In the current AGG implementation so-called negative application conditions (NAC) can be defined. Every NAC is represented by a graph and connected to a rule by a graph morphism. As for rules, every NAC is defined by a morphism using the method OrdinaryMorphism createNAC().

Types and attributes In AGG each graph object (node or arc) has exactly one graph object type. Type informations are necessary to create nodes and arcs by the methods Node createNode(Type typ) and Arc createArc(Type typ, Node src, Node tar). The incoming and outgoing arcs of a node are retained by the methods Enumeration getIncomingArcs() and Enumeration getOutgoingArcs(). Every arc has a source and a target node. These nodes can be retained by the methods Node getSource() and Node getTarget(), respectively.

Every graph object type is associated with exactly one attribute type. On the instance level every graph object holds exactly one attribute instance of the corresponding attribute type. Because an attribute may consist of several attribute members, one attribute type is called *single declaration*. Such a declaration consists then of a *declaration type* and a *declaration name*. In the context of class diagrams the class name (attribute) of a class on the logical level (abstract syntax) may be given by the declaration *CN String*. The declaration name *CN* is written in the left and the type *String* in the right of each declaration.

The declaration type of each attribute depends on the corresponding attribute handler available in AGG. This attribute handler allows managing the declaration types, proofing consistence, and evaluating during a transformation. The actual attribute handler allows the treatment of Java expressions. This implies that all declaration types are Java classes, and the evaluation of concrete attributes during a transformation step is done by calling corresponding methods. Then, generating a node or an arc instance the corresponding attribute types instances are generated. The instance of each declaration consists of a declaration type, a datatype (Java class), and either a Java expression or a constant value. The class name of a class would be indicated by *CN String "Person"*. A Java expression usually occurs in right-hand sides of rules. It must be evaluated when the corresponding rule is applied to a host graph. Such an expression is given by e.g., s.toUpperCase(), where a given string s is converted into capital letters. Variables like s are associated with concrete values. There are some requirements according to attributes:

- In the left-hand sides of rules only variables or constant values may occur.
- In the right-hand sides of rules, arbitrary Java expressions may occur. Variables have to be already declared in the left-hand side of the rule.
- In the host graph only constant values may occur.

Graph transformation In order to apply a rule to a host graph, first a match between all graph objects of rule's left-hand side and the host graph must be found. At the beginning, a match is created by the method Match createMatch(Rule r) resulting in an empty match. According to the underlying theory it is required that such a match is total. In AGG the graph matching problem is formulated as a constraint-satisfaction problem. A suitable method is available, denoted by boolean nextCompletion(). This method can be used alone, or additional user input can be regarded for a total match. User defined mappings can be defined by the method addMapping(GraphObject o, GraphObject i).

A transformation step is initiated by the method Morphism executeMatch (Match m) of the class Step. This method returns the morphism from the rule's right-hand side into the now transformed host graph. A rule Match getRule() is applied to a host graph Match getImage() under consideration of Match. The calculation of the resulting host graph is done *in place*. This means, that the host graph is directly manipulated.

Transforming a graph with attributes means to evaluate the attributes. First of all it is necessary to find a total match on the graph objects. Then, either constant values can be matched, or variables can be associated with concrete values of the corresponding graph objects of the host graph. According to the available attribute handler for Java expressions evaluating variables means to evaluate the corresponding Java expression occurring in the right-hand side of the rule for the variable in the left-hand side under consideration of the variable assignment given by the match.

5.1.2 Constraint Solving by PARCON

There is no interface to PARCON as described for the AGG system in the previous subsection. The reason is, that PARCON is implemented in C and therefore, it is used as a server. The communication between GENGED and PARCON is done via Sockets allowing multi-threading. This communication is based on input and output streams of strings. The input are constraints under consideration of the PARCON syntax and the output is given by one solution of these constraints. In this sense, we regard the syntax of PARCON as the interface (for the GENGED high-level constraints). This is the reason why we present several constraint types together with their syntax. A detailed description can be found in [Gri96].

Solving methods The major aim of PARCON is to allow for the treatment of non-directional constraint nets, cyclic nets, inequations, interval iteration, and to avoid backtracking. PARCON combines several solving methods to handle these tasks. The basic algorithm is taken from a method called *Unplanned Firing*. This method offers the property to find solutions for arbitrary variables occurring in functions. Moreover, it implies the availability of *Iteration Method* features which are necessary for calculating zero-positions of functions. In general, the *Iteration Method* allows the treatment of interval iteration. The *Unplanned Firing* method allows the treatment of non-directional and cyclic constraint nets. The *Unplanned Firing* method is extended by time stamps assigned to variables if their value has been changed. The value of that variable will be changed which is the oldest one.

In addition to *Unplanned Firing* and the *Iteration Method* PARCON considers *Label Inference* allowing the marking of variables with an interval of valid ranges. For each variable the default interval is given by $(-\infty, \infty)$. Solving a constraint, PARCON does not only calculate a concrete value, but also derive statements about interval limits for this variable from limits of further variables. The *Label Inference* method offers the possibility to treat inequations as an own constraint type in the sense that they are mapped onto restrictions of interval limits.

The *Label Inference* method together with an algorithm, called Waltz-algorithm [Wal72, Wal75], and the *Iteration Method* realizes the interval iteration.

The Waltz-algorithm allows the refinement of each constraint and variable until no new results can be achieved. The generated values for the involved variables represent the solution of the constraint net. A further aspect of *Label Inference* is the avoidance of backtracking, because each variable of the constraint net has knowledge about the valid range. By propagation of ranges it is possible to recognize and avoid contradictions and backtracking in time.

Common Properties of ParCon Based on the underlying solving methods described above we will emphasize the common properties of ParCon which result partly from these methods:

Logical connection by disjunction and conjunction

: Usually the connection of several constraints is done by a logical "and", because constraints have to be satisfied concurrently. ParCon allows the connection of several constraints also by a logical "or". In this way it is possible to describe that two rectangles must not intersect. The one rectangle has to be placed always above or below and left or right the other rectangle.

: Using disjunctions, for each alternative ParCon makes a copy of the constraint net which is then calculated separately. When for each copy valid variable values are found, the alternatives are compared according to the degree of value changes. Then, the alternative is chosen as a solution where the variables are changed by minimum values.

Equations and inequations (*Label Inference*)

: In contrast to the so-called Blue-algorithm ([FMB90] Delta-Blue) the solving algorithm of ParCon is able to treat not only equations but also inequations. This property is very important for graphical constraints where often minimum and maximum distances of graphical objects are specified.

Non-directional constraint nets (*Unplanned Firing*)

: During the specification of constraints the user does not have to specify the evaluation order. ParCon can decompose the constraint with respect to all occurring variables.

Least astonishment principle (time stamps and minimum values)

: Because the user has not to describe the evaluation order ParCon has to decide which variable must be changed to satisfy a violated constraint.

One of the main intentions of PARCON has been to choose such variables which complies the expections of the user as well as possible. The least astonishment principle can be understood as follows: PARCON considers the time a variable already keeps a value. Then, old values are preferred to be changed and it is prevented to keep new values which result from user input or calculations.

The least astonishment principle is additionally taken into account when new values of variables have to be evaluated. PARCON changes these variables by a value that is as minimal as possible. This concerns not only scalar variables but also point variables. Values are calculated with respect to the euclidic distance.

Treatment of cycles (*Unplanned Firing*)
: During constraint solving it is possible to get into a cycle. It is necessary to change a variable that has already been changed since the last user input. In contrast to the Blue-algorithms the solving algorithm of PARCON is able to treat cycles. PARCON solves cyclic constraint nets by iteration until the variable values do not be change any longer. The required exactness can be obtained by input of a value ϵ. Hereby, ϵ denotes the minimum difference between the variable value to the previous one, that the variable is passed to be changed.

Interactive insertion and deletion of constraints (*Incremental Solver*)
: During working with PARCON the constraint net can be changed interactively. Considering this fact, the constraint solver is called *incremental solver*.

5.1.2.1 PARCON **Constraints**

PARCON allows the treatment of several constraint types like linear constraints or product constraints. They can be defined under consideration of the PARCON syntax. Before we explain the constraint types let us have a brief look to the PARCON syntax:

Variable acquaintenance: set *Variable Value*
: Variables can be declared together with a certain value. Variables are initialized by 0.

Constraint acquaintenance: add *Constraint*

A new constraint is conjunctively connected to previous defined constraints. The priority of constraints is set to 0 by default.

Constraint priority: add*n Constraint*

It is possible to declare priorities where *n* denotes the priority between 0 (highest) to 9 (lowest).

Logical *or*: add| *Constraint*

A new constraint can be announced to be disjunctively connected with previous constraints using "|".

Constraint deletion: del*Constraint Number*

ParCon generates for each constraint a unique constraint number. This number allows the interactively deletion of constraints.

The syntactical descriptions introduced so far are used to illustrate the most important constraint types. Thereby, we denote numerical variables by v_1, v_2, two-dimensional point variables by P_1, P_2, and numerical constants by c_1, c_2.

- Linear constraints look like $c_1 v_1 + c_2 v_2 + \ldots + c_n v_n \otimes c$. $\otimes$ is used for the relations $=, \leq, \geq$. This constraint type is mostly applied within our graphical application. It is used to express horizontal or vertical distances between graphical objects.
- Product constraints look like $v_1 v_2 = c$. They can be used, e.g., to express constant values for the area of objects. Using several product constraints with temporary variables it is possible to denote products consisting of more than two factors.
- Point-distance constraints look like $d(P_1, P_2) = v$. This constraint type equals the (euclidic) distance between two points and a value of a variable. Defining a constant value for the variable or limiting it, then the distance of the points are correspondingly limited.
- Point-equal constraints look like $P_1 = P_2$. They can be used to express equal coordinates of points. In contrast to the component-wise equality, e.g. $P_1.x = P_2.x, P_1.y = P_2.y$, this constraint considers additionally the least astonishment principle. Using the point-equal constraint prevents that from the one point the x coordinate is chosen and from the other one the y coordinate, in the sense that both points have to change their positions.

- Point-straight line constraints look like $aP_x + bP_y = c$. This constraint type holds the point P on the straight line, characterized by $ax + by = c$.

Several *low-level* constraints can be defined during one session. User defined values have to be set by new *Variable*. Afterwards the value can be defined by val *Value*. The solution is given by a list of strings indicating constraint variables of certain domains. These strings are decomposed according to its parts. For illustration let us assume a variable P1.x indicating the x coordinate of a point P1. After constraint solving, PARCON returns the string ~P1.x 7. The variable is denoted by P1.x and its new value is set to 7. This kind of communication is used by GENGED but hidden from the user. On top of PARCON syntax, a *high-level* constraint language is developed allowing to choose constraints and the graphic's properties. These and some more about GENGED is explained in the following sections.

5.2 The Design of GENGED

The architecture of the GENGED environment has been briefly proposed in Chapter 3 and reviewed at the beginning of this chapter. In this section we focus on the design of GENGED. The design has been influenced by the desire to achieve a specification tool supporting the visual definition of visual languages. Moreover, the possibility to test the visual definition is desired as well. This possibility is obtained by a generic syntax-directed VL editor for the editing of VL sentences. The input is a VL specification comprising an alphabet and a grammar. The alphabet allows the instantiation of symbols and links. The productions occurring in the grammar define the edit commands of the VL editor.

The main criteria which guided the design of the GENGED environment in addition to the basic requirements like the availability of a specification tool, have been plattform independence and user-guidance. Platform-independence we gained using the object-oriented programming language Java. The user-guidance is realized by the generation of alphabet productions. These productions reflect the dependencies of links defined between symbols in an alphabet. Moreover, they show whether the constraints are expressive enough or not.

The GenGEd environment comprises several editors. A further design decision concerns mutualities of these editors. These mutualities are briefly introduced in Section 5.2.1 together with the overall system structure of GenGEd. Graphics and graphical constraints can be defined using the alphabet editor. As mentioned in Section 5.1.2, the low-level constraints of ParCon are not user-friendly for constraint definition (cf. Chapter 3.1.3). This is the reason why we have developed a high-level constraint language introduced in Section 5.2.2. Furthermore, we explain the communication with ParCon in that section. Thereafter, we sketch data structures of the alphabet in Section 5.2.3, and that of the grammar in Section 5.2.4. Alphabet productions are the edit commands of the grammar editor. Their data structure resembles grammar productions. Thus, we concentrate on the generation algorithms in Section 5.2.5 rather than on their data structure.

5.2.1 The System Structure

The particular classes used for the design of GenGEd are coupled according to their tasks by *packages*. Thereby, the notion of a package as proposed by UML [Rat99b] is directly mapped to the package concept of the programming language Java. These packages and their relations given in Figure 5.5 represent the system structure. The main components are given by alphabeteditor, grammareditor and the intended VL editor package, called seditor. The alphabet editor contains again two editors, namely, the symboleditor and the connectioneditor. Also the grammar editor contains several packages. The package display contains the classes describing the components of a visual language, namely, the grammar, its productions, the sentences, etc. Each of these components consists of several objects which are defined in the package objects. Some further and independent packages are surrounded by a dotted rectangle. They are used by the editors.

The system is divided into several parts under consideration of the Model-View-Controller pattern [GHJV95]. The *model* is given by the package alphabet which is the basis for the whole system. It contains all data structures for an alphabet (and its instantiation) as well as a mechanism (listener pattern [GHJV95]) informing all related components about changes. Different *views* to the data as well as *control* structures form the second important part of the system. We

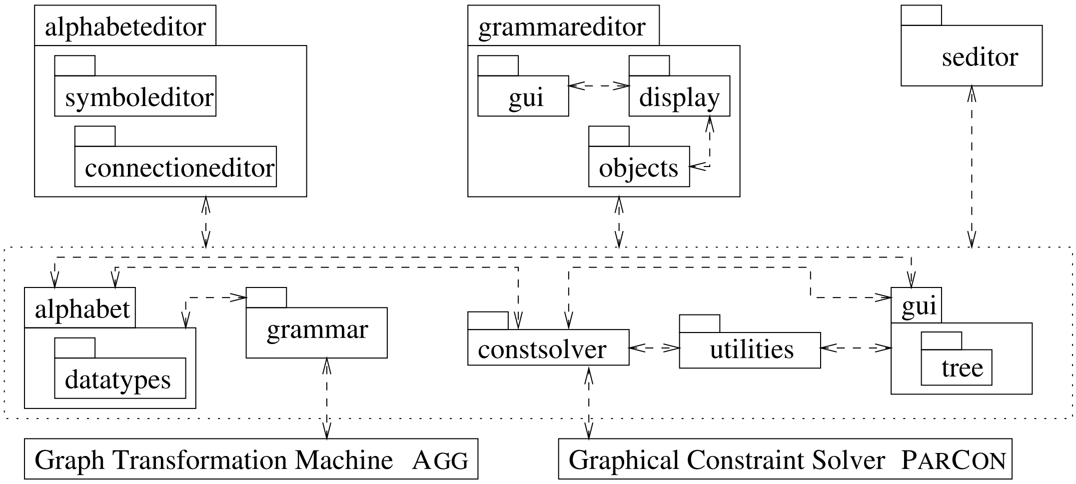


Figure 5.5: Structure of the GENGED environment.

have to mention the subpackage datatypes; here we hold all build-in datatypes like *String, StringList, Integer, Double, etc.* given as Java classes. The package grammar contains all data structures for a grammar as well as a mechanism (listener pattern [GHJV95]) informing all related components about changes.

A further important part is given by the package constsolver. This package manages the communication with the graphical constraint solver PARCON via sockets. This means that PARCON works as a server. Every drawing area is a client of this server and gets its own solver incarnation. Furthermore, the constsolver package offers the functionality for utilization of a high-level constraint language (HLCL) which is based on PARCON syntax.

Every editor has at least one drawing area for the visualization of symbol graphics. These graphics are vector graphics. All of them are derived from the abstract class GraphicObject in the package alphabet. Note, symbol graphics are distinguished from temporary graphics. Temporary graphics are, e.g., the reticule and the rubber band. The class gui.AbstractDrawArea implements a powerful drawing area. Each editor is required to derive a specialized class from this class. This is not an abstract class. It offers lots of attributes and methods concerning administration of graphics, levels of drawing, and variants of working modes.

Every editor offers a structured textual view on all its specific item names (gui.tree.GenGEdTree). E.g., these are symbol names and unique graphic names in the alphabet editor, and unique production names in the grammar editor. The class GenGEdTree is responsible for the names and their representations on the drawing area.

The GENGED specification tool offers the possibility to choose a natural language for explanations. These languages are given by “German” and “English”. The class utilities.LanguageText is used as database for all textual resources. All classes which have to be informed about changing the language have to implement the interface java.util.Observer and to announce themselves.

There is a comfortable undo mechanism available by utilities.GenGEdUndoManager. This manager extends the java UndoManager of Swing classes. The class Command is the basis for all objects which are to be treated by the undo manager. This class encapsulates every command and offers the possibility for undoing.

5.2.2 Constraints and Graphics

The package constsolver offers on the one hand the communication with PARCON via sockets. On the other hand it contains the implementation of a high-level constraint language (HLCL) realized by the class DatabaseHLCL. This HLCL allows the user to define own high-level constraints (HLCs) which are parsed and mapped onto PARCON strings. There are already various predefined HLCs available. The corresponding sources can be found in genged/resources/constraints/user.hlcl.

The HLCL distinguishes basic constraint types and composed types. Basic types are given by Value (numerical value used for composed constraint types or for local variables), ConstValue (constant numerical value used only as parameter of a constraint), and Point (comprises x and y coordinates of type Value). A composed type consists of a unique name and comprises several variables describing properties. The type of a property is either a basic type or a composed type.

Example 5.2.1 (*High-Level Constraint Definition*)
The following high-level constraint requires that a graphic object *a* of type “line” is outside a graphic object *b* of type “base”. Note, type “base” is a box graphic as proposed in Concept 3.2.3.

```
constraint Outside(Line a, Base b)

    Dialog(a, "English", "Line");              # Parameter
    Dialog(a, "Deutsch", "Linie");             # and
    Dialog(b, "English", "Object");            # constraint
    Dialog(b, "Deutsch", "Objekt");            # descriptions
    Description("English", "The line is outside of the object.");
    Description("Deutsch", "Die Linie liegt ausserhalb von Objekt.");

    Value w, h;                                # local variables
    [0] w = a.t.x - a.s.x;                     # calculation of width (w) and
    [0] h = a.t.y - a.s.y;                     # hight (h) with priority 0
    OperationMode(OR);                         # plug the OR mode; then with
                                               # priority 1 calculate:
    [1] a.s.x + w < b.lt.x;                    # a lies left of b
    [1] a.s.y + h < b.lt.y;                    # or a lies above b
    [1] a.s.x > b.lt.x + b.w;                  # or a lies right of b
    [1] a.s.y > b.lt.y + b.h;                  # or a lies below b
```

In the upper part of this example we can see the dialog for and the description of this *Outside* constraint. The dialog is given in English and in German. This supports the user's choice of the current language.

In the middle part local variables are used for the calculation of the width and the height of the line by using x and y coordinates of the line's start (s) and target point (t), respectively. As mentioned in Section 5.1.2 ParCon allows setting priorities in order to find solutions. This is used in this example.

The lower part represents the original constraints used to express that one object (here a line) is *Outside* another one. Note, the constraint variable b.lt.x indicates the x coordinate of the upper left corner (left top) of object *b*. △

As already mentioned, the basis for constraint solving are ParCon strings (low-level constraints) and constraint variables. The basis for constraint variables are graphic objects due to Concept 3.2.1. Every graphic object has a specific type, as for example, *rectangle, line,* etc. Figure 5.6 illustrates the graphic types available in GenGEd. On the instance level, these types and the constraint types ConstValue, Value and Point are the basis to generate unique constraint variables.

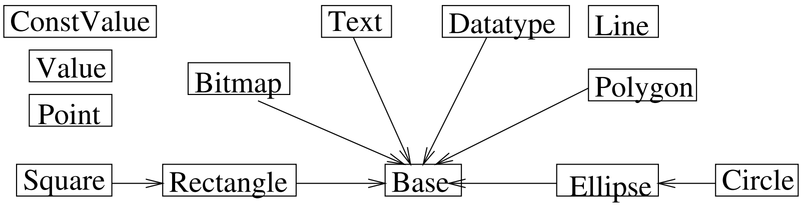


Figure 5.6: Type hierarchy for graphics.

The abstract class GraphicObject describes the graphic objects and their functionality. A high-level constraint is described by the class Constraint generating PARCON (low-level) constraints depending on dynamical graphic objects. The class ConstraintObject is used to encapsulate constraints. A constraint object stores all arguments. These are the objects participating in a constraint. The class ConstraintObject is associated with a constraint solver class ConstSolver. This class implements a mechanism maintaining constraint objects and PARCON constraints. Furthermore, it communicates with PARCON via ParconInOut by announcing variables and exchanging values. The dependencies of these classes are briefly illustrated by Figure 5.7.

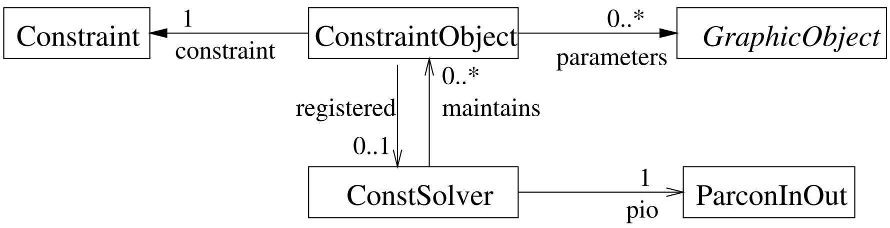


Figure 5.7: Class diagram: Dependencies between graphics, constraints and constraint solving.

Subsequently we illustrate constraint solving and the communication with PARCON by a small example. Therefore, assume the following graphic object for which suitable constraint objects (variables) are generated:

	Graphic Object 1	Graphic Object 2
identifier	`se17`	`se18`
basictype	`RECT`	`RECT`
pos = [x,y]	`[100,110]`	`[200,150]`
size = [width,height]	`[50,150]`	`[30,60]`

The variables are announced as constraint objects by the method registerObject in ConstSolver. Additionally, for these objects suitable PARCON equations (low-level constraints) are generated; they are registered according to their properties (x,y coordinates, width, height, etc.):

```
set se17.x 100 | set se17.y 110
set se17.w 50  | set se17.h 50
```

`se18` is registered analogously.

Then it is checked whether there are already some constraints defined for the types of these constraint objects. Assume we have a constraint expressing a minimum width for these object types (rectangles). Suitable PARCON constraints are generated:

```
add 1 se17.w >= 20
add 1 se18.w >= 20
```

These constraints are announced by the method registerConstraint. Note, the method unregisterConstraint countermands a constraint whenever the constraint-satisfaction problem is contradictionary. Let us further assume a new constraint expressing a fixed distance between these two constraint objects, or more concretely, between the reference points of these objects given by x, y coordinates (the upper left corners). By registerConstraint the constraints between our two objects are registered. For every registered constraint PARCON offers a unique identifier. This identifier together with the corresponding object identifiers and constraints are stored in a hashtable. This is important because the user is supported to delete constraints. On a drawing area the graphics can be manipulated by moving or resizing it. The setting of new values is done by the method resetObject. This manipulation results in new values. PARCON has to be informed about via ParconInOut. Calling the method solve or the autoSolve flag is being true, triggers PARCON to solve the constraints. For all objects that got new values, the new values are sent to PARCON by using the "new" and "val" statements of PARCON. The PARCON results are decomposed as described in Section 5.1.2.

In GENGED, the HLCL is available in the alphabet editor in order to build up symbol graphics and link constraints. Therefore, a suitable constraint menu is developed which is explained in Section 5.3.1.

5.2.3 The Alphabet

The alphabet editor comprises the symbol editor for defining the symbols, and the connection editor for defining the links between symbols. A constraint dialog supports the definition of graphical constraints that can be defined for symbol graphics as well as between two symbol graphics. The latter kind of constraints concern the layout of links. In the following, we explain the data structure of an alphabet and its instantiation.

The package alphabet comprises a class, called Alphabet, which is the main class for the treatment of an alphabet. It offers possibilities to cover and administrate alphabets. Every alphabet is enhanced with a time stamp and owner informations. It is stored using java.io.Externalizable. An alphabet offers various possibilities for the access of some alphabet constituents. These possibilities concern the access of symbol lists, datatypes, symbol constraints (internal operations) and link constraints (external operations) due to a given symbol, etc.

Symbols and links The basic object of an alphabet is represented by the interface AlphabetObject. Such an object is either a SymbolObject or a DatatypeObject. Each of these objects can be either a template or an instance. Figure 5.8 illustrates a class diagram for symbols and links (operations).

The class SymbolObject describes all lexical symbols of an alphabet. It may consist of several primitive graphics according to the box concept mentioned in Concept 3.2.3. They are defined by a GroupObject. The class DatatypeObject describes all attribute symbols. It encapsulates a graphic object described by a ClosedGraphicObject. Both, a GroupObject and a ClosedGraphicObject implements a GraphicObject for which constraints can be defined (compare Section 5.2.2). The constraints that are used to connect the primitive graphics of one symbol are signified by an *internal operation* holding all symbol constraints.

Links between two lexical symbols are defined by *external operations*. These operations regard the conceptual requirement that attribution operations may only be defined between a datatype object (attribute symbol) and a symbol object (compare Concept 3.3.1).

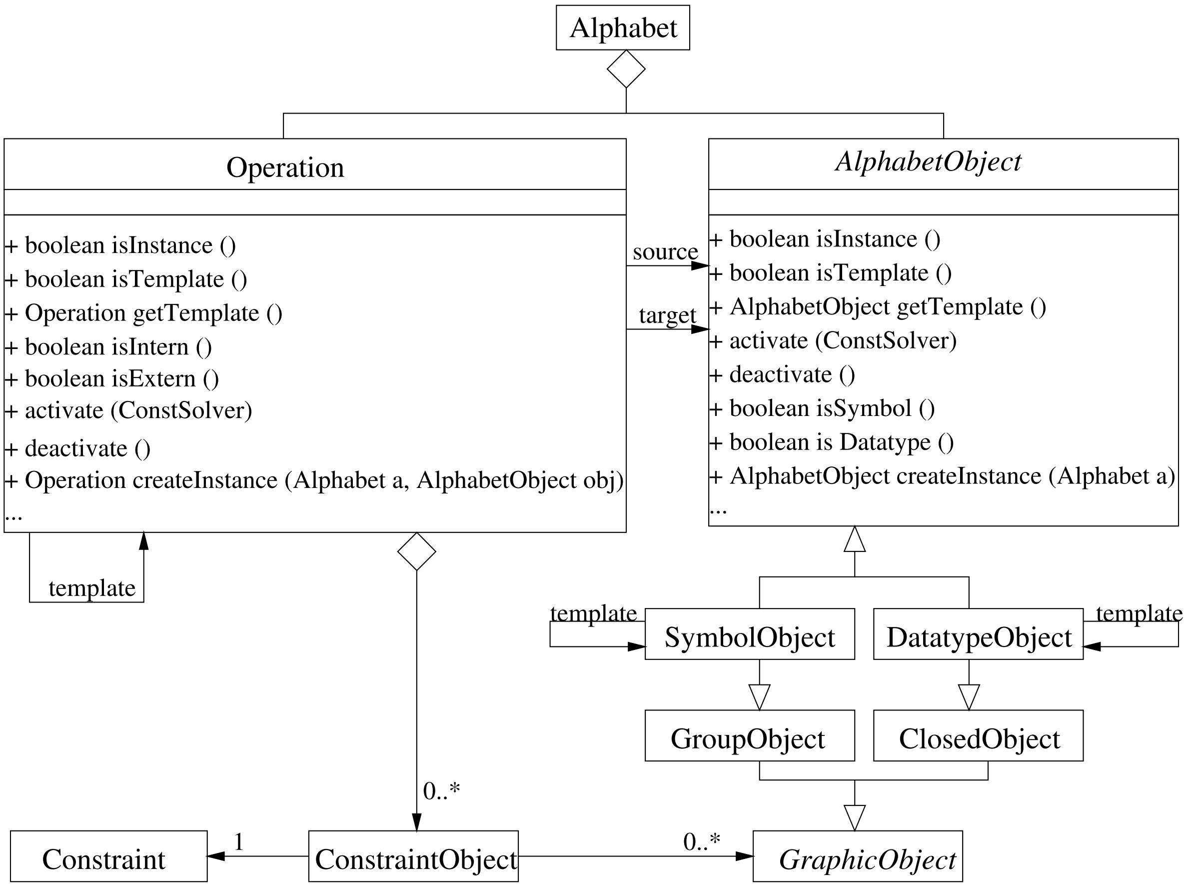


Figure 5.8: Class diagram: symbols and links (operations).

Alphabet and instances All symbol and link templates are given by an object of the class Alphabet. However, the instantiation of symbols and links is supported by the class DiagrammAlgebra, indicating a visual sentence, which extends an Alphabet. The aim of this diagram algebra is to maintain all instances of an alphabet. This is the reason that all structures of an alphabet can be used by DiagramAlgebra. An attribute alphabet is added in order to hold the corresponding alphabet for the instance. The methods setNewAlphabet and getAlphabet allow to write and read the alphabet. Figure 5.9 illustrates the class diagram for an alphabet and a diagram algebra.

The corresponding instantiation of a symbol or a link is supported by methods createInstance in AlphabetObject and Operation, respectively. The generated instance is to be added to the current diagram algebra by suitable add methods. Diagram algebras occur in the constituents of a grammar which is explained in the following section.

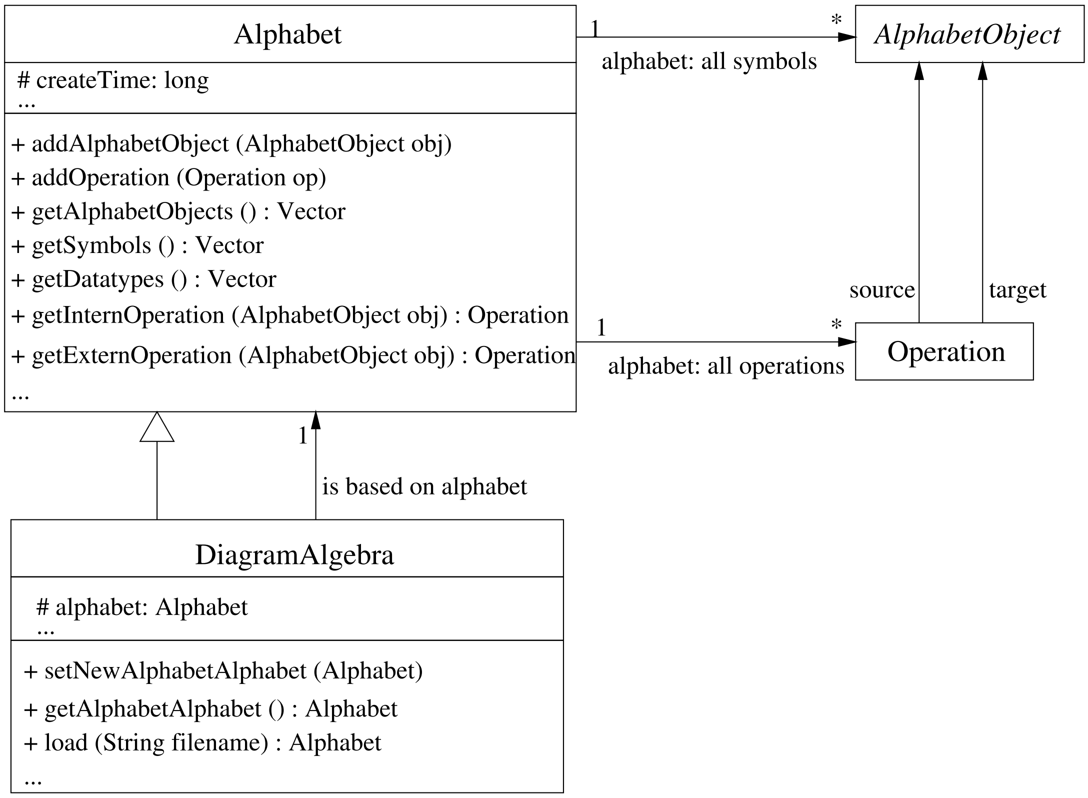


Figure 5.9: Class diagram: alphabet and diagram algebra.

5.2.4 The Grammar

The grammar editor is a generic editor. It is an implemented component which gets as input an alphabet. The alphabet is required to offer a set of symbols and links which can be instantiated (compare Figure 5.9). From the alphabet the grammar editor generates its edit commands which are called *alphabet productions*. They support the syntax-directed editing of the visual language-specific parts of a VL grammar consisting of a start sentence and a set of VL productions. Figure 5.10 illustrates the main building blocks of alphabet productions and a VL grammar. Note, the constituents of productions concern VL productions as well as alphabet productions.

In general, the left- and right-hand sides of productions comprises visual sentences. These are instances of the alphabet. According to the concepts mentioned in Chapter 3 production morphisms map instances of symbols from one sentence to another one such that links are preserved. Optionally, negative application conditions (NACs) may be bound to one production. A NAC is also

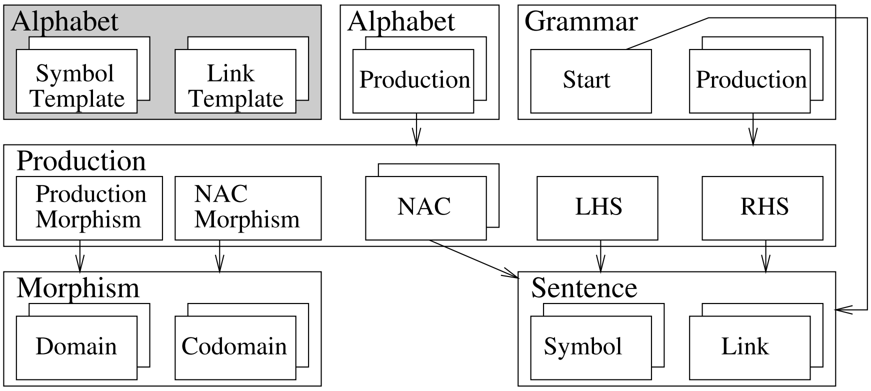


Figure 5.10: Constituents of alphabet productions and a VL grammar.

a visual sentence. The symbol instances of the production's LHS are mapped by NAC morphisms to symbols of the NACs. Alphabet productions allow the construction of a VL grammar consisting of a start sentence and a set of VL productions. The difference between alphabet productions and VL productions is only given by their expressiveness. In the following we introduce the data structures of the main building blocks of productions and sentences according to Figure 5.10. They form the basis for the data structure of a VL grammar. In Section 5.2.5 the algorithms for generating alphabet productions are introduced.

Figure 5.11 illustrates the class diagram for the constituents of productions. Except for the VL grammar all data structures used by the grammar editor occur in the package objects. The most important classes are given by GERule (production), GEDiagram (sentence), GEMorphism (morphism) and GEMatch (application of a production). These classes are based on GEObjects implementing the observer pattern [GHJV95]. The class diagram for GENGED productions (GEObjects) is similar to that of AGG as proposed in Section 5.1.1.

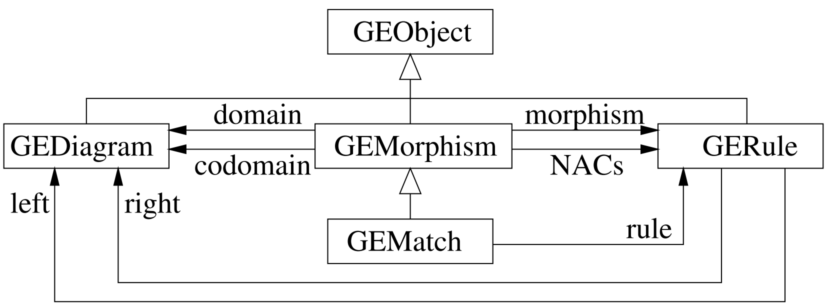


Figure 5.11: Class diagram: objects.

The data structure of a visual sentence is realized by the class GEDiagram. Because AGG generates with every transformation new nodes, the instances of symbols and links are stored in hashtables. There, nodes represent symbols and edges represent links (operations). Whenever the underlying (diagram) graph is changed by AGG the tables have to be updated accordingly. They are updated with respect to their graphics and constraints. Therefore, the default layout of symbol templates (link templates) is used in order to build up the corresponding constraint satisfaction problem. This is then solved by PARCON.

The class GERule models the data structure of a production. It is also based on a AGG-rule. It consists of two diagrams, one for the left-hand side and one for the right-hand side. Additionally, it consists of a rule morphism (GEMorphism) and several negative application condition morphisms (GEMorphism). A production can be protected. It can neither be changed nor deleted. A production can be locked, but it cannot be applied to another production. Furthermore, it can be selected in order to export it.

The class GEMorphism is based on a common AGG morphism and extends this morphism by diagrams, i.e., by domain and co-domain diagrams. Furthermore, a morphism stores the mappings between symbols allowing us to visualize the production morphisms.

A match (class GEMatch) is a special morphism storing (in addition to the domain and co-domain diagrams) the corresponding production. Note, this is due to the fact that (alphabet) productions can be applied to (VL) productions. This is achieved in the grammar editor and the VL editor as well. In the VL editor a production is applied only to one host graph (sentence), not to another production.

Datatypes (attribute symbols) As described in Section 5.2.3 datatypes are attribute symbols. They have to implement the interface genged.alphabet.Datatype. In the grammar editor VL productions are generated by applying alphabet productions. These alphabet productions contain only placeholders indicating the datatypes. Usually, such placeholders indicate variables of a specific datatype. This effects problems with the attribute component of AGG. The attribute component does not allow to map variables to variables, only to constants. In order to solve this conflict we do not map the datatypes when a production is applied to a production. The attributes are truncated and after transformation bound again to the symbols.

A further problem results from the AGG attribute component concerning constant values. They are not allowed neither in the left-hand side nor in the right-hand side of a production. We solve this problem by generating a Java expression which is to be defined by the method createDatatype in the corresponding datatype class. This expression is put into the right-hand side of productions for each new generated symbol. It is linked according to the defined attribution link.

Datatypes occur not only in production sides but also as production parameters. This means that productions may have parameters indicating datatype values. Such a parameter consists of a name, a type and a variable. Due to a variable, applying a production, the user is asked by a datatype-specific dialog for a corresponding value.

Grammar Based on the classes presented by Figure 5.11 we discuss now the data structure of a grammar for a specific visual language. Such a grammar is used by the VL editor (compare class seditor in Figure 5.5). Figure 5.12 illustrates the important classes for a grammar. The class VLGrammar realizes the construction of a visual language grammar. This is based on the class Signature which is an adapter [GHJV95] of an alphabet. This adapter allows the access of the string based alphabet. The signature is important for checking the correctness of a sentence (diagram) with respect to the defined alphabet. The class RuleCreator allows the generation of alphabet productions. The corresponding algorithms are explained in Section 5.2.5.

A VL grammar can be generated only on top of an alphabet. The Signature adapter keeps the symbol and operation names, and the domain and co-domain of the operations (the corresponding symbol names) as well. The names describe the abstract syntax level which forms the basis to construct a new grammar supported by AGG.

A grammar is generated by the constructor VLGrammar. This effects a new AGG grammar (GraGra()). This AGG grammar is enriched by types according to the signature. For each symbol a node type is created by the corresponding method createNode and for each link an arc type is created by the method createArc. Furthermore, for each attributed symbol a corresponding datatype is declared and added to the (symbol) node.

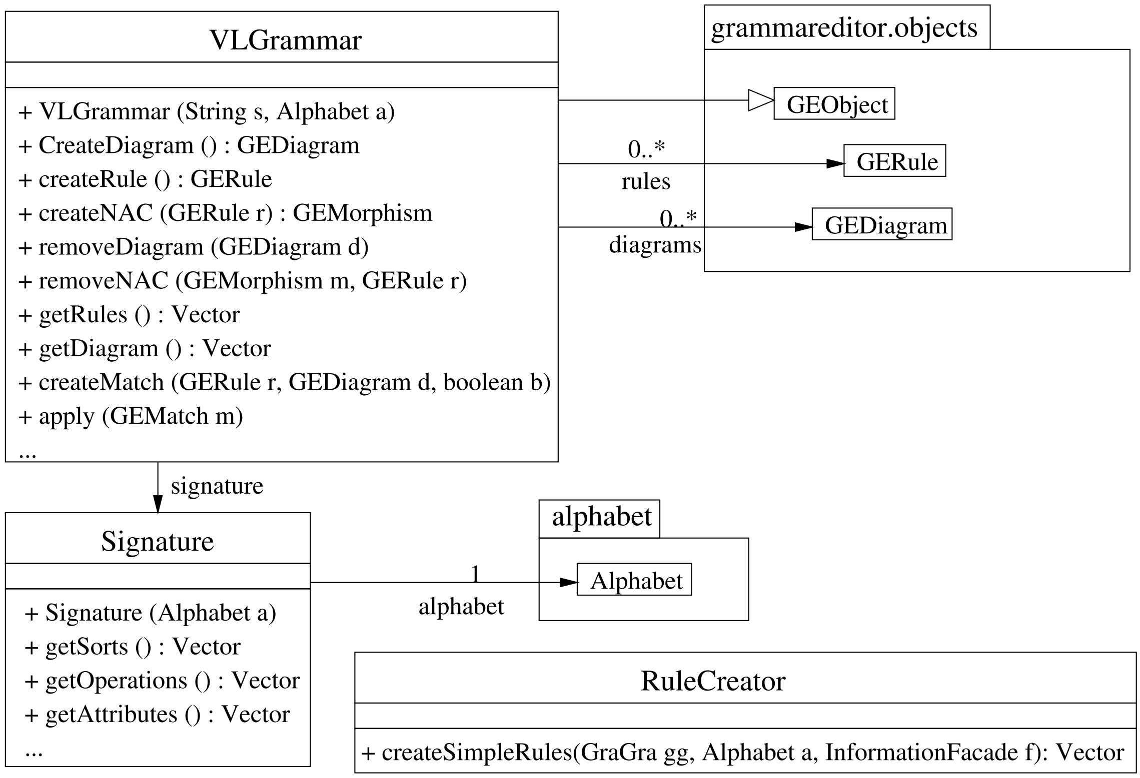


Figure 5.12: The package `grammar`.

The AGG grammar is the basis for the generation of alphabet productions by the class RuleCreator. Alphabet productions are generated due to the abstract syntax level. These generated productions are to be enriched by the corresponding graphics due to the concrete syntax level. In this sense, they get objects of class GERule. The VL grammar holds then a list of productions according to GERule and a list of diagrams (sentences) according to class GEDiagram. Note, a grammar holds not only rules (productions) but also a start diagram (sentence).

A new rule can be constructed using the method createRule of class GERule. The method createDiagram of class GEDiagram allows the creation of a new diagram (sentence). These are the constituents of a rule. By the method createNac of class GERule a negative application condition is generated for a specific rule.

Transformation The AGG system supports the definition of a match and the transformation of one sentence into another one. Figure 5.13 presents a sequence diagram for the transformation of a sentence (diagram) which is supported by AGG. A (empty) match is generated by the method createMatch. This method gets the rule r, the diagram d and a flag indicating whether the diagram belongs to a grammar production or to a VL sentence. In the grammar editor the diagram belongs to a grammar production. The data attributes are not regarded for transformation such that the flag is set to *false*. By apply the match is applied. Whenever it is not complete it will be completed by nextCompletion. The diagram is copied and AGG executes the transformation. Afterwards, every match must be destroyed supported by the method destroyMatch. This effects in the sequently toggled attribution of AGG graphs.

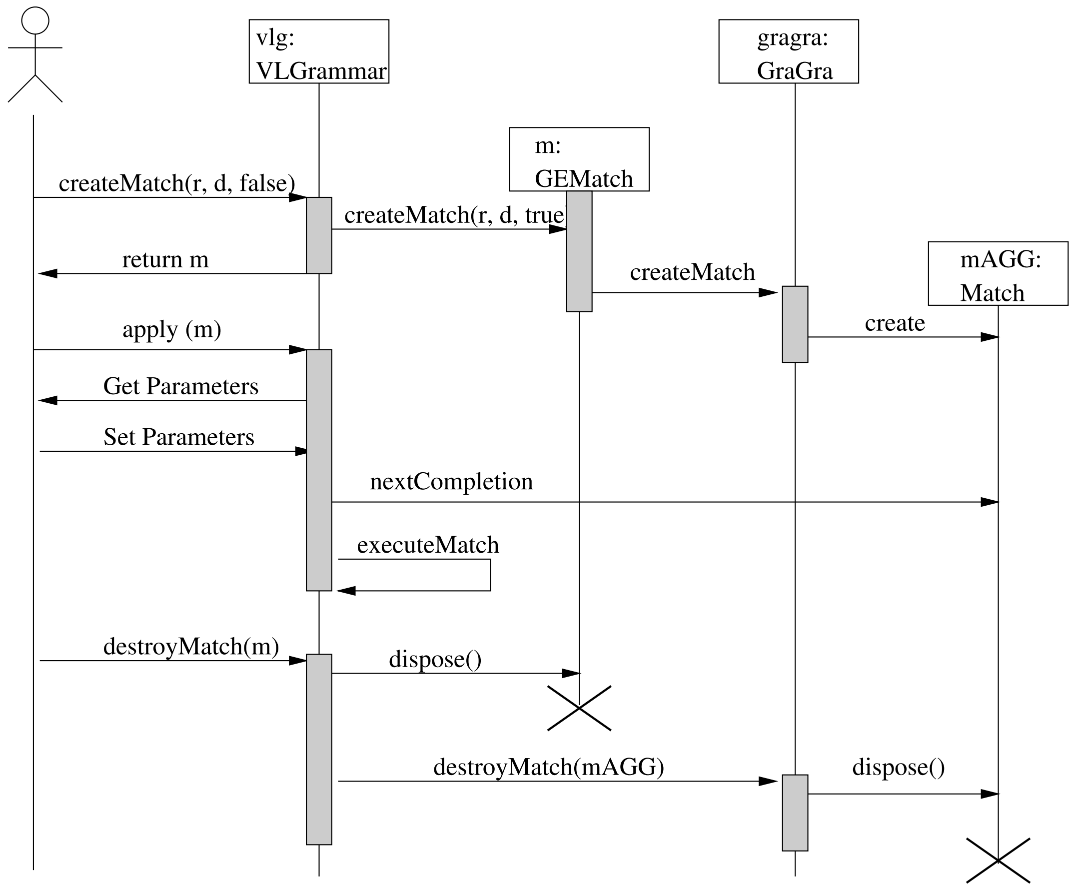


Figure 5.13: Sequence diagram: application of a rule using AGG.

The transformation result is checked according to the signature. If the transformation was successful the corresponding constraint satisfaction problem is constructed and the constraints are solved by ParCon. Thereafter, the transformation result is extended by its graphics. If either the transformation failed or the constraints cannot be solved, the previous state is restored.

5.2.5 Alphabet Productions

Alphabet productions express the insertion and the deletion of symbols and links. Furthermore, alphabet productions allow changing the datatype values. These productions offer user-guidance for constructing VL productions. They reflect the dependencies of symbols according to Concept 3.4.1. In this section we sketch the algorithms for generating the alphabet productions which are based on the abstract syntax level of an alphabet.

Algorithm 5.2.2 (*Alphabet Productions for Insertion*)
For each symbol template st occurring in an alphabet do

1. Create an empty production p with empty production morphism pm, and insert a symbol st' in the RHS of p. The symbol st' is instantiated from the symbol template st.
2. For each link template between two symbols $lt : st \rightarrow st_2$, having the symbol template st in its source do
   (a) Insert an instance of the link's target symbol (st_2') in the LHS and RHS of the production p. Extend the production morphism pm by an identic mapping from the symbol instance st_2' in LHS to st_2' in RHS of the productions.
   (b) Insert a link lt' in the RHS of the production p and connect both existing symbols by this link. The link lt' is instantiated from the link template lt.

$\triangle$

Let us illustrate this algorithm by an example, where we regard a part of the abstract syntax of the alphabet for class diagrams. Assume two symbols, these are *Class* and *Association*, and two links between these symbols: *begin: Association* $\rightarrow$ *Class* and *end: Association* $\rightarrow$ *Class*. We generate an alphabet production allowing the insertion of an association illustrated by Figure 5.14. We use the concrete syntax of symbols and numbers indicating the production morphism.

The algorithm for the generation of deletion productions is similar to the insertion algorithm except for negative application conditions (NACs). Next, we first propose the algorithm which is thereafter illustrated again by the class diagram example.

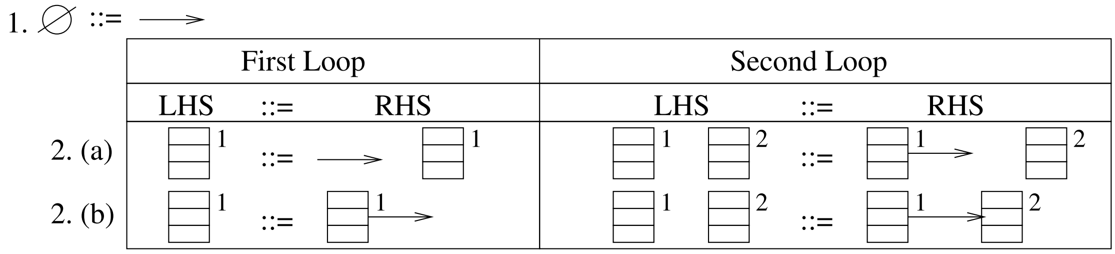


Figure 5.14: Generation of the alphabet production "InsertAssociation".

Algorithm 5.2.3 (*Alphabet Productions for Deletion*)
For each symbol template st occurring in an alphabet do

1. Create an empty production p with empty production morphism pm, and insert a symbol st' in the LHS of p. The symbol st' is instantiated from the symbol template st.

2. For each link template between two symbols, i.e., $lt : st \to st_2$, having the symbol template st in its source do

   (a) Insert an instance of the link's target symbol (st_2') in the LHS and RHS of the production p. Extend the production morphism pm by an identic mapping from the symbol instance st_2' in LHS to st_2' in RHS of the productions.

   (b) Insert a link lt' in the LHS of the production p and connect both existing symbols by this link. The link lt' is instantiated from the link template lt.

3. For each link template between two symbols, i.e., $lt : st_1 \to st$, having the symbol template st in its target, then create an empty NAC with empty NAC morphism nm from the production's LHS to this NAC.
   For each link template $lt : st_1 \to st$ do

   (a) Insert a symbol st' which is instantiated from the template st in NAC and extend the NAC morphism nm by an identic mapping from the symbol instance st' in LHS to st' in NAC.

   (b) Insert an instance st_1' from the link's target template in NAC and an instance of the link template. Connect both symbols by this link.

(c) For each further link template $lt : st_1 \rightarrow st$ do

i. Insert a symbol st'' in NAC which is instantiated from st.

ii. Insert a link instance and connect the symbol st_1 with st'' by this link instance.

△

The first two parts of the deletion algorithm express exactly the same as the insertion algorithm, however, the production sides are exchanged. The third part generates the negative application conditions which are necessary in order to obtain correct sentences.

According to class diagrams the alphabet production allowing the deletion of an association does not have NACs. The corresponding deletion production is generated similar to the insertion production where the production sides are exchanged. However, the alphabet production allowing the deletion of a class symbol is a production with two NACs because there is no link operation defined with the class symbol in its source. The construction of the alphabet production with NACs is illustrated by Figure 5.15.

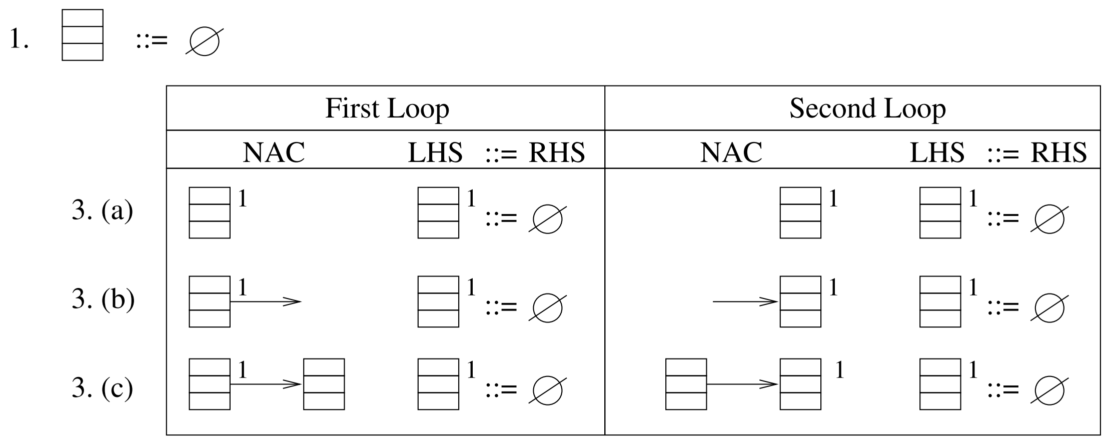


Figure 5.15: Generation of the alphabet production “DeleteClass”.

The alphabet productions define the edit commands of the syntax-directed grammar editor. Moreover, they allow building all visual sentences that are needed in VL productions. In the next section we give a brief survey of the user interface of GENGED. Thereby we show how VL productions can be constructed using the alphabet productions.

5.3 The User Interface of GENGED

In this section the GENGED environment is explained along our running example of class diagrams as discussed in Section 2.6 and Chapter 3. According to the conceptual model of Section 3.6 the environment consists of three components: the alphabet editor, the grammar editor and a VL editor. Although each component is built up similarly, their functionality differ. Figure 5.16 gives a survey of the user interface of each component, respectively editor.

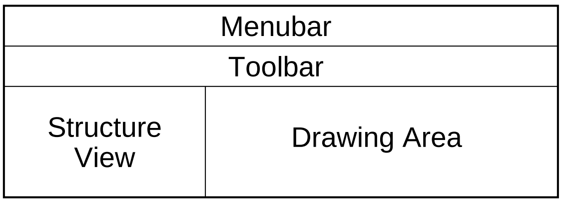


Figure 5.16: Common user interface of GENGED editors.

The menubar contains several top-down menus: file, edit, view, etc.; it depends on the editor which menu entries are available. The toolbar represents some menu entries for quick access. The structure view holds the textual representation of either alphabet or grammar constituents. Selecting one entry in the structure view this entry is visualized in the drawing area. In the following each editor is explained. Thereby, we briefly introduce its functionality which is illustrated by concrete examples.

5.3.1 The Alphabet Editor

For the visual definition of a VL-alphabet three components are available: the *Symbol Editor* (SE), the *Connection Editor* (CE) and a *Constraint Menu* (CM). The SE supports the definition of lexical symbols and the determination of implemented datatypes (attribute symbols). The lexical symbols can either be drawn or imported as a bitmap graphic. For each new symbol the user is asked to insert a name for it. Naming the symbols establishes implicitly the layout operations. These layout operations couple the symbol names with its graphical counterpart. When the symbols of a VL are determined, the CE can be used to determine the operations (links) for the logical part and the graphical constraints.

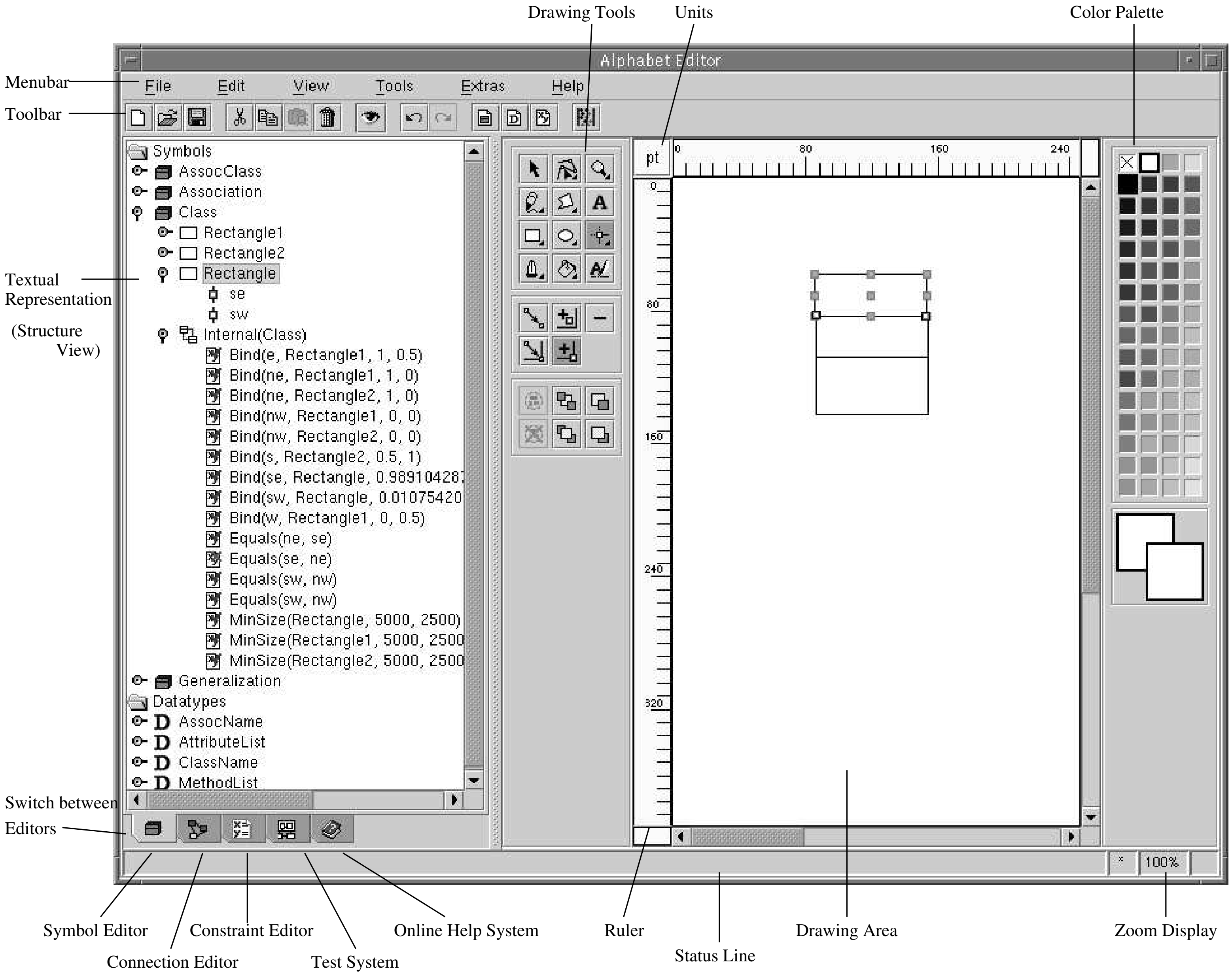


Figure 5.17: Alphabet editor with activated symbol editor.

5.3.1.1 The Symbol Editor

The SE as presented by Figure 5.17 is similar to common graphical editors. The main differences are graphical constraints that are used for the explicit grouping of several primitive graphics in order to establish one symbol graphic. The left-hand side of the figure shows textual expressions for the constituents of symbols. Selecting either a constituent or a symbol name, the corresponding graphic is visualized in the drawing area in the right.

The tools in the middle of the SE can be used to draw several graphics and to determine attachment points. Each graphic can get a color, selectable from the color palette in the right. These features are also available from the top-down menus. In the bottom, several buttons are shown. From left to right the buttons represent SE, CE, constraint editor, test system, and an online help system. The constraint editor which is intended to extend the constraint library by user-defined constraints and the help system are not implemented yet.

Each symbol consists either of a primitive graphic as given by a rectangle, circle, arrow, etc. or of several primitive graphics which are connected by graphical constraints yielding a box expression. The actual symbol presented in Figure 5.17 is a class symbol consisting of three rectangles, several attachment points and (internal) symbol constraints. The attachment points are ordered alphanumerically (see *Rectangle1*). They can be renamed according to our suggested box concept.The attachment points are automatically bound to the graphics by constraints (see *Bind(a,Rectangle1,0,0)*).

The toolbar of the symbol editor as illustrated in Figure 5.18 comprises the main operations to construct an alphabet as well as to establish symbols.

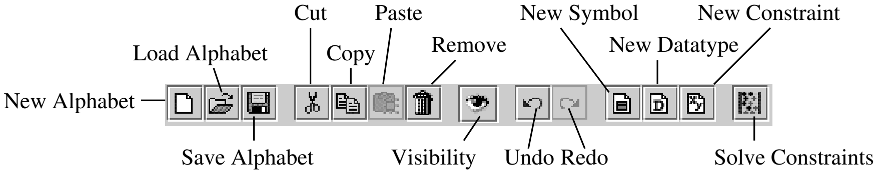


Figure 5.18: Toolbar of the symbol editor.

From the toolbar several operations can be selected. A new alphabet can be established, an alphabet can be loaded or saved. Operations on graphics are available like *remove*. It is possible to define a graphic to be invisible (*Visibility*). Such invisible placeholder symbols are introduced in order to connect graphics. The last operation can be *undone* and again, an undoing can be *redone*. The buttons in the right allow for the creation of a new symbol, a datatype or a constraint. Although PARCON is initially running permanently, it can be stopped and restarted by pressing the button in the right.

The operations of the toolbar can also be selected from the top-down menus. One menu, namely, the file menu, is illustrated in the left of Figure 5.19 in order to illustrate how bitmap graphics (GIF and JPG formats) can be loaded. Selecting *Import graphic* a dialog window pops up allowing to choose a bitmap. We have chosen two bitmaps which are illustrated in the right of Figure 5.19; it is a part of the symbol editor. The selected symbol in the left of the drawing area is the copy icon occurring in the toolbar and the right one is the icon for the constraint editor.

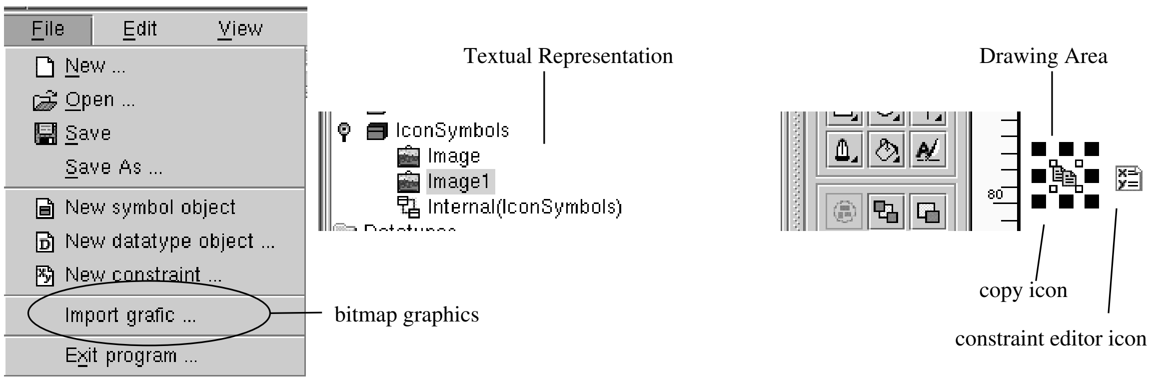


Figure 5.19: Symbol editor: importing bitmaps.

Tools for drawing a symbol graphic are available in the middle of the symbol editor between the textual representations and the drawing area (cf. Fig. 5.17). These tools are also available in the menu *Tool*. Figure 5.20 illustrates the meaning of each button. Note that there are again several tools behind some buttons. E.g., pressing the button for the rectangular tools, a palette showing all kinds of rectangles is available. These rectangles can be chosen for further progress. The tool that allows working on nodes, can be used for setting nodes that can be catched by the user furtheron.

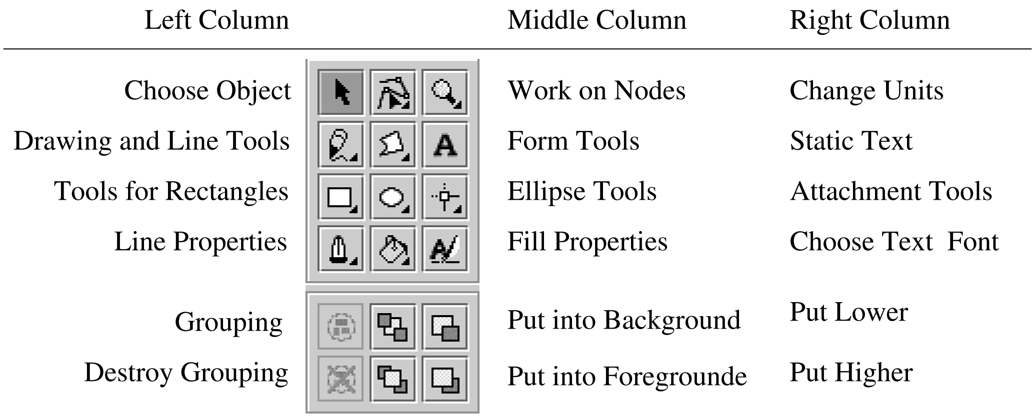


Figure 5.20: Drawing tools of the symbol editor.

The attachment tool supports the determination of attachment points for graphics occurring in the drawing area. Pushing the attachment tool button a palette with several buttons is available as illustrated in Figure 5.21. This figure shows the attachment palette in the left, and a selected rectangle and its nodes in the drawing area. The nodes of a rectangle correspond to the corners of a graphic according to our suggested box concept. They can be used as attachment points. We defined the upper left corner as an attachment point. Not only predefined nodes can be defined as attachment point. It is also possible to define an attachment point at any position of the current graphic.

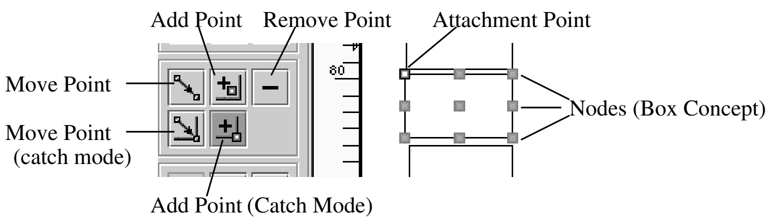


Figure 5.21: Attachment tools.

The constraint menu supports the connection of several graphics by graphic constraints. The available constraints are stored in a constraint library. This library can be extended by user-defined constraints. The definition of further constraint must be done textually under consideration of the high-level constraint language mentioned in Section 5.2.2 and fully described in [Sch99] and the PARCON syntax.

A broad variety of constraints is already available. The constraint database can be found in genged/resources/constraints/user.hlcl. How the constraints are made available in the constraint menu, is shown in Figure 5.22. A chosen constraint is shown in the top of the constraint menu. The constraint parameters must be defined by the user. This can be done visually by selecting the corresponding graphic of the drawing area or by choosing the graphic's textual representation. When all parameters are defined, the constraint can be checked whether it is contradicting or not.

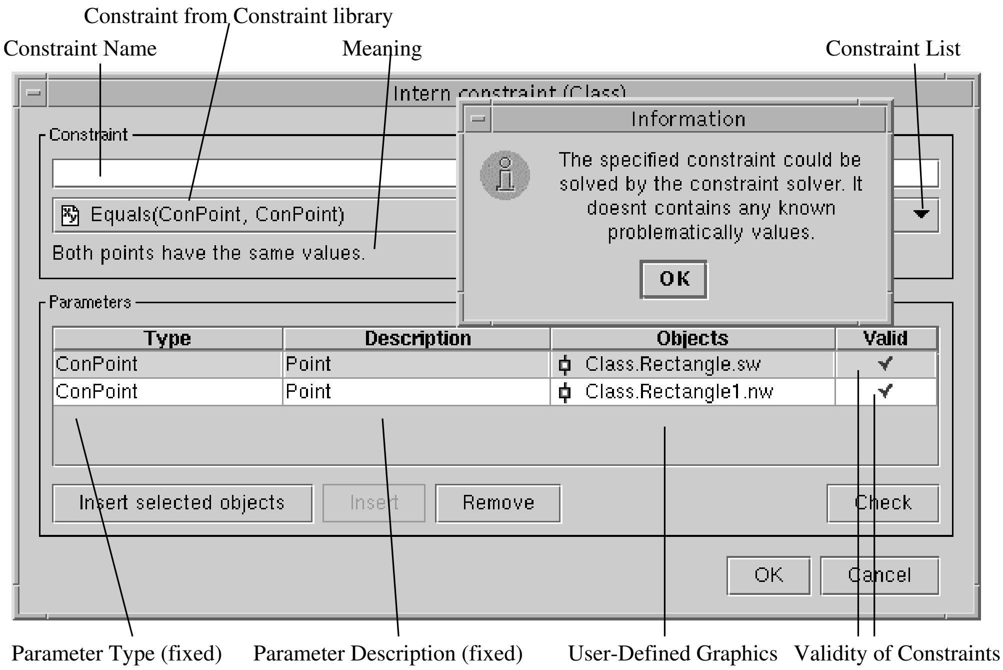


Figure 5.22: Constraint menu.

Figure 5.22 illustrates the definition of an *Equal* constraint for a class graphic. The three rectangles of a class symbol have to be connected by their corners. Furthermore, they should have the same width. Figure 5.22 illustrates the definition of an *Equal* constraint for connecting the lower left corner of the upper rectangle (*Class.rectangle.sw*) with the upper left corner of the middle rectangle (*Class.rectangle1.nw*). The other connections are defined analogously.

Datatype symbols Like the constraints, also the datatypes are textual (Java) components stored in a library. The available datatypes are *Double*, *Integer*, *String*, and *StringList*. Each datatype implements the interface Datatype.java.

The abstract class genged.alphabet.AbstractDatatype.java facilitates the implementation. Each available datatype is bound to the graphical user interface of the alphabet editor. In this sense, a datatype can be easily defined as a symbol. It depends on the chosen datatype how it is represented in the drawing area. For example, the integer datatype is represented by 0, a double by 0.0, a string by Text, and a string list by +. These representations can be changed by the user. Additionally, it is possible to define a new font, color, etc.

According to our class diagram example, we chose the *String* datatype for establishing the class name. We replaced Text by ClassName. The association name and the association class name are defined in analogy to the class name. The representations are also renamed. The properties defined for strings as text size, font, color, can be also defined for strings occurring the datatype StringList. We use the StringList datatype for the attribute list and for the method list of a class symbol. Their representations are renamed accordingly.

5.3.1.2 The Connection Editor

The symbols defined using the SE are available in the sequently toggled connection editor (CE) supporting the definition of links between symbols. Note that it is also possible to go back to the SE in order to define additional symbols or to define further attachments for an already established symbol. All changes using the SE are reflected in the CE.

The alphabet editor with the activated connection editor is shown in Figure 5.23. The left-hand side holds the textual representation of already defined links and the right-hand side is the drawing area where the symbols to be linked are illustrated. An operation can be defined by selecting the *New Operation* button (or by choosing it from the file menu). Then, the operation menu is opened allowing the selection of source and target symbols. Once we have defined source and target of an operation the textual representation is extended by an operation called *NewOperation* which can be renamed. The corresponding symbols are illustrated in the drawing area. Note, the user is forced to define at least one constraint for the operation, otherwise, the operation must be removed. The constraints can be defined using the same constraint menu as for the (internal) symbol constraints (cf. Section 5.3.1.1).

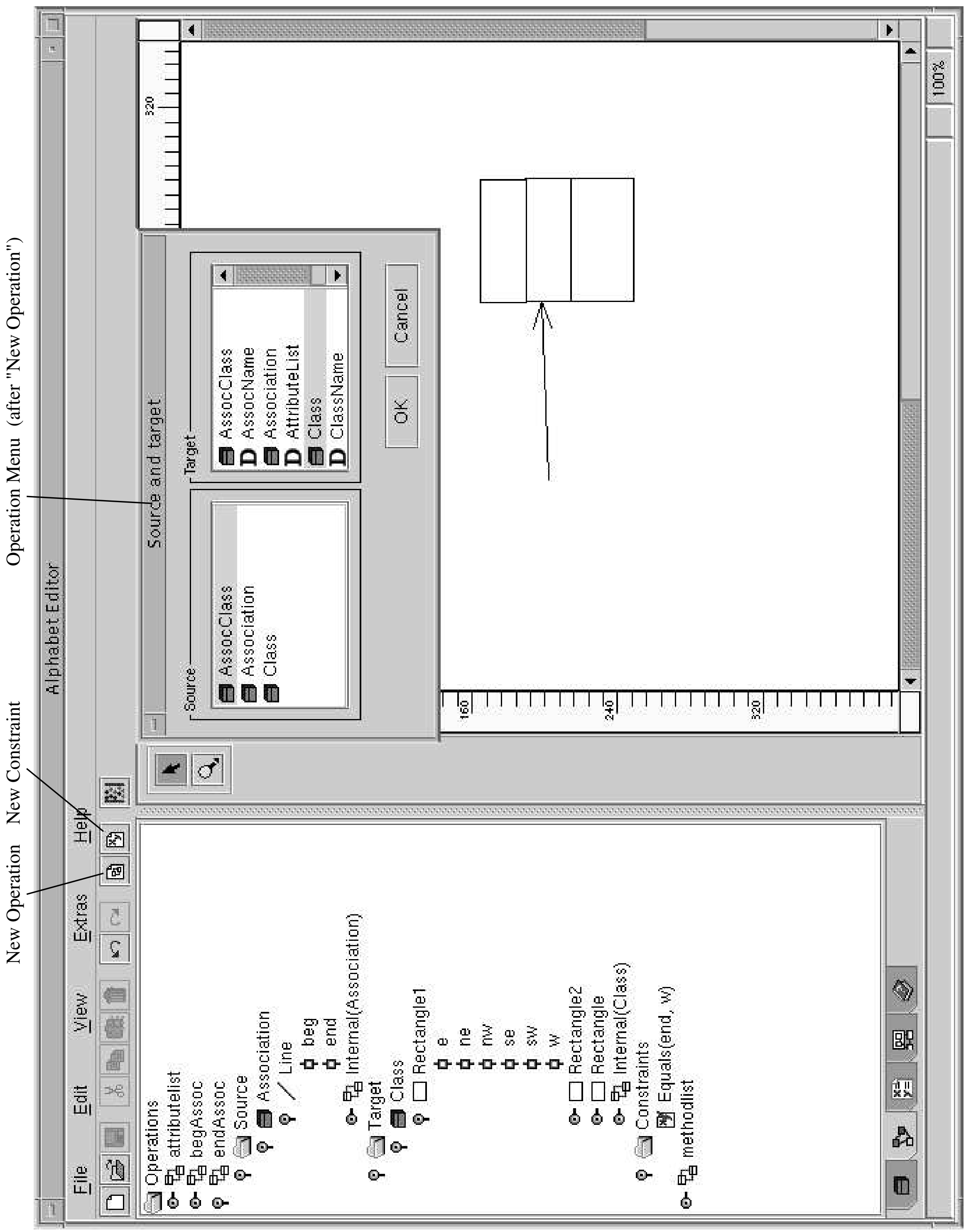


Figure 5.23: Alphabet editor with activated connection editor.

In the drawing area of Figure 5.23, the operation endAssoc is shown, where the association symbol is connected with the class symbol. A new operation is established. This can be seen on the operation menu. Remember please, an association class symbol is represented by an arrow and a vertical line. Thus we have to define three operations in order to connect this symbol with class symbols. Here, we start with the operation begAssocClass connecting the begin of the arrow with a class symbol. Pressing the OK button in the operation menu leads to an update of both, the textual representation and the drawing area along the chosen symbols. Thereafter, the corresponding constraints can be defined.

When all symbols and links are established using the alphabet editor, we can switch to the grammar editor supporting the definition of the VL grammar.

5.3.2 The Grammar Editor

The (syntax-directed) grammar editor (GE) supports the definition of VL grammars. The edit commands of the grammar editor are either alphabet productions which are generated from the VL alphabet, or VL productions already defined. These productions can be applied either to a start sentence or to several components (LHS, RHS, NACs) of a VL production which we call *visual sentences* in the following.

The grammar editor is shown in Figure 5.24. Like the alphabet editor, the grammar editor comprises a menubar, a toolbar, a structure view and a drawing area. The drawing area consists again of several drawing areas. The upper part visualizes the productions which can be applied, it is also called *production display*. The lower part comprises two areas where a production visualized in the upper part can be applied to. Hence they are called *working displays*. The working displays have broad functionality. They can be used to define the LHS and the RHS of a VL production, several NACs, and the start sentence of a grammar. The VL production tools help to solve these tasks, namely, to define morphisms from LHS to RHS as well as from LHS to NACs. The dividers available for the structure view, the production display and the working display support the dragging of the corresponding displays. Hence it is most important for the production display dragging the bar to the right where the NACs are visualized. In the structure view one can recognize a production with NACs by ; see the production called *Delete Class* in Figure 5.24.

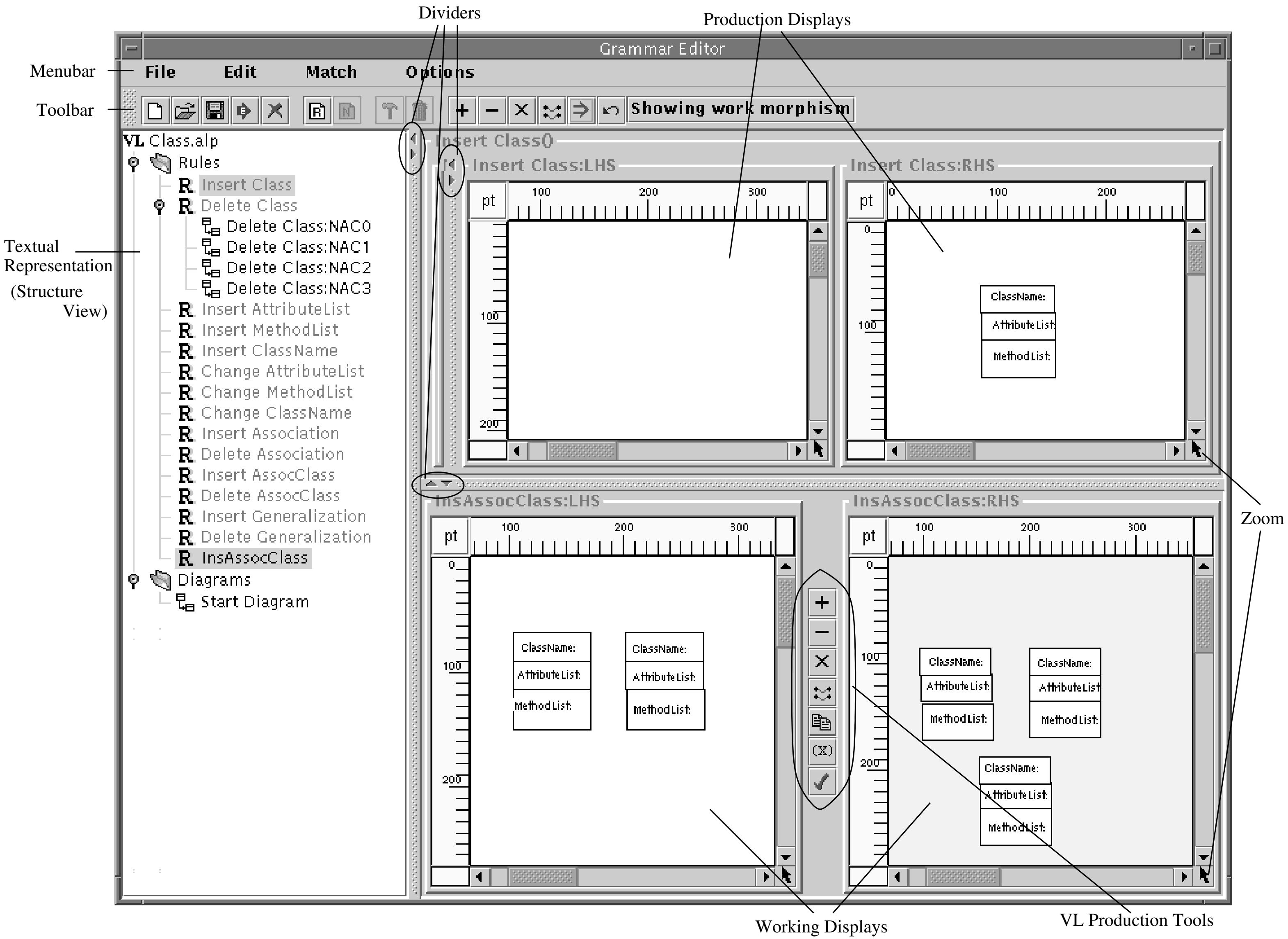


Figure 5.24: The grammar editor.

The toolbar of the grammar editor comprises the main operations to construct a VL grammar. The edit commands of the grammar editor except for those that are illustrated in Figure 5.25 are alphabet productions. Hence, either a new grammar has to be created which results in loading an alphabet, or an already existing grammar has to be loaded. In the latter case, the alphabet productions are already generated. Creating a new grammar or loading and saving a grammar, is supported by corresponding dialogs.

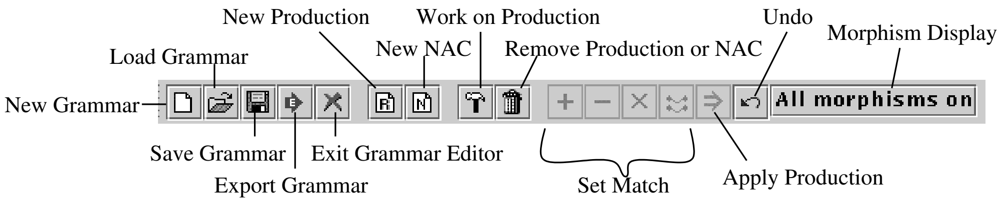


Figure 5.25: Toolbar of the grammar editor.

Whenever a VL production (NAC) should be created, first the *New Production* (*New NAC*) button has to be pushed and thereafter the button *Work on Production*. Both, a production as well a NAC can be removed from the working display and from the textual representation.

A (total) match must be defined component-wise for non-empty LHSs. For each symbol in the LHS, one symbol in the current working display has to be selected. The match can be defined by pressing the **+** button. Accordingly, the **-** button removes one match. The **X** button removes all matches. Pushing the  button, a match is automatically completed. This feature helps to solve the requirement for a total match. It is originally implemented by AGG and used by GENGED. A total match is the supposition of applying productions. All match and production morphisms can be visualized. Their visualization can be chosen from the corresponding menu in the right of the toolbar.

Visualization of morphisms There are several morphisms available within the grammar editor. These morphisms are illustrated by Figure 5.26. A morphism consists of a set of single mappings between symbols. These mappings are represented by corresponding numbers in each display. Whenever two symbols are mapped onto one symbol, the numbers of the target symbol are separated by colons. The morphism for the VL production is represented by red numbers; the others are visualized using blue numbers.

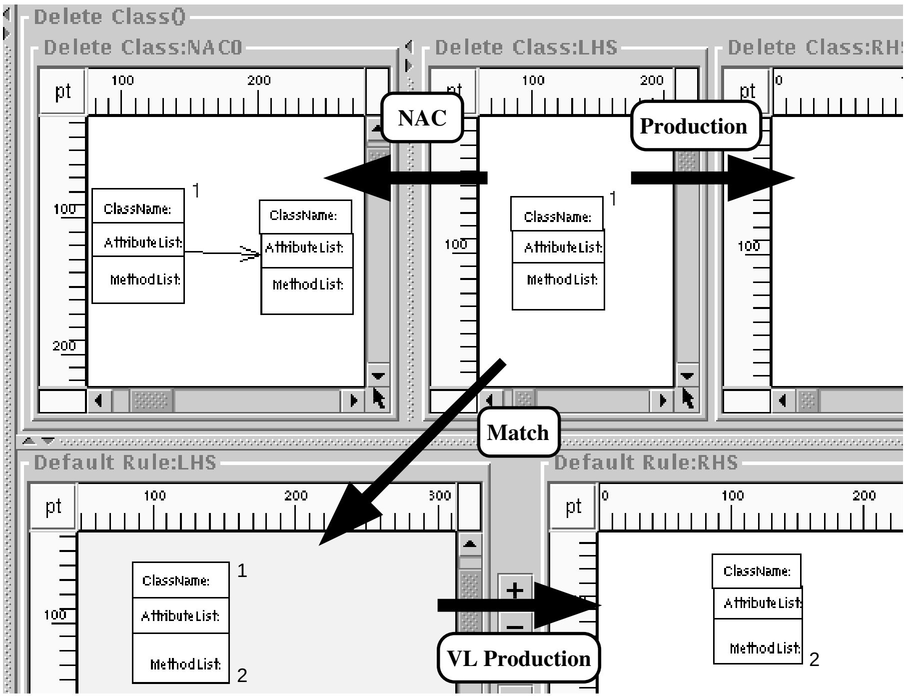


Figure 5.26: Visualization of morphisms.

The structure view is the textual representation of the VL grammar as illustrated in the left of Figure 5.24. From this representation, productions can be chosen for the visualization in the production display by pushing the left mouse button. Selecting a production and pushing the right mouse button opens a menu offering several properties. In Figure 5.27 a production called *ChangeAssocToAssocClass* is selected. This production is exported illustrated by the arrow in the structure view as well as by the hook in the menu. Although this production is already prepared for export, it is possible to work on it. Choosing the hammer in the menu, the production is taken over in the working display. Choosing the *N* symbol, it is possible to define a NAC for this production.

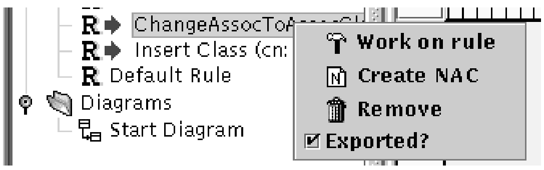


Figure 5.27: Structure view: menu after pushing the right mouse button.

The working displays and the **production tools** between the working displays support the construction of a VL production consisting of a LHS, a RHS and optionally several NACs. The NACs can be constructed in the right working display; accordingly the NAC morphism is to be set from the production's LHS to the NAC in the right. Additionally, the start sentence can be defined in the right working display. The working display offers several possibilities which can be chosen either by operations of the toolbar or by pushing the right mouse button.

The production tools are illustrated by Figure 5.28. The upper four buttons are similar to the match buttons of the toolbar. This means that it is possible to add a single mapping or to remove one or all mappings. Already defined mappings can be completed to a total morphism like the match morphism. This completion is a feature offered by AGG. When constructing a VL production, a good choice is to push the button copying all items of the LHS into the RHS.

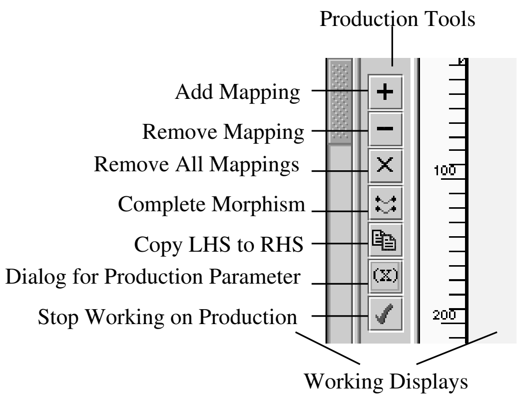


Figure 5.28: VL production tools.

A dialog window is opened when pressing on the **(X)** button. This dialog is important when a VL production should have parameters – these features are explained below. When a production is complete, it is incorporated in the structure view after pressing on the hook.

Datatypes and production parameters are important features to build up a VL grammar. As already mentioned in Section 5.3.1, the datatypes are given by Java classes peculiar for GenGEd. The datatypes used for a specific visual languages have to be established using the alphabet editor. According to our class diagram example, the datatypes for a class, e.g., are given by a class name (*StringDT*), an attribute list (*StringListDT*) and a method list (*StringListDT*).

In the grammar editor, the default values of datatypes are represented in alphabet productions whenever lexical symbols with canonical attribute symbols are added (cf. *Insert Class ()* of Figure 5.24). Defining a VL production where some data information should be given from the user, the default values have to be enriched by further variables, especially in the RHSs of VL productions. Whenever the user is forced to insert data information for applying the VL production, the variable of the production's RHS has to be defined as a production parameter. This is one conceivable application of datatypes and variables. It is also possible to define variables in all constituents of a production (LHS, NACs) as well as to define constant values. Note, that variables occurring in the RHS of productions have to be declared either in the LHS of the production or in the production parameter.

The alphabet production illustrated in Figure 5.24 is not useful for exporting it as a VL production. A class can be only inserted together with a class name. For the class name it is required that it does not occur in the current diagram. The first requirement can be expressed by a production parameter (the class name) and the latter requirement by a NAC. This is the reason for the production *Insert Class ()* to work on in the working display. Thereafter we click twice onto the default value *ClassName()* with the left mouse button. The result is shown in Figure 5.29. A dialog window pops up for defining a Java expression. Defining a variable as illustrated in Figure 5.29, it is evaluated by application of the corresponding production.

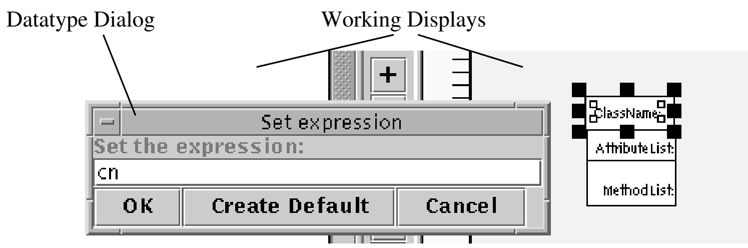


Figure 5.29: Defining a datatype variable.

The VL production for the insertion of a class is to be extended by a production parameter, namely, by the variable *cn*. In order to solve this task, we push the **(X)** button from which a dialog window is poped up. This window is illustrated by Figure 5.30. A production parameter consists of a datatype (e.g., *StringDT*), a name and a variable, initially a variable *x*. We push the *Add* button in order to change *x* by *cn* and thereafter the *OK* button. The just declared production parameter is visible when the production is selected for visualization.

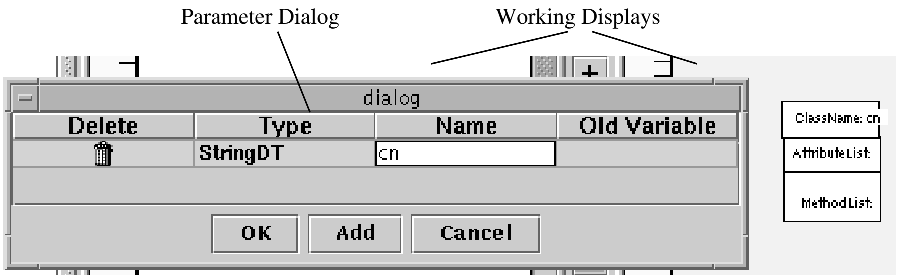


Figure 5.30: Defining a production parameter.

So far, we have explained the features available in the grammar editor. During the explanation we developed a part of the VL grammar for class diagrams. We omitted the definition of a start diagram because this is not necessary for class diagram editing. The whole class diagram grammar is saved on the disk. It is the input of the intended VL editor.

5.3.3 The Visual Language Editor

The syntax-directed VL editor as shown in Figure 5.31 is a test editor for the language designer. It is similar to the grammar editor. It contains also production displays but only one working display. Its functionality with respect to the application of a production is similar to that of the grammar editor. A VL grammar can be loaded. Its productions are then the visual language-specific edit commands of this test editor.

The test editor supports editing of visual sentences by application of VL productions. Usually, a grammar comprises not only productions but also a start sentence from which the editing can begin. Therefore, a VL production must be selected from the structure view. It is then visualized in the production display for defining the match. The match can be defined either component-wise for each symbol occurring in the LHS of the VL production or by pressing the button. This button completes the match. The user is informed whenever the match could not be completed.

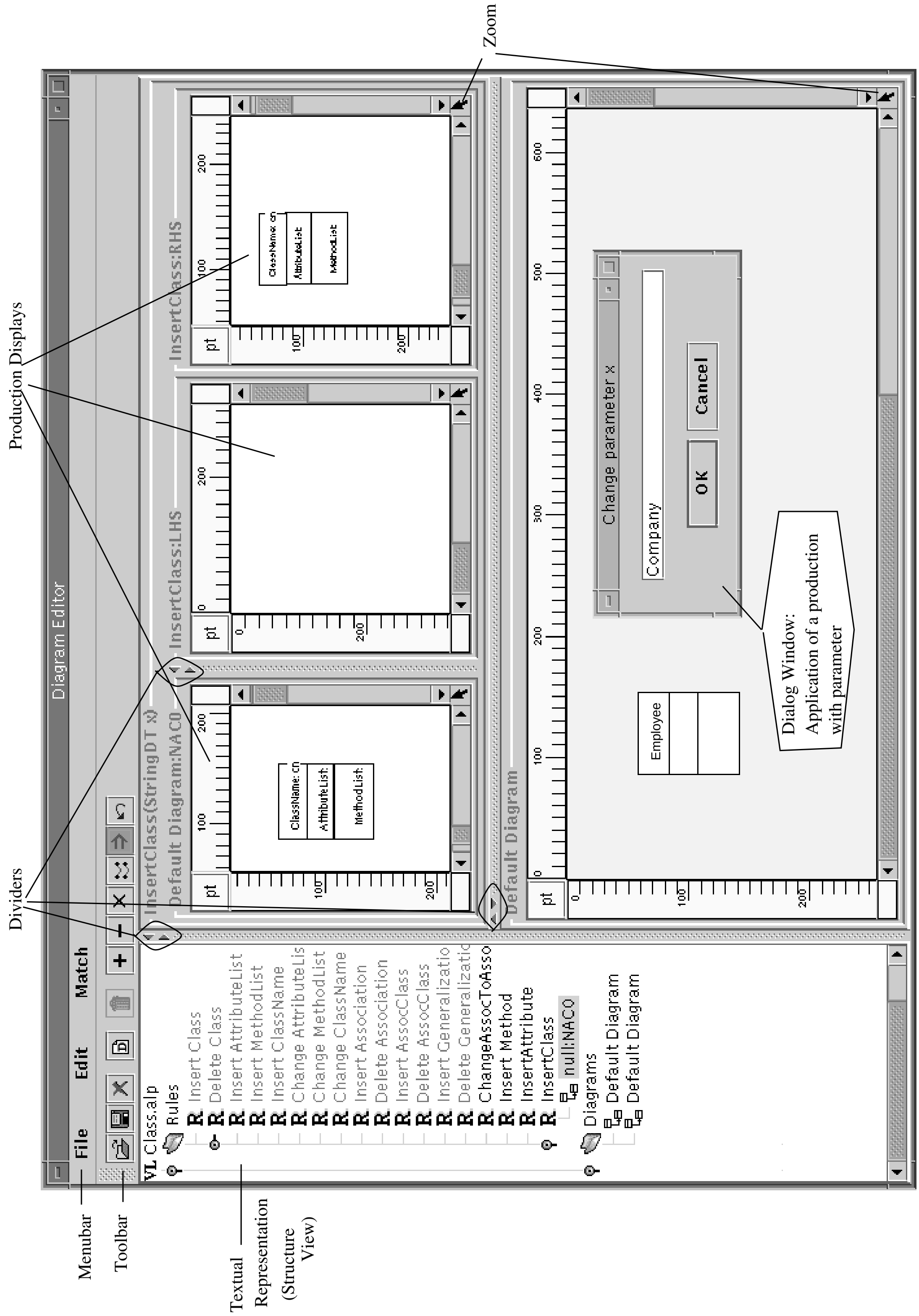


Figure 5.31: The VL (test) editor for class diagrams.

After defining the match, the visualized VL production can be applied onto the current visual sentence of the working display. This is illustrated by Figure 5.31, where the production *Insert Class (StringDT cn)* is applied. The production comprises a parameter for the class name. For this parameter, a dialog window pops up forcing the user to define a unique class name. The class name has to be unique because of the NAC.

As required, automatic layout adjustment of the adjacent graphical symbols is enforced if, e.g. one symbol is moved or resized. Moving or resizing a symbol, concerns the concrete syntax level of visual languages only. The abstract syntax level is not affected.

The user interface of the visual language editor for a common user is only slightly different to the test editor in Figure 5.31. It comprises one drawing area (the working area) for editing. The productions are not shown in the structure view which is not available here, but in the *Edit* menu. As before, the application of grammar productions is syntax-directed. A grammar productions can be chosen after pressing the *Edit* button of the menubar. A match is requested symbol by symbol due the symbol types of the (non-visible) symbols occurring in the production's LHS. As before, the application of productions with parameters effects a dialog asking the user to define concrete values for each parameter.

Chapter 6

Relation between Formal Aspects and the GENGED Environment

In this chapter we discuss the relation between the formal aspects of visual languages (VLs) proposed in Chapter 4 and the GENGED environment proposed in Chapter 5. Because the establishment and the treatment of the (formal) abstract syntax level of VLs is substantial for further developments, we focus on the abstract syntax in this chapter.

In Section 6.1, we show how symbols and links defined using the GENGED Alphabet Editor yield the abstract syntax of a visual alphabet. The abstract syntax of an alphabet is the basis for the generation of alphabet productions. Usually, these alphabet productions define a transformation system which enables us to *program* with productions as mentioned in Chapter 1. However, we focus on *languages* and thus, we regard the alphabet productions together with an empty start sentence as a grammar defining the *alphabet language*. In Section 6.2, we propose the algorithm for the generation of the alphabet productions.

The GENGED Grammar Editor supports the visual definition of VL grammars. Its language-specific edit commands are the alphabet productions. In

Section 6.3, we discuss several commands available in the Grammar Editor and illustrate why we restrict ourself to the abstract syntax level of VLs.

The transformation of visual sentences is executed on the abstract syntax level by AGG which uses a slight different graph model than GENGED. The implementation mapping between GENGED and AGG is proposed in Section 6.4.

6.1 The Visual Language Alphabet

The GENGED Alphabet Editor establishes alphabet of specific VLs. An alphabet has to meet at least Definition 4.2.2 with respect to its abstract syntax because it is our formal basis. According to our conceptual model (cf. Chapter 2), in the Alphabet Editor we distinguish lexical and attribute symbols as well as connection and attribution links. Regarding both syntactical descriptions, namely, the abstract and the concrete syntax, we get a double attribution illustrated by Figure 4.7. However, it is no problem to obtain the abstract syntax of an alphabet which will be explained in the following. Therefore, have a look to Figure 6.1 illustrating once more some symbols and links of the class diagram language meanwhile well-known.

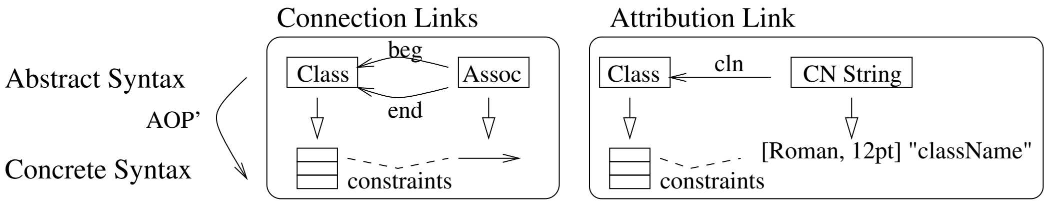


Figure 6.1: Symbols and links defined using the Alphabet Editor.

Lexical symbols consist of a symbol name and a symbol graphic, whereas attribute symbols comprise additionally the datatype of the corresponding attribute like String used, e.g. for the class name CN. Such a distinction is not necessary for the links given by directed edges between symbol names due to the abstract syntax, and by graphical constraints between symbol graphics due to the concrete syntax. Given several symbols and links, the following algorithm constructs an attributed graph structure signature for the abstract syntax of an alphabet. The graph structure signature obtained from this algorithm with respect to the symbols and links shown in Figure 6.1, is explained in Example 4.2.3.

Algorithm 6.1.1 (*Abstract Syntax of an Alphabet*)
From user-defined symbols and links using the GENGED Alphabet Editor, the following algorithm generates an attributed graph structure signature according to Definition 4.2.2.

1. For each lexical symbol name s **create** s in S_{GS}.
2. For each attribute symbol name $s' = < AttrName\ AttrType >$
   (a) **create** $AttrName$ in S_{GS}
   (b) **create** $AttrType$ in S_{AS}
3. For each connection link $op : s_1 \to s_2, s_1, s_2$ in S_{GS}
   create $op : s_1 \to s_2$ in OP_{GS}.
4. For each attribution link $op' : s' \to s$
   with $s' = < AttrName\ AttrType >$, s in S_{GS}
   (a) **create** $a_s : AttrName \to s$ in OP_{GS}
   (b) **create** $aop_s : AttrName \to AttrType$ in AOP △

Of course, algorithm 6.1.1 yields the abstract syntax level of an alphabet which is the basis for all further steps, especially for the generation of alphabet productions. Nevertheless, there must not be links between two attribute symbols. Moreover, the resulting signature has to be cycle-free. Both requirements are taken into account by the GENGED implementation. The first one is ensured by the Connection Editor allowing the definition of links. The latter one is ensured by the implementation according to Definition 4.2.1 (cycle-free signatures). A cycle-free graph structure signature forms the basis for the generation of alphabet productions.

6.2 Alphabet Productions

The algorithms generating alphabet productions have been already proposed in Chapter 5.2.5. These algorithms work correctly if the abstract syntax of an alphabet is given by a cycle-free graph structure signature. The alphabet productions generated with respect to the abstract syntax level are sufficient to generate all visual sentences. Because these alphabet productions are the language-specific edit commands of the Grammar Editor, they have to ensure the expressiveness of grammar productions. Hence it is necessary to show that the alphabet productions allow to define all VL grammars desired. Of course, a VL designer usually defines only a small set of VL productions. The VL defined is then a subset of the alphabet language. In the following, we propose the algorithm for language-generating alphabet productions with respect to the dependency relation of Definition 4.2.1. This algorithm forms the basis for a formal proof that has to be investigated in the future.

Algorithm 6.2.1 (*Construction of Alphabet Productions*)
Given a graph structure signature $GS = (S_{GS}, OP_{GS})$ with $s_1 \ldots s_n$ in S_{GS}, and $op_1, \ldots, op_m$ in OP_{GS}, $n, m \in \mathbb{N}$. The following algorithm constructs language-generating alphabet productions under consideration of cycle-free signatures (cf. Definition 4.2.1):

1. If s is the minimum element then it does not depend on any other symbol. Then, we generate a carrier set A_s for this symbol and one element $a_s \in A_s$ by an alphabet production: $\boxed{\emptyset} \Longrightarrow \boxed{a_s}$. Thus, all conceivable carrier sets of this sort symbol can be generated.
2. If s is not a minimum element, then it depends on another symbol s', i.e. $\exists\ s'$: $s' < s$. Hence, there is an operation $op : s \to s'$ with the sort symbol s in its domain. The generated alphabet production expresses this dependency: $\boxed{a_{s'}} \Longrightarrow \boxed{a_s \to a_{s'}}$ where $a_s \in A_s$ and $a_{s'} \in A_{s'}$.

   This process is repeated for all sorts that are less then s, i.e. such sorts where s depends on. The corresponding carrier sets are enriched accordingly. For each further operation symbol $op : s \to s''$ a new element $a_{s''}$ of the carrier set $A_{s''}$ is generated and the operation op^A is defined by $op^A(a_{s''}) = a_s$. Note that there is only one element a_s in A_s for sort s.
3. Other alphabet productions like editing productions can only generate algebras. This is due the construction of a pushout of graph structures as it is implemented in GENGED. Note that the match is not required to be injective.

$\triangle$

Let us illustrate the dependency of sort symbols by an example. Figure 6.2 (a) shows the abstract syntax of an alphabet for class diagrams. Note that we added a CD symbol indicating a frame for the whole class diagram. The CD symbol is the minimum element of the alphabet. There are no outgoing edges but one incoming edge representing the (connection link) operations. The alphabet production for the insertion of this symbol is constructed according to Step 1 of Algorithm 6.1.1: $\boxed{\emptyset} \Longrightarrow \boxed{\mathsf{CD}}$.

The alphabet production for the insertion of a Class symbol is constructed from Step 2, however, without recursion:

$$\boxed{\mathsf{1{:}CD}} \Longrightarrow \boxed{\mathsf{1{:}Class} \xrightarrow{in^R} \mathsf{1{:}CD}}.$$

Because of the dependency relation also the Assoc symbol has to be covered by a CD symbol. In Figure 6.2 (b), the alphabet production allowing the insertion of an assoction symbol is presented. It is generated from Step 2, but with recursion. The Assoc symbol depends on the Class symbol, and the Class symbol depends on the CD symbol. Thus, also the Assoc symbol depends on the CD symbol. Note that according to the two operation symbols beg and end, two Class symbols are generated in L and R.

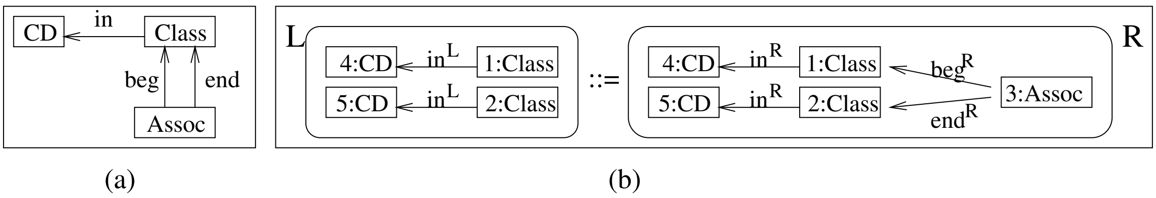


Figure 6.2: Abstract syntax of an alphabet (a) and alphabet production allowing the insertion of an association (b).

The algorithm for the construction of alphabet productions is defined on the abstract syntax level. The layout of symbols is achieved from the user-defined concrete syntax level of the alphabet. Due to the Conceptual Definition 4.2.14, each constraint variable keeps the name of the corresponding sort symbol. Thus, the layout of a concrete visual sentence as it occurs in production sides is achieved by constructing a corresponding constraint-satisfaction problem as proposed in Conceptual Definition 4.2.16. The constraints are solved and the graphic algebra describing the current layout can be constructed. Due to the constraint variables keeping the symbol names, the abstract syntax of a symbol is equipped with its concrete syntax via exactly one attribute carrier.

6.3 The Visual Language Grammar

Due to Algorithm 6.1.1 and match morphisms which are not restricted to be injective, the alphabet productions yield all algebras, i.e. visual sentences. These are the algebras partially occurring in a VL grammar. Hence, the VL grammar defined by applying alphabet productions is not restricted. We mentioned already, that the proof for this assertion exceeds the scope of this thesis and has to be investigated in the future.

The alphabet productions are the language-specific edit commands of the Grammar Editor, generated from the abstract syntax of an alphabet. Some other commands concern the concrete syntax level, namely, the moving and resizing of symbols. In the limits of the corresponding constraint-satisfaction problems solved by PARCON, these commands do not violate the syntax of visual languages.

Java expressions are used as datatypes. The corresponding commands and dialogs allowing the establishment of production names and corresponding production parameters are also implemented commands. Also they do not violate the syntax of visual languages. In the Alphabet Editor, the attribute symbols are merely declared. In contrast, in the Grammar Editor variables or constant (Java) values can be defined for a specific production. These values are then evaluated in the intended VL Editor.

Not all of these implemented commands can be specified by a graph structure grammar as proposed in this thesis. We need more powerful concepts which can be implemented by GENGED. Such concepts should allow to speak about properties of symbols and links in general, i.e. without the need to consider exactly all language-specific symbols and links, such that we are able to specify, e.g. the moving and resizing of symbol graphics. Figure 6.3 illustrates a production allowing the movement, however, this is a language-specific production.

The production parameter in Figure 6.3 indicates the new position of the Class symbol. It is moved according to the user input. This production illustrates the level on which we have to specify our features. This kind of specification does not fit to the visual definition of VLs. Nevertheless, in a VL Editor such productions can be hidden from the user as it is done, e.g. by DIAGEN.

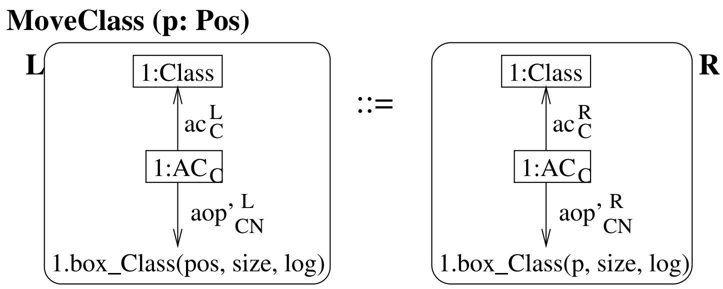


Figure 6.3: Production for moving a class symbol.

The transformation on the abstract syntax level of visual sentences is supported by the graph transformation system AGG. The implementation mapping is the topic of the next section.

6.4 Implementation Mapping

The transformation of the sentences' abstract syntax using AGG does not always result in a sentence with respect to Definition 4.2.4. AGG manages attributed and typed graphs whereas GENGED treats attributed graph structures (cf. Remark 4.1.3). Usually, typed graphs are formally defined by comma categories. However, the *typed graphs* of AGG are given in the user interface only. Internally, the types define a graph signature and the instances are graphs with respect to the graph signature. Before proposing the implementation mapping, we look at the constituents of an attributed graph G given by a tuple:

$$G = (N^G, E^G, AC^G, s^G, t^G, ac^G, Data, lab^G)$$

N^G indicates the set of nodes, E^G the set of edges. Each edge starts and ends at a node; this is indicated by source (s^G) and target (t^G) operations. They are defined as follows: $s^G, t^G : E^G \to N^G$. Attribute carriers are defined by the set AC^G. They are connected to nodes by the operation $ac^G : AC^G \to N^G$. Attributes are given by datatypes indicated by $Data$. They are coupled by a labeling operation to attribute carriers. Hence, there is a function $lab^G : AC^G \to Data$. In AGG also the edges can be attributed. This kind of attribution is not used by GENGED.

The following algorithm shows how to construct an AGG sentence from a GENGED sentence.

Algorithm 6.4.1 (*Implementation Mapping*)
Given the abstract syntax of a visual sentence which is an attributed graph structure as proposed in Definition 4.2.4. Note, the attribute carriers as well as the corresponding coupling operations are part of A^{GS}. A sentence Sen is mapped to an attributed graph as follows:

- All symbols are mapped to nodes:
  $\forall\, as \in A_s,\ A_s \in A_S^{GS} : as \mapsto n \in N^G.$
- All links are mapped to edges:
  $\forall\, op : A_s \to A_{s'} \in A_{OP}^{GS}, A_s, A_{s'} \in A_S^{GS} : op \mapsto e \in E^G$ with
  $s^G := op \mapsto dom(op)$ and $t^G := op \mapsto codom(op) \in G.$
- All attribute carrier elements are identically mapped to $AC^{\prime G}$.
- All datatypes A^{AS} are identically mapped to $Data$, such that
- the attribution operations lab^G are identical to that of A^{AOP}.

△

Figure 6.4 illustrates the implementation mapping for the abstract syntax of a class diagram consisting of two class symbols and one association symbol in between. Note, there are no attributes such that the corresponding carrier sets are empty and the operations are given by empty mappings. This sentence is an algebra with respect to the signature classdiagram$_{Abstr}$ of Example 4.2.3. However, the algebra A^{GS} is mapped to a simple graph illustrated in the right-hand side of Figure 6.4. Each symbol as of a specific carrier set A_s is mapped to a node n in N^G. Each operation $op^A(as) = as'$ defines an edge e in E^G. The domain defines the source s of an edge and the co-domain the target t.

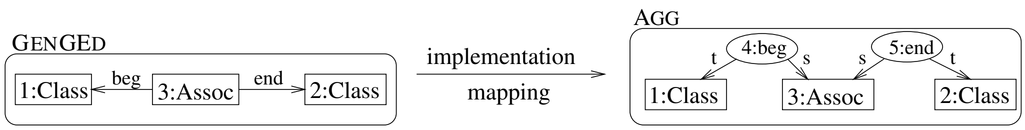


Figure 6.4: Example for the implementation mapping.

This implementation mapping may have side effects, caused by the transformation in AGG. Let us consider Figure 6.5 for illustration. In the right-hand side of this figure, an AGG graph is represented resulting from the applicaton of a production that allows to delete a class symbol. The left-hand side comprises the class symbol to be deleted and the right-hand side is empty. Assuming that no negative application condition restricts its application, it can be applied to a sentence as illustrated in the left-hand side of Figure 6.4. The implementation mapping and the transformation by AGG yield a graph whose inverse mapping is illustrated in the left-hand side of Figure 6.5. It is not a sentence with respect to the signature of Exampe 4.2.3 because the operation end is not defined.

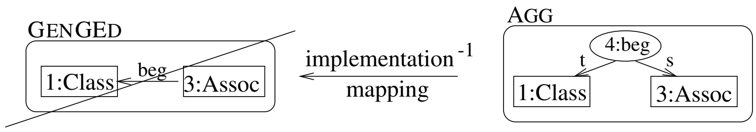


Figure 6.5: Side effects of implementation.

In the current state of GENGED, situations as described above are strictly forbidden. So, by the algorithm generating alphabet productions allowing the deletion of symbols (cf. Chapter 5.2.5), negative application conditions are added to deletion productions in order to define correct sentences only. However, the language designer is able to establish grammar productions without considering possible violations. Although a user of the VL Editor is able to apply a violating production, the transformation result of AGG is not accepted by GENGED. The previous state is restored and the user will be informed about that.

Chapter 7

Sample Visual Languages using GENGED

In this chapter we regard some visual languages (VLs) and sketch their specifications using the GENGED approach. Thus, we argue that our approach is sufficient to handle several visual languages. The class diagram language has been analyzed in Chapter 2.6. Its specification has been partly discussed in the previous chapters. In Section 7.1 we summarize and extend the specification of the class diagram language. Statecharts have been proposed by D.Harel in [Har87]. We regard a small part of the statechart language in Section 7.2. Clearly, the statechart language is a graph-like language and hence, it is similar to the class diagram language. Nevertheless, there are some more features like parallelism that have to be specified, too. Not only graph-like languages can be specified using the GENGED approach, but box-like Nassi-Shneiderman diagrams [NS73] as well. They are discussed in Section 7.3. According to the visual representation types of [BB94], Nassi-Shneiderman diagrams correspond to iconic languages because of the spatial relations. This language does not offer arrows or lines to relate symbols.

Other VLs have been already considered using the GENGED approach, but are not presented in this thesis. These are message sequence charts as proposed in Chapter 2.2 as well as algebraic High-Level Nets [PER95]. In a student's project all these languages have been analyzed and partly specified using the GenGEd approach. These languages as well as the languages treated in this chapter will be considered in more detail in [BENS99].

The definition of the sample VLs proposed in this chapter is only one of many possibilities. We present figures instead of screenshots to clarify also the abstract syntax levels of the corresponding definitions. Note that each symbol graphic can be moved and resized in the limits of constraints. The presentation of the sample VLs follows a unique scheme. We start with the proposition of the symbols which will be linked with respect to the abstract syntax level. In order to motivate the link definitions we first present the intended language-generating VL production. Except for the class diagram language which is already presented in the previous chapters, we discuss later on the links that define the dependency relation on symbols. Thus, they are important for the generation of alphabet productions. We omit the discussion concerning the construction of VL productions because this has been already explained in the previous chapters.

7.1 A Class Diagram Language

Many parts of the class diagram language have been already specified in the previous chapters. These parts are extended with respect to the analysis in Chapter 2.6. Figure 7.1 illustrates the symbols that are taken into account for the specification. These symbols form the basis to define the links between the symbols.

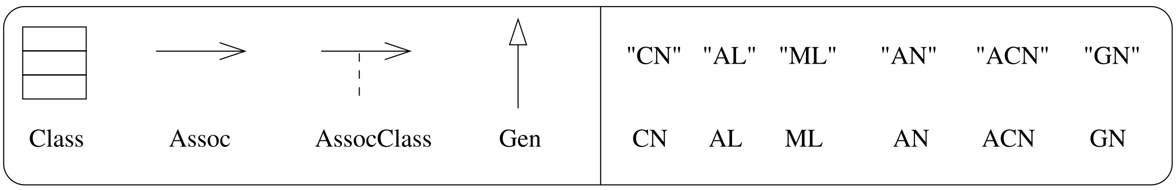


Figure 7.1: Symbols of class diagrams.

The four symbols in the left of Figure 7.1 represent the lexical symbols. The other symbols represent the attribute symbols that have to be linked to lexical symbols. We used some abbreviations in this figure. Assoc is used for the association, AssocClass for the association class, and Gen for the generalization symbol. For the attribute symbols of Figure 7.1 we used abbreviations, too. CN, AL, ML indicate the attributes of a class symbol, namely, the class name, the attribute list and the method list. AN indicates the association name and ACN the name of the association class symbol. A generalization symbol can be attributed by a name. This name is given by the GN symbol.

The symbols depicted in Figure 7.1 represent one possibility for modeling the class diagram language. The same is true for the abstract syntax graph of the alphabet as proposed in Figure 7.2 (cf. Remark 3.3.3). This graph illustrates the dependencies of the symbols due to the link edges. The dashed lines depict the concrete datatypes underlying the attribute symbols (cf. Chapter 3.1 and Remark 3.4.6). LOS indicates the datatype *ListOfStrings*.

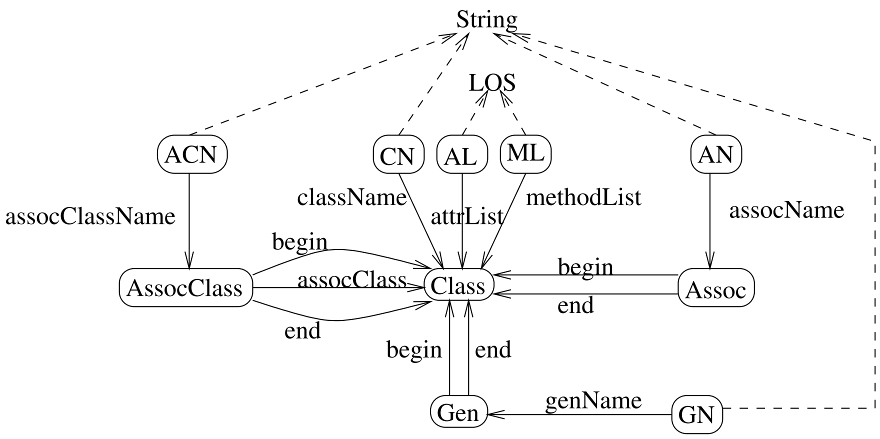


Figure 7.2: Abstract syntax graph of the class diagram alphabet.

In the following sections the dependencies are explained for each two linked symbols, inclusive the attribute symbols. In contrast to the other languages that will be treated in this chapter, we omit mainly the abstract syntax level, but concentrate on the discussions of Chapter 2.6. This means that the VL productions are in the fore.

7.1.1 Class Symbols

The class symbol is the central symbol of the class diagram language because it is not dependent on any other symbol. It has no outgoing link edges as shown in Figure 7.2. Hence, the insertion production (cf. Figure 3.8, page 59) comprises an empty LHS and one class symbol instance in its RHS. The production parameter enforces the user to define a class name if the production is applied. However, we have required uniqueness for class names in one class diagram. Therefore, we add a NAC to ensure this requirement (cf. Figure 3.12, page 62).

A production for the deletion of a (central) class symbol contains much more NACs due to the symbols that depends on the class symbol. The *DeleteClass* production in Figure 3.11 is based on two symbols only. These are the class symbol and the association symbol. Corresponding to the alphabet in Figure 7.1 and the abstract syntax graph in Figure 7.2, much more NACs are necessary for a production that allows deleting a class symbol. The deletion production together with its NACs is shown in Figure 7.3. The corresponding NAC morphisms are visualized by the number 1.

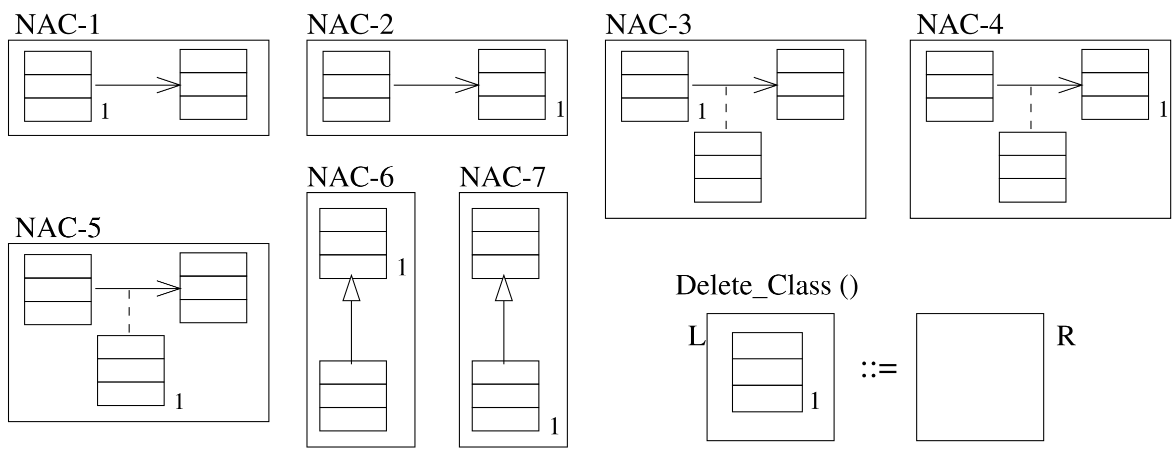


Figure 7.3: Production for the deletion of a class symbol.

In Chapter 2.6 we have required that the changing of an attribute symbol, e.g. a class name, effects an adaption of the surrounding rectangles. This happens in the limits of the constraints. Figure 7.4 illustrates the production for changing a class name. The variables cn and cn' indicate the old and the new class name. The numbers indicate the production morphism and the NAC morphism. If this production is applied to a given class diagram, the attribution link constraints enforce the enlargement of the rectangle. Furthermore, due to the symbol constraints (cf. Example 3.2.4, page 51), the rectangles enclosing the attribute list and the method list are adapted accordingly.

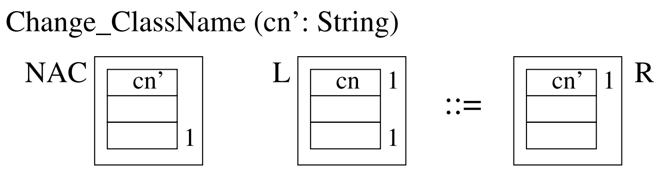


Figure 7.4: Production for changing the class name.

7.1.2 Class and Binary Association Symbols

An association symbol usually relates two class symbols. This relation is indicated by two link edges, namely, begin and end in Figure 7.2. A production that allows for inserting an association symbol is already presented in Figure 3.10 on page 61. Due to the association arrow, applying this production to one class symbol would yield a curious diagram with respect to the concrete syntax. This means that the arrow is not transformed to a polyarrow automatically. Hence, we have two possibilities. Either we allow the curiosity such that the user has to insert some bends to the association arrow or the language designer defines a further production. The production that effects a curiosity is shown in Figure 3.10, the further one is illustrated in Figure 7.5. In this figure, only one class symbol is required by the LHS. Instead of the user, the language designer has to insert bends to the association arrow in the RHS during the definition of this VL production using the GenGEd grammar editor.

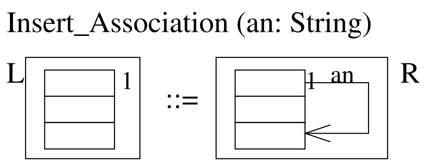


Figure 7.5: Production for the insertion of an association symbol.

The disadvantage of two productions for one task is given by the fact, that the user of the corresponding VL editor will be irritated. Moreover, in the intended VL editor where the VL productions are not shown to the user, it is questionable if a production name can clarify this fact. Nevertheless, these two possibilities illustrate the great many features of GenGEd. Independently which strategy is favored, due to the least astonishment principles of ParCon, if a class symbol is moved on the screen, the linked association arrow is lengthened accordingly.

The deletion of an association arrow causes no problems. Hence, it is similar to the insertion production shown in Figure 3.10. However, the production sides are exchanged and the variable indicating the association name is omitted.

7.1.3 Class and Association Class Symbols

The production that allows for inserting an association class symbol is already shown in Figure 3.14 on page 63. Also the production for the conversion of an association symbol to an association class symbol is already treated and illustrated in Figure 3.16. Its abstract syntax is shown in Figure 3.17. Exactly this production has been required in Chapter 2.6.

The deletion of an association class symbol causes no problems since no symbol depends on the association class symbol. The generated alphabet production for this task keeps in its LHS three class symbols where the association class symbol is linked to. Its RHS contains then the three class symbols. This alphabet production offers one possibility to delete an association class symbol. Nevertheless, Figure 7.6 illustrates a further possibility. The association class symbol and the associated class symbol is deleted if the production is applied.

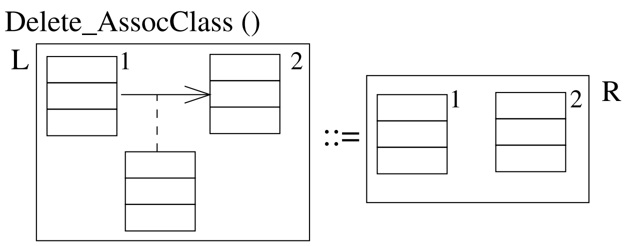


Figure 7.6: Production for the deletion of an association class symbol.

7.1.4 Class and Generalization Symbols

In Chapter 2.6 we have discussed two possibilities for generalization symbols. We favor that one which is illustrated in Figure 2.7 (b); it corresponds to the symbol depicted in Figure 7.1 at the beginning of this section. Thus, the insertion of a generalization symbol is logically similar to the insertion of an association symbol, except of the attribute symbol. A generalization symbol is not required to be named. Figure 7.7 shows the VL productions for the insertion of a generalization symbol in (a) and a generalization name in (b).

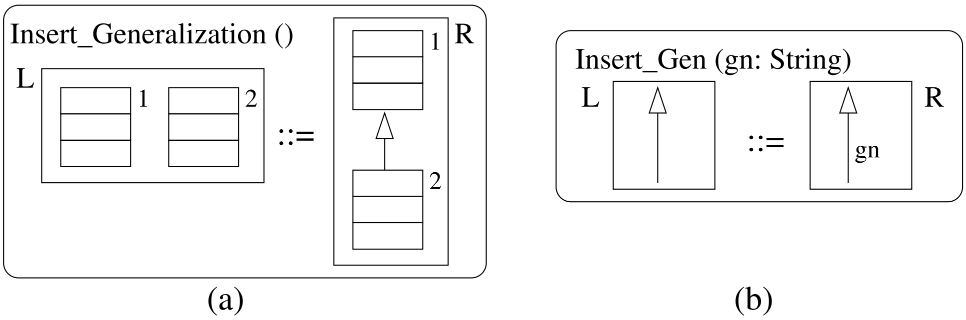


Figure 7.7: Production for the insertion of a generalization symbol (a) and a generalization name (b).

The deletion of a generalization symbol causes no problems. The corresponding deletion production looks like the insertion production of Figure 7.7 (a) where the production sides are exchanged. Due to the underlying single-pushout approach, if deleting a generalization symbol that is equipped with a generalization name, the name is deleted as well.

7.1.5 Application of Visual Language Productions

In this section we generate the class diagram that is illustrated in Figure 2.4 and reviewed in Figure 7.8. First of all we insert a class symbol by applying the

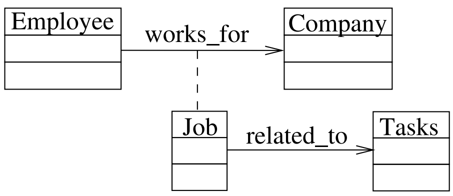


Figure 7.8: Class diagram to be derived by VL productions.

Insert_Class production of Figure 3.12 on page 3.12. Due to the production parameter we have to define a class name, that is *Employee*. The production is applied once more with the class name *Company*. Now we have two possibilities due to the VL productions. Either we insert a third class with class name

Job and insert the association class symbol afterwards, or we first insert an association symbol with association name *works_for* and convert this symbol to an association class symbol afterwards. We favor the latter case. That means, we apply the production *Insert_Assoc* to our two class symbols and define the name *works_for* as association name due to the production parameter. Thereafter, we convert the association symbol to an association class symbol. This situation is is shown in Figure 7.9. The corresponding production is illustrated in the top (cf. Figure 3.15, page 63). The sentence where this production is applied to, is marked with *before*. The dashed arrows indicate the match morphism. The resulting sentence is marked with *after*. Applying the production, the association name (*works_for*) is preserved, and the user is asked for a class name (*Job*). This class name is placed in the new generated class symbol that is linked to the association class symbol.

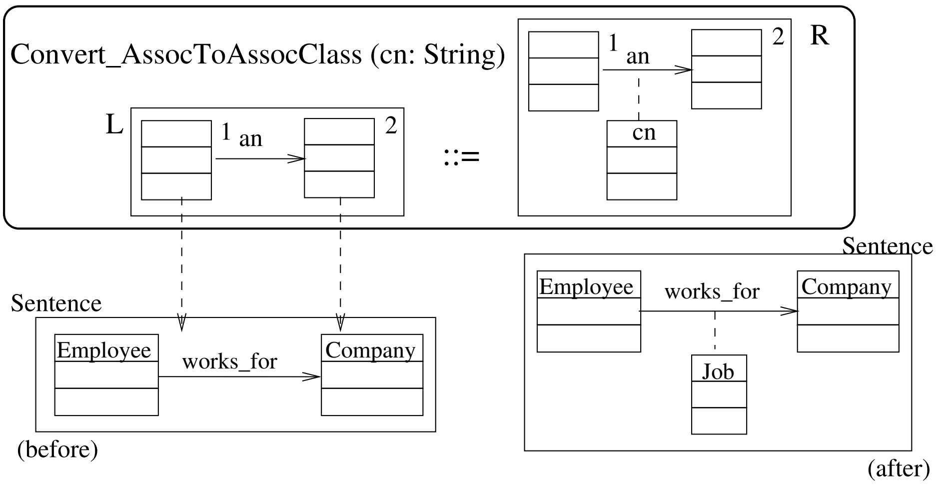


Figure 7.9: Application of a production.

The next step is the insertion of a class symbol with class name *Tasks*. Thereafter, we insert an association symbol with association name *related_to* that is linked to the class called *Job* and the class called *tasks*. So, we have obtained the intended class diagram of Figure 7.8.

7.2 A Statechart Language

In this section we consider parts of the statechart language proposed in [Har87]. Like class diagrams, the statechart language is a graph-like language. Statecharts offer an extension of simple automata. These extensions concern mainly the structuring of systems, namely, hierarchical and parallel composition and the marking of initial states (also called *default states*). Two statecharts are illustrated in Figure 7.10. Figure 7.10 (a) shows a hierarchical composition of a state called V. A hierarchical composition allows the composition of several states (*substates*) to one state (*super state*). Figure 7.10 (b) presents a parallel composition, called Y, consisting of the states K and L. Parallel composition models the combination of several substates to one super state by conjunction. This construct expresses that a system resides in all connected states. Default states are depicted by arrows with a black dot at the end. This symbol graphic indicates the initial executing states.

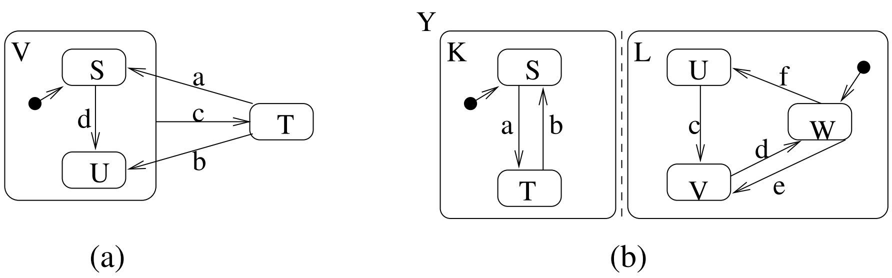


Figure 7.10: Hierarchical composition (a) and parallel composition (b) of statecharts.

Figure 7.11 depicts the symbols that are defined in the symbol editor. The symbol graphic of a hierarchy symbol is given by an invisible rectangle that is used to define the links. It is visible in the alphabet editor, but neither in the grammar nor in the VL editor. We denote this invisible rectangle as *placeholder* in the following. The symbols in the middle of Figure 7.11 are used as the common statecharts symbols. A state symbol is represented by a rounded rectangle, a transition symbol by an arrow, and a default symbol by an arrow and a filled circle. The state symbol as well as the transition symbol are equipped with attribute symbols, namely, with the state name SN and the transition name TN. Both attribute symbols represent the datatype String. In the alphabet, each attribute symbol is is represented by a textual expression. We use the same texts as the abstract syntax types. In order to construct a parallel symbol we use state symbols that are coupled by a horizontal dashed line as illustrated in the right of Figure 7.11. These symbols have to be linked by using the connection editor of the GENGED alphabet editor.

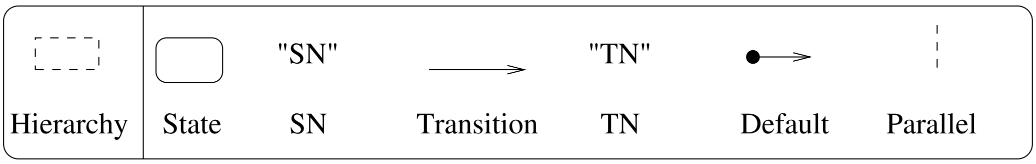


Figure 7.11: Symbols of statecharts.

We omit to present the numerous graphical constraints that are necessary to define the concrete syntax of the links. They are only briefly introduced. Instead, we concentrate mainly on the abstract syntax of links and the language-generating productions. In Figure 7.12 we present the *abstract syntax graph* (cf. Remark 3.3.3) of the alphabet for statecharts. This graph contains all the symbol names and the named link edges. These link edges define the dependency relations between symbols (cf. Concept 3.2.1).

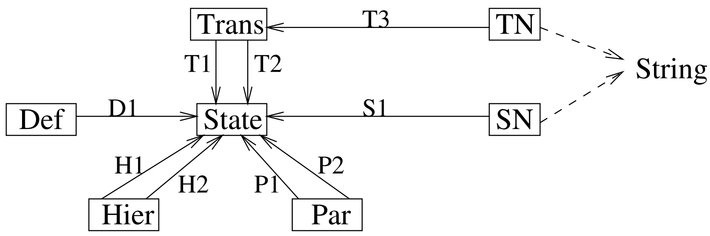


Figure 7.12: Abstract syntax graph of the statechart alphabet.

The following sections concern mainly the links and the language-generating productions. These productions allow the insertion of symbols. We start with the state symbols in Section 7.2.1. State symbols are the central symbols because there is no dependency with respect to the insertion of a state symbol. However, a production for the deletion of a state symbol has to consider all dependencies. Thus, the corresponding deletion production is proposed, too. Section 7.2.2 concerns the links between the state and default symbols. The links between state and hierarchy symbols are the topic of Section 7.2.3. Links between state and parallel symbols are the topic of Section 7.2.4, and the links between state and transition symbols are discussed in Section 7.2.5. The application of the productions proposed in these sections is illustrated in Section 7.2.6. We derive the statechart of Figure 7.10 (a). The statechart language is similar to the class diagram language. Thus, in all these sections we use a slight different notion as so far. This means that we consider mainly the abstract syntax, but use the concrete syntax for illustrations. The concrete syntax of symbols is enhanced with identifiers of the abstract syntax. These are numbers followed by the symbol type to denote symbol instances as e.g., 1:State indicating a state symbol instance.

7.2.1 State Symbols

The state symbol is the central symbol of the statechart language. There is no link edge defined from the state symbol to another symbol (cf. Figure 7.12). So there is no dependency relation between state symbols and other symbols if a state symbol is inserted. Hence, the LHS of the corresponding language-generating production is empty. This production is presented in Figure 7.13. In order to indicate a certain state symbol, it is equipped with a state name. This name is an attribute symbol. We denote the corresponding production parameter by the variable sn of type String. The user has to define a unique state name if the production is applied. This name has to be placed "inside" the new state symbol. The negative application condition (NAC) of this production states that the user-defined state name is unique in the statechart.

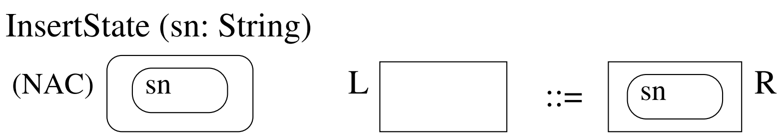
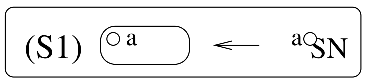


Figure 7.13: VL production for the insertion of a state symbol.

The link definition between a state symbol and its attribute symbol is shown in the figure beside. A state name must be always connected to a state.



Hence, the link edge runs from the state name to the state symbol. The attachments are illustrated by the circles called a and b. An "include" constraint ensures, that the state name is always placed inside the state rectangle. A further "equalPosition" enforces, that the symbol name is placed in the upper left corner of the state rectangle. Therefore, the reference points of the symbols are used.

Except of the production parameter and the NAC of the *InsertState* production, the automatically language-generating alphabet production for the insertion of a state symbol corresponds to the VL production of Figure 7.13.

A production for the deletion of a state symbol is inverse to the insertion production (cf. Figure 7.13). Intuitively, the LHSs and RHSs are exchanged. Furthermore, the production parameter is missing. According to the abstract syntax of statecharts in Figure 7.12 showing the dependency relation, a state should not be deleted without taking into account the context. This is expressed by the NACs of Figure 7.14 illustrating the automatically generated alphabet production.

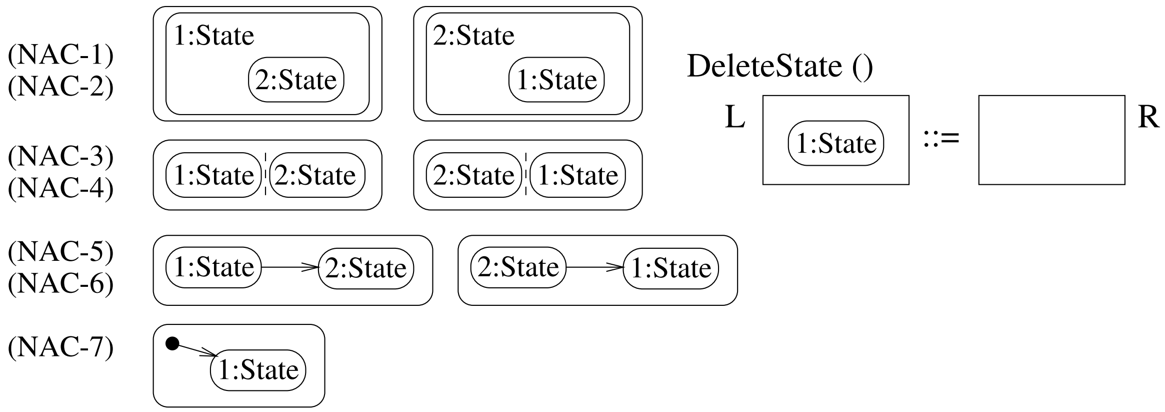


Figure 7.14: Alphabet production for the deletion of a state symbol.

NAC-1 and NAC-2 in Figure 7.14 indicate that the state to be deleted may neither be a super state nor a substate. E.g., deleting one state which is a substate, the corresponding link edge (H1) in Figure 7.12 remains undefined. The same is true if one state is in parallel to another state which is expressed by NAC-3 and NAC-4. NAC-5 and NAC-6 forbid the deletion of a state that is linked to a transition symbol. NAC-7 prohibits the deletion if the state is a default state. The morphisms between the state symbol of the LHS and the symbols of the NACs are illustrated by the numbers, e.g., 1:State in each component.

All the NACs of the alphabet production can be omitted for the VL production allowing to delete a state symbol. Nevertheless, deleting one state symbol which is in an dependency relation with another state symbol leads to the deletion of this relation and its visual elements. The corresponding links between the symbols are deleted, too.

7.2.2 State and Default Symbols

The production for the insertion of a default symbol is shown in Figure 7.15. Due to the dependency relation, the LHS requires the existence of one state symbol only. This state symbol is equipped with a default symbol in the RHS. Due to the semantics of statecharts it is required that only one state on each level (with respect to the hierarchy) is a default state. Furthermore, a state that is parallel to another one cannot be a default state. These facts are expressed by the NACs in Figure 7.15.

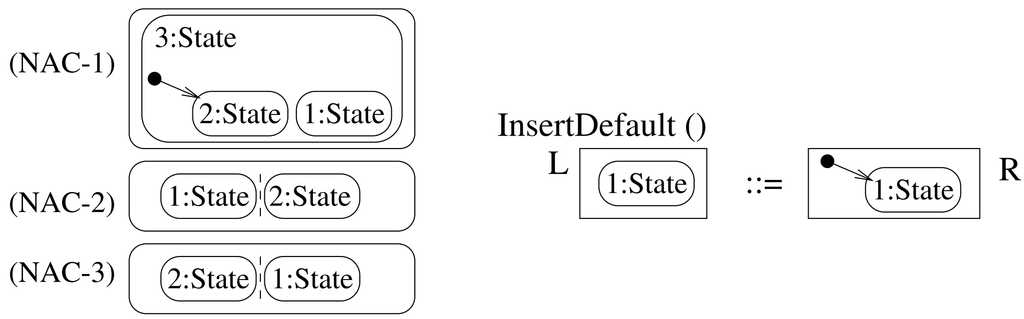
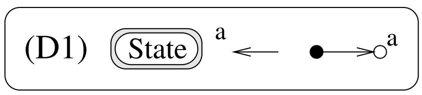


Figure 7.15: VL production for the insertion of a default symbol.

The automatically generated alphabet production for the insertion of a default symbol looks like the VL production in Figure 7.15 without the NACs.

Next we examine the link that is the basis for the generation of the corresponding alphabet production. The link between a default symbol and a state symbol is illustrated in the figure beside. Due to the abstract syntax graph (cf. Figure 7.12) the insertion of a default symbol requires the existence of a state symbol. Hence, the link is defined from the default to the state symbol. According to the concrete syntax level, the arrow head is connected to the circumference of a state rectangle by an "equalPosition" constraint.



A VL production for the deletion of a default symbol is similar to the insertion production shown in Figure 7.15 but the production sides are exchanged.

Furthermore, no NACs are necessary for restricting the deletion of a default symbol.

7.2.3 State and Hierarchy Symbols

A VL production for the insertion of a substate is illustrated in Figure 7.16. It is essential that the invisible hierarchy symbol is linked to two state symbols in the RHS. Because of the corresponding link constraints, one state is included in another state. This is shown in the RHS of the VL production. Its LHS requires the existence of two state symbols. With respect to the abstract syntax graph of Figure 7.12 a state symbol may only be once a super state. Parallel states have no substates. These facts are expressed by the NACs in Figure 7.16. NAC-1 expresses that a state symbol is a super state only once (1:State). NAC-2 and NAC-3 require that the state symbol 1:State that gets a super state by the production, may not be in parallel to another state symbol.

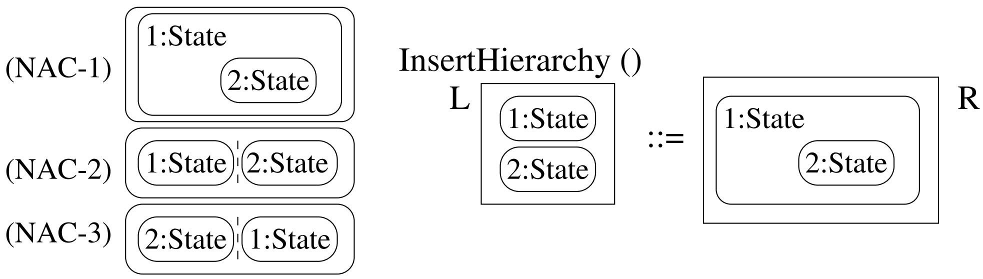
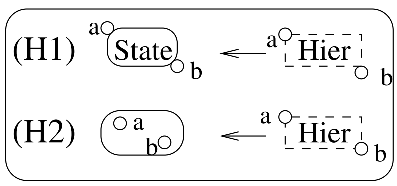


Figure 7.16: VL production for the insertion of a hierarchy symbol.

The corresponding alphabet production looks like the VL production of Figure 7.16, however, without the NACs expressing semantical features.

If one state symbol is a substate of another one, two state symbols have to be existent. Hence, we define two links as illustrated in the figure beside. With respect to the abstract syntax, the link edges run from the hierarchy symbols to the the state symbols. According to the concrete syntax, (H1) states that the hierarchy placeholder is connected to the state symbol by two "equalPosition" constraints indicated by the circles named a and b. These constraints result



in an overlapping of both symbols. (H2) expresses the inclusion of the hierarchy placeholder symbol with some distances.

A VL production for the deletion of a (invisible) hierarchy symbol is similar to the VL production for the insertion shown in Figure 7.16. As before, the production sides are exchanged. NACs are not necessary if the hierarchy symbol is deleted.

7.2.4 State and Parallel Symbols

A VL production for the insertion of a parallel symbol is illustrated in Figure 7.17. The existence of two states is required in order to insert a parallel symbol. With respect to the abstract syntax, a state symbol that has already some substates must not be in parallel to another symbol. Furthermore, a parallel state has to be given by exactly two parallel (sub)states. These facts are expressed by the NACs in Figure 7.17. NAC-1 expresses that state 1:State has no substates. NAC-2 and NAC-3 express that state 1:State or state 2:State are already in parallel to each other.

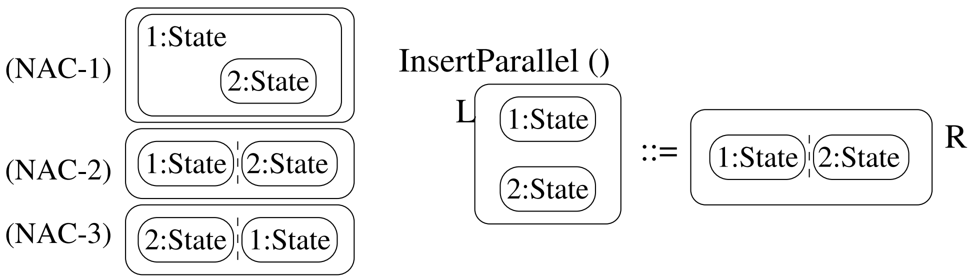
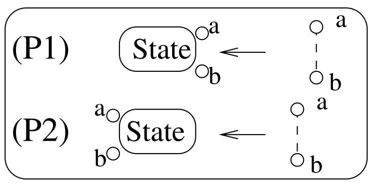


Figure 7.17: VL production for the insertion of a parallel symbol.

The alphabet production for the insertion of a parallel symbol looks like the VL production except for the NACs expressing semantical features for the statechart language. In the following we sketch the corresponding links.

The links between a state and a parallel symbol are illustrated by the figure beside. Each link edge run from the parallel symbols to the state symbols. Hence, the insertion of a parallel symbol requires the existence of two states.



The link constraints with respect to the concrete syntax level are given by "equal-Position" constraints with some distances in order to make the parallel symbol visible. Otherwise, the dashed line indicating the parallel symbol overlaps with the state rectangles and is invisible.

The VL production for deleting a parallel symbol is similar to the VL production for the insertion shown in Figure 7.17, however, the LHSs and the RHSs are exchanged. NACs are not necessary.

7.2.5 State and Transition Symbols

In order to motivate the transition link edges, we look at the VL production for the insertion of a transition symbol illustrated by Figure 7.18. The LHS requires the existence of two state symbols. These are connected by a transition symbol in the RHS. Applying this production enforces the definition of a transition name, indicated by the production parameter. The transition name is then placed "above" the transition arrow. According to the abstract syntax and the semantics of statecharts, a transition symbol may not connect a substate to a super state. This is expressed by NAC-1 in Figure 7.18. Moreover, a transition may not connect parallel states as expressed by NAC-2 and NAC-3.

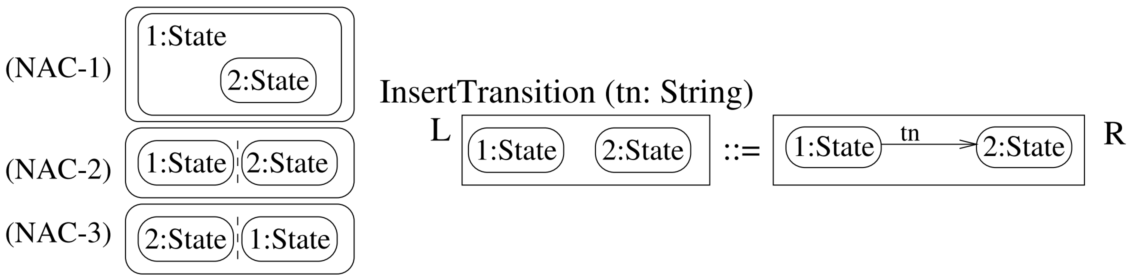
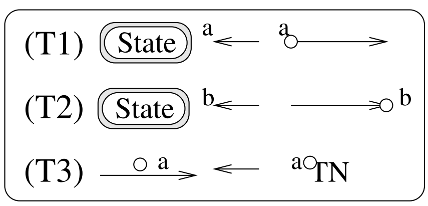


Figure 7.18: VL production for the insertion of a transition symbol.



The transition links are illustrated in the figure beside. Since a transition symbol can only be placed between two states the two link edges run from the transition to the state symbols in (T1) and (T2). Analogously, the link edge for the transition name is defined in (T3). This is due to the abstract syntax graph of Figure 7.12. With respect to the concrete syntax, the state's circumference is used to connect the arrow's begin point to the state rectangle in (T1), respectively its end point in (T2), using "equalPosition" constraints. (T3) indicates the placement of the transition name using the "above" constraint.

Except of the production parameter and the NACs of the *InsertTransition* production, the alphabet production for the insertion of a transition corresponds to the VL production of Figure 7.18.

The VL production for the deletion of a transition symbol is the same as for the insertion presented in Figure 7.18, but the other way round. No production parameter is given in the production name, and no NAC restricts the application of the production.

7.2.6 Application of Visual Language Productions

Once defined the statechart language we show now that the VL productions are sufficient for the manipulation of statecharts. The statechart which we will derive is illustrated in Figure 7.10 (a). This statechart illustrates hierarchical composition. Moreover, transitions are defined between substates and a state outside the hierarchical composition. To obtain the four states of this statechart, we use the insertion production of Figure 7.13. We define the corresponding production parameters by V, S, T and U indicating the state names. Figure 7.19 illustrates the result concerning the diagram as it is visualized (a) and its abstract syntax (b). In the latter case we write the state names directly into the nodes representing the states, i.e., we do not present the data links.

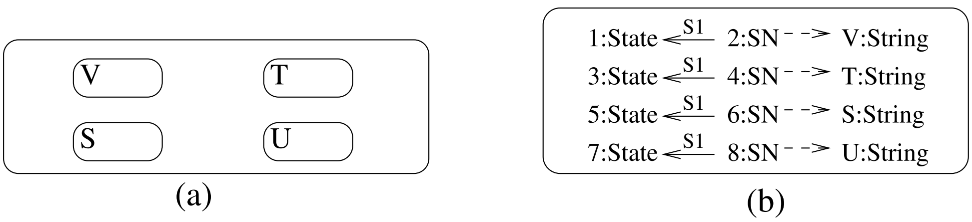


Figure 7.19: Statechart with four states (a) and its abstract syntax graph (b).

Now we are able to insert two hierarchy symbols given by invisible placeholders. The symbol V is the super state and the symbols S and U are the substates. Inserting two hierarchy symbols means, that the production of Figure 7.16 must be applied twice. For the first application we define the match morphism from 1:State to state V, and from 2:State to state S. No application condition is violated. Hence we obtain the desired result. The second application of this production, where 2:State is mapped to state U, results in an overlapping of the two substates, namely, states U and S. Moving S, effects an enlargement of the super state V. The result is illustrated in Figure 7.20.

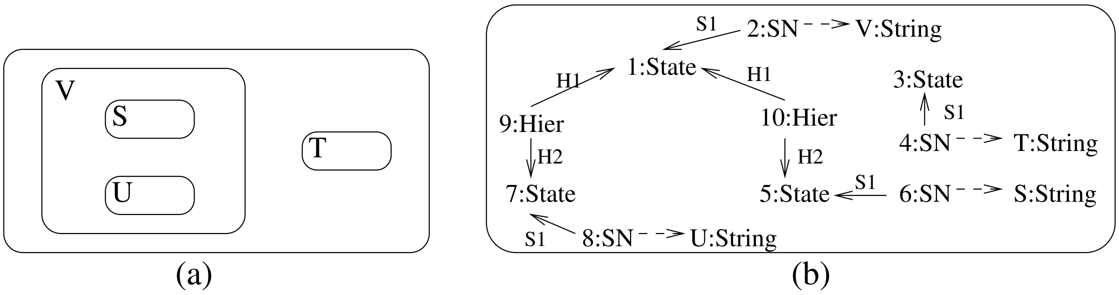
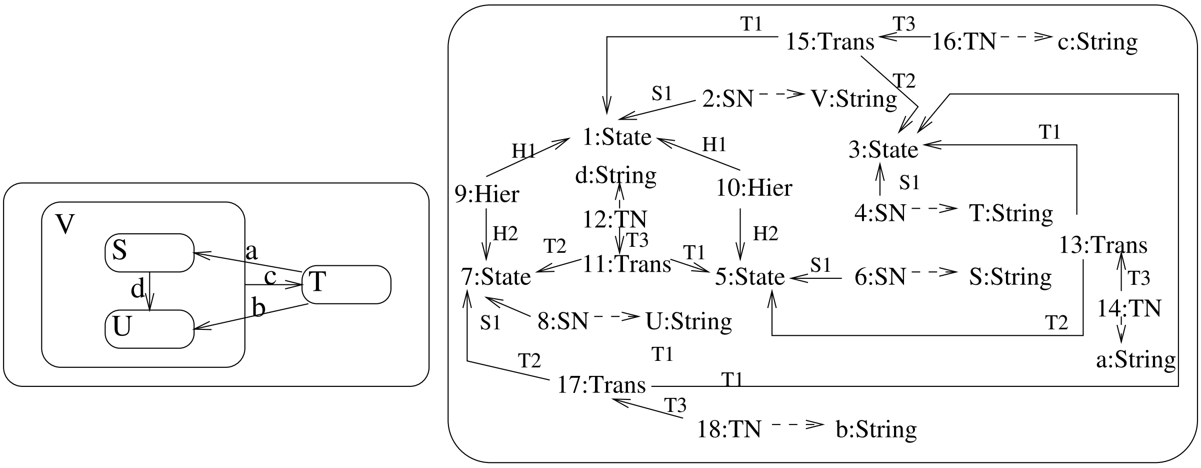


Figure 7.20: Statechart after application of the *InsertHierarchy* production (a) and its abstract syntax graph (b).

We insert now the transitions by applying the *InsertTransition* production of Figure 7.18 four times. For the first application we define the match morphism as follows. State 1:State is mapped to state S in Figure 7.20, and state 2:State is mapped to state U. No NAC is violated. The production parameter is given by the transition name d. Because of the positions of S and U the transition arrow is arranged vertically. This is due to the *least astonishment* principle implemented by the constraint solver PARCON.

The *InsertTransition* production does not prohibit a transition between a substate (of a super state) and a state that is is outside of this super state. So, we can apply the *InsertTransition* production once more yielding the transition between the states T and S. The transition name is a. The two further transitions, called b and c are inserted analogously. These applications yield nearly the desired result as illustrated in Figure 7.21.

The desired result of Figure 7.10 (a) is obtained after application of *InsertDefault* production of Figure 7.15. Therefore, we match the 1:State symbol of the LHS to state S in the current statechart.



(a) (b)

Figure 7.21: Statechart after application of the *InsertTransition* production four times (a) and its abstract syntax graph (b).

7.3 A Nassi-Shneiderman Language

The Nassi-Shneiderman language is a box-like and imperative programming language. Nassi-Shneiderman diagrams (NSDs) are represented by one rectangle (cf. Chapter 2.2). A NSD consists of several rectangular expressions representing the statements of the language. E.g., a while statement consists of a while symbol and a further NSD symbol that is represented by a rectangle. The NSD symbol is connected to the while symbol in a way that the result is a rectangular expression. Thus, two symbols build up one statement, namely, the while statement. In the GENGED approach we have to model the different statements of the language using several linked symbols. Then we are able to define the intended VL productions. Figure 7.22 illustrates the symbols we use for the visual definition of the NSD language using GENGED.

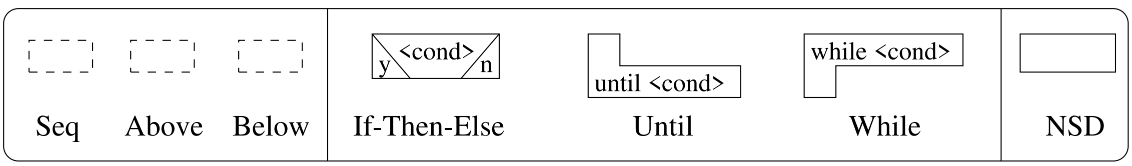


Figure 7.22: Symbols of NSDs.

The three symbols in the left of Figure 7.22 represent placeholders that are used to define spatial relations. Like for the hierarchy symbol of statecharts these placeholders indicate connections that are relevant for the concrete syntax level. In contrast to the graph-like languages presented before (class diagrams, state charts) they usually must not have a logical counterpart. The symbols in the middle of Figure 7.22 are the usual symbols occurring in a NSD (cf. Chapter 2.2). These symbols are already equipped with their attribute symbols like while <cond>. Furthermore, the symbols have to be linked with a further rectangle illustrated in the right of Figure 7.22. This rectangle depicts a NSD symbol that can be usually derived by application of productions. Building up statements, we enclose each resulting statement by a further rectangle that represents a NSD symbol in order to ensure rectangular expressions. So, the (visual) definition of a box-like language using the GENGED approach is slightly different to graph-like languages.

In Figure 7.23, we propose one possible *abstract syntax graph* (cf. Remark 3.3.3) concerning the alphabet of NSDs. The attribute symbols are omitted in this graph. However, with respect to the concrete syntax level they are depicted in the following. We assume that each While, Until and If-then-else (ITE) symbol "encloses" its attribute symbols and further static text as illustrated in Figure 7.22.

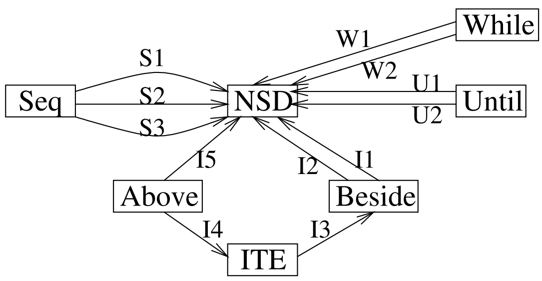


Figure 7.23: Abstract syntax graph of the NSD alphabet.

The abstract syntax graph is explained in the following sections. Thereby, we regard for each two symbols the corresponding links. In addition, some generated alphabet productions are proposed because they differ from the VL productions. These productions are already presented and discussed in Chapter 2.3 (cf. Figure 2.2) [1]. Actually, some of these productions are reviewed briefly.

[1]The Stmt symbol is identic to the NSD symbol used in this section.

In Section 7.3.1 we start with the discussion of NSD symbols. The main features of the NSD language are given by sequences and nestings. A nesting occurs for the definition of the while, until and if-then-else statements. We do not need a special symbol to indicate the nesting. The nestings for each statements are defined by links. A sequence describes the connection of two NSDs, where one NSD symbol is "above" another one. The links between NSD and sequence symbols are discussed in Section 7.3.2. The links between NSD and while symbols is the topic of Section 7.3.3. In Section 7.3.4 the if-then-else statement is discussed. The VL productions proposed in these sections are proved in Section 7.3.5. There we show how to obtain the NSD illustrated in Figure 2.1 (a).

7.3.1 NSD Symbols

The central symbol of the NSD language is the NSD symbol to be seen in the abstract syntax graph. So, the alphabet production for the insertion of a NSD symbol consists of an empty LHS. The RHS comprises then the NSD symbol. The alphabet production for the deletion of a NSD symbol is the insertion production where the production sides are exchanged. It is not useful to define this production as a VL production because its application would lead to incorrect diagrams. It must be prohibited that a NSD symbol is deleted if it occurs in a while, until or if-then-else statement. The VL production for the deletion of a NSD symbol is illustrated by Figure 7.24. It is equipped with NACs. Each condition expresses the non-existence of a further NSD symbol. The corresponding NAC morphisms are illustrated by equal numbers.

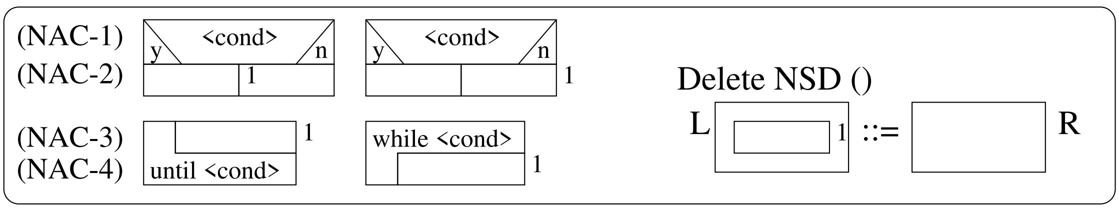


Figure 7.24: VL production allowing the deletion of a NSD symbol.

In the following section we first explain how to achieve the VL production for the derivation to a sequence. Thereafter, we discuss how to achieve the statements occurring in the NACs.

7.3.2 NSD and Sequence Symbols

The sequence symbol is used to connect two NSD symbols in such a way that the one is "above" the other. This is due to the fact that we need a VL production as illustrated in Figure 7.25 (a). Its LHS requires the existence of a NSD symbol. This NSD symbol is derived by the RHS of the production to two further NSD symbols. Figure 7.25 illustrates the abstract syntax level of the production.

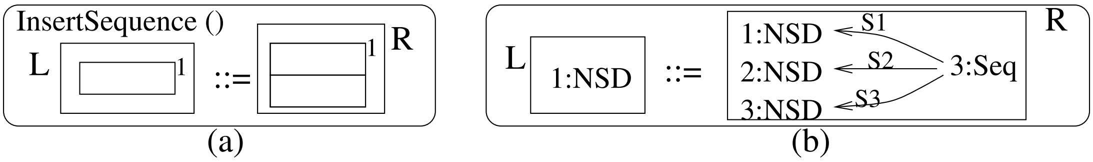
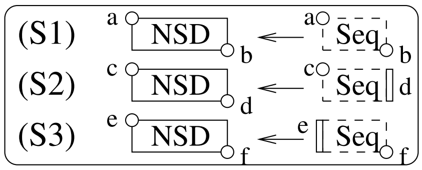


Figure 7.25: VL production for the insertion of a sequence symbol (a) and its abstract syntax (b).

All the link edges illustrated in the figure beside run from the sequence symbols to the NSD symbols. The corners illustrated by equal numbered points are connected by "equalPosition" constraints. The link (S1) expresses that the circumferences of the symbols are identical. As shown in the RHS of Figure 7.25, it is intended that this NSD symbol encloses two further NSD symbols expressed by (S2) and (S3). In (S2), the lower corner of the NSD symbol (d) is connected to the the right line of the sequence symbol, called d. In (S3), the upper left corner of the NSD symbol is connected to the left line of the sequence symbol (e). The corresponding "equalPosition" constraints yield an overlapping of the NSD symbols in the alphabet production for the insertion of a sequence symbol. The definition of a suitable VL production for the insertion of a sequence symbol must ensure further constraints, namely, that one NSD symbol is "above" the other.



Because three links are defined to build up a VL production (cf. Figure 7.25), there are three NSD symbols in the LHS of the alphabet production for the insertion of a sequence symbol shown in Figure 7.26. The symbol graphics of the RHS in Figure 7.26 (a) overlap because of the link constraints. Moving the three NSD symbols so that they do not overlap, enlarges the invisible sequence placeholder, and hence the symbol 1:NSD. For illustrational reasons, the abstract syntax of the alphabet production is presented in Figure 7.26 (b).

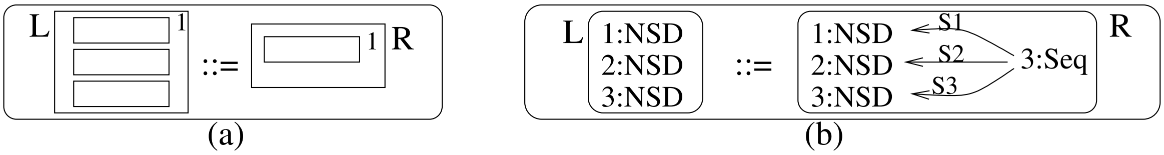


Figure 7.26: Alphabet production for the insertion of a sequence (a) and its abstract syntax (b).

In order to obtain the VL production presented in Figure 7.25, we manipulate the alphabet production. Therefore, we delete the 2:NSD symbol in the LHS of the alphabet production. To ensure that 2:NSD is linked to 3:NSD in the RHS, we connect the lower corners of 2:NSD to the upper corners of 3:NSD by "equalPosition" constraints. The production morphism from 1:NSD in the LHS to 1:NSD in the RHS is already defined by the alphabet production and hence, it must not be defined again. Note, the deletion of a symbol effects implicitly in a deletion of corresponding morphisms.

The alphabet production for the deletion of a sequence symbol looks like the alphabet production for the insertion, but with exchanged production sides. This is not the desired VL production. The desired VL production for the deletion of a sequence symbol is the VL production for the insertion with exchanged production sides.

7.3.3 While Statement

Let us consider now the VL production for the derivation of a while statement. The corresponding production is presented in Figure 7.27 (a). Its abstract syntax level is depicted in Figure 7.27 (b). In its LHS, the existence of a NSD symbol is required. This NSD symbol contains a while symbol and a further NSD symbol in its RHS.

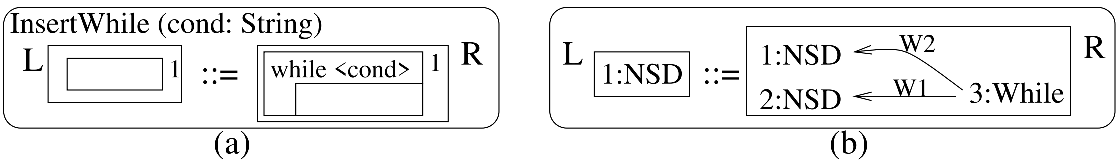
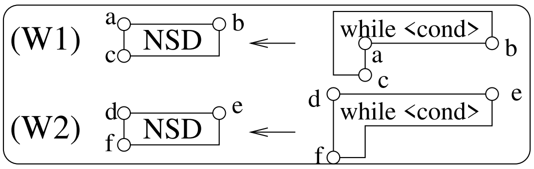


Figure 7.27: VL production for the insertion of a while statement (a) and its abstract syntax (b).



All link edges illustrated in the figure beside run from the while symbol to the NSD symbol. The corners illustrated by the numbered points are connected by "equalPosition" constraints. (W1) expresses the nesting of a while symbol. (W2) indicates that the NSD symbol encloses the while statement, namely, the while symbol and its inherent NSD symbol.

Two links are defined to build up a while statement. These two links enforce the existence of two NSD symbols in the LHS of the corresponding alphabet production which is illustrated by Figure 7.28. Deleting the symbol 2:NSD leads to the desired VL production.

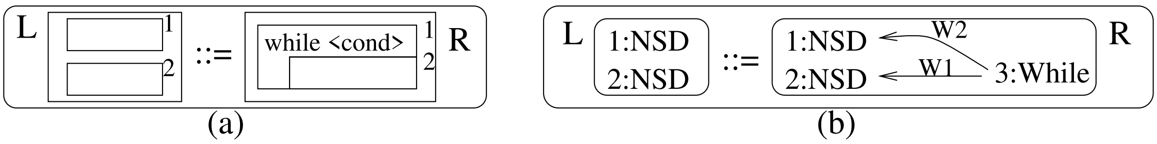


Figure 7.28: Alphabet production for the insertion of a while statement (a) and its abstract syntax (b).

The alphabet production for the deletion of a while statement is the similar to the insert production with exchanged production sides. So we have two NSD symbols in the RHS and delete the symbol which usually encloses the while statement. The VL production is illustrated in Figure 7.29 (a). Figure (b) shows its abstract syntax. The LHS contains the while statement which is to be deleted. The NSD symbol 1:NSD which encloses the while statement remains undeleted in the RHS.

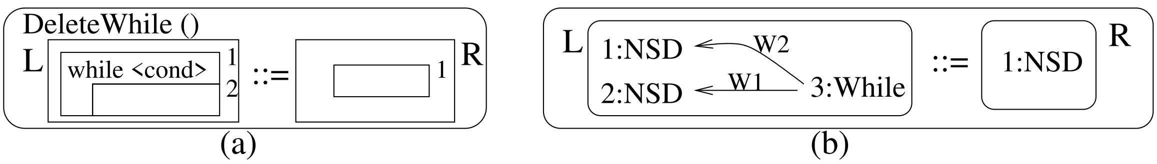


Figure 7.29: VL production for the deletion of a while statement (a) and its abstract syntax (b).

7.3.4 If-then-else Statement

The VL production for the insertion of an if-then-else statement is illustrated by Figure 7.30 (a). Its abstract syntax level is depicted in Figure 7.30 (b). Its LHS requires the existence of one NSD symbol which encloses several symbols in the RHS. These symbols are given by the if-then-else symbol and two NSD symbols which are placed “below” the if-then-else symbol. Moreover, these two NSD symbols have to be placed one “beside” the other and have to be horizontally aligned.

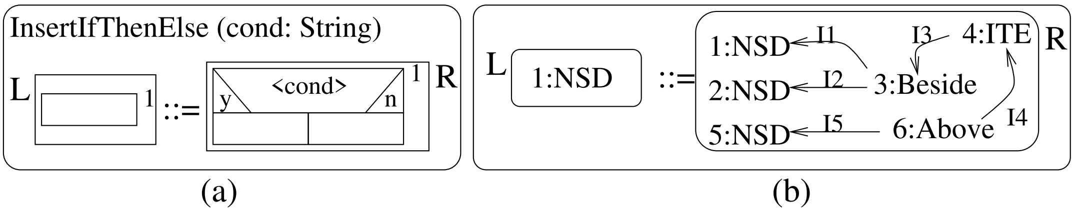
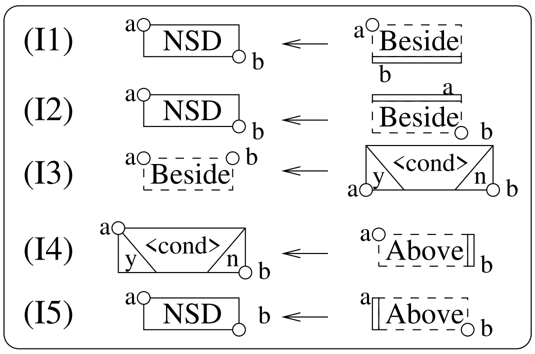


Figure 7.30: VL production for the insertion of an if-then-else statement (a) and its abstract syntax (b).



The links are illustrated in the figure beside. As before, the points indicate “equalPosition” constraints. The lines indicate, that the same numbered point of the other symbol is to be placed on this line. The beside placeholder in (I1) to (I3) expresses the connection of the two NSD symbols and the if-then-else-statement in the RHS of the VL production (cf. Figure 7.30). (I1) states that the upper left corners (a) of both symbols are connected and the point b of the NSD symbol is on the lower line of the beside placeholder. (I2) is similar to (I1), but with different corners. In (I3) the if-then-else statement is connected with the beside placeholder. It is intended that the beside placeholder is “below” the if-then-else statement. (I4) and (I5) are used to enclose an if-then-else expression together with its inherent two NSD symbols. These link constraints effect an overlapping of both NSD symbols.

The three alphabet productions are illustrated by Figure 7.31. The productions are presented with their visual representation (a) and their abstract syntax (b). Production (1) allows inserting a beside placeholder. Its LHS requires the existence of two NSD symbols. These NSD symbols are then linked according to (l1) and (l2). Because of the corresponding link constraints, the NSD symbols overlap. Note that the beside placeholder is invisible, but it is a symbol with respect to the abstract syntax. Production (2) in the middle allows the insertion of an if-then-else statement. It requires in its LHS the existence of three symbols, namely, two NSD symbols and a beside placeholder. As for the first production the two NSD symbols in the LHS of the production overlap because of the constraints for (l1) and (l2). The constraints of (l3) ensure that the if-then-else statement is placed "above" the other symbols. Production (3) requires in its LHS the existence of several symbols as well. These symbols are the same as the symbols of the RHS of the second production. A further NSD symbol is required, denoted by 5:NSD. According to the links (l4) and (l5) the rectangle representing the 5:NSD symbol contains all the other symbols.

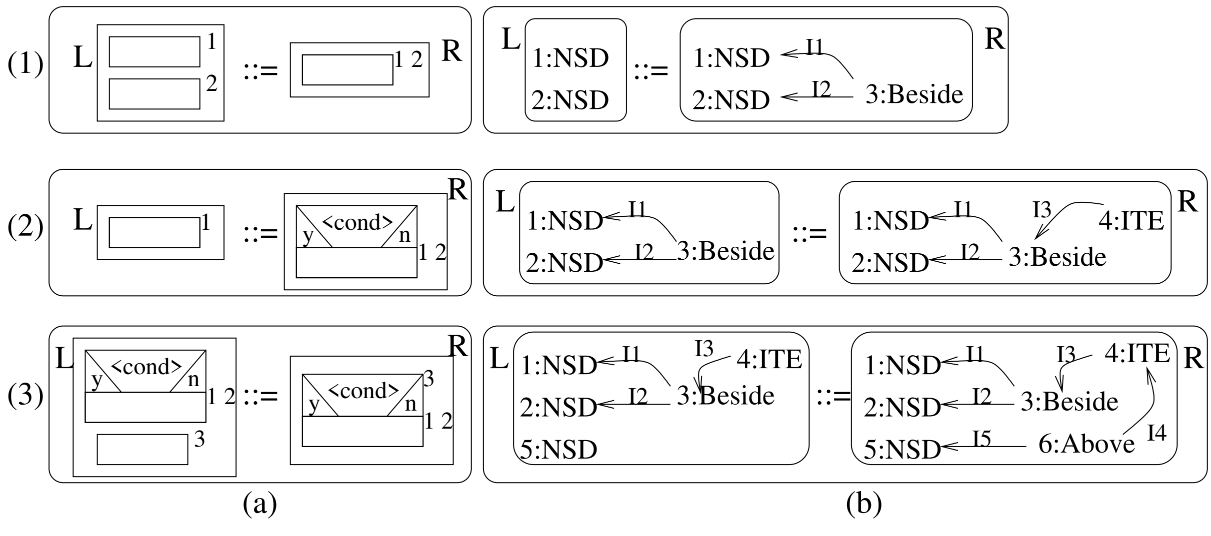


Figure 7.31: Alphabet productions for the insertion of an if-then-else statement (a) and their abstract syntax (b).

The simplest way to construct the VL production as illustrated by Figure 7.30 is to use the third alphabet production and to manipulate the LHS. The RHS is nearly the intended RHS of the VL production. Similar to the sequence production we must define a production constraint according the placement of the two

NSD symbols below the if-then-else symbol. Therefore we move the overlapping rectangle (2:NSD) and connect it to the 1:NSD symbol by "equalPosition" constraints according to the corners of both symbols. Constraints have to be defined between the upper right (lower right) and the upper left (lower left) corners of both symbols. In the LHS the upper construct can be deleted by the corresponding deletion production. Because the morphism between the 5:NSD symbol of the LHS is already defined in the alphabet production we have yet the intended VL production.

7.3.5 Application of Visual Language Productions

We have shown that box-like languages like the NSD language can be easily defined using the GENGED approach. In this section we snow that the VL productions are sufficient for the generation of NSDs. The diagram to generate has been already illustrated in Figure 2.2 (a). For illustrational reasons it is again presented in Figure 7.32.

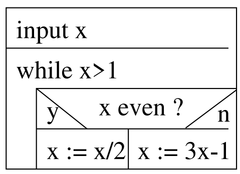


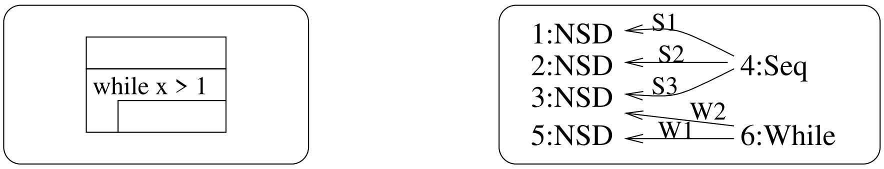
Figure 7.32: NSD to be derived by the VL productions.

Assume that a NSD symbol is the start symbol of the VL grammar for NSDs. Given this NSD symbol it is possible to insert a sequence by application of the production illustrated in Figure 7.25. This yields the two NSD symbols presented in Figure 7.33.

(a) (b)

Figure 7.33: NSD after application of the *InsertSequence* production (a) and its abstract syntax graph (b).

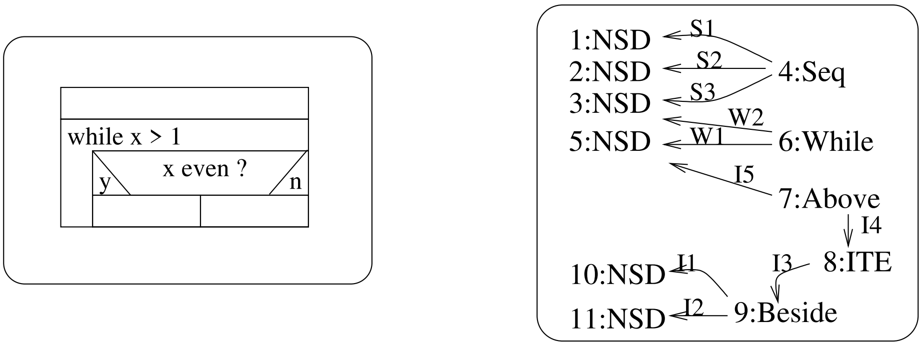
Now we are able to apply the production inserting a while statement as proposed by Figure 7.27. We match the NSD symbol of the production's LHS to the 3:NSD (the lower rectangle) of the current NSD. In addition, we define the production parameter by $x > 1$. This effects the NSD illustrated by Figure 7.34.



(a) (b)

Figure 7.34: NSD after application of the *InsertWhile* production (a) and its abstract syntax graph (b).

Last but not least we are able to apply the production for the insertion of an if-then-else statement as proposed by Figure 7.30. We match the NSD symbol of the production's LHS to the 5:NSD symbol (rectangle nested with the while symbol) of the current diagram. The application of the production yields the diagram illustrated by Figure 7.35.



(a) (b)

Figure 7.35: NSD after application of the *InsertIfThenElse* production (a) and its abstract syntax graph (b).

In order to achieve the intended diagram we must apply a production for the derivation of a NSD symbol to a textual expression to each of the NSD symbols. This yields the intended diagram presented in Figure 7.32.

Chapter 8

Comparison of GenGEd with Related Approaches

There are still many different formalisms for the specification of visual languages (VLs) which have been investigated for more than two decades now [MM97]. These are grammar-like and non-grammar-like formalisms. Grammar-like formalisms range from early approaches like array and web grammars [Ros76], and shape grammars [Gip75] to recent formalisms like picture-layout grammars [Gol91b, Gol91a], positional grammars [CC91, CLOT98], relational grammars [Wit92, WW96], constraint multiset grammars [Mar94, MM98a], etc. and several types of graph grammars [Roz99]. Non grammar-like formalisms include algebraic approaches [DÜ96] and logic-based approaches [HM90a, Mey92, Haa96]. Most of these formalisms are described in [MM98b], the first book about *Visual Language Theory*. Note that these formalisms are mainly motivated by applications involving parsing. Furthermore, most of the formalisms are rather concerned with visual programming languages than with visual modeling languages.

Parsing restricts the expressiveness Since there has been no common underlying framework up to now for defining the syntax and semantics of visual languages, it has been difficult to define precisely what context-freeness or context-sensitiveness means for visual languages. However, when parsing is involved in a visual language framework, the underlying formalisms are mostly restricted to some certain classes of visual languages. Parsing is rather concerned with the recognition of visual languages than with its generation. We concentrate on the generation which is the reason that the aspect of parsing is omitted in this thesis. The reader is referred to [MM98b] where each of the authors compare their formalisms with some others with respect to the computational effort of parsing. The reader is also referred to [BTMS99] where the hypergraph grammar formalism used by DiaGen and the graph grammar formalism used by Progres are compared with some other formalisms with respect to parsing.

Most formalisms use textual notations Beside the wide-spread usage of visual modeling, if not visual programming languages (VLPs), there is a considerable *lack of formalisms* for defining their syntax and semantics. Until now, there exists no equivalent to Backus-Naur-Forms or attribute grammars, which would be the notation for defining the syntax and semantics of a visual language or for generating visual language editors, interpreters, and compilers. Even worse, almost all currently used VPL syntax definition formalisms have a textual instead of a visual notation. They are processed using standard text editors, parsers, and so on. As a consequence, most published visual languages come with informal and imprecise definitions, and the development of their tools often requires way far too many person years.

Graph grammars itself offer visual notations Graph grammars are used as a powerful visual syntax definition formalism. Graph transformation-based (meta) tools like Progres [Sch91, SWZ99, BTMS99] are used to generate VL programming environments (with a main emphasis on VL editors) and they are used to generate VL parsers. Graph transformation-based languages like Progres are a special brand of (visual) rule-based programming languages, which have a precisely defined syntax and semantics, and which offer thereby appropriate means for checking the correctness of constructed programs.

Purpose of a formalism Progres is a general purpose rule-based programming language in contrast to the formalisms analyzed in Chapter 2.4, namely the positional grammar formalism in Vlcc [COL94, COL95, CLOT98] and the hypergraph grammar formalism in DiaGen [Min97, Min98, BTMS99]. The purpose of the these formalisms as well as the graph transformation formalism used by GENGED is specifically the specification of visual languages. This may be the reason why PROGRES allows for specifying diagrammatic visual languages whereas the other formalisms (DIAGEN, VLCC, GENGED) allow for specifying any visual representation type according to [BB94].

Rekers and Schürr [RS95, RS96] make a clear distinction between abstract and concrete syntax of visual languages referenced by a triple graph grammar. Note that this triple graph grammar is not put into practice, e.g. by the PROGRES environment. In [AER96] a graph grammar for Message Sequence Charts is sketched as a PROGRES specification. This specification merges the abstract syntax graph and the concrete syntax graph of a visual sentence into a single, too complex, graph. Nevertheless, the PROGRES language and environment [Sch91, SWZ99] supports comprehensive means of visual programming by graph transformation. PROGRES has its strong points when complex analysis operations or animations have to be specified.

Graph grammars compared with other grammar formalisms The main (dis)advantage of graph grammars compared with relational grammars [WW96] is that they enforce a strict distinction between objects (nodes) and relations between objects (edges). Both approaches are, therefore, more or less equivalent to each other, especially in the case of hypergraph grammars, which use hyperedges as n-ary relationship objects.

The differences between graph grammars on one hand and picture-layout grammars [Gol91b, Gol91a] or constraint multiset grammars [Mar94] on the other hand are more important. The latter two approaches enforce a language designer to represent all needed spatial or abstract relationships implicitly as constraints over attribute values of objects. Therefore, attribute assignments within a production have the implicit side effect of creating new (derived) relationships to unknown objects (from the production's point of view). Graph grammars follow a quite different approach. They use explicitly manipulated edges for any kind of relationships between objects.

Contents of this chapter The hypergraph grammar formalism in DIAGEN is compared with the graph transformation formalism in GENGED because both formalisms offer visual notations. These visual notations may help the language designer to model a visual language. Hypergraph grammars used by DIAGEN are topic of Section 8.1. The GENGED formalism is based on *algebraic* graph transformation. This is the reason why we compare the GENGED formalism with the algebraic but non-grammar-like approach proposed by S.Üsküdarlı and T.B.Dinesh [DÜ96] in Section 8.2. We do not compare VLCC because the positional grammar formalism is restricted to context-free productions. Additionally, this formalism is given by textual notations. Although in [COL95] a visual framework is proposed the grammar editor of the available tool offers no visual notations.

It may be possible to judge some aspects of a specification framework, such as how difficult it is to specify some particular languages, on an intuitive level. But for a full comparison of the various types of specification formalisms, a formal comparison is usually required to understand their tradeoffs and comparative advantages. A formal comparison is not regarded in this thesis. Instead, we regard the internal structure of visual sentences, the formalism used to specify a language and the tool support.

8.1 The Hypergraph Formalism in DIAGEN

In Chapter 2.4 we have proposed the hypergraph grammar formalism in DIAGEN [Min97, Min98, BTMS99]. General attributed and labeled graph structures as introduced in Chapter 4 can be used to model all graph classes, e.g. hypergraphs as they are used as internal data model for visual sentences by DIAGEN [1]. Although it is possible to model hypergraphs using the graph structure approach, their application to visual languages is different.

8.1.1 Representation of Visual Sentences

DIAGEN uses labeled and attributed hyperedges but unlabeled attributed nodes. The set of hyperedge labels is defined as the set of pictorial object types due to the abstract syntax of visual languages. Each pictorial object of a specific type consists of its visual appearance and of some specific attachments where it can be attached (related) to other objects. Each object is represented by a hyperedge

[1] An algebraic graph structure signature for hypergraphs supposing a slight different notation can be found by Def. 4.5.4 in [EHK+97].

with the object's attachments being represented by the hyperedge's tentacles. Two hyperedges that visit a common node represent two related pictorial objects. The tentacles connected to the common node identify the objects' attachments which are involved in the relationship. The attachments are used for constraints (constraint variables) specifying the correct layout.

Graph structures usually do not distinguish between nodes and edges. Graph items hold the pictorial objects (as attributes), and operations between graph items define the relationships, i.e. the links between the graph items. The attachments on pictorial objects define constraint variables occurring in constraints. Constraints specify the layout of a sentence. However, the graph items can be interpreted as nodes and the operations as a kind of edges. The nodes play the central role in the case of GENGED but are of minor importance in the case of DIAGEN. Nodes of DIAGEN are only used as attachments between hyperedges and do not have any additional label. Edges, on the other hand, model labeled attributed n-ary relationships (hyperedges) with labeled tentacles in DIAGEN. In GENGED edges model labeled, attributed binary relationships with identifiers. In GENGED parallel relationships as well as m:n or 1:n relationships are supported. Note that in [Min98] the hypergraph representation concepts are extended in the sense that additional hyperedges are introduced in order to model (arbitrary) relationships like "touch" and "inside" necessary for the VEX language.

8.1.2 The Grammar

The hypergraph grammar comprises a set of (non-terminal and terminal) symbols, a start sentence and a set of productions. Note that the alphabet is given by symbols only; relationships are specified by productions.

Concerning the definition of graph transformation productions three groups of properties are distinguished. They concern (1) some properties of LHS matches, (2) additional possibilities for the definition of application conditions which prohibit certain LHS matches, and (3) the properties of the rewriting mechanism itself.

Homomorphic and isomorphic matching It is remarkable that DIAGEN and GENGED allows one to choose between homomorphic and isomorphic matching (two LHS elements are allowed or disallowed to match the same host graph element) due to the fact that almost all earlier graph transformation approaches adhere either to the homomorphic or to the isomorphic graph transformation approach.

Application conditions Additional *conditions*, which prohibit production applications to otherwise legal LHS matches, are treated very similar in DIAGEN and GENGED. DIAGEN and GENGED support the definition of negative application conditions which may not exist as extensions of a determined LHS occurrence.

Conflicts The graph transformation mechanisms of DIAGEN and GENGED, which remove and add certain graph elements after a distinct LHS match has been fixed, are rather straightforward. There are only three specific points which have to be taken into account. First of all we have to decide what happens if a graph element, which is part of a production's LHS and RHS, and another graph element, which is only part of the rule's LHS, match the same host graph element. GENGED votes for deletion in the case of such a *delete/preserve conflict*, whereas DIAGEN simply forbids such a graph transformation step. Furthermore, we have to make the decision what happens to those edges of deleted host graph nodes (tentacles of hyperedges) which are not part of the applied production's match. DIAGEN favors again the conservative approach and forbids such a graph transformation step, whereas GENGED simply deletes the otherwise dangling edges, too. Note that because of the graph structure approach it may happen that also graph items are deleted in GENGED. This is due to the fact that total algebras are required with respect to the abstract syntax.

8.1.3 Tool Support

Switching from formalisms to tools (systems) we have to regard two tools, one for editing, analyzing, and one for executing graph transformations, whereas the second one deals with generated visual (programming) tools only. Tools of the first category are, therefore, used to develop tools of the second category. They are sometimes called *meta programming tools*.

DIAGEN follows the traditional compiler compiler approach, i.e., it processes a pure textual specification of a hypergraph grammar plus the associated constraint definitions, Java methods, and interaction automata. It checks its input with respect to a given text grammar as well as some additional consistency conditions and produces a visual language editor as output. At the current implementation state of DiaGen II, however, this compiler compiler is not yet fully available for generating syntax-directed editors.

GenGEd is famous for its integration with the layout constraint solver ParCon and its flexibility concerning the notation of LHS and RHS of its productions. It is the only graph transformation-based tool which allows one to adapt its grammar editor such that it uses directly the visual constructs of a studied application domain.

Attributes DiaGen and GenGEd use a host language for the definition of attribute types and the manipulation of attribute values. This is Java in the case of DiaGen II (parsing) and GenGEd , and C++ in the case of DiaGen I. GenGEd offers furthermore a number of built-in datatypes which are part of the language itself and not imported from a host programming language.

Controlling the application of productions DiaGen uses the imperative programming means of Java (C++ in the case of DiaGen I) to control the application of productions as well as so-called interaction automata. Such a mechanism is not (yet) available in GenGEd.

Derived data elements Until now we had our main focus on those data elements which are explicitly manipulated by graph transformation productions. But there is another category of so-called *derived* data elements which are automatically updated whenever some related data elements are modified. DiaGen and GenGEd use the concept of arbitrary constraints (undirected equations, inequalities, . . .) for the definition of derived attributes. DiaGen has additionally some grouping mechanism for viewing and handling collections of hyperedges selected by the user as composite derived hyperedges.

8.1.4 Summary and Discussion

It is obvious that the differences between the hypergraph grammar formalism in DiaGen and the graph structure formalisms in GenGEd are less important. This is due to the properties of the basic elements of hypergraphs and graph structures as well as the kind of productions. The difference between the handling of conflicts, however, is to be regarded by a language designer.

DIAGEN lies its main emphasis on free-hand editing and parsing of diagrams, the visual definition of visual languages is not the purpose of DIAGEN. GENGED is rather strong concerning the visual definition of visual languages, however, it generates syntax-directed VL editors *only*.

The graph structure formalism can be used to model all graph classes such as hypergraphs. An interesting topic is whether a VL specification generated by the GENGED specification tool can be transformed into a hypergraph specification which is used by DIAGEN.

8.2 The VAS Formalism in VASE

The visual algebraic specification formalism (VAS) is an algebraic specification formalism for the visual specification of syntax and semantics of visual languages [DÜ96]. In general, the formalism allows for the visual specification of algebraic specifications. These visual specifications specify visual syntax and the editing operations of language-specific editors. The VAS formalism is based on the ASF+SDF formalism [Kli93].

The major aspect of the VAS formalism in VASE, the visual algebraic specification environment, is the generation of so-called *visual programming environments* (or *term editors*). In order to achieve such a programming environment the language designer must follow several steps. In a first step the visual syntax is to be specified together with some behavior functions for an editor. From this specification a structured visual editor (SVE) is generated as well as a visual equation editor. The SVE allows for editing visual sentences over the syntax defined. The equation editor allows for the specification of conditional equations based on the specified visual syntax. The specification of equations forms the second step, namely, the specification of the semantics of specific visual languages. These equations are (context-sensitive) term-rewriting rules.

The VAS formalism allows basically context-free syntax definitions where the 2D lexicals are defined using VODL [DÜ95], a constraint-based declarative specification language. Concerning the tools mentioned above only a VODL editor is available allowing the drawing of circles and rectangles. Although tools show that a formalism is feasible the development of tools needs many person years. The interesting point is the VAS formalism which is proposed in the following. In the end of this section we summarize this formalism and compare it with the GENGED formalism.

8.2.1 Visual Syntax Specification

The VODL language is used to specify visual notations in terms of *vods*. These are primitive or composite visual object definitions and spatial relations definitions. Primitive *vods* serve as a small foundational set of *vods*, whereas composite *vods* are defined in terms of less complex *vods*. The spatial relationships define the spatial constraints that exist among the sub-*vods*. A primitive rectangle, e.g. having the width of 50 units can be defined as:

$$\textit{rectangle()} \otimes [\text{width} = 50].$$

The visual layout of *vods* involves the description of the spatial relationships among *vods*. These are similar to Golin's constraints defined in his picture-layout grammar [Gol91a, Gol91b]. Note that there are some basic spatial relationships such as "touches", "left-of", "over", "contains", and "followed-by". This means that arbitrary relationships cannot be defined like in GENGED. Additionally, the spatial relationships are not mirrored on the abstract syntax level of a visual language.

The visual syntax of a language is defined by a function v which maps the abstract syntax of the language to the domain of *vods*. The mapping is defined by specifying a mapping for every syntactic sort in the language. Figure 8.1 shows the visual mapping v giving a part of a simple visual syntax for expressions [Üsk97]. In (1) the *plus* operator is mapped to an operator string "+"; Exp_1, Exp_2 are variables of sort *EXP*. *0, . . . , 9* are constants of sort *Digit* in (2) to (11). Each digit is mapped to a corresponding string. The mapping of a term evaluation in (12) results in recursive component-wise mappings which is common for algebraic specifications. In (1) and (12) the corresponding spatial relationships for each expression are indicated by $\lhd$. (12) indicates the mapping of a so-called *semantic function* which is defined as part of the language specification. The following visual expression is a valid expression according to these mappings: $\boxed{1 + 0 + 3}$.

(1)	*v(plus(Exp₁, Exp₂)) =*	{	*e1: v(Exp₁),* *op: text()* ⊗ [**string** = "+"], *e2: v(Exp₂)* }
		◁ {	*e1* **followed-by** *op* *op* **followed-by** *e2* }
(2)	*v(0) = text()* ⊗ [**string** = "0"]		
	...		
(11)	*v(9) = text()* ⊗ [**string** = "9"]		
(12)	*v(eval(Exp)) =*	{	*e: v(Exp),* *op:* **rectangle()** }
		◁ {	*op* **contains** *e* }

Figure 8.1: Visual mappings for an expression language.

The visual syntax specification as proposed above is enriched by the specification of replacement behavior in order to achieve a structured visual editor (SVE) definition. Note that visual syntax specification comprises also semantic functions such as *eval* in (12) of Figure 8.1. Semantic functions are either applied to the entire visual sentence or to a focus in an editor. Such a focus can be selected via user interaction. The replacement behavior is defined as the set of permissible replacements for each non-terminal of the language. The replacement behavior for a language is obtained from its syntax specification. For each sort s a so-called meta-variable is generated representing the non-terminals of a language. The default layout of a meta-variable is defined by its textual representation similar to (2) in Figure 8.1. When creating visual sentences in an editor, these meta-variables bearing the name of corresponding non-terminals are used such as Exp_M in *plus(1, plus(*Exp_M*, 3)* which corresponds to $\boxed{1 + \langle EXP\rangle + 3}$.

8.2.2 Visual Semantics Specification

The semantics of a visual language can be defined on top of the visual syntax specification. The semantics is defined by a set of conditional equations between terms of the same sort of the syntax specification. The equations are based on an equation language, i.e. for each sort s of the visual syntax specification an equation signature is generated with sort Eq and operation $s = s \to Eq$. Each sort symbol s is represented by a meta-variable which can be derived according to the replacement behavior of the visual syntax specification. This means that the equation signature allows the language designer to write equations of the sorts.

In [Üsk97] some example languages and editors are introduced. One example is given by a visual language for sets, another one by the specification of a move-around editor. This editor is slight similar to the well-known packman game. The semantics of the move-around editor is defined by evaluation equations and an interaction equation. The evaluation equations express the possible movements of a cursor (above, below, left, right). The interaction equation causes a direction to be repeatedly requested during rewriting. The underlying ASF+SDF formalism [Kli93] ensures then the correct term-rewriting based on the abstract syntax level of specifications.

8.2.3 Summary and Discussion

The VAS formalism is based on algebraic specification techniques. It is used to specify the syntax of visual languages and the behavior of a language-specific syntax-directed editor [DÜ96]. These specifications are only sketched in the available articles. The reader is referred to [CEW93] where a fully specification of a syntax-directed editor can be found. Such a specification is visualized using $\mathcal{G}\mathcal{Y}\mathcal{T}$ [BC94], a textual language allowing the visualization of terms over algebraic specifications. In our opinion, a textual language allowing the visualization of algebraic specifications is much more adequate than a language or formalism offering a mixture of visual and algebraic expressions for specifying the features and the behavior of a syntax-directed visual language editor.

The visual languages which can be specified using VAS are very strong restricted to context-free languages. Because of the VODL language all visual representation types [BB94] can be defined. However, relationships due to the abstract syntax level of visual languages are not regarded or interpreted. Furthermore, there is a predefined set of spatial relationships (for the concrete syntax). This means that arbitrary relationships are not supported. Also tool support is not available except for a small graphics editor for the VODL language.

Nevertheless, this work offers one formalism (algebraic specification) for several specifications which are necessary in order to achieve a visual language editor. Note that the semantic specification of a visual language is similar to the productions of a VL specification generated by the GENGED specification tool. These productions define already the editor operations of a generated visual language editor. However, an interesting question results from the so-called semantics definition in the VASE formalism: is it possible to extend a visual language editor to a visual language grammar editor (VL-GE) similar to the grammar editor available in the GENGED specification tool? A VL-GE could be used to define dynamic semantics of a language as, e.g. used for simulation, interpretation, dynamic verification, etc.

Chapter 9

Conclusion

We first summarize the main results of this thesis and then give an outlook to future work.

9.1 Summary

In this thesis we have introduced the GENGED approach for the visual definition of visual languages yielding a visual language specification. Visual languages are specified by an alphabet and a grammar ensuring Requirement 2.4.1. The alphabet is represented by an attributed graph structure signature together with a constraint-satisfaction problem. An algebraic graph structure grammar represents the grammar. Both, the signature and the grammar define the static aspects of a visual language specification. Visual sentences can be manipulated with respect to the corresponding visual language specification. The manipulation process is defined by algebraic graph structure transformation and graphical constraint solving. Both define the dynamic aspects of a visual language specification.

With respect to the abstract syntax level of visual sentences, symbols are represented by nodes and links are represented by edges. The concrete syntax level is defined by symbol graphics and graphical constraints. Each symbol node is attributed with a symbol graphic. For each link exists at least one constraint between the corresponding symbol graphics. Visual sentences being represented by graphs or, more precisely, by attributed graph structures ensure Requirement 2.2.1. Moreover, the distinction between abstract syntax and concrete syntax of visual languages according to [AER96, RS96] offers a modular design forming the basis for future improvements and extensions. Together with the underlying grammar formalism based on [LKW93, EHK+97], it ensures Requirement 2.1.1.

Visual language specifications can be visually defined by the GENGED specification tool. Thus, Requirement 2.5.1 is ensured. In analogy to the parts of visual language specifications, the specification tool comprises an alphabet editor and a grammar editor as well. The visual language specification is the input of a visual language editor supporting the manipulation of visual sentences in a syntax-directed way. Exactly this was desired and formulated in Requirement 2.5.2. However, the transformation process is supported by the graph transformation system AGG [TER99]. The constraints are solved by the graphical constraint solver PARCON [Gri96]. Without any doubt, the GENGED environment shows that our conceptual model is realistic and feasible.

Beside theoretical considerations and implementational aspects, we have shown that a variety of visual languages can be captured by the GENGED approach. This concerns the expressiveness of the formalism as well as the visual representation types of [BB94]. The underlying graph grammar formalism is not restrictive. Moreover, it allows using negative application conditions in grammar productions. Thus, Requirement 2.3.1 is ensured. The GENGED approach supports a broad spectrum of visual means. Both, the powerful formalism and the powerful constraint system allow for specifying numerous visual languages.

The visual definition of visual languages captures numerous of aspects. These aspects have been introduced in Chapter 1 and refined in Chapter 2. They have determined the requirements and hence, they have influenced the development of the GENGED approach. Nevertheless, the GENGED approach offers many interesting topics for discussion. These topics concern possibly future extensions that are proposed in the following.

9.2 Outlook

Most formalisms that are applied to visual languages support parsing of visual sentences. Thus, they are mostly restricted to context-free languages. Although syntax-directed editing offers user-guidance, free-hand editing should be supported, too. The advantage of parsing facilities is given by the concise visual language specification. The visual language designer must not specify all the language-specific edit commands of a visual language editor. Nevertheless, the user interactions must be specified in order to get an operational semantics of the visual language editor. Therefore, at least two possibilities are conceivable. On the one hand, user interactions can be tightly integrated in a visual language editor. On the other hand, it is imaginable that the visual language editor gets the interaction specification as parameter. In the latter case, the interactions must be specified for each visual language. Reusability aspects should be regarded. Independently which kind of approach is favored, if offering parsing facilities, the GENGED approach needs to be extended with integration concepts for interaction specifications.

Specification of user interaction User interactions can be specified by automata as in DIAGEN [VM94]. Of course, the automata language can be specified by using the GENGED specification tool. We obtain a visual language editor allowing the editing of automata. This editor must be extended to a rule editor that supports the specification of interaction rules. Due to the modular design of GENGED, it seems easy to achieve such a specific rule editor. The GENGED grammar editor can be used for this purpose. Instead of alphabet productions, the corresponding edit commands are then the user-defined visual language productions. Nevertheless, it remains an open question how to integrate a specific visual language specification with an interaction specification in a visual language editor. The basis may be given by the abstract syntax level of both visual languages.

Specification of simulation and animation The integration of different specifications in the GENGED approach is a major topic concerning future extensions. Integration concepts are necessary with respect to several purposes. User interaction is in the fore, but animation and simulation of visual sentences as well. The main use of Petri-nets as proposed in [Pad96, Pad99] is based on animation and simulation. The encouragement of the GENGED approach in the work of the Petri-net group [WER98] is in progress. The first step towards this direction has been the specification of *Algebraic High-Level nets* (AHL nets) using the GENGED approach [Erm98, BER00a, BEE00]. The language is described by rules where the markings can be visualized as icons. These icons represent datatypes that can be shown to the user if desired. This kind of refinement requires at least an extension of the datatype concepts. The specification of AHL nets forms the basis for further developments like the specification of a visual language for parameterized net classes. The corresponding language then allows an easy and intuitive usage with different Petri-net types.

Animation and simulation aspects of a specific visual languages can be specified by the visual means of the visual language like the automata language. But the execution process must also be specified. Perhaps it is possible to use the graph transformation system AGG allowing to apply one rule n times. However, it is necessary that several rules can be applied. Moreover, the application of rules should be driven by control structures. These features must be analyzed and suitable interfaces have to be developed [BER00b].

Code generation The availability of scanners and parsers in a visual language editor concern not only the possibility to allow free-hand editing. They allow also generating some code. The basis is given by the consice visual language specification if parsing facilities are available. The generation of output code then depends on the specific visual language. For visual programming languages, executable code can be generated. Class diagrams belong to the category of visual modeling languages. Suitable classes of an object-oriented programming language like Java can be generated. The program logic has to be added by the user. Suitable concepts for code generation based on parsing facilities should be developed for the GENGED approach.

ViewPoints In order to support multiple perspectives in software development, a scheme which expresses explicitly all the views held by the various humans involved is needed. These are requirements engineer, software architect, client, user, etc. The ViewPoint framework has been proposed in [GEMT99] as a conceptual framework for expressing such a multiple perspective setting in software development projects. It is based on the graph transformation system AGG that is being extended by distribution aspects [GPST00]. Since most methods in software development are based on visual languages, the encouragement of GENGED is planned.

User's perception and acception The topics mentioned so far define a great challenge to the GENGED approach. Nevertheless, realizing these topics allow to proof the user's perception and acception. The first step towards this direction will be the comprehensive specification of several visual languages [BENS99]. The specification of visual languages concerns the abstract syntax and the concrete syntax as well. Without any doubt, the distinction between abstract syntax and concrete syntax offers the basis for various extensions like, e.g. model evolution as proposed in [PEB00]. The use of abstract syntax is not restricted to semantic definitions. It forms also the basis for different layouts [BEE00].

Bibliography

[AER96] M. Andries, G. Engels, and J. Rekers. How to represent a Visual Program? In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[Bar98] R. Bardohl. GENGED - A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars. In *Proc. IEEE Symposium on Visual Languages*, pages 48–55, Halifax, Canada, September 1998.

[BB94] M. M. Burnett and M. J. Baker. A Classification System for Visual Programming. *Journal of Visual Languages and Computing*, 5(3):287–300, 1994.

[BC94] R. Bardohl and I. Claßen. Graphical Support for Prototyping of Algebraic Specifications. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, pages 19–26. Informatik aktuell, 1994.

[BE99] R. Bardohl and H. Ehrig. Conceptual Model of the Graphical Editor GENGED for the Visual Definition of Visual Languages. In *Theory and Application of Graph Transformation*, number 1764 in LNCS, pages 252–266. Springer, 1999.

[BEE00] R. Bardohl, H. Ehrig, and C. Ermel. Generic Description, Behaviour and Animation of Visual Modeling Languages. In *Proc. Integrated Design and Process Technology (IDPT 2000)*, Dallas (Texas), USA, June 2000.

[BENS99] R. Bardohl, C. Ermel, M. Niemann, and T. Schultzke. Specification of Visual Languages using GENGED. Technical report, TU Berlin, 1999. In preparation.

[BER00a] R. Bardohl, C. Ermel, and L. Ribeiro. Towards Visual Secification and Animation of Petri Net Based Models. In *Proc. Joint APPLIGRAPH/GETRATS Workshop on Graph Transformation Systems (GRATRA 2000, Satellite Event of ETAPS 2000)*, pages 22–31, Berlin, Germany, March/April 2000.

[BER00b] R. Bardohl, C. Ermel, and L. Ribeiro. A Modular Approch to Animation of Simulation Models. In *Proc. 14th Brazilian Symposium on Software Engineering*, Joao Pessoa, Brazil, October 2000.

[BGL95] M. M. Burnett, A. Goldberg, and T. G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications Co., Greenwich, 1995.

[BNS00] R. Bardohl, M. Niemann, and M. Schwarze. GENGED – A Development Environment for Visual Languages. In *Application of Graph Transformations with Industrial Relevance*, number 1779 in LNCS, pages 233–240. Springer, 2000.

[Bor81] A. H. Borning. The Programming Language Aspects of ThingLab, A Constraint Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[Bor90] A. H. Borning. The Programming Language Aspects of ThingLab, A Constraint Oriented Simulation Laboratory. In *[Gli90]*, pages 416–449. 1990. Reprint from [Bor81].

[Bor94] A. H. Borning, editor. *Principles and Practice of Constraint Programming, Proc. 2nd Int. Workshop (PPCP'94)*, number 874 in LNCS. Springer, 1994.

[BRR87] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets: Central Models and Their Properties*. Springer, number 254 in LNCS. Springer, 1987.

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In [Roz99], pages 103–180.

[Bur96] P. Burmeister. A Model Theoretic Oriented Approach to Partial Algebras. *Mathematical Research – Mathematische Forschung*, 32:257–274, 1996.

[CC91] G. Costagliola and S.-K. Chang. Parsing 2-D languages with positional grammars. In *Proc. of 1991 International Workshop on Parsing Technologies (IWPT'91)*, pages 218–224, Cancun, Mexico, 1991. Association for Computational Linguistics.

[CEW93] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development - The ACT Approach*. AMAST Series in Computing Vol. 1. World Scientific, 1993.

[CHZ95] W. Citrin, R. Hall, and B. Zorn. Programming with Visual Expressions. In *Proc. IEEE Symp. on Visual Languages*, pages 294–301, Darmstadt, Germany, September, 5-9 1995.

[CIL86] S.-K. Chang, T. Ichikawa, and P.A. Ligomenides, editors. *Visual Languages*. Management and Information Systems Series. Plenum Press, New York, 1986.

[CL95] I. Claßen and M. Löwe. Scheme Evolution in Object-Oriented Models - A Graph Transformation Approach. In *ICSE '95 - 17th Workshop on Formal Methods Application in Software Engineering Practice*, Seattle, Washington, April 1995.

[CLOT97] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A Framework of Syntactic Models for the Implementation of Visual Languages. In *Proc. IEEE Symp. on Visual Languages*, pages 58–65, Capri, Italy, September 1997.

[CLOT98] G. Costagliola, A.De Lucia, S. Orefice, and G. Totora. Positional Grammars: A Formalism for LR-Like Parsing of Visual Languages. In *[MM98b]*, pages 171–192. 1998.

[COL94] G. Costagliola, S. Orefice, and A. De Lucia. Towards Efficient Parsing of Diagrammatic Languages. In *Proc. IEEE Workshop Theory of Visual Languages*, 1994.

[COL95] G. Costagliola, S. Orefice, and A. De Lucia. Automatic Generation of Visual Programming Environments. *IEEE Computer*, 28(3):56–66, March 1995.

[COP+93] G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M Tucci. Automatic parser generation for pictorial languages. In *Proc. IEEE Symp. on Visual Languages*, pages 306–313, 1993.

[Cou90] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 193–242. Elsevier, Amsterdam, 1990.

[CRM78] S. K. Chang, J. Reuss, and B. H. McCormick. Design considerations of a pictorial database system. *Int. Journal of Policy Anal. Inf. Syst.*, 1(2):49–70, 1978.

[DÜ95] T.B. Dinesh and S.M. Üsküdarlı. Visual Object Definition Language. In R.C. Veltkamp and E.H. Blake, editors, *Proceedings of the fifth Eurographics Workshop on Programming Paradigms in Graphics*, pages 109–124, 1995.

[DÜ96] T.B. Dinesh and S.M. Üsküdarlı. Specifying Input and Output of Visual Languages. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[DvB97] R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In [Roz97], pages 247–312.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop*, number 73 in LNCS, pages 1–69. Springer, 1979.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EACTS Monographs on Theoretical Computer Science*. Springer, Berlin, 1985.

[Erm98] C. Ermel. Generierung eines graphischen Editors für Algebraische High-Level Netze mit GENGED. TU Berlin, Student's Project Status Report, 1998.

[FMB90] B. N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, 1990.

[FTTV96] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello. Relation Grammars: A Formalism for Syntactic and Semantic Analysis of Visual Languages. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[GEMT99] M. Goedicke, B. Enders, T. Meyer, and G. Taentzer. Tool Support for ViewPoint-oriented Software Development: Towards Integrating Multiple Perspectives by Distributed Graph Transformation. In *Proc. Application of Graph Transformations with Industrial Relevance*. RWTH Aachen, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gip75] J. Gips. *Shape Grammars and Their Uses, Artificial Perception, Shape Generation and Computer Aesthetics*. Birkhäuser Verlag, Basel, 1975.

[Gli90] E. P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Gol91a] E.J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

[Gol91b] E.J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.

[Göt87] H. Göttler. Graph Grammars and Diagram Editing. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, number 291 in LNCS, pages 216–231. Springer, 1987.

[GP98] M. Gogolla and F. Parisi-Presicce. State Diagrams in UML - A Formal Semantics using Graph Transformation. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, pages 55–72. Technical University of Munich, Technical Report TUM-I9803, 1998.

[GPST00] M. Große-Rhode, F. Parisi-Presicce, M. Simeoni, and G. Taentzer. Modeling Distributed Systems by Modular Graph Transformation Based on Refinement via Rule Expressions. In *Application of Graph Transformations with Industrial Relevance*, number 1779 in LNCS, pages 31–45. Springer, 2000.

[GR90] E.J. Golin and S.P. Reiss. The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing*, 1(2):141–156, 1990.

[Gri96] P. Griebel. *ParCon - Paralleles Lösen von grafischen Constraints*. PhD thesis, Paderborn University, February 1996.

[Haa96] V. Haarslev. A Fully Formalized Theory for Describing Visual Notations. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8):231–274, 1987.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special Issue of Fundamenta Informaticae*, 26(3,4), 1996.

[HM90a] R. Helm and K. Marriott. Declarative Specification of Visual Languages. In *Proc. IEEE Symp. on Visual Languages*, pages 98–103, Skokie, Illinois, October, 4-6 1990. Extended version can be found in [HM90b].

[HM90b] R. Helm and K. Marriott. Declarative specification of visual languages. Technical Report RC 15813, IBM T.J. Watson Research Center, 1990.

[HMTW95] R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed Graph Transformations with Controlled Application of Rules. In G. Valiente and F. Rossello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. Technical Report B - 19, Universitat de les Illes Balears, 1995.

[HW95] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[ITU93] ITU-T, Geneva. *Recommendation Z.120: Message Sequence Chart (MSC)*, 1993. See also http://www.win.tue.nl/win/cs/fm/sjouke/msc.html.

[Kli93] P. Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Koc99] M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems*. PhD thesis, Technical University, Berlin, 1999.

[Kor96] M. Korff. *Generalized Graph Structure Grammars with Applications to Concurrent Object-Oriented Systems*. PhD thesis, TU Berlin, 1996.

[KS90] K. M. Kahn and V. Saraswat. Complete Visualizations of Concurrent Programs and Their Executions. In *Proc. IEEE Symp. on Visual Languages*, pages 7–15, Skokie, Illinois, October, 4-6 1990.

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley & Sons Ltd, 1993.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[Mar94] K. Marriott. Constraint Multiset Grammars. In *Proc. IEEE Symp. on Visual Languages*, pages 118–125, St. Louis, Missouri, October, 4-7 1994.

[Mey92] B. Meyer. Pictures Depicting Pictures. In *Proc. IEEE Symp. on Visual Languages*, pages 41–47, Seattle, WA., October 1992.

[MG97] M. Minas and J. Gottschall. Specifying Animated Diagram Languages. In *Proc. Workshop on Theory of Visual Languages*, pages 51–59, Capri, Italy, 27 September 1997.

[Min97] M. Minas. Diagram Editing with Hypergraph Parser Support. In *Proc. IEEE Symp. on Visual Languages*, pages 226–233, Capri, Italy, September 1997.

[Min98] M. Minas. Hypergraphs as a Uniform Diagram Representation Model. In *Proc. Theory and Application of Graph Transformation*, pages 24–31, Paderborn, Germany, November 1998.

[MM97] K. Marriott and B. Meyer. On the Classification of Visual Languages by Grammar Hierarchies. *Journal of Visual Languages and Computing*, 8(4):375–402, 1997.

[MM98a] K. Marriott and B. Meyer. The CCMG Visual Language Hierarchy. In *[MM98b]*, pages 129–170. 1998.

[MM98b] K. Marriott and B. Meyer, editors. *Visual Language Theory*. Springer, New York, 1998.

[MV93] M. Minas and G. Viehstaedt. Specification of Diagram Editors Providing Layout Adjustments. In *Proc. IEEE Symp. on Visual Languages*, pages 324–329, 1993.

[MV95] M. Minas and G. Viehstaedt. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In *Proc. IEEE Symp. on Visual Languages*, pages 203–210, Darmstadt, Germany, September, 5-9 1995.

[Mye86] B. A. Myers. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proc. CHI'86: Human Factors in Computing Systems*, pages 59–66, New York, 1986. ACM Press.

[Mye90] B.A. Myers. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *[Gli90]*, pages 33–40. 1990. Reprint from [Mye86].

[NB00] M. Niemann and R. Bardohl. Tool-based Specification of Visual Languages and Graphic Editors. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in LNCS, pages 456–470. Springer, 2000.

[Nie99] M. Niemann. Konzeption und Implementierung eines generischen Grammatikeditors für visuelle Sprachen. Master's thesis, TU Berlin, 1999.

[NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), 1973.

[Pad96] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement*. PhD thesis, Technical University Berlin, 1996. Shaker Verlag.

[Pad99] J. Padberg. The Petri Net Baukasten: An Application-Oriented Petri Net Technology. In *Int. Colloquium on Petri Net Technologies for Modeling Communication Based Systems*, pages 191–209. Fraunhofer Gesellschaft ISST, October 1999.

[PEB00] J. Padberg, C. Ermel, and R. Bardohl. Rule-Based and Visual Model Evolution using GENGED. In *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (Satellite Event of ICALP'2000)*, Geneva, Switzerland, July 2000.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[PN83] M. C. Pong and N. Ng. PIGS - A System for Programming with Interactive Graphical Support. *Software, Practice and Experience*, 13(9):847–855, 1983.

[PSG95] M. Pöpping, G. Szwillus, and P. Griebel. Objection - eine Entwicklungsumgebung zum Entwurf constraint-basierter Benutzungsoberflächen. In *Proc. GI Softwaretechnik'95-Tagung Braunschweig*, pages 136–143, 1995.

[Rat99a] Rational Software Corporation. Rational Products - Rational Rose. http://www.rational.com/products, 1999.

[Rat99b] Rational Software Corporation. UML – Unified Modeling Language. http://www.rational.com/uml, 1999.

[Ros76] A. Rosenfeld. Array and Web Languages: An Overwiew. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 517–529. North Holland, Amsterdam, 1976.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 1: Foundations*. World Scientific, Singapore, 1997.

[Roz99] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformaitons, Volume 2: Applications*. World Scientific Publishing, Singapore, 1999.

[RS95] J. Rekers and A. Schürr. A Graph Grammar Approach to Graphical Parsing. In *Proc. IEEE Symp. on Visual Languages*, pages 195–202, Darmstadt, Germany, September, 5-9 1995.

[RS96] J. Rekers and A. Schürr. A graph-based Framework for the Implementation of Visual Environments. In *Proc. IEEE Symp. on Visual Languages*, pages 148–155, Boulder, Colorado, September 1996.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[San95] M. Sannella. The SkyBlue Constraint Solver and Its Applications. In *[Sv95]*, pages 385–406, 1995.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, Cambridge, MA, 1993.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen Anwendungsbeispiele und Werkzeugunterstützung*. Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[Sch98] J. Schiffer. *Visuelle Programmierung*. Addison-Wesley, 1998.

[Sch99] M. Schwarze. Konzeption und Implementierung eines generischen Alphabeteditors für visuelle Sprachen. Master's thesis, TU Berlin, 1999.

[Shu86] N. C. Shu. Visual Programming Languages: A Perspective and a Dimensional Analysis. In *[CIL86]*. 1986.

[Shu90] N. C. Shu. Visual Programming Languages: A Perspective and a Dimensional Analysis. In *[Gli90]*. 1990. Reprint from [Shu86].

[SMFB93] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software – Practice and Experience*, 23(5):529–566, 1993.

[Sut63] I. E. Sutherland. SketchPad: A Man-Machine Graphical Communications System. In *Proc. 23rd IFIPS Spring Joint Computer Conference*, pages 329–345, 1963.

[Sv95] V. Saraswat and P. van Henterrynck, editors. *Constraint Programming*. MIT Press, Cambridge, MA, 1995.

[SWZ99] A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES Approach: Language and Tool Environment. In [Roz99], pages 487–550.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.

[TER99] G. Taentzer, C. Ermel, and M. Rudolf. The AGG Approach: Language and Tool Environment. In [Roz99], pages 551–604.

[Üsk97] S. Üsküdarlı. *Algebraic Specification of Visual Languages*. PhD thesis, University of Amsterdam, 1997.

[VM94] G. Viehstaedt and M. Minas. Interaction in Really Graphical User Interfaces. In *Proc. IEEE Symp. on Visual Languages*, pages 270–277, St. Louis, Missouri, October, 4-7 1994.

[Wag97] A. Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras*. PhD thesis, TU Berlin, 1997.

[Wal72] D. L. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report TR AI271, MIT, MA, 1972.

[Wal75] D. L. Waltz. Understanding Line Drawings of Scenes with Shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, Maidenhead, England, 1975.

[WER98] H. Weber, H. Ehrig, and W. Reisig. Konzeption, theoretische Fundierung, und Validierung einer anwendungsbezogenen Petrinetz-Technologie. Fortsetzungsantrag für die Petri net Forschergruppe an die DFG, 1998.

[Wit92] K. Wittenburg. Earley-style Parsing for Relational Grammars. In *Proc. IEEE Symp. on Visual Languages*, pages 192–199, Seattle, WA., October 1992.

[Wol97] D. Wolz. *Colimit Library for Graph Transformations and Algebraic Specification Techniques*. PhD thesis, TU Berlin, May 1997.

[WW96] K. Wittenburg and L. Weitzman. Relational Grammars: Theory and Practice in a Visual Language Interface for Process Modeling. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 1996.

[Yam79] Yamamoto Tsunetomo. Hagakure – The Book of the Samurai. Kodansha International, 1979. Original: Yamamoto/Tsuramoto, 1710.

Index